XX SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE

ANAIS

16 a 20 de outubro de 2006
Florianópolis, Santa Catarina, Brasil

Promoção

SBC – Sociedade Brasileira de Computação
Comissão Especial de Engenharia de Software

ACM – Association for Computing Machinery
SIGSOFT Special Interest Group on Software Engineering

Edição

Paulo Cesar Masiero
Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo

Organização

UFSC – Universidade Federal de Santa Catarina
IDESTI – Instituto de Desenvolvimento, Pesquisa e Capacitação em Gestão Social de Tecnologia de Informação

Realização

Departamento de Informática e Estatística – UFSC

FICHA CATALOGRÁFICA

Catalogação na fonte pela Biblioteca Universitária da UFSC

S612a Simpósio Brasileiro de Engenharia de Software (20. : 2006 : Florianópolis, SC).
Anais do XX Simpósio Brasileiro de Engenharia de Software / editor Paulo Cesar Masiero. – Florianópolis : Sociedade Brasileira de Computação, 2006.
xvi, 328 p. : il.

ISBN: 85-7669-079-9

1. Engenharia de Software – Congressos. I. Masiero, Paulo Cesar. II. Sociedade Brasileira de Computação. III. SBES (20. : 2006 : Florianópolis, SC). IV. Título.

CDU: 681.31:519.683.2

"Esta obra foi impressa a partir de originais entregues, já compostos pelos autores".

Editoração: Frank Siqueira
Arte Gráfica do Evento/Capa: Sandro Coutinho de Azevedo
Impressão: Mattes Agência Gráfica

20th BRAZILIAN SYMPOSIUM ON SOFTWARE ENGINEERING

PROCEEDINGS

October 16-20, 2006
Florianópolis, Santa Catarina, Brazil

Promotion

SBC – Brazilian Computer Society
Special Committee on Software Engineering

ACM – Association for Computing Machinery
SIGSOFT Special Interest Group on Software Engineering

Editor

Paulo Cesar Masiero
Instituto de Ciências Matemáticas e de Computação
Universidade de São Paulo

Organization

UFSC – Universidade Federal de Santa Catarina
IDESTI – Institute for Development, Research and Education on Social Management of Information Technology

Realization

Departamento de Informática e Estatística – UFSC

Apresentação

O Simpósio Brasileiro de Engenharia de Software (SBES) é um evento promovido anualmente pela Sociedade Brasileira de Computação (SBC), por intermédio da Comissão Especial de Engenharia de Software (CEES). Sua criação se deu em 1987 e neste ano, portanto, está em sua 20ª edição. É para mim uma alegria muito grande ter participado do primeiro SBES realizado em Petrópolis e estar coordenando esta 20ª edição.

O SBES é classificado no Qualis da Área de Ciências de Computação da CAPES como Nacional A. Este evento conta, desde 2001, com o apoio do Grupo de Interesse em Engenharia de Software (SIGSOFT) da *Association for Computing Machinery* e, desde o ano de 2005, com o apoio da *IEEE Computer Society*. Além disso, neste ano os três melhores artigos, a serem selecionados por uma comissão especialmente designada para esse fim, serão publicados em uma edição especial do *Journal of Systems and Software* – juntamente com três artigos do Simpósio Brasileiro de Bancos de Dados – que é classificado no Qualis-CC como Internacional A. Isso tudo mostra a maturidade atingida pelo evento e pelo grupo de pesquisadores brasileiros em Engenharia de Software.

O público-alvo do SBES reúne as comunidades acadêmica e profissional de Engenharia de Software no Brasil. O evento oferece as seguintes atividades: apresentação de artigos técnicos com conteúdos inovadores; apresentação de palestras e mini-tutoriais por pesquisadores convidados e de renome internacional; e, apresentação de tutoriais selecionados e de mini-cursos. O SBES agrega atividades tradicionais que são: o XI Workshop de Teses em Engenharia de Software e a Sessão de Ferramentas. São também realizados *workshops* que tratam de assuntos atuais de Engenharia de Software. Cada uma dessas atividades é supervisionada por um comitê de programa especializado que é responsável pela seleção criteriosa dos trabalhos a serem apresentados.

Em 2006, o SBES será realizado em Florianópolis, Santa Catarina, em conjunto com o 21º Simpósio Brasileiro de Bancos de Dados (SBBD). A Coordenação Geral institucional é da Universidade Federal de Santa Catarina, por meio de uma equipe de docentes e funcionários do Departamento de Computação coordenada pelo Prof. Raul S. Wazlawick. As atividades se estenderão pelos cinco dias da semana de 16 a 20 de outubro.

Nesta edição, teremos como palestrante destaque nacional o Prof. Julio César Leite da Pontifícia Universidade Católica do Rio de Janeiro. As palestras convidadas e mini-tutoriais serão apresentadas pelos seguintes pesquisadores e professores: Richard W. Selby, da Northrop Grumman Space, Linda Northrop e Paulo Merson, do Software Engineering Institute, atuando profissionalmente nos Estados Unidos da América, além de Carlo Ghezzi do Instituto Politécnico de Milão, na Itália.

O Comitê de Programa do SBES envolve pesquisadores representativos da comunidade de engenharia de software nacional de todas as regiões. Além disso, cerca de 30% dos membros deste comitê são estrangeiros ou brasileiros que trabalham em universidades no exterior. Neste ano, foram recebidos 134 artigos, dos quais foram selecionados vinte (20) para apresentação durante o simpósio e publicação em seus anais. Dentre os artigos aceitos, 25% deles têm o autor ou um co-autor residindo no exterior. O processo de avaliação garantiu que cada artigo tivesse três avaliações. A primeira etapa de avaliação envolveu este coordenador e os membros do comitê de programa. Em alguns casos, foram designados avaliadores externos ao comitê para auxiliar no processo de revisão. Com base nessas avaliações, o Comitê Diretivo fez a avaliação final e a escolha dos vinte (20) trabalhos selecionados.

O SBES contou com a colaboração dos seguintes professores para a realização dos diversos eventos que o compõem: o Prof. Julio César S. do P. Leite (PUC-Rio), Coordenador do Comitê de Workshops; o Prof. Márcio Delamaro (UNIVEN), Coordenador do Comitê da Sessão de Ferramentas; a Profa. Itana Gimenes (UEM), Coordenadora do Comitê de Tutoriais, o Prof. Manoel Mendonça (UNIFACS), Coordenador do XI WTES; e, o Prof. Raul S. Wazlawick, também coordenador da Sessão de Mini-cursos.

Gostaria de agradecer de um modo geral a todos os que colaboraram na organização do XX SBES e especialmente aos membros do Comitê de Programa e demais revisores de artigos pela pontualidade e alta qualidade do trabalho produzido; ao Comitê Diretivo, pelo apoio em todas as fases do processo; ao Prof. Raul S. Wazlawick, pela dedicação e constante apoio, aos coordenadores dos eventos, listados acima, e aos alunos que comigo colaboraram: Valter Camargo e Otávio Lemos, do ICMC, e ao Ricardo Neisse, da UFRGS/SBC, pelo apoio ao uso do Sistema JEMS.

Florianópolis, outubro de 2006.

Paulo Cesar Masiero

Coordenador do Comitê de Programa do SBES 2006

Presentation

The Brazilian Symposium on Software Engineering – in Portuguese: Simpósio Brasileiro de Engenharia de Software (SBES) – is an event promoted by the Brazilian Computer Society – Sociedade Brasileira de Computação (SBC) – through its Special Committee on Software Engineering. The first edition of the Symposium was in 1987 and this year it is in its 20th edition. I am very pleased to have participated in the first SBES held in Petrópolis and to be the chairman of the Program Committee in this 20th edition.

SBES is classified in Computer Science profile of quality of CAPES as National A event. Since 2001 the Special Interest Group on Software Engineering (SIGSOFT) of the Association has supported this event for Computing Machinery; and, since 2005, the IEEE Computer Society has supported it. Additionally, in this year the three best papers, to be selected by a committee assembled specially for it, will be published in a special issue of the Journal of Systems and Software together with three papers of the Brazilian Symposium on Data Bases, classified by the CAPES profile of Computer Science as International A. All these achievements show the maturity reached by our event and by the group of Software Engineering Brazilian researchers.

SBES is an event that gathers together both Brazilian and international researchers, students and practitioners. Several activities take place during the event: technical sessions where original research papers are presented; keynote speeches; mini-tutorials presented by invited and internationally renowned speakers; and selected tutorials. In addition, the SBES has several associated activities: selected mini-courses; a thesis and dissertations workshop; several selected workshops; and presentation of selected software tools. Specific program committees that are responsible for the careful selection of the presentations supervise all of theses activities.

In 2006, SBES will be held in Florianópolis, Santa Catarina, together with the 21st Brazilian Symposium on Data Bases (SBBD). The Federal University of Santa Catarina, through a group of faculty members of the Computer Science Department, is responsible for the general coordination of both events. The General Chair is Prof. Raul S. Wazlawick. The SBES´s activities will take place for five days during the week of 16th to 20th of October,

This year we will have as national keynote speakers the Prof. Julio César Leite, from the Pontifícia Universidade Católica of the Rio de Janeiro, Brazil. The international keynote speakers are Dr. Richard W. Selby, from Northrop Grumman Space; Linda Northrop and Paulo Merson, from Software Engineering Institute, working professionally in the United States of America, and Carlo Ghezzy, from the Instituto Politecnico of Milan, Italy.

The Program Committee is composed by researchers representatives of the Brazilian community of software engineering, from all regions of the country. Additionally, foreigners or Brazilians working in Universities abroad comprise 30% of the PC. In this year, 134 articles have been submitted and 20 have been chosen for presentation. Among the accepted articles, 25% have an author or co-author living abroad. Three reviewers reviewed each article. The first step of the reviewing process involved this chair and tthe program committee members. In several cases, external reviewers have been assigned to help in the reviewing process. Based on these reviews, the steering committee has made its final evaluation and the choice of the twenty (20) selected articles.

The following professors collaborated with the organization of the SBES: Prof. Julio César S. do P. Leite (PUC-Rio), Chairman of the Workshops track; Prof. Márcio Delamaro (UNIVEN), Chairman of the Software Engineering Tools track; Prof. Itana Gimenes (UEM), Chairwoman of the Tutorials track; Prof. Manoel Mendonça (UNIFACS), Chairman of the XI WTES – Thesis and Dissertation Workshop; and Prof. Raul S. Wazlawick, Chairman of the mini-courses track.

I would like to express my gratitude to all that collaborated with the organization of the XX SBES and give my special thanks to the members of the Program Committee and all other reviewers for the timeliness and high quality of their work; to the Steering Committee for the support in all phases of this process; to Prof. Raul S. Wazlawick, for the dedicated work and constant support, to the chairmen of the events mentioned above and to the students that helped me: Valter Camargo and Otávio Lemos, from ICMC, and Ricardo Neisse, from UFRGS/SBC, for the support in using the JEMS system.

Florianópolis, October 2006.

Paulo Cesar Masiero

Program Committee Chair, SBES 2006

Paulo Cesar Masiero é Professor Titular do Departamento de Ciências de Computação e Estatística do Instituto de Ciências Matemáticas e de Computação (ICMC), da USP. Foi o criador do Curso de Sistemas de Informação e é atualmente o Coordenador desse curso na Escola de Artes Ciências e Humanidades da USP, em São Paulo. É atualmente membro da Congregação dessa Escola e também seu representante no Conselho Universitário da USP. Foi Presidente da Comissão Central de Informática e, até recentemente, da Coordenadoria de Tecnologia da Informação da Universidade de São Paulo, da qual foi o criador. Foi também Diretor do Centro de Informática de São Carlos, da USP. Foi Vice-diretor e depois Diretor do Instituto de Ciências Matemáticas e de Computação da USP e também chefe do Departamento de Ciências de Computação desse Instituto. Foi membro da Comissão de Especialistas de Ensino de Computação e Informática do MEC. Foi membro do Conselho Diretor da Sociedade Brasileira de Computação em dois mandatos. É orientador do programa de pós-graduação em Computação e Matemática Computacional no ICMC, do qual já foi coordenador. Orientou seis doutorados e trinta e três mestrados. Tem dois livros publicados, cerca de vinte artigos científicos publicados em periódicos nacionais e internacionais e mais de cem artigos científicos completos em conferências nacionais e internacionais com arbitragem. É atualmente pesquisador do CNPq, nível I-D. Foi também editor do *Journal of the Brazilian Computer Society* durante três anos. É licenciado em Matemática pela Universidade Estadual Paulista (1975), Mestre em Ciências de Computação pelo Instituto de Ciências Matemáticas de São Carlos, USP – atual ICMC – (1979) e Doutor em Administração (Sub área: Métodos Quantitativos e Tecnologia da Informação) pela Faculdade de Economia e Administração (FEA) da USP (1984). Realizou programas de Pós-doutorado na *University of Michigan* (02/85 a 02/86), USA, *Strathclyde University* (12/89 a 02/90), UK, e *Technical University of Denmark* (12/92 a 08/93), Dinamarca.

XX Simpósio Brasileiro de Engenharia de Software
20th Brazilian Simposium on Software Engineering

Comitê Diretivo
Steering Committee

Arndt von Staa, PUC-Rio
Guilherme Travassos, UFRJ
Itana Maria Gimenes, UEM
Jaelson Brelaz de Castro, UFPE
Paulo Cesar Masiero, USP

Coordenador Geral
General Chair

Raul Sidnei Wazlawick, UFSC

Coordenador do Comitê de Programa
Program Committee Chair

Paulo Cesar Masiero, USP

Coordenador de Tutoriais
Tutorials Chair

Itana Maria de Souza Gimenes, UEM

Coordenador Workshop de Teses em Engenharia de Software
Software Engineering Theses Workshop Chair

Manoel Gomes de Mendonça Neto, UNIFACS

Coordenador da Sessão de Ferramentas
Tools Session Chair

Marcio Delamaro, UNIVEM

Coordenador de Workshops
Workshops Chair

Julio César Sampaio do Prado Leite, PUC-Rio

Membros do Comitê de Programa
Program Committee Members

Adenilso Simão, ICMC-USP, BR
Alessandra Russo, Imperial College, GB
Alessandro Garcia, Un. of Lancaster, GB
Alexandre Vasconcelos, UFPE, BR
Alvaro Moreira, UFRGS, BR
Ana Moreira, Un. Nova de Lisboa, PT
Ana C. V. de Melo, IME/USP, BR
Ana Regina Rocha, COPPE/UFRJ, BR
Antonio Francisco Prado, UFSCar, BR
Arndt von Staa, PUC-Rio, BR
Augusto Sampaio, UFPE, BR
Cecilia Rubira, UNICAMP, BR
Christina v.Flach G. Chavez, UFBA, BR
Cláudia Werner, COPPE / UFRJ, BR
Daniel Berry, Un. of Waterloo, CA
David Déharbe, UFRN, BR
Emilia Mendes, Un. of Auckland, NZ
Fernanda Alencar, UFPE, BR
Gustavo Rossi, Un. Nac. de La Plata, AR
Guilherme Travassos, COPPE/UFRJ, BR
Itana Gimenes, UEM, BR
Jaelson Freire B. de Castro, UFPE, BR
Jeffrey Carver, Mississippi St. Un., USA
João Araújo, Un. Nova de Lisboa, PT
João Cangussu, UT-Dallas, USA
João Falcão e Cunha, Un. do Porto, PT
José Carlos Maldonado, ICMC/USP,BR
Jorge Luis Becerra, POLI/USP, BR
Julio Leite, PUC-Rio, BR
Karin Breitman, PUC-Rio, BR
Leila Ribeiro, UFRGS, BR
Leila Silva, UFSE, BR
Leonor Barroca, Open Un., GB
Manoel Mendonça, UNIFACS, BR
Marcello Visconti, Un. Téc. FSM, CL
Marcelo Morandini, UEM, BR
Marcelo Yamaguti, PUCRS, BR
Marcelo Augusto S. Turine, UFMS, BR
Marcio Delamaro, UNIVEM, BR
Marcos Chaim, EACH/USP, BR
Mario Jino, FEEC-UNICAMP, BR
Mariza Bigonha, UFMG, BR
Mauro Pezze, Un. D. Studi di Milano, IT
Márcio Barros, UNIRIO, BR
Nabor Mendonca, UNIFOR, BR
Nelson Rosa, UFPE, BR
Oscar Pastor, Un. Polit. de València, ES
Paolo Giorgini, Un. di Trento, IT
Patricia Machado, UFCG, BR
Paulo Borba, UFPE, BR
Paulo C. Masiero , ICMC/USP, BR
Ricardo Falbo, UFES, BR
Roberto Barros, UFPE, BR
Rodrigo Reis, UFPA, BR
Rosana Braga, ICMC/USP, BR
Regina Braga, UFJF, BR
Reiko Heckel, Un. of Leicester, GB
Ricardo Choren, IME/RJ, BR
Rosângela Penteado, UFSCaR, BR
Rossana Andrade, UFC, BR
Sandra Fabbri, UFSCar, BR
Silvia Vergilio, UFPR, BR
Stephen Fickas, Un. of Oregon, EUA
Toacy Oliveira, PUCRS, BR
Viviane Silva, Un. Complutense, ES
W. Eric Wong, UT-Dallas, EUA
Xavier Franch, Un. Pol. De Catalunya, ES
Yadran Eterovic, PUC-Chile, CL

Revisores
Reviewers

Abraham L. Rabelo, Unilasalle, BR
Adailton Lima, UFPA, BR
Adalberto Crespo, CenPRA, BR
Adriano Albuquerque, UFRJ, BR
Alex Gomes, UFPE, BR
Ana Natali, COPPE/UFRJ, BR
Auri Vincenzi, UFGO, BR
Ayla Débora D. de Souza, UFCG, BR
Breno França, UFPA, BR
Carla Lima Reis, UFPA, BR
Carla G. do N. Macário, Embrapa, BR
Carlos André G. Ferraz, UFPE, BR
Carme Quer, Un. Polit. Catalunya, ES
Clarindo Pádua, UFMG, BR
Claudio Sant'Anna, PUC-Rio, BR
Cleidson Souza, UFPA, BR
Daniela Cruzes, Salvador Un., BR
Donizete Bruzarosco, UEM, BR
Eduardo Kessler Piveta, UFRGS, BR
Elaine Sousa, USP, BR
Elói Favero, UFPA, BR
Elisa Huzita, UEM, BR
Ellen Francine Barbosa, ICMC/USP, BR
Emerson Lima, UFCG, BR
Fabiano Cutigi Ferrari, ICMC/USP, BR
Fabio Campos, UFRJ, BR
Flavia Delicato, UFRJ, BR
Flávio Soares C. da Silva, IME/USP, BR
Francisco Vasconcellos, MB/COPPE, BR
Gisele Craveiro, EACH/SP, BR
Gleison Santos, COPPE/UFRJ, BR
Guilherme Telles, ICMC/USP, BR
Helio Santos, UFPE, BR
Hermano Moura, UFPE, BR
Hyggo Oliveira de Almeida, UFCG, BR
Ivandré Paraboni, EACH/USP, BR
João Dantas, UFMG, BR
Jair Cavalcanti Leite, UFRN, BR
Jobson Silva, UFRJ/COPPE, BR
Jordi Marco, Un. Polit. de Catalunya, ES
Junia Anacleto, UFSCar, BR
Jussara Almeida, UFMG, BR
Kalinka Castelo Branco, UNIVEM, BR
Leliane Nunes de Barros, IME/SP, BR
Leonardo Murta, COPPE/UFRJ, BR
Luciana Foss, UFRGS, BR
Marc Alier, Un. Polit. de Catalunya, ES
Marcelo Cezar Pinto, UFRGS, BR
Marcilio de Souto, UFRN, BR
Marco A, Araújo, COPPE/UFRJ, BR
Marco Tulio Valente, PUCMG, BR
Maria Claudia Emer, UNICAMP, BR
Maria Istela Cagnin, UNIVEM, BR
Mariano Montoni, COPPE / UFRJ, BR
Michel Wermelinger, Open Un., GB
Miguel Argollo, CenPRA, BR
Otávio Lemos, ICMC/USP, BR
Paula Mian, COPPE / UFRJ, BR
Paulo Bueno, CenPRA, BR
Paulo Guerra, Unicamp, BR
Paulo Pires, UFRJ, BR
Patricia Oliveira, EACH/SP, BR
Pere Botella, Un. Polit. de Catalunya, ES
Plinio Leitao Junior, UNICAMP, BR
Rafael Bordini, Un. of Durham, GB
Regiane Brito, UFPR, BR
Reginaldo Ré, ICMC/USP, BR
Renata Fortes, ICMC-USP, BR
Renata Vieira, UNISINOS, BR
Ricardo Souza, UFPR, BR
Rodrigo Ramos, UFPE, BR
Rodrigo R. de Almeida, UFCG, BR
Rodrigo Spínola, COPPE / UFRJ, BR
Roseli Sanchez, ICMC/USP, BR
Sandro R. B. Oliveira, Un. Amazônia, BR
Simone da Costa, UFRGS, BR
Simone Souza, ICMC-USP, BR
Tania Tait, UEM, BR
Thais Vasconcelos Batista, UFRN, BR
Uirá Kulesza, PUC-Rio, BR
Valter Camargo, UNIVEM, BR
Virginia de Paula, UFRN, BR
Viviane Malheiros, SERPRO, BR

Comissão de Seleção de Tutoriais
Tutorials Committee

Ana C. V. de Melo , IME/USP
Antonio Francisco Prado, UFSCar
Arndt von Staa, PUC-Rio
Edmundo Sérgio Spoto, UNIVEM
Gledson Elias, UFPB
Itana Maria de Souza Gimenes, DIN/UEM (Coord.)
Toacy Oliveira, PUC-RS

Comissão de Seleção Workshop de Teses em Engenharia de Software
Software Engineering Theses Workshop Committee

Arndt von Staa, PUC-Rio
Daltro José Nunes, UFRGS
Guilherme Horta Travassos, UFRJ
Itana Maria de Souza Gimenes, UEM
José Carlos Maldonado, USP
Manoel Gomes de Mendonça Neto, UNIFACS (Coord.)
Mario Jino, UNICAMP

Comitê da Sessão de Ferramentas
Tools Session Committee

Auri Vincenzi, UFG
Carla Lima Reis, UFPA
Daltro Nunes, UFRGS
Edmundo Spoto, UNIVEM
Elisa Huzita, UEM
Ellen Francine Barbosa, ICMC/USP
Fabio Lucena, UFG
Frank Siqueira, UFSC
Guilherme Travassos, COPPE/UFRJ
Joao Cavalcanti, UFAM
João Cangussu, UT-Dallas-USA
Jorge Fernandes, UnB
José C. Maldonado, ICMC/USP
Manoel Mendonça, UNIFACS
Marcelo Turine, UFMS
Marcio Delamaro, UNIVEM (Coord.)
Marcos Chaim, EACH/USP
Maria Istela Cagnin, UNIVEM
Mario Jino, FEEC-UNICAMP
Plinio Vilela, UNIMEP
Renato Cerqueira, PUC Rio
Ricardo Falbo, UFES
Ricardo Pereira e Silva, UFSC
Rosângela Penteado, UFSCar
Sandra Fabbri, UFSCar
Silvia Vergilio, UFPR
Simone Souza, ICMC/USP

Comitê da Sessão de Workshops
Workshops Session Committee

Alberto H F Laender, UFMG
Alexandre M. L. de Vasconcelos, UFPE
José Carlos Maldonado, USP
José Palazzo M. de Oliveira, UFRGS
Julio César Sampaio do Prado Leite, PUC-Rio (Coord.)
Renato Cerqueira, PUC-Rio
Rubens Nascimento Melo, PUC-Rio

Comitê Organizador
Organizing Committee

Raul Sidnei Wazlawick, UFSC
Coordenador Geral

Ronaldo dos Santos Mello, UFSC
Coordenador SBBD

Daniele Wazlawick, UFSC
Coordenador Administrativo-Financeiro

Eugênio C. E. Vieira, PGE-SC
Coordenador de Relações Institucionais

Olinto José Varela Furtado, UFSC
Coordenador de Inscrições

Frank Siqueira, UFSC
Coordenador de Publicações

Maria Marta Leite, UFSC
Coordenador de Eventos

Ricardo Pereira e Silva, UFSC
Coordenador de Publicidade

Masanao Ohira, UFSC
Coordenador de Equipamentos

Everton Luiz Vieira, UFSC
Webmaster

Juliana Sellmer, UFSC
Rogério Cid Bastos, UFSC
Álvaro G. Rojas Lezana, UFSC
Rosvelter Coelho da Costa, UFSC
José Eduardo de Lucca, UFSC
Eratóstenes Araújo, SOFTEX
Heitor Blum S. Thiago, SUCESU-SC
Maria Angela Alves, CenPRA
Adoniran B. Cantelli, PowerSolutions
Gislaine Parra Freund, CTAI-SENAI-SC
Everaldo Artur Grahl, FURB
Mirela S. M. A. Notare, Faculdades Barddal
Fernanda dos Santos Cunha, UNIVALI
Mauro N. Madeira, UNISUL
Jolmar Luís Hawerroth, Universidade Estácio de Sá

Sumário/Contents

Sessões Técnicas (ST) /Technical Sessions (TS)

I - Arquitetura de Software

II - Verificação, Validação, Certificação e Teste de Software

III - Especificação de Software

IV - Desenvolvimento de Software baseado em Componentes

V - Desenvolvimento baseado em Aspectos e Frameworks

VI - Gestão de Projetos de Software

VII - Engenharia de Software para Computação Distribuída e Aplicações Ubíquas

Palestras Convidada/Invited Talks

Mini-tutoriais Convidados/Invited Mini-tutorials

Tutoriais/Tutorials

Software Architecture for the Real Time Scheduling of Workflow Management Systems based on a Petri net model

Fernanda Francielle de Oliveira [1], **Stéphane Julia**[1]

[1]Faculdade de Computação, Universidade Federal de Uberlândia,
FACOM-UFU, Av. João Naves de Avila, 2160, P.O. Box 593, 38400-902
Uberlândia-M.G.-Brazil

fernandafrancielle@yahoo.com.br, stephane@facom.ufu.br

Abstract. *In this paper, an approach is proposed to solve the scheduling problem of Workflow Management Systems. The proposed approach uses an activity diagram to show the main activities and the different routings of the Workflow Process. Based on the obtained activity diagram, the corresponding p-time Petri net model is produced. Hybrid resource allocation mechanisms are modelled by an hybrid Petri net in order to represent discrete and continuous resources. Then, a token player algorithm is applied to an example of "Handle Complaint Process" in order to obtain an acceptable scenario corresponding to a specific sequence of activities which respects the time constraints. Finally, a software architecture is proposed for the real time scheduling of Workflow Management Systems.*

Resumo. *Este artigo apresenta uma abordagem para o problema do escalonamento dos Sistemas de Gerenciamento de Workflow. A abordagem proposta utiliza um diagrama de atividades para mostrar as principais atividades e os diferentes roteiros do processo. Baseado no diagrama obtido, o modelo de rede de Petri correspondente é produzido. Mecanismos de alocação de recursos híbridos são modelados por uma rede de Petri híbrida a fim de representar recursos contínuos e discretos. Em seguida, um Jogador de Redes de Petri é aplicado ao exemplo de um "Serviço de Reclamações" a fim de obter um cenário admissível correspondente a uma sequência de atividades que respeita as restrições temporais. Finalmente, é proposta uma arquitetura de software para o escalonamento em tempo real dos Sistemas de Gerenciamento de Workflow.*

1. Introduction

The purpose of Workflow Management Systems [Aalst and Hee 2002] is to execute Workflow Processes. Workflow Processes represent the sequence of activities which have to be executed within an organization to treat specific cases and to reach a well defined goal.

Of all notations used in the Software Industry, UML [OMG 2005] is one of the best accepted. In particular, the activity diagrams of the UML notation seem to be very suitable for proposing approaches to represent Workflow Processes as these diagrams represent basic routings encountered in these Processes which are the sequential routing (the sequential execution of activities), the parallel routing (two or more activities executed

simultaneously) and the selective routing (when a choice must be made between two or more activities).

Several authors have already used UML notations for the specification of Workflow Processes [Eshuis 2002] and [Hruby 1998]. However, UML notations have their limitations when they are used for specifying the real time characteristics of Workflow Management Systems. For example, activity diagrams do not represent real time constraints in a formal way and they do not show in an explicit manner resource allocation mechanisms. As a matter of fact, late deliveries in an organization are generally due to the problem of resource overload. Consequently, the model used at a Workflow Management System level should consider resource allocation mechanisms. In particular, time management of Workflow Processes is crucial for improving the efficiency of Business Processes within an organization.

Petri nets [Murata 1989] are very well adapted to model Real Time Systems, as they allow for a good representation of conflict situations, shared resources, synchronous and asynchronous communication, precedence constraints and explicit quantitative time constraints in the time Petri net case. Some authors have already considered Petri net theory as an efficient tool for the modelling and analysis of Workflow Management Systems [Aalst and Hee 2002], [Covès et al. 1998], [Li et al. 2003], [Wirtz e Giese 2000].

The dynamic behavior of a system imposes a scheduling of control flow. The scheduling problem [Esquirol and Lopez 1995] consists of organizing in time the sequence of activities considering time constraints (time intervals) and constraints of shared resources utilization necessary for activity execution. From the traditional point of view of Software Engineering [Pressman 1995], the scheduling problem is similar to the activity of scenario execution. A scenario execution becomes a kind of simulation which shows the system's behavior in real time. In the real time system case, several scenarios (several cases in a Workflow Management System) can be executed simultaneously and conflict situations which have to be solved in real time (without a backtrack mechanism) can occur if a same non-preemptive resource is called at the same time for the execution of activities which belong to different scenarios.

The fundamental difference between the traditional scheduling problem of Production Systems [Lee and DiCesare 1994] and the scheduling problem of Workflow Management Systems is the nature of the resources used to treat the activities [Julia and de Oliveira 2004]. In the Production System case, resources represent physical equipment and are represented by a simple token in a place. They are discrete type resources. In the Workflow Management System case, resources can represent physical equipment as well as human employees [Aalst and Hee 2002]. For example, it is possible to allocate a nurse in a hospital service to take care of several patients at the same time during her working day. In this case, the nurse could not be seen as a simple discrete token and a model based on an ordinary Petri net would not be able to represent the real features which exist at a Workflow Management System level. Some recent works [Aalst and Hee 2002], [Tramontina et al. 2004] emphasize the lack of results which significantly cover the characteristics of Workflow Management Systems when considering the scheduling problem.

In this paper, an approach based on UML notations and on a p-time hybrid Petri

net model is proposed to solve the scheduling problem of Workflow Management Systems.

2. Workflow Management Systems modelling based on activity diagrams

The "Handle Complaint Process" presented in [Aalst and Hee 2002] will be used to illustrate the specification activity of Workflow Processes. First, an incoming complaint is recorded. Then, the client who has complained and the department affected by the complaint are contacted. The client is approached for more information. The department is informed of the complaint and may be asked for its initial reaction. These two tasks may be performed in parallel. After this, the data is registered and a decision is taken. Depending upon the decision, either a compensation payment is made or a letter is sent. Finally, the complaint is filed.

The proposed approach uses an activity diagram to show the main activities of the system and the different routings of the Workflow Process. The corresponding activity diagram for the "Handle Complaint Process" is the one of figure 1. The advantage of such an example is that it shows the different kind of routings which can exist in a Workflow Process. Swim lanes are used on the activity diagram to indicate in an informal way the principal resources (human and/or equipment resources) used to treat the different activities.

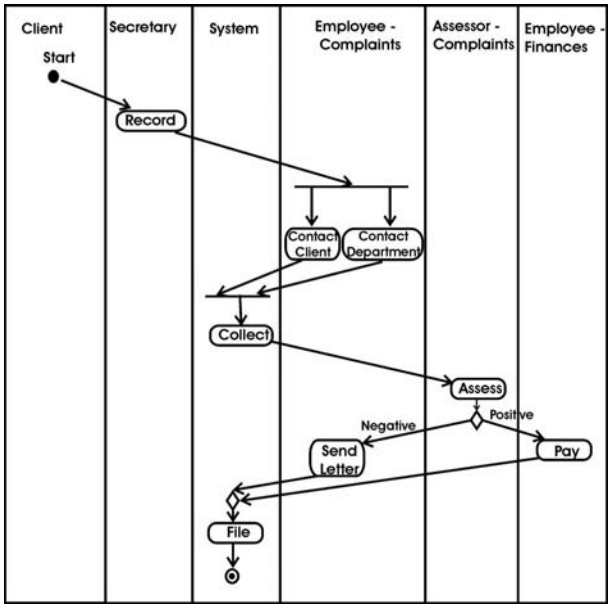


Figure 1. Activity Diagram

3. Workflow Management Systems modelling based on p-time hybrid Petri net

A way of guaranteeing that software specifications are consistent is to use formal methods. But, in practice, the software engineers, because of the complex mathematical definitions of formal notations, prefer to work with methods based on semi-formal notations, like the UML notation for example. It is then of fundamental interest to find a good compromise between the conviviality of semi-formal methods and the rigor of formal methods

by progressively incorporating formal notations within semi-formal specifications, in the context of a multi-formalism approach.

3.1. Routings definition based on ordinary Petri nets

Translating UML diagrams in Petri Net models, as was presented in [Cardoso and Silbertin-Blanc 2001] for example, allows one to define an operational semantic for UML dynamic diagrams in order to know how these diagrams are executed in real time. Nevertheless, it is important to emphasize the difficulty of producing algorithms that allow the automatic translation of a semi-formal model into a formal model. As a matter of fact, an activity diagram could only be seen as an intermediary level of specification between the textual specification of a Workflow Management System and the corresponding formal specification because of the limitations of UML notations for specifying real time characteristics such as resource allocation mechanisms for example.

Based on the activity diagram of figure 1, it is possible to obtain the corresponding Petri net model by representing each activity by a specific place with an input transition which shows the beginning of the activity and an output transition which shows the end of the activity.

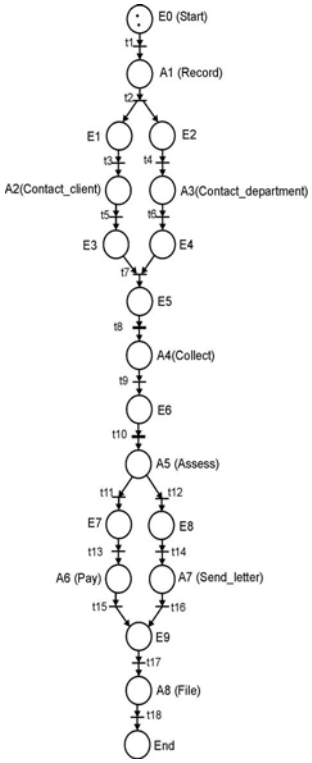


Figure 2. Routing Definition

Such a Petri net based model is shown in figure 2. The activities of the "Handle Complaint Process" are represented by the places A_i (for i=1 to 8) and the waiting times between the sequential activities are represented by the places E_j (for j=0 to 9). As a matter of fact, at the end of an activity, the next can only be initiated if the corresponding

resource is immediately available, which is not necessarily the case. For example, looking at the activity diagram of figure 1, at the end of the "record" activity, the activities "Contact Client" and "Contact Department" could only be performed if at least one employee of the complaints department is available. On the contrary, there will exist waiting times between the "record" activity and the activities "Contact Client" and "Contact Department". Such waiting times are represented in the Petri net of figure 2 by the places E_1 and E_2. The waiting places E_3 and E_4 are then necessary to synchronize the end of the activities "Contact Client" and "Contact Department". These waiting places are not represented in an explicit manner on the activity diagram.

3.2. Explicit time constraints based on a p-time Petri net model

As the actual time taken by an activity in a Workflow Management System is non-deterministic and not easily predictable, a time interval can be assigned to every workflow activity. The existing waiting times between sequential activities can be represented in the same way by an interval whose minimum and maximum bounds will depend on the earliest and latest delivery dates of the considered case (for each Client Complaint, there exists a specific case represented by a token in the Petri net model which represents the Workflow process). As was shown in [Julia and Valette 2000], explicit time constraints which exist in a Real Time System, can be formally defined using a p-time Petri net model. The static definition of a p-time Petri net [Khansa 1996] is based on static intervals which represent the permanency duration (sojourn time) of tokens in places and the dynamic evolution of a p-time Petri net depends on the time situation of the tokens (date intervals associated with the tokens).

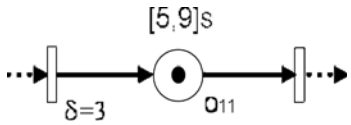


Figure 3. Visibility Interval

For example, in figure 3, if the arrival date of the token in the place O_{11} is $\delta = 3$, knowing that the static interval of this place is $[5, 9]_s$, then, the visibility interval of this token is $[5 + 3, 9 + 3]_V = [8, 12]_V$.

In the Petri net of figure 4, static intervals are associated with the activity places. For example, the static interval $[20, 30]_s$ associated to the place A_2 which represents the "Contact Client" activity means that the employee who has to contact the client will need at the minimum 20 time units and at the maximum 30 time units. The static intervals associated with the "Collect" activity in A_4 and the "File" activity in A_8 are equal to $[0; 0]$ because the duration of these activities are negligible compared to other activities of the "Handle Complaint Process".

As was said before, the waiting times will depend on the earliest and latest delivery dates of each case. It is considered that the maximum allowed duration of a case for the "Handle Complaint Process" is 105 time units. Knowing the beginning date of a case, it is possible to calculate the previsional visibility intervals associated to the tokens in the waiting places using constraint propagation techniques classically used in scheduling problems based on activity-on-arc graphs [Esquirol and Lopez 1995].

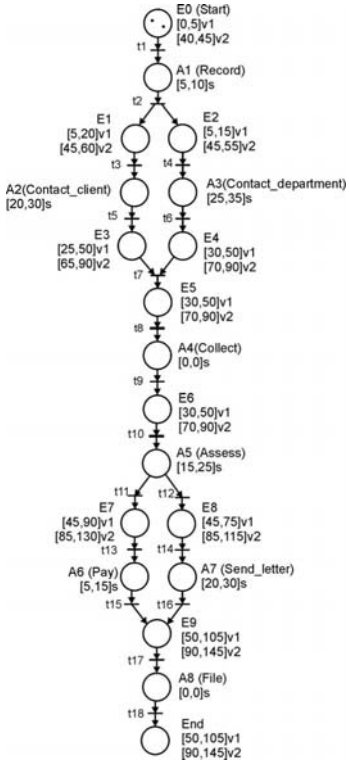


Figure 4. p-time Petri net

On the Petri net of figure 4, two cases represented by two tokens in place E_0 are considered. The first one can be initiated at date 0 which is represented as the minimum bound of the visibility interval v1 in E_0. The second case can be initiated at date 40 which is represented as the minimum bound of the visibility interval v2 in E_0. Because each case has to be processed in 105 time units at the maximum, the maximum bound of the previsional visibility intervals attached to the last place "End" of the process are equal to 0 (beginning date of case 1) + 105 (maximum duration of a case) = 105 (maximum date to deliver the first case) for the first case and 40 (beginning date of case 2) + 105 (maximum duration of a case) = 145 (maximum date to deliver the second case) for the second case. Bounds of other previsional visibility intervals attached to the waiting places are calculated applying propagation techniques. For example, the minimum bound of the visibility interval of case 1 in E_1 is obtained considering the minimum duration of the "Record" activity in A_1 making: 0 (minimum bound of the visibility interval v1 attached to E_0) + 5 (minimum bound of the static interval in A_1) = 5 (minimum bound of the previsional visibility interval v1 in E_1 corresponding to case 1). In the same way, the maximum bound of the visibility interval v1 in E_9 is obtained by considering the maximum duration of the "File" activity in A_8 making: 105 (maximum bound of the visibility interval v1 in the place "End") - 0 (maximum bound of the static interval in A_8) = 105 (maximum bound of the visibility interval v1 in E_9). Applying a similar reasoning to all waiting places and considering the different routings of the process, the minimum and maximum bounds of the waiting places for both cases (visibility intervals v1 for case 1 and v2 for case 2) are obtained as shown in figure 4.

3.3. Resource allocation mechanisms based on hybrid Petri nets

The last step of the modelling activity is to allocate the different types of resource used to realize the different activities. As already mentioned before, there exists different kinds of resources in Workflow Processes. Some of them are discrete type and can be represented by a simple token. For example, a printer used to treat a specific class of documents will be represented as a non-preemptive resource and could be allocated to a single document at the same time. On the other hand, some other resources can not be represented by a simple token. This is the case of most human type resources. As a matter of fact, it is not unusual for an employee who works in an administration to treat simultaneously several cases. For example, in an insurance company, one employee could treat normally several documents during a working day and not necessarily in a pure sequential way. In this case, a simple token could not model human behavior in a proper manner. As a solution to this problem, some of the human type resources will be represented by a real number (equal to one hundred when the resource is one hundred percent available) which will show the resource's availability. For example, if a maximum of ten patients can be allocated to a nurse in a hospital during her working day, the nurse's availability will be represented by a continuous resource equal to the real number one hundred and ten percent of this value will be allocated to each patient she will take care of.

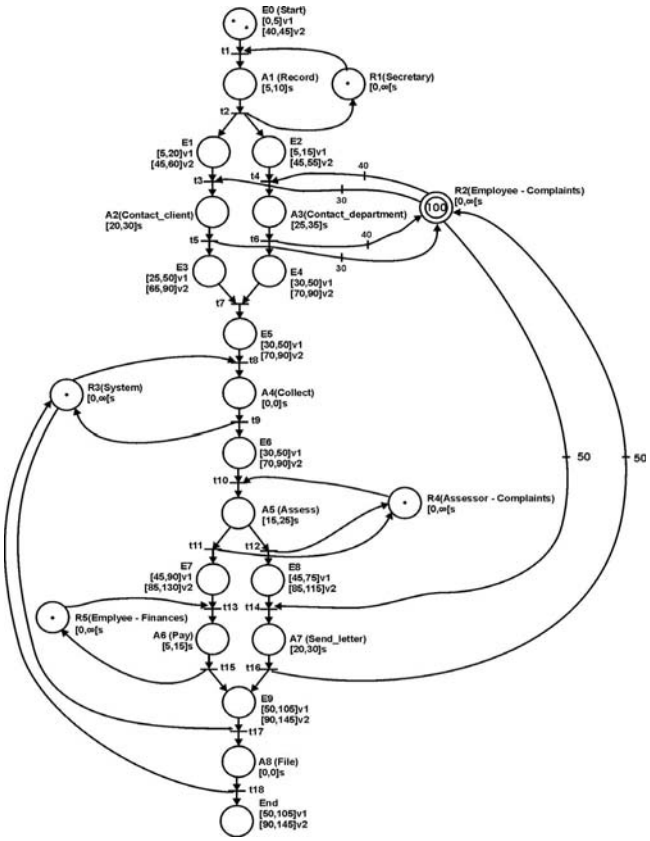


Figure 5. Resource allocation mechanisms

Hybrid resource allocation mechanisms (discrete + continuous resources) can be modelled by the hybrid Petri nets of Rene David and Hassane Alla [David and Alla 1992]. In particular, when considering Workflow Systems, all transitions of the hybrid Petri net

model will be discrete ones (none of the transitions of the model will be of continuous type).

The resource allocation mechanisms of the "Handle Complaint Process" are represented in figure 5. The "Secretary", the "Assessor-Complaints", the "Employee-Finances" and the "System" are represented as discrete resources. These resources will treat activities in a pure sequential way (one after the other). For example, the "Secretary" represented by a token in R1 can only register the data of a client at a single stroke and in a non-preemptive way. On the contrary, the employee of the complaints department is represented by the real number one hundred in R2. This continuous resource can be used to treat the activities: "Contact-Client", "Contact-Department" and "Send-Letter". The "Contact-Client" activity will need 30 percent of the employee's availability. The "Contact-Department" activity will need 40 percent of the employee's availability. And, finally, the "Send-Letter" activity will need 50 percent of the employee's availability. This means that, to a certain extent, the employee represented in place R2 could process more than one activity on a given period of time. For example, he could write a letter (50 percent of R2 for the"Send-Letter" activity) and, at the same time, try to phone a client (30 percent of R2 for the "Contact-Client" activity) to obtain more information about a case.

4. Scheduling principle

A natural approach to solve the scheduling problem of production systems represented by activity-on-arc graphs with disjunctive constraints (resource allocation mechanisms in the non-preemptive case) [Esquirol and Lopez 1995] is to combine a constraint propagation mechanism which allows for the calculation of admissible date intervals (visibility intervals in the p-time Petri net case) and a branch and bound algorithm whose goal is to find an optimal sequence of operations which respect the time constraints (date intervals). The limitation of such an approach is that the solution is generally given through a perfect sequence of operations which will be difficult to follow in real time when the system is a Workflow Process that considers human behavior.

Other scheduling approaches are based on token player algorithms which are specialized inference mechanisms applied in real time to Petri net models. The normal working of a token player algorithm is to fire transitions as soon as they become available. Such a strategy allows one to schedule the operations of a system in real time and gives the necessary flexibility to take into account possible perturbations (like delays for example). Nevertheless, it was shown in [Silva and Valette 1989] that it is not always the best strategy to fire transitions as soon as they are enabled. As a consequence, a traditional token player algorithm will hardly respect the time delivery constraints.

The approach proposed in this paper is based on a combination of time constraint propagation mechanisms (the ones used to obtained the visibility interval attached to the waiting places in the Petri net of figure 2) and a token player algorithm which uses a conflict resolution mechanism whose purpose is to calculate a sequence of activities that respects the disjunctive constraints (resource allocation mechanisms) as well as the time constraints (visibility intervals).

4.1. Conflict resolution principle

In the p-time Petri Nets, as presented in [Julia and Valette 2000], conflict situations for shared resources are visible during a time interval (there exists the notion of Conflict

Time Interval) and not at a single time point. For example, considering the p-time Petri net of figure 6 (the Petri net fragment involved in the conflict for the shared resource R2), the conflict time intervals associated to the pairs of transitions (t3,t14) and (t4,t14) are given by the intersections of the visibility intervals $[45, 60]_{v2} \cap [45, 75]_{v1} = [45, 60]_{(t3,t14)}$ and $[45, 60]_{v2} \cap [45, 75]_{v1} = [45, 60]_{(t4,t14)}$. It means that effective conflicts between t14 and t3 and between t14 and t4 are able to occur during the date interval $[45, 60]$.

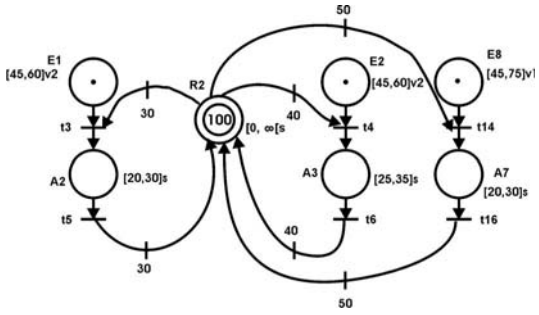


Figure 6. Conflict for a p-time Petri net

When a conflict is detected between two or more transitions in structural conflict, a decision mechanism has to look at a local level the consequences of a particular decision. For example, when considering the Petri net fragment of figure 6, if the decision of firing t14 at date $\delta = 45$ is made, then a token appears in A7 with the visibility interval $[20 + 45, 30 + 45] = [65, 75]_{v1}$ and the real number associated to the place R2 becomes equal to 100-50=50. From this new time situation, it is easy to see that a time constraint violation will happen on one of the visibility intervals associated to the tokens in E1 and E2. As a matter of fact, after the firing of t14, if t3 is fired before t4, a token is produced in place A2 with the visibility interval $[45 + 20, 45 + 30] = [65, 75]_{v2}$ and the real number associated to the place R2 becomes 50-30=20. In this case, the transition t4 could not be fired before 60 which is the maximum bound of the visibility interval in E2 because the real number in R2 will remain equal to 20 (which is less than the weigh necessary to fire t4) until date 65 (minimum bounds of the visibility interval in A2 and in A7) and a time constraint violation will happen. It is possible to show that after the firing of t14 at date $\delta = 45$, if t4 is fired before t3, a time constraint violation will happen in the same way with the visibility interval $[45, 60]_{v2}$ in E1. It is then possible to conclude that the firing of t14 at date $\delta = 45$ as soon as the token in E8 becomes available will not be allowed by the decision mechanism. The tool which allows one to represent all possible evolutions of a p-time Petri net is the class graph defined by Khansa [Khansa 1996]. Such a graph allows one to define the concept of "death token class" which corresponds to the possibility of a time constraint violation. The working of the decision mechanism based on the construction of a class graph when considering Petri net fragments involved in conflicts for shared resources was presented in [Julia and Valette 2000].

4.2. Token Player Algorithm

Figure 7 shows how the token player algorithm works. This algorithm has a decision making system which has to be used each time a conflict for a resource appears in the meaning of a p-time Petri net. The token player uses a calendar containing a sequence of events scheduled in time. These events are the minimum and the maximum bounds

of the visibility intervals. Each time a minimum bound of a visibility interval is reached in the calendar, it eventually means that a token becomes available. If the corresponding token enables a transition and if the transition is not in structural conflict, then the transition is fired. If the transition is in structural conflict, the conflict state (Petri net fragment) is isolated and the conflict resolution mechanism is called. If the conflict resolution mechanism allows one to fire the corresponding transition as soon as the token becomes available, then the transition is fired. If a maximum visibility interval is reached in the calendar, the "death" of a token occurs. The only solution is then to relax a constraint: increase the value of the maximum bound of a time interval; in this case, there will not have the guarantee of respecting the delivery delays.

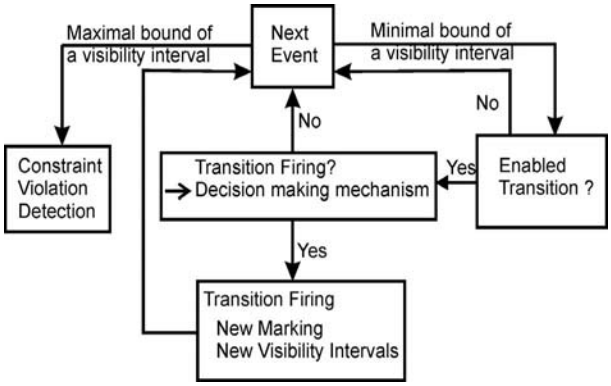


Figure 7. Token Player

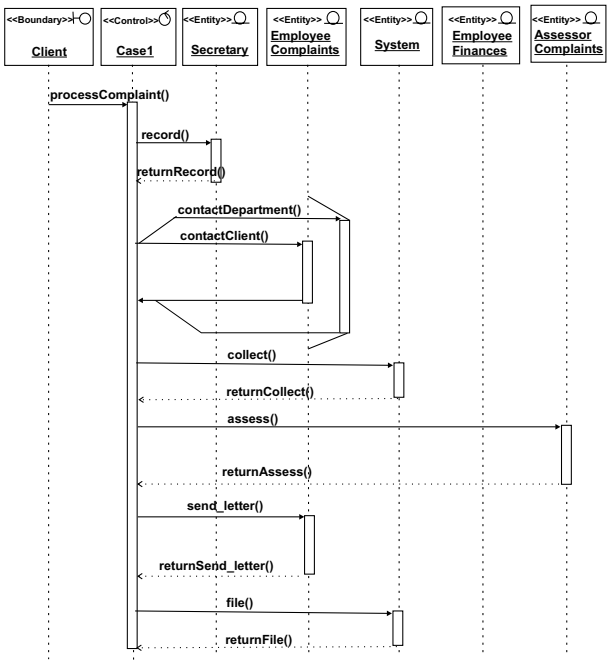


Figure 8. Sequence Diagram for case 1

When considering the previsional scheduling problem, for each case of the workflow process, a well defined scenario have to be specified. As a matter of fact, depending on the way the selective routings are treated, different scenarios can occur. The UML

diagram which allows one to specify scenarios is the sequence diagram which explicitly shows the chronological order of activities.

The sequence diagram in figure 8 is associated with the first case of the "Handle Complaint Process" (the token in place E0 in figure 5 with the visibility interval $[0, 5]_{v1}$). In this diagram, the object Case 1 associated with the UML stereotype "Control" represents the execution of case 1 when the complaint is not accepted (the compensation payment is refused). The objects associated with the UML stereotype "entity" are the ones which are informally defined through the swim lanes of the activity diagram in figure 1.

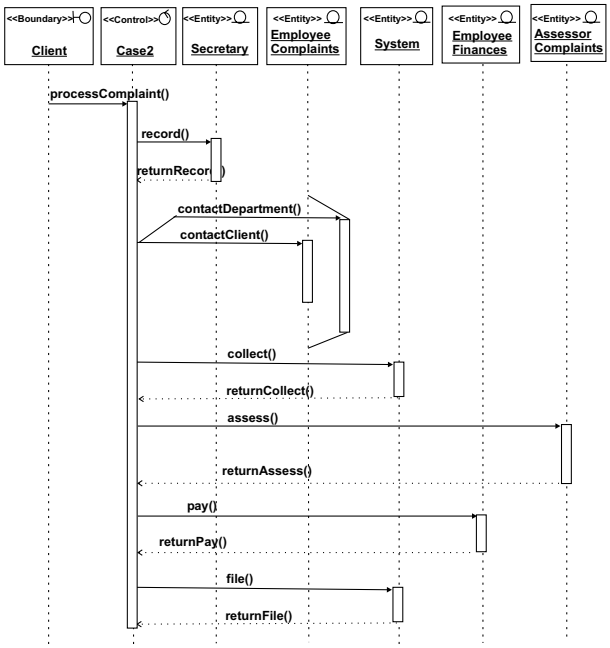


Figure 9. Sequence Diagram for case 2

The sequence diagram in figure 9 is associated with the second case of the "Handle Complaint Process" (the token in place E0 in figure 5 with the visibility interval $[40, 45]_{v2}$). In this diagram, the object Case 2 represents the execution of case 2 when the complaint is accepted (the compensation payment is accepted).

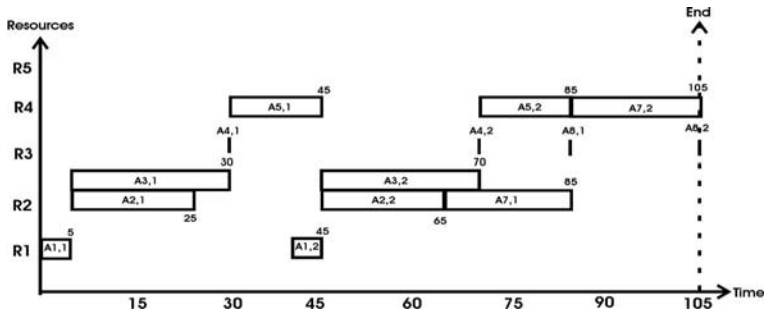


Figure 10. Admissible Sequence

Considering the scenarios specified by the sequence diagrams in figure 8 and 9, and applying the token player algorithm to the global model of figure 5, the result given

by the Gantt diagram of figure 10 is obtained. This diagram shows that the activity $A_{2,2}$ and $A_{3,2}$ (activities A2 and A3 of the second case of the Workflow Process) have to be processed before the activity $A_{7,1}$ (activity A7 of the first case of the Workflow Process) as was shown through the conflict resolution of the Petri net fragment in figure 6. It is also interesting to note that both cases respect their final visibility intervals in place "End": case 1 terminates on date $\delta = 85$ which belongs to the visibility interval $[50, 105]_{v1}$ and case 2 terminates on date $\delta = 90$ which belongs to the visibility interval $[90, 145]_{v2}$.

5. Software Architecture for the Real Time Scheduling

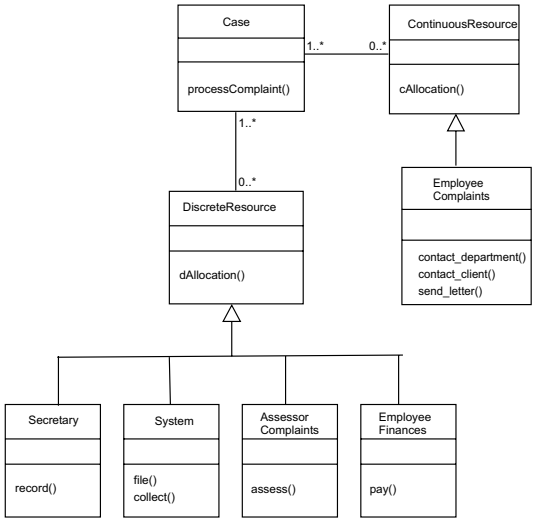


Figure 11. Class Diagram

One way of extending the results of the quantitative analysis (previsional schedule) to the design level is to consider a generic software architecture which allows one to apply the proposed scheduling approach (token player algorithm) in real time for the real time execution of workflow processes. In particular, the design of the chosen software architecture should show how components are structurally organized and the possible links between them.

Figure 11 shows how the objects of the sequence diagrams in figures 8 and 9 can be structured in a set of classes. The objects "Secretary", "System", "Employee Finances" and "Assessor Complaints" will inherit the characteristics of a discrete resource class whose main purpose is to allocate services in a discrete way through the method "dAllocation()". The object "Employee Complaints" will inherit the characteristics of a continuous resource class whose main purpose is to allocate services in a continuous way through the method "cAllocation".

Based on such a class structure, the collaboration diagram in figure 12 can be derived for the study of a specific case. It represents the existing links between the objects and the kind of communication. The methods "dAllocation" and "cAllocation" are called by the object Case 1 through asynchronous callings in order to allocate the continuous and discrete resources which will provide the methods necessary for the execution of the activities of the workflow process.

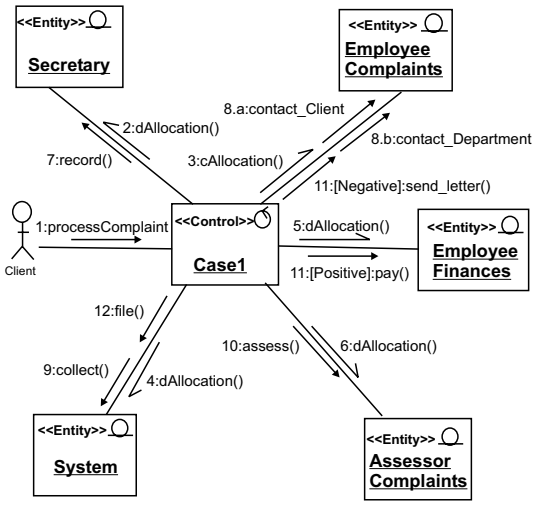


Figure 12. Collaboration Diagram for case 1

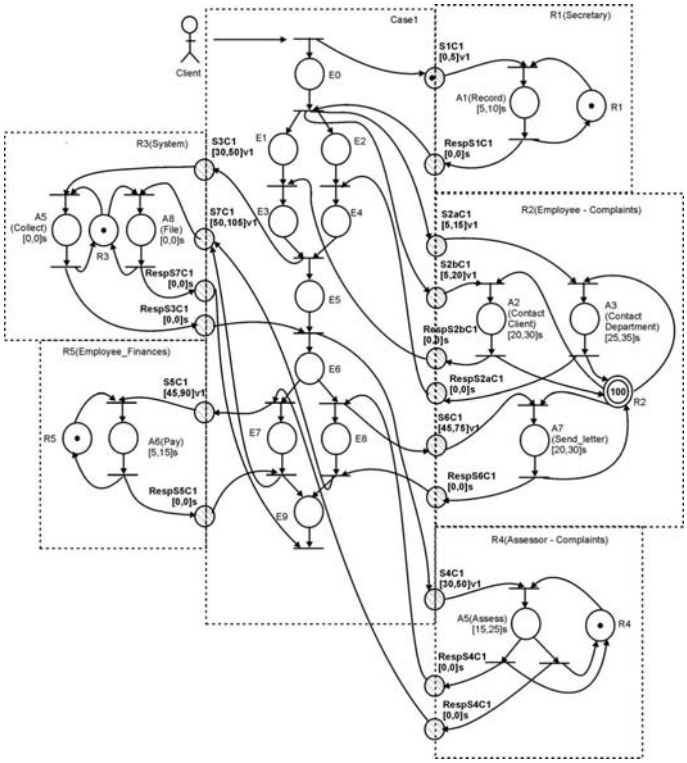


Figure 13. Petri net objects for case 1

After the last asynchronous calling from object Case 1, the Petri net model of figure 13 is obtained. Such a model shows the interactions between the different objects involved in the process of case 1. The internal behavior of each object is represented by a specific p-time Petri net model with continuous or discrete allocation mechanisms. The communication between the different Petri net objects is represented through communication places which are semaphore type. In particular, the methods necessary to execute the activities of the workflow process are called by the object Case 1 through synchronous callings as specified on the collaboration diagram in figure 12.

It would be easy to show that applying the reduction rules of the Petri net theory [Murata 1989] on the Petri net model of figure 13, eliminating the waiting places of the object Case 1 and some of the communication places, the global model of figure 5 would be obtained, which can be seen as a way of guaranteeing that the proposed software architecture will be equivalent to the analysis model used for the previsional schedule.

Finally, Figure 14 shows the internal behavior of the Petri net object R2 (Employee Complaints) when both cases (case 1 and case 2 in figure 2) are executed simultaneously. It is interesting to note that the mechanism used to solve conflict situations could be directly applied to this object which explicitly isolates a Petri net fragment. The visibility intervals associated with the communication places are calculated using the constraint propagation mechanisms at the level of the objects Case1 and Case 2 and are communicated to this object through asynchronous callings like the ones represented in the collaboration diagram of figure 12. Like that, the scheduling approach presented previously could be directly applied to such an architecture for the real time execution of workflow processes.

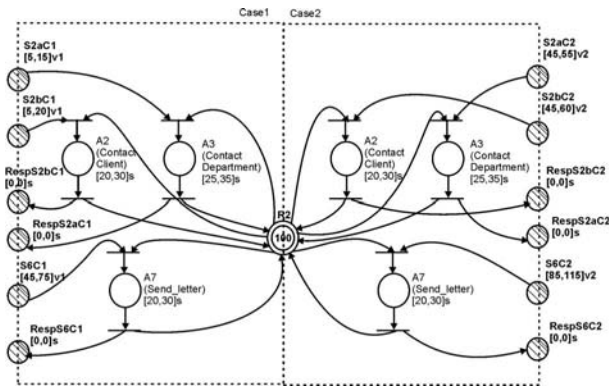


Figure 14. Petri net object "Employee-Complaints" for case 1 and case 2

6. Conclusions

This paper has shown that the initial UML specification (the activity diagram) of a Workflow Management System can be transformed into a unique p-time hybrid Petri net which represents the global behavior of the entire system with explicit time considerations and hybrid resource allocation mechanisms. Based on this model, a token player algorithm used for the scheduling problem of Workflow Management Systems has been applied. The final result has been given through an object based software architecture which allows one to apply the presented scheduling approach in real time for the real working of the Workflow Management System.

The advantages of such an approach are diverse. The fact of working with hybrid resources permits to represent the human behavior in a more realistic way. The originality of the time constraint propagation mechanism comes from the idea of combining on a same Petri net model duration intervals (static intervals) which represent durations of activities in a Workflow Process and date intervals (visibility intervals) which represent the waiting times of the cases between activities. The conflict resolution principle which can be used for shared hybrid resources allows to solve conflict situations in real time without a backtrack mechanism. Finally, the proposed software architecture is a way of

extending the result of the analysis level (previsional schedule) to the design level (real time schedule).

As a future work proposal, it will be interesting to study the possibility of introducing some fuzzy sets in the p-time hybrid Petri net model in such a manner that the natural flexibility of human behavior will be represented in a more expressive way at the Workflow Management System level.

References

Aalst, W. van der; Hee, K. van (2002). *Workflow Management: Models, Methods and Systems.* The MIT Press. Cambridge, Massachusetts. 368p.

Cardoso, J.; Silbertin-Blanc, C. (2001). Ordering actions in Sequence Diagrams of UML. *Invited talk in* 23^{rd} *International Conference on Information Technology Interfaces.* Croatia.

Covès, C.; Crestani, D.; Prunet, F. (1998). Design and Analysis of Workflow Processes with Petri nets. *In: 1998 IEEE International Conference on Systems, Man, and Cybernetics Proceedings of IEEESMC´1998.* vol. 1. p. 101-106.

David, R., Alla, H. (1992). *Petri Nets and Grafcet: tools for modelling discrete event systems.* Prentice Hall. 339p.

Eshuis, H. (2002). *Semantics and Verification of UML Activity Diagrams for Workflow Modelling.* PhD thesis, University of Twente, Enschede, The Netherlands.

Esquirol, P., Huguet, M.J., Lopez, P. (1995). (1999). Modelling and managing disjunctions in scheduling problems. Journal of Intelligent Manufacturing 6. pp. 133-144.

Hruby, P. (1998). Specification of Workflow Management Systems with UML. *In: Workshop on Implementation and Application of Object-oriented Workflow Management Systems.* Proceedings of the 1998 OOPSLA. Vancouver.

Julia, S.; de Oliveira, F. F. (2004). A p-time hybrid Petri net model for the scheduling problem of the Workflow Management Systems. *In: 2004 IEEE International Conference on Systems, Man and Cybernetics. Proceedings of IEEESMC´2004.* p. 4947-4952. The Hague, The Netherlands.

Julia, S.; Valette, R.; (2000). *Real time scheduling of batch Systems.* Simulation Practice and Theory. Elsevier Science. p. 307-319.

Khansa, W., Aygaline, P., Denat, J. P.(1996). Structural analysis of p-time Petri Nets. *Symposium on discrete events and manufacturing systems. CESA'96 IMACS Multiconference*. Lille, France.

Lee, D. Y.; DiCesare, F.k (1994). Scheduling flexible manufacturing systems using Petri nets and heuristic search. *IEEE Transactions on Robotics and Automation.*

Li, J., Fan, Y., Zhou, M. (2003). Timing Constraint Workflow Nets for Workflow Analysis. *IEEE Transactions on Systems, Man and Cybernetics. Part A: Systems and Humans. Vol. 33, N. 2, March 03.*

Murata, T. (1989). Petri nets: Properties, Analysis and Applications. *Proceedings of the IEEE.* 77(4), p. 541-580.

OMG - Object Management Group. (2005). “htp://www.omg.org”. Acesso em: 5 mar. 2005. 10(3). p. 123-132.

Pressman, R. S. (1995). *Software Engineering: A Practioner´s Approach.* 3 ed. Mc-Graw Hill.

Silva, M.; Valette, R. (1989). Petri nets and Flexible Manufacturing. *Advances in Petri nets (G. Rozemberg, Ed.), Lecture Notes in Computer Science. (Springer Verlag).*

Tramontina, G. B.; Wainer, J.; Ellis, C. (2004). Applying Scheduling Techniques to Minimize the Number of Late Jobs in Workflow Systems. *In: 2004 ACM Symposium on Applied Computing.* p. 1397-1403.

Wirtz, G.; Giese, H. (2000). Using UML and Object-Coordination-Nets for Workflow Specification. *In: 2000 IEEE International Conference on Systems, Man and Cybernetics. Proceedings of IEEESMC´2000.* vol.5, p. 3159-3164

Aspectual Connectors: Supporting the Seamless Integration of Aspects and ADLs

Thaís Batista[1], Christina Chavez[2], Alessandro Garcia[3], Uirá Kulesza[4], Cláudio Sant'Anna[4], Carlos Lucena[4]

[1]Computer Science Department, UFRN - Brazil

[2]Computer Science Department, UFBA - Brazil

[3]Computing Department, Lancaster University, United Kingdom

[4]Computer Science Department, PUC-Rio - Brazil

thais@ufrnet.br, flach@ufba.br, garciaa@comp.lancs.ac.uk

{uira,claudios,lucena}@inf.puc-rio.br

Abstract. *With the emergence of Aspect-Oriented Software Development (AOSD), there is a need to understand the adequacy of Architecture Description Languages (ADLs) connection abstractions for capturing the crosscutting nature of some architectural concerns. In this paper, we present the Aspectual Connector (AC), a special kind of architectural connector, as the only necessary enhancement to an ADL in order to support a seamless integration of AOSD and Software Architecture. We also present AspectualACME, an extension to ACME that incorporates ACs and additional facilities to modularize architectural crosscutting concerns. We use a Web-based information system as the main case study.*

Resumo. *Com o amadurecimento das pesquisas em Desenvolvimento de Software Orientado a Aspectos (DSOA) é necessário investigar se as abstrações das Linguagens de Descrição de Arquitetura (ADLs) são adequadas para modelar interesses arquiteturais transversais. Nesse artigo apresentamos o conceito de Conector Aspectual (AC), um tipo especial de conector arquitetural, como a única abstração adicional necessária em ADLs para permitir a integração entre DSOA e arquitetura de software. Apresentamos também AspectualACME, uma extensão de ACME que incorpora ACs e mecanismos adicionais para modularizar interesses arquiteturais transversais. Um sistema de informação Web é usado como estudo de caso para ilustrar a expressividade de AspectualACME.*

1. Introduction

Aspect-Oriented Software Development (AOSD) (Filman *et al*. 2005) aims to provide systematic support for the identification, modularization, and composition of crosscutting concerns throughout the software lifecycle. At the architecture design level, a crosscutting concern can be any concern that cannot be effectively modularized using the given abstractions of Architecture Description Languages (ADLs) (Shaw and Garlan 1996), leading to increased maintenance overhead, reduced reuse capability and

generally resulting in architectural erosion over the lifetime of a system. Since the emergence of Software Architecture as a discipline, the main focus of ADLs has been on the conception of architectural connection abstractions, such as interfaces, connectors, and configurations (Shaw and Garlan 1996). Hence, there is a pressing need for understanding to what extent these abstractions are able to capture the crosscutting interaction of certain architectural components. Ideally, ADL designers should promote a natural blending of conventional architectural abstractions and aspects.

Some Aspect-Oriented Architecture Description Languages (AO ADLs) (Navasa *et al*., 2002)(Pérez *et al*., 2003)(Pessemier *et al*., 2004)(Pinto *et al*., 2005) have been proposed, either as extensions of existing ADLs or developed from scratch employing AO abstractions commonly adopted in programming frameworks and languages, such as aspects, joinpoints, pointcuts, advice, and inter-type declarations. Although these AO ADLs provide interesting first contributions and viewpoints to the field, there is little consensus on how AOSD and ADLs should be integrated, especially with respect to the interplay of aspects and architectural connection abstractions. The main problem is that existing proposals typically provide heavyweight solutions (Batista *et al*. 2006), thereby hardening their adoption and the exploitation of the available tools for supporting ADLs.

In a previous work (Batista *et al*. 2006) we have discussed seven issues relating to the integration of AOSD and ADLs. We have discussed how and why extensions are required or not to conventional interconnection ADL elements, such as interfaces, connectors, and architectural configurations. Our conclusion was that ADLs promote the principle of Separation of Concerns (SoC) by explicitly separating components from their interactions (described by connectors). A systematic integration of architectural abstractions and AOSD would enhance the existing support for separation and modular representation of crosscutting concerns at the architectural level. The idea is to reuse the abstractions provided by conventional ADLs, with minor adaptations to support effective modeling of crosscutting concerns without introducing additional complexity into architecture specification.

This work presents the *Aspectual Connector* (AC) as the only necessary enhancement to an ADL in order to support a seamless integration between AOSD and Software Architecture. The AC specializes the conventional connector abstraction to support the description of interactions among components that have a crosscutting impact and other components. Instead of defining a new AO ADL, we extend ACME (Garlan *et al* 1997), a well-known ADL, with aspectual connectors. The resulting extension is AspectualACME, an ADL that supports the seamless exploitation of AOSD composition mechanisms in architecture design. To illustrate and evaluate AspectualACME, we present a web-based information system that exhibits some traditional crosscutting concerns in architecture description, such as persistence and distribution. We also assess the simplicity and generality of our approach with respect to related work and according to an evaluation framework that is also proposed in this paper.

The remainder of this paper is organized as follows. Section 2 presents background concepts related to AOSD and ADLs, and introduces the example that will be used throughout the paper. Section 3 presents an evaluation framework for AO ADLs that encompasses seven important issues related to aspects and architectural connection. Section 4 presents the notion of Aspectual Connectors. Section 5 illustrates

how to incorporate ACs into ACME. Section 6 compares our proposal with related work and Section 7 presents the final remarks.

2. ADLs and Aspect-Oriented Software Development

Architecture Description Languages (Sections 2.1 and 2.2) and Aspect-Oriented Software Development (Section 2.4) encompass abstractions and techniques that promote the principle of separation of concerns (SoC). In this section, we also present an initial description of an example used in this paper to illustrate the manifestation of crosscutting concerns in ADL representations.

2.1 Architecture Description Languages

Architectural concerns are typically expressed by using abstractions supported by Architecture Description Languages (ADLs). According to a well-known conceptual framework (Medvidovic and Taylor, 2000), the building blocks of an architectural description are components, connectors, and architectural configurations. In fact, ADLs enforce the SoC principle by explicitly distinguishing architectural elements used to specify computation (components) from those used to express interaction between components (connectors). *Components* are the units of computation, while *connectors* are the locus of interaction. Components and connectors may have associated interfaces, types, semantics and constraints, but only explicit component interfaces are a required feature for ADLs. A *component's interface* is a set of interaction points between it and the external world. An interface specifies the *services* (messages, operations, and variables) a component provides and also the services it requires from other components. Component types are templates that encapsulate functionality into reusable blocks and can be instantiated many times. *Connectors* model interactions among components and specify rules that govern those interactions. Similarly, connector types are templates that encapsulate component communication, coordination, and mediation decisions. A *connector's interface* specifies the interaction points between the connector and the components attached to it. A connector enables proper connectivity between components by exporting as its interface those services it expects from its attached components. *Configurations* define architectural structure and how components and connectors are connected.

2.2 ACME

ACME (Garlan *et al*. 2000) supports the definition of: (i) architectural structure, that is, the organization of a system into its constituent parts, (ii) properties of interest, information about a system or its parts that allow one to reason abstractly about overall behavior, both functional and nonfunctional, and (iii) types and styles, defining classes and families of architecture. Architectural structure is described in ACME with components, connectors, systems, attachments, ports, roles, and representations. *Components* are potentially composite computational encapsulations that support multiple interfaces known as *ports*. *Ports* are bound to ports on other components using first-class intermediaries called *connectors* which support so-called *roles* that attach directly to ports. *Systems* are the abstractions that represent configurations of components and connectors. A system includes a set of components, a set of connectors, and a set of attachments that describe the topology of the system. *Attachments* define a

set of port/role associations. *Representations* are alternative decompositions of a given element (component, connector, port or role) to describe it in greater detail. Thus, the representation may be seen as a more refined depiction of an element. For instance, ports may have a representation to encapsulate a large set of API calls as a single port. Inside the representation, a set of ports is used to represent individual API calls.

Other ACME elements support more sophisticated architectural features. *Properties* of interest are <*name, type, value*> triples that can be attached to any of the above ACME elements as annotations. Properties are a mechanism for annotating designs and design elements with detailed, generally non-structural, information. Architectural *styles* define sets of types of components, connectors, properties, and sets of rules that specify how elements of those types may be legally composed in a reusable architectural domain. The ACME fragment in Figure 1 illustrates the main ACME elements. These architectural elements organize software architecture as a graph of components and connectors. However, they do not provide the adequate means to capture some architectural crosscutting concerns, as discussed in the next section.

2.3. Crosscutting Concerns in ADL Representations: An Example

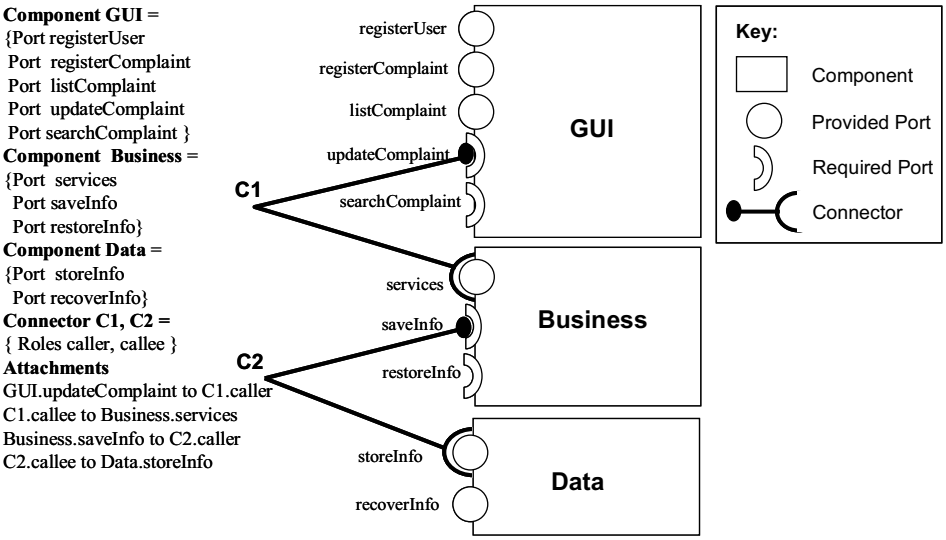


Figure 1. ACME Description of the HealthWatcher System

The HealthWatcher (HW) system is a Web-based information system developed by the Software Productivity research group from the Federal University of Pernambuco (Soares *et al*. 2002). The HW system supports the registration of complaints to the Public Health System. The HW is composed of the three main architectural components: (i) the *GUI (Graphical User Interface)* component provides a web interface for the system, (ii) the *Business* component defines the business rules, and (iii) the *Data* component stores the information to be processed. Figure 1 depicts ACME textual and graphical descriptions for this example. The interactions between the HW components are modeled using provided and required ports, and connectors. In Figure 1, for example, the *GUI* component uses the functionalities provided by the *Business* component by means of the connector C1. This connector has two roles which are used to attach the component ports. The attachment textual description for the HW system

(Figure 1) shows, for example, the binding of: (i) the *updateComplaint* required port to the caller role from the C1 connector; and (ii) the *services* provided port to the callee role from the C1 connector.

However, some architectural concerns cannot be modularly captured with traditional abstractions supported by ADLs, such as ACME. Some concerns are *crosscutting* even at the architectural design level, since they cannot be easily localized and specified with individual architectural units such as traditional interfaces, components, connectors, and configurations. Similar to the notion of aspect at the programming level (Kiczales *et al*., 1997), we say that these concerns crosscut the architectural units and denote the so-called *architectural aspects* (Araújo *et al*., 2005)(Baniassad *et al*., 2006)(Chitchyan *et al*., 2005)(Cuesta *et al*., 2005)(Krechetov *et al*., 2006).

Three crosscutting concerns affect the components of the HW system: (i) Persistence – supports issues related to the data management in web-based systems (transaction management, data update, repository configuration); (ii) Distribution – supports the distribution of the Business component services; (iii) Concurrency – specifies mechanisms to apply different concurrency strategies to the functional components. The problem is that, very often, the crosscutting property of these architectural concerns remains either implicit or is described in informal ways leading to reduced uniformity, impeding traceability and hindering detailed design and implementation decisions.

2.4 Aspect-Oriented Software Development

Aspect-Oriented Software Development (AOSD) (Filmann *et al*., 2005) provides new *abstractions* and *composition mechanisms* to support the explicit representation of aspects through software development stages, including software architecture design. The use of such new abstraction and composition mechanisms supports the encapsulation of crosscutting concerns into separated modular units, which are composed with other system modules at well-defined *join points*. Hence AOSD supports the *modularization* of structures and behaviors relative to a concern, which otherwise would be tangled and scattered through the representation of other concerns in software artifacts. Structural and behavioral enhancements can be typically applied *before*, *after* and *around* certain join points. In general, some quantification mechanism is provided to specify the extent of validity of such enhancements, that is, the extent to which each enhancement holds over a range of join points.

3. A Framework for Evaluation of Aspect-Oriented ADLs

This section presents a conceptual framework that subsumes a set of core issues that need to be considered while dealing with architectural aspects. Our goal is to use such a conceptual framework to support the systematic evaluation of existing aspect-oriented (AO) ADLs with respect to their proposed abstractions and extensions on the top of existing non-AO ADLs. The proposed framework is a result of a conceptual blending involving an AOSD glossary (van den Berg *et al*., 2005) and a widely-recognized terminology for software architecture descriptions (Medvidovic and Taylor, 2000). The conceptual framework was also derived from our extensive experience on: (i) the design of aspect-oriented software architectures in different application domains (Garcia *et al*,

2004)(Kulesza *et al*., 2004)(Kulesza *et al*., 2006)(Kulesza *et al*., 2006b), (ii) the development of modeling approaches to handle different categories of crosscutting concerns at the architectural stage (Chavez *et al*., 2006)(Garcia *et al*., 2006)(Krechetov *et al*., 2006)(Kulesza *et al*., 2004), and (iii) analysis of the suitability of existing ADLs to support architectural aspects (Chitchyan *et al*., 2005)(Batista *et al*., 2006).

Our comparison framework is composed of seven main elements, which are described in Table 1. The first column lists the framework issues, while the second column defines the purpose of the respective issue and describes potential choices in the design of an AO ADL. The first issue is dedicated to understanding which architectural elements (e.g. components and interfaces) in an architectural description are typically affected by a crosscutting concern. The following six issues correlate AOSD concepts with conventional abstractions of ADLs (Section 2.1). For example, the fourth issue is related to the specification of aspect interfaces. The last issue is particularly concerned with the need of a new abstraction for aspects at the architectural level. We recommend that the interested readers explore the details of our extensive discussion on the issues that inspired the conception of our evaluation framework (Batista *et al*., 2006).

Architectural Issue	**Description**
Base Elements	An AO ADL must define which architectural building blocks may be affected by aspects. The main architectural building blocks are components, connectors, configurations and interfaces. Hence, the design of an AO ADL is expected to define a subset or all of them as base elements.
Aspectual Composition	An AO ADL must support the composition between base elements and aspects. The issues here are whether and where the aspectual composition should be defined.
Quantification	An AO ADL can support or not quantification mechanisms over join points. If so, it must define where and how quantification should be specified.
Aspect Interfaces	An AO ADL should allow the explicit description of aspect interfaces. The issue is whether the conventional notion of architectural interfaces should be changed or not to express the boundaries of aspects.
Join point Exposition	An AO ADL must support join point exposition. Architectural join points are the instances of base elements in an ADL-based specification that can be affected by a certain aspect. The issue is whether the base elements should have a different interface exposing the join points to the aspectual components.
Interface Enhancements	Interface enhancement is the enrichment of component interfaces with new elements, such as services and attributes. An AO ADL may support or not interface enhancements.
Aspect	An AO ADL must support the description of aspects. The issue is whether it should provide or not a new architectural abstraction for describing them.

Table 1. An Evaluation Framework for Aspect-Oriented ADLs

In a previous work (Batista *et al*., 2006), we used our conceptual framework to evaluate several AO and non-AO ADLs. We analyzed how different ADLs address each issue of the framework. One of the main conclusions of our analysis was that no additional architectural abstractions were needed to represent aspects. We proposed extensions to the connector abstraction and to the configuration abstraction to support the modeling of the composition mechanism used in the crosscutting concern representation at the architectural level. These extensions are related with the need to support new ways of composition, as well as the quantification supported by a number of AO approaches. Next section describes aspectual connectors as the core of our proposal.

4. Aspectual Connectors

As already stated, software architecture descriptions rely on a *connector* to express the interactions between components. This section discusses why crosscutting interactions (Section 4.1) involving architectural components can be localized through the use of an extended notion of traditional connectors, called *Aspectual Connectors* (Section 4.2). From herein, we use the term *aspectual component* to refer to a component that implements a crosscutting concern (architectural aspect).

4.1. Modularizing Crosscutting Interactions in ADL Representations

A connector is a fundamental building block to model simple or complex interaction protocols as discussed in the taxonomy of connectors (Mehta *et al*. 2000). In addition, since ADLs (Section 2.1) explicitly distinguish components (units of computing) from connectors (units of interaction), this SoC approach should also play a key role in the integration of ADLs and AOSD. First of all, the component abstraction should be enough to model any kind of architectural concern independently from its crosscutting interaction with other components. In fact, a central goal of architecture specifications is to come up with a unifying abstraction – the component – to capture different types of computing units defined in specific architectural styles (Medvidovic and Taylor, 2000), such as objects, layers, meta-objects, and aspects. The key distinction between aspectual and regular components is in the way aspects compose with the rest of the system – the scope of the composition is broad and affects multiple components or multiple architectural elements.

Second, as connectors are widely used for different interconnection purposes, they are enough to model the interaction between traditional components and components that represent a crosscutting concern. However, the way that an aspectual component composes with a regular component is slightly different from the composition between traditional components. A crosscutting concern is represented by a provided service of an aspectual component and it can affect provided or required services of other components. As in ADLs valid configurations are those that connect provided and required services, it is impossible to represent a connection between a provided service of an aspectual component and a provided service without extensions to the traditional notion of architectural connections.

4.2. The Structure of Aspectual Connectors

In order to address the issues mentioned in Section 4.1, we propose an innovative abstraction, called *Aspectual Connector (AC)*, which is a regular connector with a new interface. The purpose of such a new interface is twofold: to make a distinction between the elements playing different roles in a crosscutting interaction – i.e. affected base components and aspectual components; and to capture the way both categories of components are interconnected. The AC interface contains: (i) a *base role*, (ii) a *crosscutting role,* and (iii) a *glue* clause. Figure 2 depicts a high-level view of the composition between an aspectual component and two components. C1 and C2 are examples of aspectual connectors. Note that we do not have a distinct abstraction to represent architectural aspects, which are similarly represented as regular components; the different colors in Figure 2 are only to emphasize which one is playing the role of aspectual component in the crosscutting collaborations.

The base role is specified to be connected to a port of the regular component and the crosscutting role is specified to be connected to a port of an aspectual component. The pair base-crosscutting roles do not impose the constraint of connecting provided and required ports. A crosscutting role defines the place at which an aspectual component joins a connector. In Figure 2 the aspectual connector C1 connects a provided port of the aspectual component with a provided port of Component 1. C2 connects another provided port of the aspectual component with a required port of Component 2. The glue clause specifies the details about a connection such as the place where the connector joins the component – after, before, around, and others.

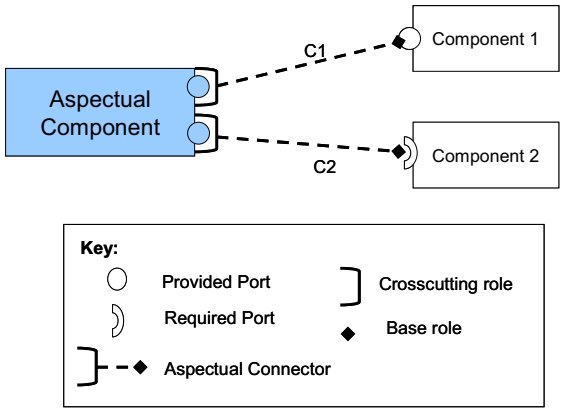


Figure 2. Aspectual Composition

4.3 Aspectual Composition

In ADLs, the connections between components and connectors are defined in the *configuration* section. The configuration description picks up architectural join points at which an aspectual component acts. The join points of interest are certain elements of the component interfaces, which are captured and associated with a base role of a specific AC. Thus, such elements of component interfaces are the collection of join points where the regular components and aspectual connector are combined. In fact, the concept of configuration already defines the point where a component joins a connector. Thus, we are just taking advantage of this concept to identify the join points affected by a crosscutting interaction. Wildcards and logical expressions can be used in the configuration part to specify several join points in a single statement, or to quantify over join points.

5. AspectualACME: An Aspect-Oriented ADL

This Section presents the description of AspectualACME, an extension of ACME with the goal of supporting a seamless integration of aspects and ADLs. In Section 5.2 we evaluate AspectualACME according to the framework presented in Section 3.

5.1. Extending ACME

We address the integration of aspects and ADLs to conform to the issues discussed in Sections 3 and 4, by extending ACME to introduce aspectual connectors and quantification support at the configuration level. Additionally, AspectualACME is expected to support simplicity, expressiveness, and to provide a conservative extension so that software architects can foster reuse of ACME libraries and tools. We have

selected ACME as our base ADL because it presents a relatively simple core set of concepts for defining system structure and it captures the essential elements of architectural modeling (Medvidovic and Taylor 2000). In addition, unlike most ADLs, ACME is not domain-specific and provides generic structures to describe a wide range of systems. It comes with tools that provide a good basis for designing and manipulating architectural descriptions and generating code. The complete BNF of AspectualACME is available at (AspectualACME, 2006).

5.1.1. ACME extension for aspectual connectors

The first extension that we propose is a specialization of ACME's connector abstraction. This extension allows the expression of aspectual connectors and their inner constructs: base roles, crosscutting roles, and the composition between them denoted by *glue*. We extend the connector interface in order to support the specification of base and crosscutting roles. The base role may be connected to the port of a component (provided or required) and the crosscutting role may be connected to a port of an aspectual component. The distinction between base and crosscutting roles addresses the constraint typically imposed by many ADLs about the valid configurations between provided and required ports. An aspectual connector must have at least one base role and one crosscutting role. Figure 3a and 3b present examples of a regular connector and an aspectual connector in ACME.

```
Connector aConnector = {
  Role aRole1;
  Role aRole2;
}
```

(a) regular connector in ACME

```
Connector aConnector = {
  Base Role aBaseRole;
  Crosscutting Role aCrosscuttingRole;
  Glue glueType; }
```

(b) aspectual connector in AspectualACME

Fig. 3. Regular and Aspectual Connectors

We also introduce a new construct - the *glue* clause - to specify details about the composition between components and aspectual components, such as the place where the port from an aspectual component will affect the regular component. There are three types of aspectual glue: *after*, *before*, and *around*. The semantics are similar to that of advice composition from AspectJ (AspectJ Team, 2006). For binary aspectual connectors (only one crosscutting role and one base role), the glue clause is simply a declaration of the glue type (Figure 3b), but whenever more than one base role and one crosscutting role are declared inside an aspectual connector, the glue clause must be more elaborated (Figure 4).

```
Connector aConnector = {
  Base Role aBaseRole1, aBaseRole2;
  Crosscutting Role aCrosscuttingRole1,
                    aCrosscuttingRole2;
  Glue { aCrosscuttingRole1 before aBaseRole1;
         aCrosscuttingRole2 after aBaseRole2;
  }
}
```

Fig. 4. Glue Clause

5.1.2. ACME extension for quantification

The second extension addresses quantification to avoid the need to refer explicitly to each join point in an architectural description. Since the Attachments part is the place where structural join points are identified in ACME, we have decided for defining the quantification mechanism by extending the configuration part. It is also possible to use wildcards in order to denote names or part of names of components and their ports. The quantification must be used in the attachment of a base role with target component(s). In Figure 5, the star symbol ('*') is used to specify that `aConnector.aBaseRole` is bound to all components that offer a port with a name that begins with `prefix`.

```
System Example = {
Component aspectualComponent = { Port aPort }
Connector aConnector = {
  baseRole aBaseRole;
  crosscuttingRole aCrosscuttingRole;
  glue glueType;
}
Attachments {
      aspectualComponent.aPort to aConnector.aCrosscuttingRole
      aConnector.aBaseRole to *.prefix* }
}
```

Fig. 5 ACME Description of the Composition

5.1.3. Example

In this section, we present the modeling of the Distribution and Persistence concerns in the context of the HealthWatcher (HW) system (Section 2.2). We discuss two different configurations of the HW system architecture. This allows us to illustrate the flexibility and expressivity of AspectualACME to represent different architectural decisions when modeling an architecture. Figures 6 and 7 show the modeling of the two HW configurations using AspectualACME.

In the first system configuration (Figure 6) Persistence is modeled as a crosscutting concern and Distribution is specified as a non-aspectual component which allows the GUI component to remotely access the services provided by the Business component. The Persistence aspectual component addresses: (i) the modularization of an update protocol in order to persist information that is modified by the GUI component; and (ii) the transaction demarcation of the services provided by the Business component using a transaction service available in the Data component.

Figure 6 depicts the AspectualACME description of the HW system including the Persistence concern. Persistence affects the GUI component and the Business component. The composition of the Persistence component with the GUI component is modeled by the *Persist* aspectual connector. In the attachments section, the *Persist* connector connects *updateStateControl* with *registerUser* and with *registerComplaint* (both are referred by the * wildcard in the attachments description). The glue clause of *Persist* specifies that the element bound to the crosscutting role (source) acts after the execution of the element bound to the base role (sink). This means that, whenever a user or a complaint is registered, a function is activated by the *Persistence* component. The internal implementation of *updateStateControl* needs to invoke the service of the *Distribution* Component, modeled by the C3 connector. However, this internal feature is not explicit in the AspectualACME description. The reason is that in ACME, as well

as in other ADLs, implementation details are not described by the architectural specification. Nevertheless, if the architect decides to expose some internal feature, ACME properties can be used for this purpose.

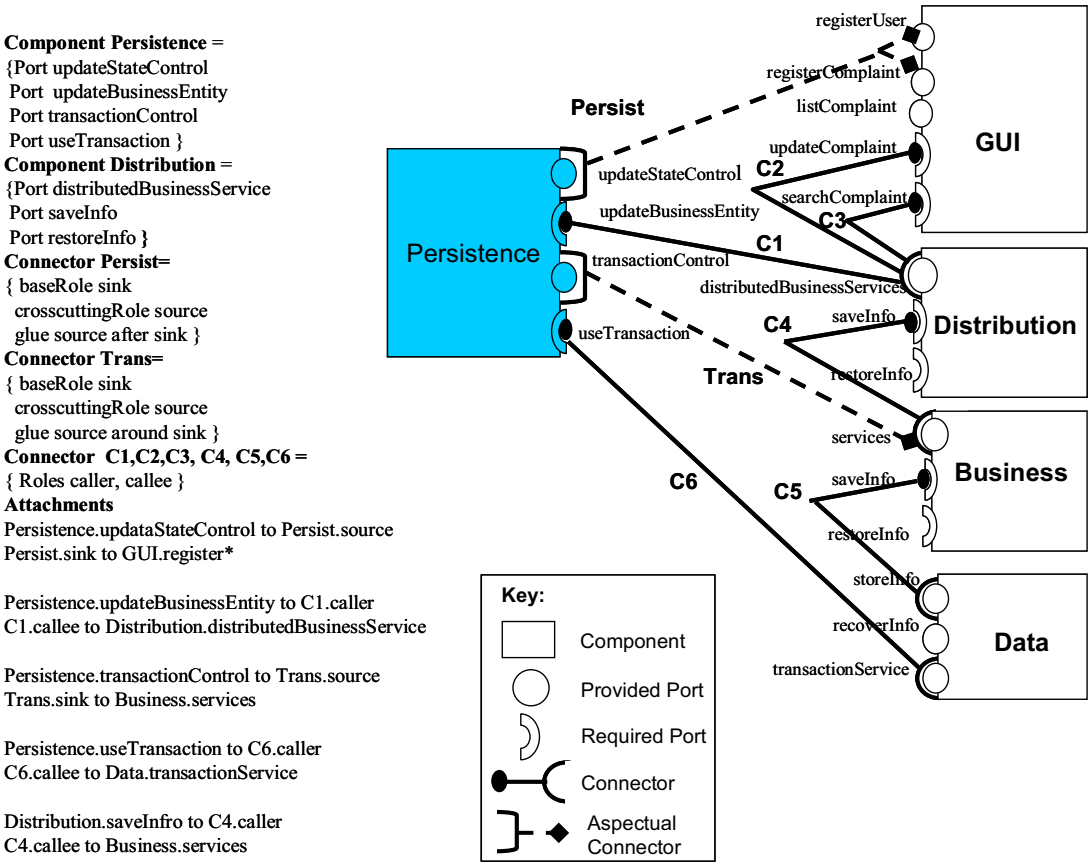


Fig. 6 HW AspectualACME Description with Persistence

The composition of the *Persistence* component with the *Business* component is modeled by the *Trans* aspectual connector. It connects the *services* with the *transactionControl*. It defines that whenever a *service* is requested, a transaction control mechanism acts during this action. The idea is that the transaction control mechanism of the *Persistence* component uses the transactional operations (*begin_transaction*, *comit_transaction*, and *rollback*) provided by the *transactionService* provided port of the Data component. However, again, as this information is not specified in the architectural description since it is internal to the *transactionControl* implementation. This interaction is modeled by a conventional connector (C6) and it can be explicitly described by means of ACME properties.

The second configuration shows both Persistence and Distribution modeled as aspectual components addressing crosscutting concerns (Figure 7). This configuration corresponds to the architectural modeling presented by an aspect-oriented implementation of the HW system (Soares *et al*. 2002). Persistence is responsible only for the transactional demarcation of the Business services. The Distribution aspectual component modularizes: (i) the transparent configuration of the calls from the GUI component to the Business to be realized through remote access; and (ii) the update protocol that persists information modified by the GUI component. This functionality is

now implemented by the Distribution component because it requires the remote invocation of the Business component.

Component Persistence =
{ Port transactionControl
Port useTransaction
}
Component Distribution =
{Port remoteUpdateStateControl
Port remoteAccess
Port useBusinessServices
}
Connector Persist=
{ baseRole sink
crosscuttingRole source
glue source after sink }
Connector Trans=
{ baseRole sink
crosscuttingRole source
glue source around sink }
Connector Distrib=
{ baseRole sink
crosscuttingRole source
glue source around sink }
Connector C3, C4, C5 =
{ Roles caller, callee }
Attachments
Persistence.transactionControl to Trans.source
Trans.sink to Business.services
Persistence.useTransaction to C5.caller
C5.callee to Data.transactionService

Distribution.remoteUpdateStateControl to Persist.source
Persist.sink to GUI.register*
Distribution.remoteAccess to Distrib.source
Distrib.sink to GUI.*Entity
Distribution.useBusinessSerivce to C3.caller
C3.callee to Business.services
…

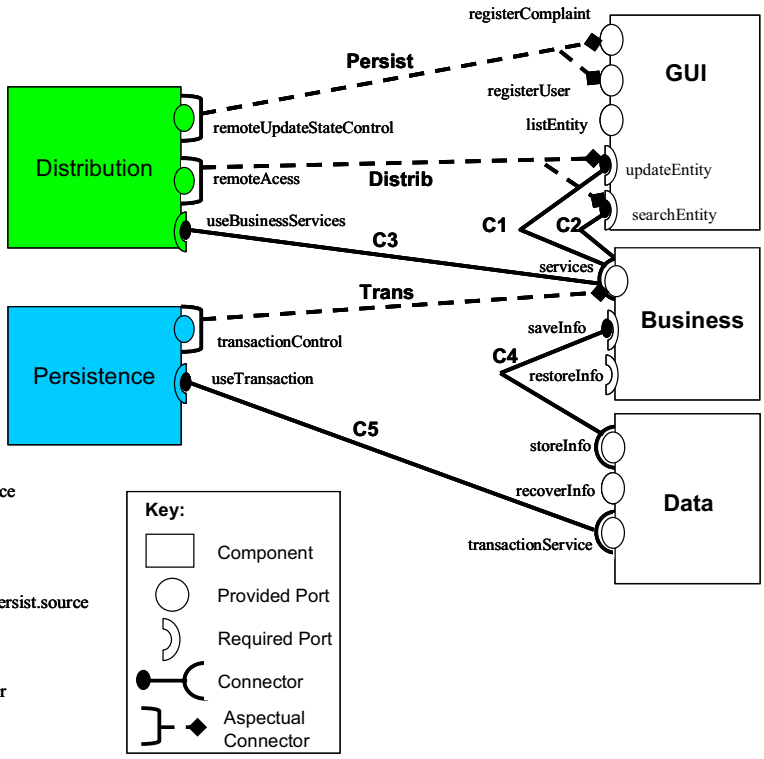


Fig. 7 HW AspectualACME Description with Persistence and Distribution

Figure 7 shows the AspectualACME description for the second configuration of the HW. In order to support the update protocol, the Distribution aspectual component affects the *registerComplaint* and *registerUser* by quantifying over them using wildcard expressions (*register**). The protocol is localized within the *Persist* aspectual connector. The *Persist* glue clause states that the service bound to the crosscutting role is invoked after the execution of the services bound to the base role. The *Distribution* component also models the transparent distributed access of the Business component by the GUI component. The *Distrib* aspectual connector is responsible for this task. The attachments section defines that the *remoteAccess* service affects *updateEntity* and *searchEntity*. The idea is that internally, the *remoteAccess* service redirects (using around) every invocation to *services* to be executed by means of the C3 connector. As this information represents implementation details of the *remoteAccess* service, it is not described in the AspectualACME specification. The Persistence aspectual component models the transaction control as described previously (first configuration).

5.2. Evaluation

We have evaluated the applicability of Aspectual Connectors (Section 4) and the extensions supported by AspectualACME (Section 5.1) in the context of two case studies: the HealthWatcher system (Section 2.3), which has been partially discussed in the previous section, and a context-sensitive tourist guide system described in (Batista *et al.*, 2006). The second case study encompasses the manifestation of three

architectural aspects: replication, security, and performance. In fact, the choice of such systems was driven by the heterogeneity of the aspects, and the way they affect regular components and each other.

Our approach has scaled up well in both case studies mainly by the fact that ACs and AspectualACME are following a symmetric approach, i.e., we assume that there is no explicit distinction between regular components and aspectual components. The modularization of the crosscutting interaction into connectors has promoted, for example, the reuse of the persistence component description in the second case study. Persistence was described as a crosscutting concern only in the HealthWatcher architecture (Figure 6). Hence, we have not applied an aspectual connector in the second case. The definition of quantification mechanisms (Section 5.1.2) in attachments also has shown to be the right design choice as it improves the reusability of connectors. Furthermore, it is possible to determine how multiple interacting aspects affect each other by looking at a single place in the architectural description: the attachments section.

	AspectualACME
Base Elements	Aspectual components affect components through aspectual connectors.
Aspectual Composition	Modeled by aspectual connectors with base and crosscutting roles and by configurations.
Quantification	Supported by wildcards defined at the configuration section.
Aspect Interfaces	No extension required
Join Point Exposition	Components can expose their internal events
Interface Enhancement	Not supported
Aspects	Aspects are modeled by means of connectors, crosscutting roles, base roles, and components.

Table 2. An Evaluation of the Proposed ADL

Table 2 presents an evaluation of the proposed ADL according to the framework presented in Section 3. Our proposal advocates that no new architectural abstractions are needed to represent aspects. Regular components are used for this purpose. In addition, we have argued that no changes are required in components interfaces. AspectualACME defines a composition model that takes advantage of existing architectural connection abstractions – connectors and configurations – and extends them to support the definition of some composition facilities. In this way, AspectualACME promotes simplicity and avoids introducing complexity in the architectural description. Compared to existing solutions (Section 6), AspectualACME proposes a smaller set of required extensions to deal with architectural crosscutting concerns. As a result, the architects can model crosscutting concerns using the same abstractions, with minor adaptations, used in the conventional ADL description.

6. Related Work

There is a diversity of viewpoints on how aspects (and generally concerns) should be represented by ADLs. However, so far, the introduction of AO concepts into ADLs has been experimental in that researchers have been trying to incorporate mainstream AOP concepts into ADLs. In contrast, we argue that most of existing ADLs abstractions suffice to model crosscutting concerns, with minor adaptations, including the specialization of connectors and a minor extension to the syntax of attachments.

Contrary to AspectualACME, most AO ADLs introduce new abstractions in the ADL to model AO concepts (aspects, joinpoints, and advices). DAOP-ADL (Pinto *et al*. 2005) defines components and aspects as first-order elements. Aspects can affect the components' interfaces by means of: (i) an *evaluated interface* which defines the messages that aspects are able to intercept; and (ii) a *target events interface* responsible for describing the events that aspects can capture. The composition between components and aspects is supported by a set of *aspect evaluation rules*. They define when and how the aspect behavior is executed. In the Prisma approach (Perez *et al*. 2003), aspects are new ADL abstractions used to define the structure or behavior of architectural elements (component and connectors), according to specific system viewpoints. Components and connectors include a weaving specification that defines the execution of an aspect and contains weaving operators to describe the temporal order of the weaving process (after, before, around). Pessemier *et al.* (Pessemier *et al*. 2004) extend the Fractal ADL with Aspect Components (ACs). ACs are responsible for specifying existing crosscutting concerns in software architecture. Each AC can affect components by means of a special interception interface. Two kinds of bindings between components and ACs are offered: (i) a direct crosscut binding by declaring the component references and (ii) a crosscut binding using pointcut expressions based on component names, interface names and service names.

Similarly to our proposal, FuseJ (Suvée *et al*. 2005) defines a unified approach between aspects and components. FuseJ provides the concept of a gate interface that exposes the internal implementation of a component and offers access-point for the interactions with other components. FuseJ concentrates the composition model in a special type of connector that extends regular connectors by including constructs to specify how the behavior of one gate crosscuts the behavior of another gate. However, differently from our work, FuseJ defines the gate interface that exposes internal implementation details of a component, while our compositional model works in conjunction with the component (conventional) interface. We consider that FuseJ introduces an additional level of complexity for component reuse - the gate interface. Moreover, the exposition of the component internals is against object-oriented principles. In addition, configurations are not explicitly dealt by the FuseJ approach. The connection between components and connectors is defined inside the connector itself. This contrasts with the traditional way that ADLs work, by declaring a connector and binding connectors' instances at the configuration section.

7. Conclusions

In this paper we have proposed the Aspectual Connector as a central element to support the integration of crosscutting concerns in ADL descriptions. We have also instantiated this concept in the context of a general-purpose ADL – ACME – and we have illustrated the concept with an example that presents two crosscutting concerns. Our proposal defines a composition model, centered on the concept of an aspectual connector, which takes advantage of existing architectural connection abstractions – connectors and configurations – and extends them to support the definition of some composition facilities such as a quantification mechanism. In this way, our proposal avoids introducing complexity in the architectural description and comparing with existing solutions, we identified a reduced set of required extensions to deal with architectural crosscutting concerns. As a result, architects can model crosscutting

concerns using the same abstractions, with minor adaptations, used in the conventional ADL description. As such, our proposal is based on enriching the composition semantics supported by architectural connectors instead of introducing new abstractions that elevate programming language concepts to the architecture level. Our proposal, therefore, supports effective modeling of crosscutting concerns without introducing additional complexity into the architecture specification. Planned future work includes evaluating our ADL by modeling a large-size system.

Acknowledgments

This work has been partially supported by CNPq-Brazil under grant No.479395/2004-7 for Christina and grant No.140214/04-6 for Cláudio. Uirá is partially supported by FAPERJ under grant No. E-26/151.493/2005. Alessandro is supported by European Commission as part of the grant IST-2-004349: European Network of Excellence on Aspect-Oriented Software Development (AOSD-Europe), 2004-2008. The authors are also supported by the ESSMA Project under grant 552068/02-0.

References

Aldrich, J. (2005) "Open Modules: Modular Reasoning about Advice". In: Proc. of the European Conf. on Object-Oriented Programming (ECOOP'05), LNCS 3586, pp. 144-168, July.

Araújo, J. *et al*. (2005) "Early Aspects: The Current Landscape". Technical Notes, CMU/SEI and Lancaster University.

AspectJ Team (2006). "The AspectJ Programming Guide". http://eclipse.org/aspectj/.

AspectualACME (2006) "AspectualACME". http://www.teccomm.les.inf.pucrio.br/aspectualacme.

Baniassad, E. *et al*. (2006) "Discovering Early Aspects". IEEE Software, 23(1): 61-70, January.

Batista, T. *et al*. (2006) "Reflections on Architectural Connection: Seven Issues on Aspects and ADLs". In: Workshop on Early Aspects, ICSE'06, pp. 3-9, May, Shanghai, China.

van den Berg, K., Conejero, J., Chitchyan, R. (2005), "AOSD Ontology 1.0 – Public Ontology of Aspect-Orientation". AOSD-Europe Report, Deliverable D9, 27 May.

Chavez, C., Garcia, A., Kulesza, U., Sant'Anna, C., Lucena, C. (2006). "Crosscutting Interfaces for Aspect-Oriented Modeling". Journal of the Brazilian Computer Society, 12(1), June.

Chitchyan, R. *et al.* (2005) "Survey of Analysis and Design Approaches". AOSD-Europe Report, Deliverable D11, 18 May.

Cuesta, C. *et al*. (2005) "Architectural Aspects of Architectural Aspects". In: 2nd European Workshop on Software Architecture (EWSA), LNCS 3527, pp. 247-262.

Filman, R. *et al.* (2005). "Aspect-Oriented Software Development". Addison-Wesley.

Garcia, A., Kulesza, U., Lucena, C. (2004). "Aspectizing Multi-Agent Systems: From Architecture to Implementation". In: Software Engineering for Multi-Agent Systems III, Springer-Verlag, LNCS 3390, pp. 121-143, December.

Garcia, A., Batista, T., Rashid, A., Sant'Anna, C. (2006) "Driving and Managing Architectural Decisions with Aspects". Workshop on Sharing and Reusing architectural Knowledge (SHARK'2006), Torino, Italy.

Garlan, D., Monroe, R., Wile, D (1997) "ACME: An Architecture Description Interchange Language". In: Proc. of CASCON '97, Electronic Edition, November.

Garlan, D., Monroe, R., Wile, D. (2000) "ACME: Architectural descriptions of component-based systems". In: Foundations of Component-based Systems. Gary T. Leavens and Murali Sitaraman (eds), Cambridge University Press, pp. 47-68.

Kiczales, G. *et al*. (1997) "Aspect-Oriented Programming". European Conference on Object-Oriented Programming (ECOOP), LNCS 1241, pp. 220-242, Springer, June, Finland.

Krechetov, I., Tekinerdogan, B., Garcia, A., Chavez, C., Kulesza, U. (2006) "Towards an Integrated Aspect-Oriented Modeling Approach for Software Architecture Design". 8th Workshop on Aspect-Oriented Modeling (AOM'06), AOSD'06, March, Bonn, Germany.

Kulesza, U., Garcia, A., Lucena, C. (2004), "Towards a Method for the Development of Aspect-Oriented Generative Approaches." Workshop on Early Aspects, OOPSLA'04, November, Vancouver, Canada.

Kulesza, U., Garcia, A., Bleasby, F., Lucena, C. (2005) "Instantiating and Customizing Aspect-Oriented Architectures using Crosscutting Feature Models". Workshop on Early Aspects, OOPSLA'05, November, San Diego, USA.

Kulesza, U. *et al*. (2006) "Quantifying the Effects of Aspect-Oriented Programming: A Maintenance Study". In: 22nd IEEE Intl. Conf. on Software Maintenance (ICSM'06), Sept.

Kulesza, U. *et al*. (2006b) "Improving Extensibility of Object-Oriented Frameworks with Aspect-Oriented Programming". In: Proc. 9th Intl. Conf. on Software Reuse, June, Torino, Italy.

Medvidovic, N., Taylor, R. (2000). "A Classification and Comparison Framework for Software Architecture Description Languages". IEEE Trans. Soft. Eng., 26(1), pp.70-93, January.

Mehta N., Medvidovic, N., Phadke, S. (2000) "Towards a Taxonomy of Software Connectors". In: Proc. 22nd Intl Conf. on Software Engineering (ICSE'00), pp. 178-187, Limerick, Ireland.

Navasa, A. *et al*. (2002) "Aspect Oriented Software Architecture: a Structural Perspective". In: Workshop on Early Aspects, AOSD'2002, April, The Netherlands.

Pérez, J., Ramos, I., Jaén, J., Letelier, P., Navarro, E. (2003) "PRISMA: Towards Quality, Aspect-Oriented and Dynamic Software Architectures". In: Proc. of 3rd IEEE Intl Conf. on Quality Software (QSIC 2003), November, Dallas, Texas, USA.

Pessemier, N., Seinturier, L., Duchien, L. (2004) "Components, ADL and AOP: Towards a Common Approach". In: ECOOP Workshop on Reflection, AOP and Meta-Data for Software Evolution (RAM-SE04), June.

Pinto, M., Fuentes, L., Troya, J., (2005) "A Dynamic Component and Aspect Platform". The Computer Journal, 48(4), pp. 401-420.

Shaw, M. and Garlan, D. (1996): "Software Architecture: Perspectives on an Emerging Discipline". Prentice Hall.

Soares, S. *et al.* (2002). "Implementing Distribution and Persistence Aspects with AspectJ". In: Proc. of the 17th Annual ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA'02), pp. 174-190, November, Seattle, USA.

Suvée, D., De Fraine, B. and Vanderperren, W. (2005) "FuseJ: An Architectural Description Language for Unifying Aspects and Components". Software Engineering Properties of Languages and Aspect Technologies Workshop @ AOSD2005, March, Bonn, Germany.

Geração automatizada de drivers e stubs de teste para JUnit a partir de especificações U2TP

Luciano B. Biasi, Karin Becker

Programa de Pós-Graduação em Ciência da Computação – Faculdade de Informática
Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
Porto Alegre – RS – Brasil

{lbiasi, kbecker}@inf.pucrs.br

Abstract. *Testing has become essential to ensure the quality of software products. Within the test process, unit testing is performed on the smallest functional part of the software with the aim of discovering defects in these units. JUnit is a unit testing tool that helps developers in test automation and verification of the results. However, much time, cost and effort are spent on the programming of the necessary drivers and stubs, minimizing the benefits expected from its use. UML 2.0 Test Profile (U2TP) allows one to specify all test artifacts using a standard, high level and visual notation, which is independent of any specific programming language. This work addresses the automated generation of test drivers and stubs for JUnit, given a set of test cases specified with U2TP. The proposed models and algorithms were applied in a case study, and all corresponding test code was correctly generated.*

Resumo. *Teste é amplamente usado para garantir a qualidade em produtos de software. O teste de unidade é realizado na menor parte funcional de um software e visa descobrir defeitos nestas unidades. JUnit é uma ferramenta de apoio ao teste unitário, a qual auxilia desenvolvedores na automação dos testes e verificação dos resultados. Porém, muito tempo e esforço são gastos para codificar os drivers e os stubs de teste necessários a esta ferramenta, minimizando os benefícios esperados. O Perfil de Teste da UML 2.0 (U2TP) permite especificar artefatos de teste em uma notação padronizada, de alto nível, gráfica, e independente de linguagem de programação. Este trabalho aborda a geração totalmente automatizada de drivers e stubs de teste para ferramenta JUnit a partir de especificações de testes modeladas com o U2TP. Os modelos e algoritmos desenvolvidos foram aplicados a um estudo de caso, gerando corretamente todo o código correspondente.*

1. Introdução

Teste de software é o processo de executar um sistema de maneira controlada, a fim de revelar seus defeitos e avaliar sua qualidade. Um teste unitário consiste em testar unidades individuais de uma aplicação a fim de descobrir defeitos nas mesmas. O nível de abstração destas unidades depende muito do tipo de sistema sendo desenvolvido. Segundo [Burnstein, 2003], o processo de teste unitário deve envolver as seguintes atividades: Planejamento, Especificação e Projeto dos casos de teste, Preparação do código auxiliar e Execução propriamente dita. O código auxiliar (*Test Harness*) é

constituído de *drivers* e *stubs* de teste. Um *driver* é uma unidade que implementa chamadas às funcionalidades testadas. *Stubs* servem para substituir funcionalidades que ainda não foram implementadas ou que estão subordinadas ao módulo sendo testado.

A automação da execução de testes tem sido uma tendência, pois diminui o tempo gasto na execução dos testes e possibilita a verificação automática dos resultados. Para teste unitário, ferramentas bastante populares são aquelas da família denominada genericamente de *xUnit,* entre elas cppUnit (C++), VBUnit (Visual Basic), NUnit (.net) e JUnit (Java). Entretanto, alguns problemas são encontrados no uso deste tipo de ferramenta, entre eles: a) o esforço necessário para preparar o código auxiliar, em particular os *drivers*, que representam implementações dos casos de teste; b) nem sempre há uma distinção clara entre as atividades de especificação dos casos de teste e de programação do código auxiliar correspondente; c) frequentemente é o próprio programador da unidade quem fica encarregado da especificação do caso de teste, muitas vezes levando em conta a programação realizada na unidade e não os requisitos da aplicação; e d) dependência da especificação do caso de teste da linguagem de programação, própria a cada ferramenta.

O Perfil de Teste da UML 2.0 [OMG, 2004], conhecido como U2TP, tem por objetivo oferecer uma notação padronizada à especificação de diversos aspectos de teste. Assim, é possível especificar os testes em um mais alto nível de abstração e independente de linguagem de programação, resultando em uma documentação adequada, objetiva e sem ambigüidades sobre os testes que devem ser realizados, independentemente de ferramentas que possam vir a ser utilizadas para automatizá-los. Neste sentido, o U2TP estabelece um mapeamento dos seus conceitos com ferramentas de testes automatizados como JUnit e TTCN-3 [Dai e Gabrowski, 2003]. Assim, através de especificações de testes modeladas com o U2TP é possível não apenas representar os artefatos utilizados no processo de teste, mas também utilizá-los como ponto de partida para a geração automatizada do código de teste para diferentes ferramentas.

A geração automatizada de testes para ferramenta JUnit tem sido a preocupação de vários trabalhos encontrados na literatura. Nestes, as especificações de teste são documentadas em UML (e.g. [Wittevrongel e Maurer, 2001] [Fraikin e Leonhardt, 2002]), ou linguagens de especificação de teste que abstraem linguagens de programação (e.g. [Cheon e Leavens, 2002] Zongyuan et al., 2004]). Nenhum dos trabalhos encontrados é baseado no U2TP, nem aborda por completo a geração automatizada de todo código de teste, incluindo *drivers* e *stubs*.

Este artigo propõe a geração automatizada de *drivers* e *stubs* de teste para a ferramenta JUnit, considerando especificações de teste unitário em U2TP, e sem interação com o usuário.

O restante deste trabalho está estruturado como segue. A Seção 2 apresenta os conceitos relevantes da ferramenta JUnit. A Seção 3 discute o U2TP, com foco em seu uso para especificação em teste unitário. Na Seção 4 é apresentada a proposta do trabalho, descrevendo o cenário para geração dos *drivers* e *stubs* de teste, bem como as convenções adotadas para especificação de testes em U2TP. Na Seção 5 são apresentados o extrator e o gerador de código. Um estudo de caso é relatado na Seção 6 e trabalhos relacionados são discutidos e comparados na Seção 7. A Seção 8 apresenta as conclusões e as direções futuras de pesquisa.

2. Junit

JUnit é um framework para teste de unidade usado por programadores que desenvolvem aplicações em linguagem Java [Beck, 2003]. Casos de teste no JUnit são constituídos por um ou mais métodos, sendo que estes podem estar agrupados em suítes de teste.

Um *driver* JUnit é constituído por uma classe principal, cujos métodos representam os casos de teste. Adicionalmente, os métodos *setup* e *teardown* permitem especificar pré e pós condições comuns a todos os casos de teste. A execução de cada caso de teste no JUnit resulta na ativação dos seguintes métodos: (1) *setup*; (2) método correspondente ao caso de teste; e (3) *teardown*. O JUnit permite o uso de diferentes assertivas para comparar os testes, tais como *assertTrue*, *assertEquals e assertFalse.* Um teste no JUnit pode apresentar como resultado três valores: *pass*, *fail* ou *error*.

JUnit foi adaptado a uma variedade de IDEs (*Integrated Development Evironment)* que favorecem a escrita do código para uma melhor interpretação dos procedimentos de teste. As IDEs mais conhecidas são Eclipse [Eclipse, 2005] e NetBeans [Netbeans, 2005]. No Eclipse, por exemplo, é possível gerar a estrutura do *driver* automaticamente a partir da classe a ser testada, sendo que este esqueleto deve ser completado manualmente com código para os casos de teste e a declaração de variáveis.

3. Perfil de Teste da UML 2.0

3.1. Conceitos do Perfil U2TP

O U2TP define uma linguagem para projetar, visualizar, especificar, construir e documentar artefatos de teste para sistemas [OMG, 2004]. O U2TP pode ser usado somente para a manipulação dos artefatos de teste, ou de uma maneira integrada com UML, para a manipulação conjunta de um sistema e respectivos artefatos de teste. Seus conceitos estão organizados em quatro grupos lógicos: Arquitetura, Dados, Comportamento e Tempo. Para este trabalho, foram adotados elementos pertencentes aos grupos de arquitetura de teste e comportamento de teste, por serem os mais relevantes ao teste unitário. O grupo de dados, igualmente útil, não foi considerado nesta etapa do trabalho por questões de simplificação. Os construtors destes grupos que foram utilizados neste trabalho são descritos abaixo, e representados no meta-modelo da Figura 1.

- *Arquitetura de teste*: um *TestContext* representa o agrupamento de vários casos de teste, os quais se referem a um ou mais *Sut*s *(system under test). Sut* corresponde ao que será testado. *TestCase* é uma especificação de uma situação de teste do sistema, incluindo o que testar, entradas, e resultados esperados de acordo com as condições. *TestControl* determina como o *Sut* deve ser testado em um determinado contexto de teste (*TestContext*). Um *TestComponent* é uma classe de um sistema em teste que tem por objetivo auxiliar na execução de um ou mais casos de teste. Em teste unitário, é usado para representar um *stub*.
- *Comportamento de teste*: O comportamento de um caso de teste é especificado por diagramas comportamentais da UML (e.g. diagrama de seqüência). Um caso de teste sempre deve retornar um *verdict* (i.e. *pass*, *fail, inconclusive* ou *error*).

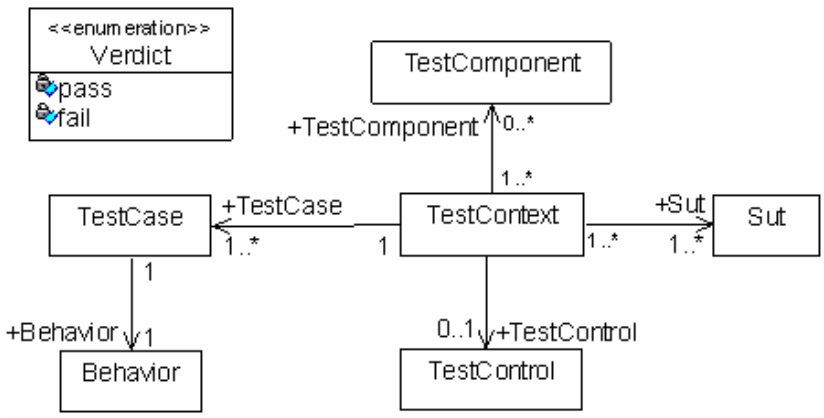


Figura 1. Construtos do U2TP adotados no trabalho.

3.2. Mapeamento do U2TP para JUnit

Os elementos descritos anteriormente possuem um mapeamento direto com os conceitos do JUnit, apresentado na Tabela 1, adaptada de [OMG, 2004].

Tabela 1. Mapeamento entre conceitos U2TP e JUnit.

U2TP	JUnit
TestContext	Subclasse de TestCase.
Sut	Qualquer classe Java.
TestCase	Uma operação pertencente a subclasse TestCase.
Behavior	Implementação de um método.
TestComponent	Uma classe Java.
TestControl	Método *runTest* que deve ser subscrito.
Verdict	Pass, fail e error.

3.3. Metodologia de Uso do U2TP

Uma metodologia para usar o U2TP é proposta em [Dai et al., 2004]. Ela assume que o U2TP é usado efetivamente a partir de um modelo de projeto UML existente, o qual deve ser enriquecido de detalhes para desenvolver as especificações de teste com o U2TP. A metodologia descreve e ilustra, passo a passo, como especificar testes para um sistema utilizando os construtos do U2TP. Para cada grupo do U2TP, são especificados os elementos mais importantes destes, e como podem ser usados. Ela também define para cada grupo do U2TP os elementos obrigatórios e opcionais. Contudo, ela não estipula o uso de cada elemento de acordo com o nível de teste (unidade, integração e sistema). Esta metodologia foi adotada neste trabalho. Com seu apoio, foram identificados os grupos do U2TP, junto com seus respectivos elementos, relevantes para o contexto de teste unitário. Após, foram definidas algumas convenções de uso destes construtos visando o mapeamento para código JUnit.

4. Uma abordagem para geração automatizada de drivers e stubs de teste para JUnit

O presente trabalho propõe uma abordagem para geração automatizada de *drivers* e *stubs* para JUnit, considerando um núcleo de conceitos do U2TP (Figura 1) e o

respectivo mapeamento destes nos conceitos do Junit (Tabela 1). A Figura 2 ilustra o cenário proposto para geração automatizada de *drivers* e *stubs* de teste.

As entradas necessárias são o projeto UML do software que será testado, junto com o projeto de teste unitário correspondente especificado com o U2TP. Ambos devem estar de acordo com uma série de pressupostos, discutidos na Seção 4.1. Estes dois modelos são gerados usando uma ferramenta de modelagem, e exportados como um documento XMI (XML Metadata Interchange). A geração dos *drivers* e *stubs* correspondentes é realizada em duas etapas: *Extração* e *Geração*. A Extração é baseada em dois modelos definidos neste trabalho: Modelo de Mapeamento U2TP-JUnit e Modelo de Mapeamento U2TP-XMI. O primeiro define quais elementos do U2TP são necessários para geração de *drivers* e *stubs* para ferramenta JUnit. O segundo define quais rótulos (*tags*) do documento XMI correspondem aos conceitos buscados e como os relacionamentos entre estes devem ser explorados. O Extrator então instancia o Modelo de Mapeamento U2TP-JUnit com base nos documentos XMI de entrada e, juntamente com um Analisador (*parser*), captura todos elementos que comporão os *drivers* e *stubs*. Finalmente, o Gerador de Código gera o código Java baseado em um *template* de *drivers* para JUnit, além de classes Java representando os *stubs*. É importante ressaltar que este processo não requer interação com o usuário.

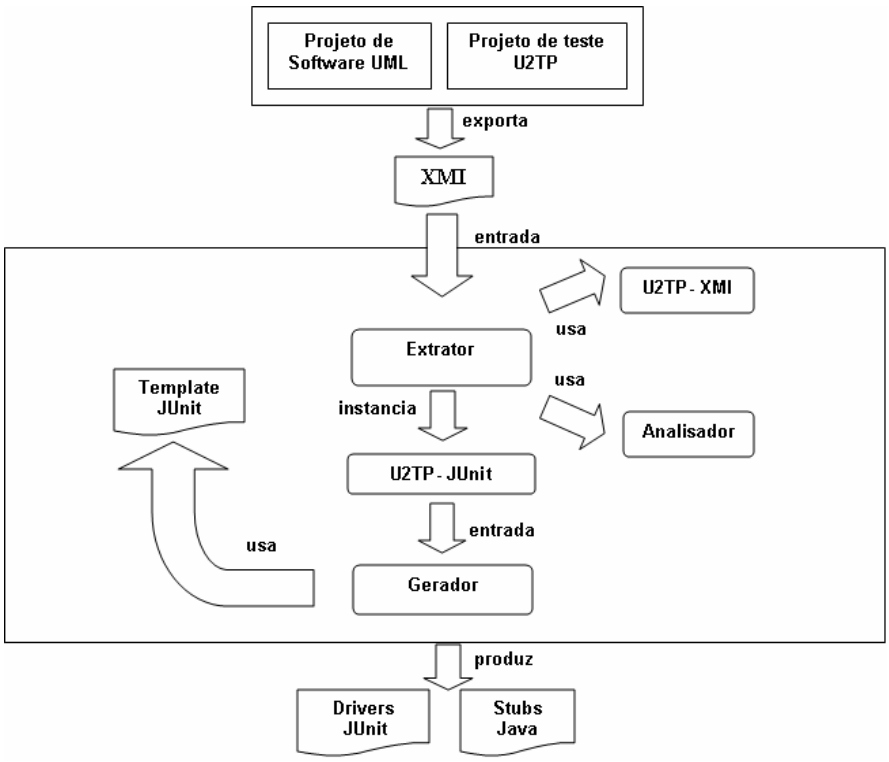


Figura 2. Cenário proposto.

4.1. Pressupostos para modelagem

Além da adoção da metodologia de [Dai et al., 2004], foram assumidos pressupostos necessários para especificar e modelar teste em nível de unidade com o U2TP. Os

pressupostos estabelecem simplificações ou padronizações sobre o uso da UML ou do U2TP, a fim de viabilizar a geração de *drivers* e *stubs*, e estão divididos em dois grupos: Projeto de Software e Projeto de Teste U2TP.

4.1.1 Projeto de Software

Assume-se a existência de um projeto de software UML correspondendo ao que será testado. Este deve conter obrigatoriamente um diagrama de classes em nível de projeto, pois define as operações das classes que podem ser testadas. Diagramas de seqüência são relevantes para a geração de *stubs*, mas não são obrigatórios. A Figura 3.(a) ilustra um projeto de software simples denominado Venda.

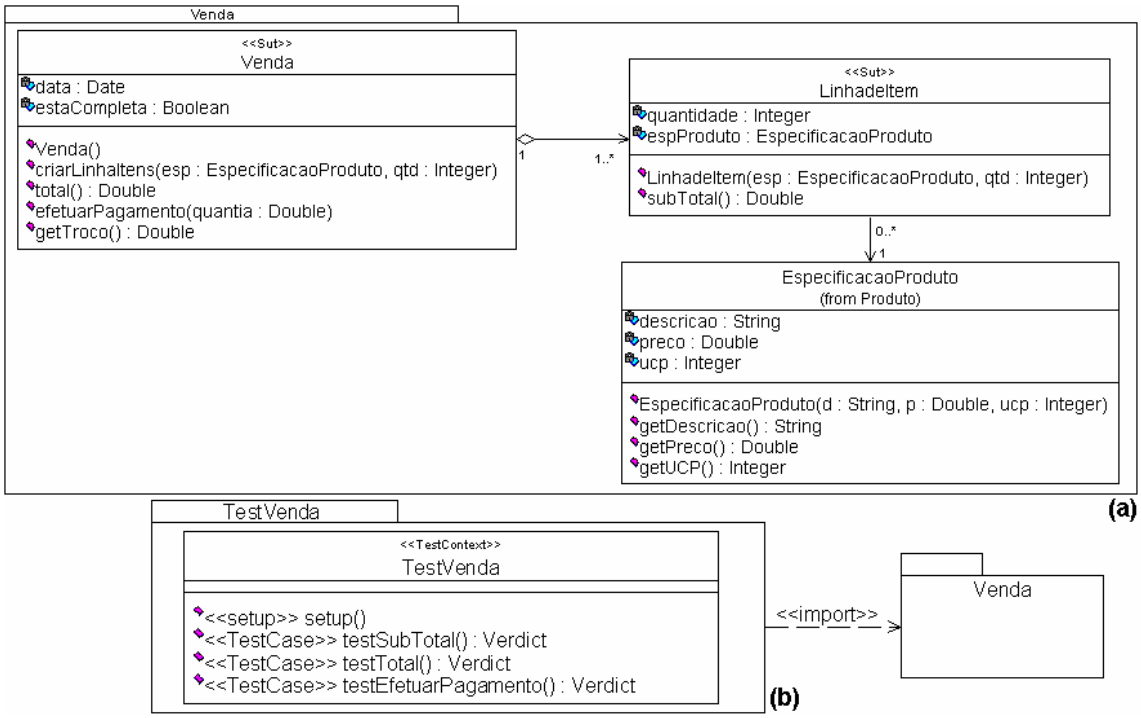


Figura 3. Projeto de Software e Projeto de Teste U2TP.

4.1.2 Projeto U2TP

Assume-se a definição de um pacote de teste referente ao projeto que será testado, como ilustrado na Figura 3.(b).

Arquitetura de Teste

- *Sut*: Todas as classes de software que serão testadas devem ser estereotipadas no modelo de projeto com <<sut>>, de acordo com a metodologia de [Dai et al., 2004]. Pelo menos um elemento <<sut>> é obrigatório (Figura 3.(a)).
- *TestContext* e *TestCase*: Deve haver uma e somente uma classe no pacote de teste estereotipada com <<testcontext>>. Nesta, cada caso de teste deve ser representado através de uma operação estereotipada com <<testcase>>. Deve haver no mínimo uma operação <<testcase>>.

- *TestComponent*: Podem ser especificadas várias classes estereotipadas com <<testcomponent>>. Estas podem ser especificadas tanto no projeto de software, quanto no pacote de teste. Quando definido no projeto de software, especifica-se que as classes *sut* dependem de funcionalidades de classes <<testcomponent>>. No pacote de teste, é possível criar novas classes representando apenas aqueles aspectos necessários ao teste.

Comportamento de Teste

- *TestCase*: o comportamento de um caso de teste deve ser especificado através de diagramas de seqüência, como na Figura 4 para o caso de teste testSubTotal.

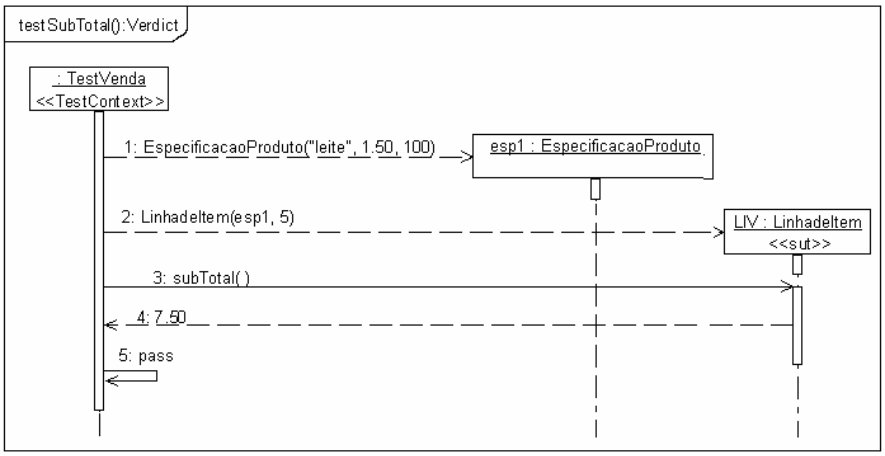


Figura 4. Comportamento do caso de teste testSubTotal.

Ainda, são assumidas as seguintes convenções para este tipo de diagrama:

1. O diagrama de seqüência deve ter o nome do caso de teste;
2. O objeto iniciador da interação do diagrama de seqüência deve ser uma instância da classe estereotipada com *TestContext*;
3. Os demais objetos do diagrama de seqüência são instâncias de classes do tipo *Sut*, *TestComponent* ou de qualquer classe do projeto de software, a qual será interpretada com um *stub* necessário. Estas instâncias devem ser nomeadas, e podem ser usadas como variáveis em outras mensagens do diagrama;
4. As mensagens representadas por setas pontilhadas partindo do objeto *TestContext* e recebidas pela classe representam a criação de um novo objeto;
5. Só é permitido um veredito por caso de teste, representado pela última mensagem do diagrama, na forma de uma auto-delegação ao objeto *TestContext*. A mensagem deve ser *pass* ou *fail*;
6. O veredito é estabelecido comparando o método representado pela antepenúltima mensagem, normalmente correspondente ao método testado, com o valor especificado para o teste. Assume-se que a penúltima mensagem corresponde ao valor especificado para o teste. O conjunto das três últimas mensagens constitui a assertiva para o caso de teste;

7. É possível declarar mensagens onde uma variável recebe o retorno de uma operação da seguinte forma: <variável> := <operação>.

Admite-se o uso de diagramas de seqüência para representar pré ou pós-condições comuns a todos os casos de teste. Para este fim, foram definidos dois novos estereótipos, <<setup>> e <<teardown>>, os quais devem ser associados a operações na classe <<testcontext>>. Pode haver no máximo uma operação <<setup>> e uma <<teardown>>, sendo que estes elementos são opcionais. Quando definidos, seu comportamento também deve ser especificado através de diagramas de seqüência.

4.2. Modelo de Mapeamento U2TP-JUnit

Na Tabela 1 foi apresentado o mapeamento do U2TP para os conceitos correspondentes na ferramenta JUnit. Além destes, foram acrescentados neste trabalho dois novos elementos para especificação de testes, a saber, *Configuration* e *Stub*. O diagrama de classes da Figura 5 estende o meta-modelo apresentado na Figura 1, apresentando todos os elementos usados para geração do código no JUnit. As classes em cor escura representam os elementos adicionados. O elemento *Stub* representa todos os *Stubs* de teste necessários à execução dos testes, os quais podem ter sido explicitamente especificados pelo projetista de software ou teste, ou ser derivados dos diagramas de seqüência do modelo de projeto. Um *testcomponent* é portanto um tipo de *stub*. *Configuration* é uma classe que representa os métodos *setup* e *teardown* do JUnit.

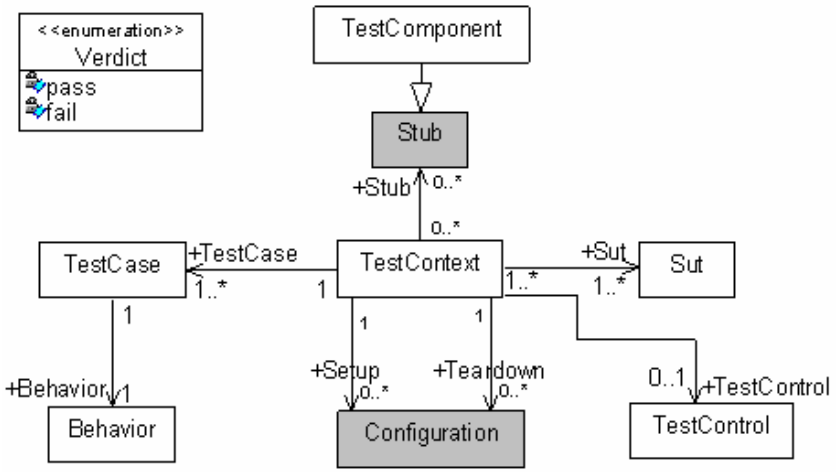


Figura 5. Modelo de mapeamento U2TP-JUnit.

4.3. Modelo de Mapeamento U2TP-XMI

Este modelo define como mapear os elementos do U2TP e extensões propostas em um documento XMI, indicando quais *tags* do documento XMI são necessários para extrair os elementos necessários à geração do código. Este mapeamento, sumarizado na Tabela 2, foi definido a partir do modelo de mapeamento U2TP-JUnit e das convenções adotadas. Cada elemento corresponde a uma ou mais *tags* XMI.

Tabela 2. Mapeamento dos elementos do U2TP e extensões para tags XMI.

Elemento U2TP e Extensões	XMI	Descrição
Sut, TestComponent, Stub e TestContext	XMIStereotype	identifica o tipo de elemento.
	XMIClass	representa o nome do elemento.
	XMIPackage	representa o nome do pacote onde se encontra o elemento.
	XMIClassifierRole e XMIDiagram	representa as instâncias do elemento usadas em diagramas de seqüência.
TestCase, Setup e Teardown	XMIStereotype	identifica o tipo da operação.
	XMIOperation	representa o nome da operação.
	XMIInteraction	representa a interação do testcase, setup e teardown.
	XMIMessage	representa as mensagens referentes àquela operação em diagramas de seqüência.
	XMIClassifierRole	representa os objetos destinatários das mensagens.
TestControl	XMIActionState	representa o nome das atividades do diagrama de atividades.

5. Extração e Geração de Drivers e Stubs

O extrator é responsável por identificar os elementos necessários contidos no documento XMI e armazená-los em uma instância do modelo de mapeamento U2TP-JUnit. O mecanismo de extração foi baseado no *parser* DOM – *Document Object Model* e no XMI gerado pela ferramenta Rational Rose. O *parser* DOM permite percorrer o documento XMI sem nenhuma ordem pré-definida. Dessa forma, é possível armazenar os elementos contidos no documento XMI em memória e manipulá-los mais facilmente. Para identificar e extrair os elementos necessários à geração de código, foram definidos algoritmos de mapeamento, os quais identificam os elementos no XMI com base no modelo U2TP-XMI, e os mapeiam para uma instância do modelo de mapeamento U2TP-JUnit.

Por fim, o gerador é responsável por gerar os elementos de código Java. No caso de *drivers*, esta geração segue um template da ferramenta alvo Junit. No caso de *stubs*, classes Java com os métodos necessários são geradas. O restante desta seção detalha os aspectos importantes da extração e geração de *drivers* e *stubs*.

5.1. Extração de Drivers

O parser DOM permite recuperar coleções de elementos possuindo um mesmo rótulo. Desta forma, para extrair os elementos necessários da especificação e instanciar o modelo U2TP-Junit são percorridas todas as *tags* correspondentes a cada elemento e através de um conjunto de métodos, extraídas as informações pertinentes.

O funcionamento do extrator será ilustrado considerando o caso de teste *testSubTotal*, detalhado no diagrama de seqüência da Figura 4. De acordo com a Tabela 2, cinco *tags* estão envolvidas. A *tag* que representa as interações envolvidas no detalhamento deste caso de teste através de um diagrama de seqüência é <UML:Interaction>, como ilustra a Figura 6.

```
- <UML:Interaction xmi.id="G.21" name="{Use Case View::Comportamento}testSubTotal"
    visibility="public" isSpecification="false">
  - <UML:Interaction.message>
      <UML:Message xmi.id="G.16" name="EspecificaçãodeProduto("leite", 1.50, 100)"
        visibility="public" isSpecification="false" sender="G.10" receiver="G.15" message3="G.17"
        communicationConnection="G.12" action="XX.26.1247.53.38" />
      <UML:Message xmi.id="G.17" name="LinhadeItemVenda(esp1, 5)" visibility="public"
        isSpecification="false" sender="G.10" receiver="G.14" message3="G.18"
        predecessor="G.16" communicationConnection="G.11" action="XX.26.1247.53.39" />
      <UML:Message xmi.id="G.18" name="subTotal( )" visibility="public" isSpecification="false"
        sender="G.10" receiver="G.14" message3="G.19" predecessor="G.17"                    antepenúltima
        communicationConnection="G.11" action="XX.26.1247.53.40" />
      <UML:Message xmi.id="G.19" name="7.50" visibility="public" isSpecification="false"
        activator="G.18" sender="G.14" receiver="G.10" message3="G.20" predecessor="G.18"
        communicationConnection="G.11" action="XX.26.1247.53.41" />                         penúltima
      <UML:Message xmi.id="G.20" name="pass" visibility="public" isSpecification="false"
        sender="G.10" receiver="G.10" predecessor="G.19" communicationConnection="G.13"
        action="XX.26.1247.53.42" />                                                        última
    </UML:Interaction.message>
  </UML:Interaction>
```

Figura 6. Tag <UML:Interaction>.

O atributo *name* da *tag* <UML:Interaction> identifica o nome do caso de teste. As *tags* <UML:Message> representam as mensagens trocadas por objetos neste caso de teste. Para cada mensagem, existe um objeto que recebe (atributo *receiver*) e outro que envia a mensagem (atributo *sender*). Na Figura 6, o objeto que recebe a primeira mensagem é representando pelo identificador "G.15". Dessa forma, para saber o nome da instância do objeto que recebe esta mensagem, são percorridas todas *tags* <UML:ClassifierRole> até encontrar aquela com identificador igual a "G.15". De acordo com a Figura 7, o nome desta instância é esp1.

```
- <UML:ClassifierRole xmi.id="G.15" name="esp1" visibility="public" isSpecification="false"
    isRoot="false" isLeaf="false" isAbstract="false" base="S.298.1247.52.12"
    availableFeature="S.298.1247.52.16" message1="G.16">
  - <UML:ClassifierRole.multiplicity>
    - <UML:Multiplicity>
      - <UML:Multiplicity.range>
          <UML:MultiplicityRange xmi.id="id.2991547.18" lower="1" upper="1" />
        </UML:Multiplicity.range>
      </UML:Multiplicity>
    </UML:ClassifierRole.multiplicity>
  </UML:ClassifierRole>
```

Figura 7. Tag <UML:ClassifierRole> correspondente ao objeto esp1.

Para extrair o comportamento do caso de teste são buscadas no diagrama de seqüência todas as mensagens, com os respectivos destinatários, à exceção das três últimas mensagens. O formato da mensagem extraída varia conforme o tipo de mensagem: instanciação de classe (e.g. esp1 = new EspecificaçãodeProduto("leite", 1.50, 100)) ou mensagem à instância (e.g. LIV.subTotal()).

As três últimas mensagens são usadas para montar a assertiva para o caso de teste, como ilustra a Figura 8. A última mensagem corresponde ao veredito do caso de teste (*pass* ou *fail*), a qual equivale ao tipo de assertiva usada no caso de teste. A antepenúltima mensagem normalmente corresponde ao método testado, que será igual à mensagem representada pelo caso de teste. A penúltima mensagem corresponde ao valor esperado da execução da operação.

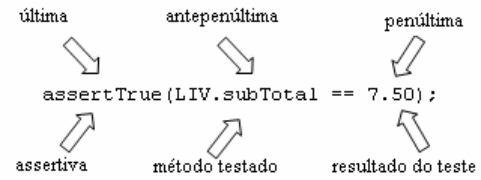


Figura 8. Três últimas mensagens constituem a assertiva.

Cabe salientar que neste exemplo foi extraído o comportamento do caso de teste testSubTotal. Na busca do nome do caso de teste, conforme a Tabela 2, estão envolvidas as *tags* <UML:Stereotype> e <UML:Operation>. Assim, o procedimento para buscar os demais elementos é o mesmo, mudando apenas as *tags* envolvidas na extração de cada elemento.

5.2. Gerador de Código para Driver

O template Junit define a estrutura básica de um *driver* de teste, descrita na Figura 9. Considerando este template, o nome e o comportamento do caso de teste testSubtotal, o código correspondente é gerado como segue:

```
public void testSubTotal(){
       esp1 = new EspecificaçãdeProduto("leite", 1.50, 100);
       LIV = new LinhadeItemVenda(esp1, 5);
       assertTrue(LIV.subTotal() == 7.50);}
```

```
(1)package pacote_testcontext; //obrigatório
(2)import junit.framework.TestCase;
(3)import nome_pacote_sut.nome_classe_sut;
(4)import nome_pacote_testcomponent.nome_classe_testcomponent;
(5)import nome_pacote_stub.nome_classe_stub;
...
(6)if (testcontrol != "){
(7)import junit.framework.TestSuite;
(8)import junit.framework.Test;
}

(9) public class nome_testcontext extends TestCase{ //obrigatório
(10)     private nome_classe_sut nome_objeto_sut; //obrigatório
(11)     private nome_classe_testcomponent nome_objeto_testcomponent;
(12)     private nome_classe_stub nome_objeto_stub;

(13)     public nome_testcontext(String testName){
(14)          super(testName);
(15)     }

(16)     protected void setup() throws Exception{
(17)          super.setup();
(18)          nome_objeto = new nome_construtor(parâmetros);
(19)          nome_classe nome_objeto = new nome_construtor(parâmetros);
(20)     }

(21)     protected void teardown() throws Exception{
(22)          super.teardown();
(23)          nome_objeto = null;
(24)     }

(25)     public void test_nome_testcase(){
(26)          nome_objeto.nome_mensagem;
(27)          //declaração de variáveis locais;
(28)          //assertiva, ex: asserTrue(método_testado == resultado_teste);
(29)     }
         ...
(30)     public static Test suite(){
(31)          TestSuite suite = new TestSuite();
(32)          suite.addTest(new nome_testcontext("nome_testcase"));
(33)          . . .
(34)          return suite;
(35)     }
(36)}
```

Figura 9. Template JUnit.

5.3. Extração e Geração de Stubs de Teste

Os s*tubs* são extraídos: a) a partir de especificações do elemento *testcomponent,* definidas tanto no projeto de software, como na especificação do teste U2TP; b) quando não explicitamente definidos, através de outros diagramas de seqüência do projeto de software. No primeiro caso, o extrator busca as *tags* correspondentes ao componente de teste, como definido na Tabela 2. No segundo, o extrator deve percorrer os diagramas de seqüência. A Figura 10 mostra um diagrama de seqüência do projeto de software do Sistema Venda referente ao comportamento do método *subTotal*.

Para extrair os *stubs*, são percorridas todas as *tags* que representam as interações dos diagramas de seqüência (exceto aquelas correspondentes ao comportamento de casos de teste) até encontrar uma mensagem correspondente ao método testado no caso de teste. Se o objeto que receber esta mensagem por sua vez enviar uma mensagem para um ou mais objetos, então é instanciado um *stub* para cada classe correspondente a estes objetos. Cada *stub* possui apenas um construtor e os métodos invocados. Cabe salientar que os *Stubs* são incluídos tanto no código do *driver* de teste, na forma de variáveis (Figura 9), como fora do *driver*, como classes Java.

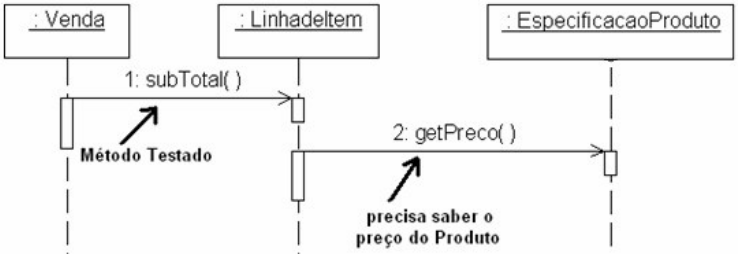


Figura 10. Diagrama de seqüência do método subTotal.

6. Estudo de Caso

Um estudo de caso foi realizado no centro de pesquisa em teste de software – CPTS, localizado na PUCRS. Para a geração dos *drivers* e *stubs* de teste foi usado um sistema para locações de DVDs desenvolvido em Java, o qual foi originalmente desenvolvido para experimentos de teste de software. Este sistema contém uma série de artefatos que foram usados para especificar os casos de teste usando o U2TP, entre eles: diagrama de classes, diagrama de casos de uso, casos de uso expandido, bem como alguns *drivers* de teste que já haviam sido previamente codificados por um testador do CPTS.

Para possibilitar a comparação entre o código produzido manualmente, e aquele gerado automaticamente, foram especificados apenas os testes referentes aos *drivers* previamente codificados no CPTS, os quais correspondem a métodos de 3 classes do sistema: CatalogoCategoria, Categoria, e CatalogoCliente. Juntas, estas classes contemplavam todas as situações contempladas pelos modelos e algoritmos propostos neste trabalho, como resumido na Tabela 3.

Tabela 3. Construtos U2TP encontrados em cada classe do estudo de caso.

Elementos/Classes	Categoria	CatalogoCategoria	CagalogoCliente
TestCase	4	4	3
TestComponent/Stub	0	3	2
Setup	0	1	1
Teardown	0	1	1
TestControl	0	0	1

A especificação dos casos de teste para estas classes utilizando o U2TP foi baseada na documentação recebida, e de acordo com os pressupostos descritos na Seção 4.1. A Figura 11 ilustra a especificação de teste unitário referente à classe CatalogoCategoria. A especificação para o comportamento do caso de teste testNew_config é mostrado na Figura 12.

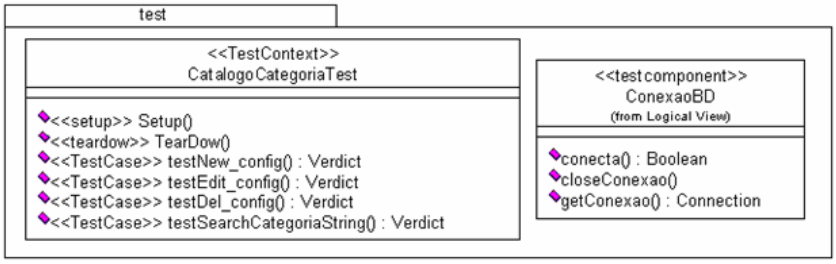


Figura 11. Pacote de teste unitário.

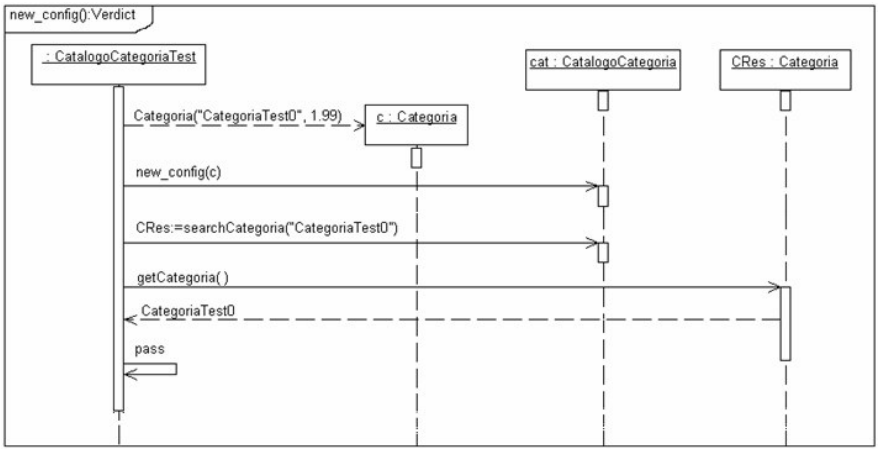


Figura 12. Comportamento para o caso de teste testNewConfig.

Foi especificado um elemento *TestComponent* no projeto de teste U2TP denominado ConexaoBD (Figura 11). Este foi especificado, para conexão com o banco de dados, uma vez que esta classe não existia no projeto de software. Já no projeto de software, a classe Categoria foi estereotipada com <<testcomponent>>, pois é necessária à execução dos casos de teste. A Figura 12 o uso do componente de Categoria no comportamento de um caso de teste.

O *driver* Junit gerado para os testes modelados com o U2TP é mostrado na Figura 13.(a). Também foram gerados os *stubs* de teste referente às classes Categoria e ConexaoDB, juntamente com seus métodos e atributos.

```
(1)package tmm.test;
(2)import tmm.bdi.CatalogoCategoria;
(3)import tmm.entities.Categoria;
(4)import junit.framework.TestCase;

(5)public class CatalogoCategoriaTest extends TestCase{

(6)    private CatalogoCategoria cat;

(7)    protected void setUp() throws Exception {
(8)          super.setUp();
(9)           cat = new CatalogoCategoria( );
(10)          cbd = new conecta( );
(11)   }

(12)   protected void tearDown() throws Exception {
(13)         super.tearDownn();
(14)          cat = null;
(15)   }

(16)   public void testNew_config(){
(17)          Categoria c = new Categoria("CategoriaTest0", 1.99);
(18)          cat.new_conf(c);
(19)         CatalogoCategoria CRes = cat.searchCategoriaString("CategoriaTest0");
(20)          assertEquals(0, CRes.getCategoria( ),"CategoriaTest0");
(21)   }

(22)   public void testDel_config(){
(23)          Categoria c = new Categoria("CategoriaTest1", 1.99);
(24)          cat.new_conf(c);
(25)          cat.del_conf(c);
(26)          CatalogoCategoria CRes = cat.searchCategoriaString("CategoriaTest1");
(27)          assertEquals(0, CRes,null);
(28)   }

(29)   public void testEdit_config(){
(30)          Categoria c = new Categoria("CategoriaTest0", 2.99);
(31)          cat.new_conf (c);
(32)          c.setPreco(1.0);
(33)          cat.edit_conf (c);
(34)          CatalogoCategoria CRes = cat.searchCategoriaString("CategoriaTest0");
(35)          assertTrue(CRes.getPreco( ) == 1.0);
(36)   }

(37)   public void testSearchCategoriaString(){
(38)          Categoria c = new Categoria("CategoriarTest0", 1.99);
(39)          cat.new_conf(c);
(41)          CatalogoCategoria CRes = cat.searchCategoriaString("CategoriaTest0");
(42)          String nomeCategoria = CRes.getCategoria( );
(42)          assertEquals(0, c.getCategoria( ),"nomeCategoria");
(43)}  }                                                                    (a)
```

```
(17)  public void testNew_config() {
(18)          Categoria c = new Categoria("CategoriaTest0", 1.99);
(19)          cat.new_config(c);
(20)          String res = "CategoriaTest0";
(21)          Categoria cRes = cat.searchCategoria("CategoriaTest0");
(22)          assertEquals("", cRes.getCategoria(), res);
(23)   }

(24)  public void testDel_config() {//funcionoiu
(25)          Categoria c = new Categoria("CategoriaTest1", 1.99);
(26)          cat.new_config(c);
(27)          cat.del_config(c);
(28)          Categoria cRes = cat.searchCategoria("CategoriaTest1");
(29)          assertNull("E null??", cRes);
(30)   }                                                                   (b)
```

Figura 13. Driver de teste gerado para classe CatalogoCategoria.

6.1. Resultados

Os resultados deste estudo de caso foram bastante satisfatórios, pois foi gerado 100% do código necessário para os *drivers* e *stubs*. O estudo de caso foi bastante abrangente, pois explorou todos os elementos propostos nos modelos, exercitando assim diferentes aspectos dos algoritmos desenvolvidos.

Outro aspecto importante está relacionado à qualidade do código gerado. A Figura 13.(a) mostra o *driver* gerado automaticamente, enquanto que a Figura 13.(b) mostra uma porção do código produzido pelo testador referente a dois casos de teste , a saber, testNew_config e testDel_config. Observa-se que o código produzido automaticamente é equivalente em termos de funcionalidade, mas levemente menor.

A principal vantagem deste trabalho está na qualidade da documentação dos artefatos de teste. Além de ser uma documentação padronizada, visual e independente de linguagem de programação, ela facilita a manutenção dos testes, bem como testes de regressão. Em comparação, a única documentação produzida pelo testador neste estudo de caso era o próprio código-fonte, o que dificulta a manutenção dos testes, e seu reaproveitamento em caso de implementação do sistema em outras linguagens de programação. Outro fator é que a codificação é fortemente dependente da habilidade e experiência do programador do *driver*, o que muitas vezes pode resultar em um código fonte de baixa qualidade.

Observou-se ainda uma pequena redução do *workload* para este estudo de caso. O testador gerou a partir da IDE Eclipse o esqueleto de cada *driver*, e dispunha-se do tempo gasto por ele para programar o comportamento dos casos de teste de cada *driver* . No total, o testador despendeu cerca de 70 minutos na codificação dos 3 *drivers*. Para razões de comparação, contabilizou-se o tempo para modelagem dos mesmos casos de teste com o U2TP, partindo do diagrama de classes e da especificação dos casos de uso, o que totalizou menos de 60 minutos. Portanto, neste estudo a redução de tempo foi de aproximadamente 15%. Contudo, devem ser realizados mais experimentos para saber se este tipo de ganho é generalizável, em que proporção, e sob quais condições.

7. Trabalhos Relacionados

A geração de *driver*s de teste para JUnit é proposta em vários artigos. Entretanto, nenhuma abordagem propõe a geração totalmente automatizada de *driver*s, incluindo o comportamento dos casos de teste, tampouco a geração dos *stubs* de teste. Por exemplo, algumas IDEs como *Eclipse* e *NetBeans* possibilitam gerar automaticamente a estrutura do *driver* de teste para JUnit, porém é necessário completar esta com as variáveis, importações de pacotes e código referente ao comportamento dos casos de teste. O mesmo dá-se para o *addin* TestExpert [IBM, 2005] da ferramenta Rational Rose.

Outras abordagem geram parcialmente *drivers* e *stubs* de teste para JUnit a partir de modelos UML. Em [Wittevrongel e Maurer, 2001], uma ferramenta denominada *Scensor* gera *drivers* de teste para JUnit a partir de diagramas de seqüência. Entretanto, esta abordagem não adota os conceitos do U2TP e também não gera *stubs* de teste. Já em [Fraikin e Leonhardt, 2002] também é proposta a geração automatizada de código de teste para aplicações Java, através de uma ferramenta denominada *Seditec*. Esta abordagem também usa diagramas de seqüência para geração de testes automatizados, incluindo *stubs* de teste. Porém, o código gerado é específico da ferramenta *Seditec*, de uso muito mais restrito que o JUnit. Duas outras limitações desta ferramenta são: a) não considera o U2TP; e b) os *stubs* de teste gerados estão relacionados com as pré-condições para a execução dos testes, não considerando a geração de *stubs* a partir do relacionamento com outras classes, como no presente trabalho.

Outras abordagens propõem a geração de casos de teste para JUnit a partir de notações que são abstrações de linguagens de programação [Cheon e Leavens, 2002] [Zongyuan et al., 2004]. Nestas abordagens é necessário codificar os casos de teste usando as linguagens JML e AspectJ, respectivamente, não permitindo beneficiar das vantagens preconizadas na adoção do U2TP para especificação de teste.

8. Considerações Finais

Este trabalho propôs uma abordagem para geração automatizada de *drivers* e *stubs* de teste para JUnit a partir de especificações de testes modeladas com o U2TP. Com a adoção do U2TP, é possível especificar testes em diferentes níveis usando uma linguagem visual e padronizada. Desta forma, problemas referentes à falta de documentação, entendimento das especificações, bem como dependência de linguagem de programação são resolvidos.

Em relação aos trabalhos relacionados, as principais contribuições desta proposta são: a) considerar especificações padronizadas em U2TP como entrada para a geração

automática; b) gerar *drivers*, incluindo o código para os casos de teste; c) gerar *stubs*, considerando tanto a especificação explícita quanto implícita destes; d) não necessitar de intervenção do usuário neste processo. Além disto, o código gerado é padronizado, não sendo dependente de habilidade, estilo ou experiência do codificador do *driver*.

Entre as principais limitações deste trabalho estão: a) dependência dos algoritmos desenvolvidos do *parser* DOM e do código XMI produzido pela ferramenta Rational Rose; e b) não considerar o grupo de Dados de Teste do U2TP.

Como trabalhos futuros, podem ser citados, entre outros: a) a extensão desta proposta para outras ferramentas da família xUnit (e.g. cppUnit, n.Unit); b) a consideração dos conceitos do U2TP para Dados de Teste; c) a extensão da proposta para outros níveis de teste, com a geração de código correspondente a ferramentas de teste próprias a estes níveis, etc.

9. Referências

Beck, K (2003). "Test-Driven Development". Addison-Wesley, 220p.

Burnstein, I (2003). "Practical software testing: a process-oriented approach". Springer-Verlag, 709p.

Cheon, Y.; Leavens, G. T (2002). "A Simple and practical approach to unit testing: the JML and JUnit way". In: Proceedings of 16th European Conference Object-Oriented Programming (ECOOP), 231-255p.

Dai, J. Grabowski, A. R (2003). "The UML 2.0 Testing Profile and its Relation to TTCN-3". In: 15th IFIP International Conference on Testing of Communicating Systems (TestCom), pp. 79-94.

Dai, Z.R., Grabowski, J., Neukirchen, H., Pals, H (2004). "From Design to Test with UML". In: 16th IFIP International Conference on Testing of Communicating Systems (TestCom), pp. 33-49.

Eclipse (2005). "Eclipse". http://www.eclipse.org, May.

Fraikin, F.; Leonhardt, T (2002). "SeDiTeC – Testing Based on Sequence Diagrams." In: 17th IEEE International Conference on Automated Software Engineering (ASE), pp. 262-266.

IBM (2005). Test Expert Rose AddIn. http://www-128.ibm.com/developerworks/rational/content/03July/2500/2834/Rose/rational_rose.html, June.

NetBeans (2005). "NetBeans". http://www.netbeans.org, April.

OMG (2004). "UML 2.0 Testing Profile Specification". Technical Report, OMG. http://www.omg.org/docs/ptc/04-04-02.pdf, June 2004, 114p.

Wittevrongel, J.; Maurer, F (2001). "SCENTOR: Scenario-Based Testing of E-Business Applications". In: 10th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), pp. 41-48.

Zongyuan, Y.; Guoqing, X; Haitao, H; Qian, C; Ling, C; Fengbin, X (2004). "JAOUT: Automated Generation of Aspect-Oriented Unit Test". In: 11th Asia-Pacific Software Engineering Conference (APSEC), pp. 374-381.

Tratamento de Exceções Sensível ao Contexto

Karla Damasceno[1], Nélio Cacho[2], Alessandro Garcia[2], Carlos Lucena[1]

[1]Departamento de Informática – PUC-Rio – Brasil

[2]Computing Department – Lancaster University, United Kingdom

{karla,lucena}@inf.puc-rio.br, {n.cacho,a.garcia}@lancaster.ac.uk

Abstract. *Exception handling on mobile systems is not a trivial task due to their intrinsic characteristics: context-awareness, asynchrony, and intermittent connectivity. Conventional mechanisms of exception handling can not be used for many reasons. One of the main problems is that the error recovery and the exception handling strategies frequently need to be selected according to contextual information. This paper identifies limitations of existent mechanisms, as well as it proposes a new model and architecture for context-sensitive exception handling. The evaluation of our proposal has been performed through the implementation of three applications from heterogeneous domains.*

Resumo. *Tratamento de exceções em aplicações móveis não é uma tarefa trivial devido às características destas aplicações, como sensibilidade ao contexto, comunicação assíncrona e conexão instável. Mecanismos convencionais de exceções não podem ser utilizados por várias razões. Um dos principais problemas é que a busca e execução dos tratadores e a propagação de exceções precisam freqüentemente considerar informações de contexto. Este artigo identifica limitações dos mecanismos existentes, bem como propõe um novo modelo e uma arquitetura para tratamento de exceções sensível ao contexto. A avaliação da solução proposta foi conduzida através da implementação de três aplicações de domínios heterogêneos.*

1. Introdução

Os avanços recentes da Computação Móvel têm possibilitado a construção de aplicações sensíveis ao contexto capazes de monitorar e utilizar dinamicamente informações que provêm do ambiente ou usuário (Coutaz et al., 2005). Tais aplicações precisam tratar variações freqüentes em seus contextos de execução, tais como, mudanças de temperatura, bateria e localização. Embora a incorporação de mecanismos apropriados para tratamento de exceções seja fundamental para o desenvolvimento de sistemas móveis confiáveis, o projeto de tais mecanismos não é uma tarefa trivial em função das próprias características destes sistemas, tais como mobilidade, abertura, conexão instável e comunicação assíncrona. Estas características incorrem em maior imprevisibilidade das exceções e na necessidade de novos mecanismos modulares apropriados para realizar a propagação de eventos excepcionais. Além disso, existe um sério conflito da natureza síncrona do tratamento de exceções tradicional com a instabilidade de conexão e a comunicação assíncrona inerentes as aplicações móveis sensíveis ao contexto.

Até o momento, o que se observa é que a maior parte das aplicações utiliza apenas o mecanismo de exceções fornecido pelas linguagens de programação subjacentes (Garcia et al., 2001). Entretanto, as abstrações e mecanismos convencionais não são satisfatórios por vários motivos (Cacho et al., 2006; Garcia, 2005; Tripathi & Miller, 2000). Primeiro, a propagação de exceções deve considerar mudanças contextuais que constantemente ocorrem nas aplicações móveis. Segundo, as atividades de recuperação de erros e a estratégia de tratamento de exceções freqüentemente precisam ser selecionadas de acordo com informações de contexto, podendo ser necessário ativar diferentes tratadores para uma mesma exceção de acordo com o contexto. Além disso, a própria caracterização de uma exceção pode depender do contexto dos dispositivos - isto é, um estado do sistema pode ser considerado errôneo em uma dada localização onde o dispositivo se encontra, mas não em outra localização.

Não existem trabalhos na literatura que provêem suporte a tratamento de exceções sensível ao contexto para o desenvolvimento de aplicações móveis robustas. Em geral, as propostas existentes são restritas a solucionar questões gerais relacionadas às aplicações móveis baseadas em agentes (Tripathi & Miller, 2000; Souchon et al., 2004; Iliasov & Romanovsky, 2005). Estes mecanismos não possuem as abstrações necessárias para realizar um tratamento de exceções sensível ao contexto, como exemplo, a possibilidade de definir escopos que estejam associados diretamente a localização física dos dispositivos. Embora os sistemas de *middleware* que oferecem suporte à construção de aplicações móveis sensíveis ao contexto (Iliasov & Romanovsky, 2005; Sacramento et al., 2004; Capra et al., 2003) considerem as abstrações referentes a sensibilidade ao contexto, não existe suporte para engenheiros de software lidarem explicitamente com o tratamento de exceções contextuais.

Neste sentido, as contribuições deste trabalho são as seguintes: (a) identificação de um conjunto de requisitos necessários ao desenvolvimento de um mecanismo de exceções para aplicações móveis; (b) um modelo com abstrações adequadas para tratamento de exceções sensível ao contexto, desenvolvido para atender os requisitos anteriores; (c) uma arquitetura de software para um mecanismo de tratamento de exceções sensível ao contexto, que pode ser adotada por outras implementações; e (d) implementação de um mecanismo de tratamento de exceções sensível ao contexto usando MoCA (Sacramento et al., 2004), baseado no modelo e na arquitetura propostos.

Este artigo está organizado como a seguir. A seção 2 descreve conceitos referentes a sensibilidade ao contexto e ao tratamento de exceções e analisa os problemas para um efetivo tratamento sensível ao contexto. As seções 3 e 4 apresentam um modelo e uma arquitetura para um mecanismo de tratamento de exceções em aplicações móveis sensíveis ao contexto. A seção 5 apresenta uma avaliação do mecanismo. A seção 6 apresenta os trabalhos relacionados. Finalmente, a seção 7 apresenta as conclusões e os trabalhos futuros.

2. Sensibilidade ao Contexto e Tratamento de Exceções

Inicialmente apresentamos as definições de contexto, aplicações sensíveis ao contexto e sistemas de *middleware* para desenvolver estas aplicações (Seção 2.1). Em seguida, descrevemos os principais conceitos relacionados ao tratamento de exceções e apresentamos a evolução dos mecanismos de exceções e a necessidade de um

mecanismo específico para estas aplicações (Seção 2.2). Finalmente, a partir da análise da aplicação *Virtual Lines* desenvolvida em MoCA, identificamos um conjunto de requisitos necessários para incorporar a sensibilidade ao contexto no tratamento de exceções em aplicações móveis (Seção 2.3). Esta mesma aplicação será utilizada na Seção 5 para avaliar a aplicabilidade do mecanismo proposto.

2.1. Sensibilidade ao Contexto

Contexto é qualquer informação utilizada para caracterizar a situação de uma pessoa, lugar ou objeto relevante para a interação entre um usuário e uma aplicação (Dey & Abowd, 1999). Uma aplicação é *sensível ao contexto* se utiliza informações de contexto para fornecer serviço ou informação relevante para o usuário (Dey & Abowd, 1999). Tais aplicações podem usar diversas informações como perfil de usuários, bateria, localização, temperatura e pressão sanguínea. Desta forma, além de ter conhecimento sobre sua "situação" em um dado momento, tais aplicações possuem a habilidade para interpretar e usar o contexto como base para um comportamento adaptativo.

Para permitir esta adaptação baseada em contexto, diversos paradigmas de coordenação têm sido adotados para a implementação de sistemas de *middleware* específicos, como (a) baseado em eventos ou *publish-subscribe* (Sacramento et al., 2004), (b) baseado em espaços de tuplas (Iliasov & Romanovsky, 2005) e (c) reflexivo (Capra et al., 2003). Tais sistemas são responsáveis por realizar a coleta das informações de contexto, a disseminação destas informações e a notificação de alterações de contexto de acordo com o interesse dos clientes. Neste artigo, também discutiremos como o nosso modelo (Seção 3) e a arquitetura (Seção 4) para tratamento de exceções sensível ao contexto foram implementados em um *middleware publish-subscribe*, chamado MoCA (Sacramento et al., 2004).

De fato, o modelo e arquitetura propostos são especialmente adequados para mecanismos comumente definidos em sistemas de *middleware publish-subscribe*. A principal justificativa para a seleção deste paradigma está relacionada à própria natureza das aplicações móveis, uma vez que o paradigma *publish-subscribe* se apresenta como uma alternativa interessante por duas características fundamentais: (i) comunicação assíncrona, a comunicação pode ocorrer mesmo se o destino está indisponível; e (ii) comunicação anônima e interação desacoplada, um publicador e um subscritor não precisam conhecer um ao outro. Estas duas características impulsionaram a utilização deste paradigma, de modo que hoje ele é adotado por um número crescente de sistemas de *middleware* sensíveis ao contexto (Pietzuch & Bacon, 2002; Meier & Cahill, 2002; Sacramento et al., 2004; Muthusamy et al., 2005; Cugola & Cote, 2005), inclusive por produtos de grandes indústrias de software como a IBM (MQTT, 2006). Ademais, o modelo de tratamento de exceções também pode ser estendido para arquiteturas baseadas em espaços de tuplas, pois estas soluções apresentam vários mecanismos similares ao paradigma *publish-subscribe*. De fato, a maioria dos sistemas para aplicações móveis se apóia nestes dois paradigmas.

2.2. Tratamento de Exceções

Desenvolvedores de sistemas confiáveis freqüentemente se referem a erros como exceções porque erros raramente se manifestam durante a atividade normal do sistema (Goodenough, 1975; Parnas & Würges, 1976). Em situações de erro, um componente

gera exceções que modelam a condição de erro e o sistema deve realizar o tratamento daquelas exceções (Lee & Anderson, 1990). Desta forma, *tratamento de exceções* é a capacidade que um software possui de reagir apropriadamente diante da ocorrência de exceções, continuando ou interrompendo sua execução, a fim de preservar a integridade do estado do sistema (Garcia et al., 2001).

Quando um serviço requerido não pode ser realizado, ele retorna uma *resposta anormal*, ou exceção (Figura 1a). Ao receber uma resposta anormal de um outro componente (exceção externa) ou levantar uma exceção durante sua própria atividade normal (exceção interna), as atividades adequadas para tratamento da exceção devem ser executadas. Um *tratador* de exceções é a parte da *atividade anormal* (ou excepcional), é, portanto, a parte do código da aplicação que fornece medidas específicas da aplicação para o tratamento da exceção levantada (Garcia et al., 2001). Este é vinculado a uma região particular do código normal, chamada *região protegida* ou *escopo de tratamento*. Se uma exceção é levantada em uma região protegida, o fluxo de controle normal é desviado para um *fluxo de controle excepcional* (Garcia et al., 2001). Após o tratamento da exceção, o componente pode voltar a prover o serviço normal ou *propagar* a exceção para um componente de mais alto nível.

Mecanismos de tratamento de exceções provêem suporte explícito a propagação de exceções, e mudanças no fluxo de controle normal para o fluxo de controle excepcional (Goodenough, 1975; Parnas & Würges, 1976). Eles também são responsáveis por suportar diferentes estratégias de fluxo excepcional e procurar os tratadores apropriados depois que a ocorrência de uma exceção for detectada. Além disso, eles devem ter um projeto simples, serem fáceis de usar, e fornecer uma separação explícita entre o código normal e excepcional (Garcia et al., 2001). Mecanismos de tratamento de exceções são tipicamente parte de linguagens de programação ou são uma característica de sistemas de *middleware* dedicadas a diferentes domínios de aplicação e seguindo diferentes estilos de arquitetura.

A Figura 1 ilustra a evolução dos mecanismos de tratamento de exceções, considerando (a) programas seqüenciais, (b) programas distribuídos concorrentes e (c) aplicações móveis sensíveis ao contexto. O modelo tradicional para tratamento de exceções em programas seqüenciais e apresentado na Figura 1a. Este modelo consiste basicamente de um componente ideal tolerante a falhas (Lee & Anderson, 1990), com sua parte de atividade normal e anormal (tratadores), as requisições de serviços e as exceções internas que são tratadas de forma síncrona ou precisam ser propagadas sincronamente aos componentes de mais alto nível. Entretanto este mecanismo torna-se inapropriado para programas distribuídos concorrentes (Xu et al., 1995), onde uma exceção que ocorre em uma *thread*, envolve todas as *threads* participantes da colaboração. Assim, como ilustra a Figura 1b, para tratar exceções que ocorrem em programas distribuídos concorrentes, é necessário adicionar o conceito de transação, e as exceções precisam ser propagadas de forma síncrona para todas as *threads* envolvidas.

Entretanto, o modelo do componente ideal (Figura 1a) e o conceito de transação (Figura 1b) não são satisfatórios para incorporação de tratamento de exceções em aplicações móveis sensíveis ao contexto. Isto ocorre porque a instabilidade nas conexões e a presença de comunicação assíncrona inviabilizam a implementação da sincronicidade exigida pela manutenção das transações.

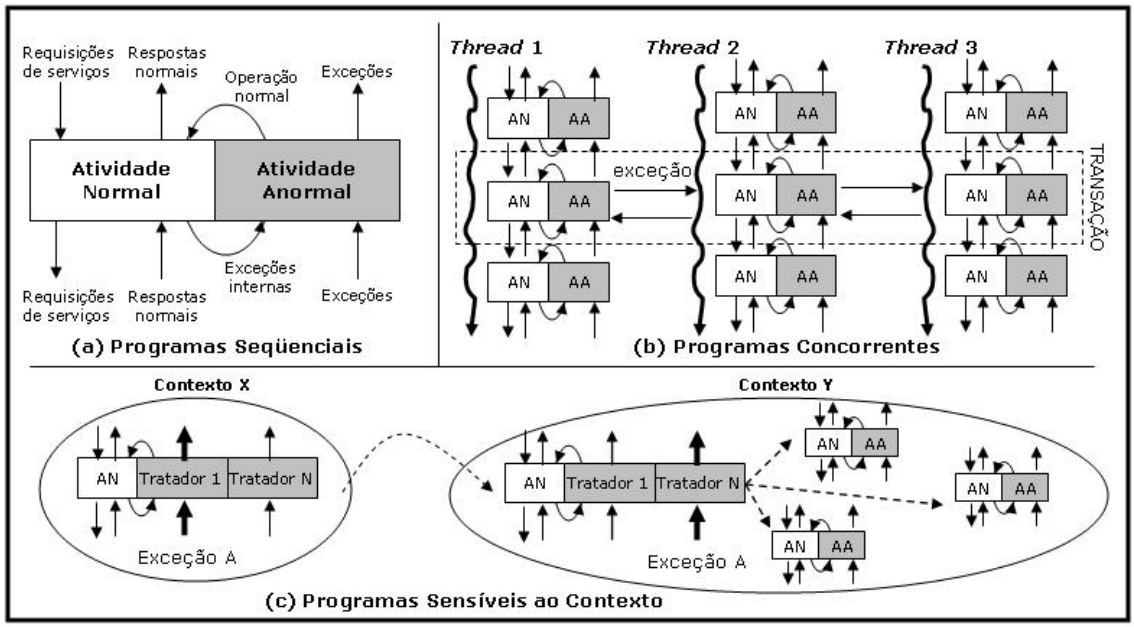


Figura 1. A evolução dos mecanismos de tratamento de exceções

Devido suas características de dinamicidade e mobilidade, as aplicações usualmente devem considerar os contextos onde estão atuando e quais usuários do sistema estão inseridos (Figura 1c). Assim como informações de contexto devem ser consideradas na execução das atividades normais do sistema, as variações de contextos também precisam ser consideradas nas atividades relacionadas ao tratamento de exceções. Além disso, diversas outras abstrações precisam ser consideradas, como contexto, localização, múltiplos tratadores, exceções dependentes de contexto, diferentes estratégias de propagação dependendo do contexto onde uma exceção é levantada, e assim por diante.

2.3. Exemplo de Aplicação Sensível ao Contexto

Virtual Lines é uma aplicação móvel sensível ao contexto cujo protótipo foi desenvolvido com a arquitetura MoCA (Sacramento et al., 2004). Esta aplicação realiza o controle de filas virtuais em parques de diversão e tem como principal objetivo impedir que as pessoas esperem muito tempo nas filas e aproveitem melhor as atrações existentes em um parque. Ao passar próximo a uma atração, um dispositivo móvel pode coletar um *ticket* virtual que corresponde a um lugar na fila. O sistema avisa ao usuário sobre a proximidade de sua vez para que ele retorne e participe da atração. Quando o usuário não retorna a tempo, o sistema emite um alerta avisando que perdeu a vez. Esta aplicação envolve contextos excepcionais que serão discutidos e analisados na próxima seção de tal forma a ilustrar as soluções de tratamento de exceções sendo propostas.

3. Modelo de Tratamento de Exceções Sensível ao Contexto

O nosso modelo de tratamento de exceções para aplicações móveis sensível ao contexto é constituído pelas características sendo apresentadas nas seções a seguir. Cada característica do modelo é apresentada da seguinte forma. Primeiro, discute-se a problemática envolvendo cada elemento do modelo, o qual é exemplificado com a

aplicação *Virtual Lines* (Seção 2.3). Quando apropriado, também argumenta-se por que soluções existentes (Seções 2.1 e 2.2), tais como *middleware publish-subscribe*, não provêem mecanismos adequados para cada um dos elementos apresentados.

3.1. Especificação de Contextos Excepcionais

De modo similar a definição de contexto proposta por Dey e Abowd (1999) (Seção 2.1), um "contexto excepcional" pode ser visto como um contexto que caracteriza uma situação excepcional de uma entidade, onde "excepcional" pode variar de acordo com os requisitos da aplicação. Um contexto excepcional corresponde então a um conjunto indesejável de condições que podem estar relacionadas a diferentes tipos de informação tais como uma região geográfica específica, a temperatura de uma sala e batimentos cardíacos de um paciente. O objetivo de definir um contexto excepcional é facilitar a definição de situações excepcionais, pois a ocorrência de um contexto excepcional está diretamente relacionada à ocorrência de uma exceção.

Entretanto, a especificação de uma situação de contexto excepcional possui uma semântica diferente e, como tal, precisa lidar com diferentes elementos em sua especificação, incluindo o escopo de tratamento, tratadores gerais alternativos, tipos de informação contextual que devem ou não ser propagadas junto com a ocorrência de exceção, e assim por diante. Para realizar o tratamento de contextos excepcionais é necessário prover suporte ao "fluxo de controle excepcional", que é inerentemente diferente do fluxo normal; este último consiste simplesmente em reações baseadas em notificação, enquanto o primeiro requer a propagação de contextos excepcionais aos tratadores apropriados, que também podem ou não ser selecionados dependendo de sua localização física. Por outro lado, a tarefa de propagação de um contexto excepcional pode exigir o envolvimento entre várias entidades. No caso da aplicação *Virtual Lines* (Seção 2.3), tais entidades podem ser o grupo de dispositivos registrado em uma atração, os dispositivos na mesma localização da atração defeituosa e a equipe responsável pelo suporte no parque. Denominamos tal envolvimento entre entidades como uma "colaboração excepcional".

O mecanismo *publish-subscribe* não provê suporte direto e efetivo para a implementação de um fluxo de controle excepcional sensível ao contexto. Para sistemas *publish-subscribe,* existe a necessidade de subscrições que devem registrar explicitamente o interesse em um ou mais contextos. Além disso, é importante que dispositivos recebam notificações de contextos excepcionais, independente de terem registrado interesse nesta informação. No caso da *Virtual Lines*, um contexto excepcional poderia especificar a situação em que uma atração deixa de funcionar por algum defeito ou manutenção urgente.

3.2. Níveis de Escopo: Dispositivo, Servidor, Regiões ou Grupos

Há muitas situações em que o tratamento de exceções requer que diversos dispositivos estejam envolvidos dependendo da região física e outros tipos de informação contextual. Por exemplo, para a aplicação *Virtual Lines* o tratamento dos contextos excepcionais que se referem a parada de uma atração pode envolver diferentes conjuntos de dispositivos, como o servidor do parque, os dispositivos presentes na região física onde se encontra a atração, o grupo responsável pela manutenção no parque, ou os dispositivos registrados na fila desta atração.

Além disso, deve ser possível que dependendo da gravidade de uma situação excepcional, a exceção seja propagada para todas as regiões e grupos envolvidos. Conseqüentemente, as regiões físicas ou um determinado grupo de dispositivos são também exemplos de escopos de tratamento contextual que devem ser suportados pelo *middleware* subjacente. Contudo, os sistemas de *middleware* não suportam tais escopos de tratamento de exceções sensível ao contexto, diminuindo a modularidade do sistema na presença de contextos excepcionais.

A fim de suportar uma abordagem sensível ao contexto para o tratamento adequado de erros, exceções podem ser capturadas através de escopos em quatro níveis diferentes: um dispositivo, um grupo de dispositivos, um servidor e uma região. Escopos de dispositivo e servidor compreendem as unidades operacionais básicas do sistema sensível ao contexto, o que permite que a funcionalidade do tratador de exceções seja encapsulada dentro do escopo do próprio dispositivo ou servidor. Tais escopos estão relacionados a abstrações próprias dos sistemas *publish-subscribe*.

Um escopo de grupo envolve um conjunto de dispositivos definidos pela aplicação para suportar o tratamento cooperativo de uma exceção entre os dispositivos que pertencem ao grupo. É possível inserir ou remover elementos do grupo de acordo com a necessidade da aplicação. Por exemplo, para a aplicação *Virtual Lines*, agentes que representam a equipe de manutenção podem formar um grupo específico, assim quando ocorrem exceções relacionadas a problemas em uma atração, todos os membros do grupo são notificados. Diferentemente dos outros três escopos, um escopo de região tem um comportamento mais dinâmico para identificar os dispositivos que estão participando deste escopo. Um escopo de região está diretamente relacionado a uma localização física do ambiente onde estão os dispositivos móveis.

3.3. Busca Sensível ao Contexto por Tratadores

Para ser sensível ao contexto, a busca dos tratadores apropriados para uma ocorrência de exceção deve ser realizada considerando os escopos e tratadores sensíveis ao contexto associados a exceção. Entretanto, tais elementos não são considerados pelos mecanismos convencionais, não sendo possível utilizar informações de contexto para realizar a busca. Por exemplo, dependendo das informações de contexto, a busca pode ser feita primeiramente entre os tratadores associados a regiões. Esta ordem pode ser previamente definida por uma aplicação, entretanto, se nenhuma ordem for explicitamente definida, o mecanismo de tratamento de exceções deve fornecer uma seqüência crescente pré-estabelecida para os diversos tipos de escopo, como exemplo, dispositivo < grupo < região < servidor. Esta seqüência estabelecida entre os diversos escopos é utilizada também para executar a propagação sensível ao contexto.

3.4. Tratadores Sensíveis ao Contexto

A seleção dos tratadores de exceções apropriados depende de condições de contexto dos dispositivos envolvidos no tratamento de exceções. Para a mesma exceção, é necessário criar tratadores específicos para diferentes condições de contexto e garantir que eles sejam corretamente executados. Na aplicação *Virtual Lines* por exemplo, para a situação excepcional que indica a parada de uma atração, pode ser necessário utilizar informações de contexto sobre a localização dos dispositivos e o tipo de problema ocorrido. No caso da parada na atração indicar uma situação excepcional do ambiente,

como fogo, os tratadores associados são executados nos dispositivos localizados na mesma região da atração. Novamente, temos que implementar tal controle de tratadores sensíveis ao contexto como parte da aplicação móvel, uma vez que não existe uma facilidade que dê suporte a este requisito nos sistemas de *middleware*.

Antes de disparar a execução de um tratador, é interessante que sejam feitas algumas verificações de contexto de acordo com as necessidades específicas da aplicação. Assim, de acordo com o contexto de execução, pode ser apropriado executar um tratador, ao passo que para a mesma exceção em outro contexto, o tratador não deva ser utilizado. Neste caso, a pesquisa pelo tratador apropriado deve continuar até encontrar outro tratador que satisfaça as condições de contexto. O propósito desta abordagem é promover flexibilidade extra que ofereça suporte a definição de tratadores sensíveis ao contexto. Além disso, após a execução dos tratadores, deve ser feita uma checagem para verificar a necessidade de realizar a propagação automática de exceções.

3.5. Propagação Sensível ao Contexto

A propagação é considerada sensível ao contexto, pois as informações de contexto e escopo são utilizadas para que o mecanismo decida quando deve ocorrer a propagação das exceções. A verificação da necessidade de realizar a propagação automática deve ser realizada após busca do tratador e sua posterior execução. Três situações podem ocorrer após a execução do tratador: (i) o tratador foi executado com sucesso e está associado a um escopo de execução local; (ii) o tratador foi executado com sucesso e está associado a um escopo de execução remota; (iii) o tratador não foi executado com sucesso. No primeiro caso não é necessário realizar a propagação automática. No segundo caso, após a execução local do tratador, a exceção deve ser propagada para o escopo associado a ele. Finalmente, no terceiro caso, deve ser realizada uma nova busca por tratadores que estejam associados a escopos no próximo nível de granularidade.

Além da propagação automática, o mecanismo pode permitir que a propagação seja solicitada explicitamente pelas aplicações sensíveis ao contexto.

3.6. Tratamento de Exceções Proativo

Em uma aplicação móvel aberta, não podemos esperar que todos os dispositivos, nos quais os agentes de software são desenvolvidos por diferentes projetistas, sejam capazes de prever todos os possíveis contextos excepcionais. No caso de *Virtual Lines*, por exemplo, durante a ocorrência de um incêndio em uma atração, se por alguma razão a equipe de manutenção do parque não estiver disponível, é útil que a exceção seja propagada para outros dispositivos realizarem o suporte emergencial, os quais podem ser selecionados a partir de informações de perfil dos usuários, como profissão, idade e sexo. Entretanto, estes usuários podem não ser capazes de tratar tal notificação. Adicionalmente, mesmo que os dispositivos em uma região não tenham registrado interesse em uma exceção, é fundamental notificar a ocorrência de uma exceção grave, aos dispositivos que pertencem a mesma região.

Em outras palavras, a exceção contextual deve ser proativamente levantada em outros agentes colaborativos e/ou dispositivos móveis pertencentes à mesma região ou escopo. Isto pode ser feito explorando a infra-estrutura móvel de colaboração fornecida pelo *middleware publish-subscribe* para desenvolver aplicações sensíveis ao contexto, na qual um dispositivo pode colaborar, por exemplo, com os dispositivos que estão na

mesma localização que ele. A proatividade precisa ser considerada sob dois pontos de vista (i) a notificação de exceções contextuais deve ser proativa: os dispositivos precisam tomar conhecimento da ocorrência de exceções mesmo que não tenham registrado interesse em obter informações sobre o contexto excepcional relacionado à exceção, e (ii) o tratamento de exceções precisa ser proativo: deve ser possível que os dispositivos colaborem para adquirir os possíveis tratadores para uma exceção contextual que eles não são capazes de tratar.

4. Arquitetura de Software para Tratamento de Exceções Contextuais

Esta seção apresenta uma arquitetura de software que objetiva modularizar os interesses de tratamento de exceções considerando a sensibilidade ao contexto. Neste sentido, os componentes da arquitetura separam os interesses de tratamento de exceções dos componentes correspondentes às funcionalidades básicas e aos outros interesses de aplicações sensíveis ao contexto. Vários componentes da arquitetura proposta são responsáveis por gerenciar o fluxo resultante da ocorrência de contextos excepcionais, não suportados comumente por mecanismos e sistemas de *middleware* existentes (Seção 3). As aplicações sensíveis ao contexto podem reusar facilidades fornecidas pelos componentes (Seção 4.1) e interfaces (Seção 4.2) da arquitetura a fim de tratar seus contextos excepcionais. Além disso, a arquitetura proposta é independente de linguagens de programação ou plataformas específicas, descrevendo um projeto em alto nível de abstração dos componentes, relacionamentos, responsabilidades e interfaces.

4.1. Componentes

A Figura 2 ilustra a arquitetura proposta para tratamento de exceções sensível ao contexto. Ela possui os seguintes componentes: (i) `ContextualException`, (ii) `Handler`, (iii) `ExceptionHandlingStrategy`, (iv) `PropagationManager`, (v) `ExceptionPropagation`, e (vi) `HandlingScope`. O componente `Middleware` é um componente externo que interage com os componentes da arquitetura de tratamento de exceções. Este componente representa a infra-estrutura *publish-subscribe* subjacente utilizada para recuperação das informações de contexto.

O componente `ContextualException` é responsável pela especificação de exceções, bem como gerenciamento das informações relacionadas, tais como nome, descrição, contexto excepcional, e assim por diante. Como esta arquitetura está relacionada à utilização do paradigma *publish-subscribe*, sempre que um contexto excepcional é associado com uma exceção, este componente possui a tarefa de subscrever interesse no recebimento de eventos com aquela informação. Existindo portanto uma relação direta entre este componente e o modelo de contexto adotado pelo *middleware* orientado a contexto. Além disso, este componente é responsável por manter atualizadas as informações sobre ocorrências de contextos excepcionais. Por exemplo, se um contexto excepcional está relacionado à entrada de dispositivos em uma região, uma informação útil associada a ocorrência deste contexto excepcional é qual dispositivo entrou na região especificada. Para manter estas informações atualizadas, este componente precisa interceptar todas as ocorrências de contextos excepcionais enviadas pelo *middleware*. Este componente também possui a tarefa de levantar as exceções contextuais, tão logo sejam sinalizadas as ocorrências de suas condições de contexto, ou quando solicitadas pelo componente de propagação ou aplicação.

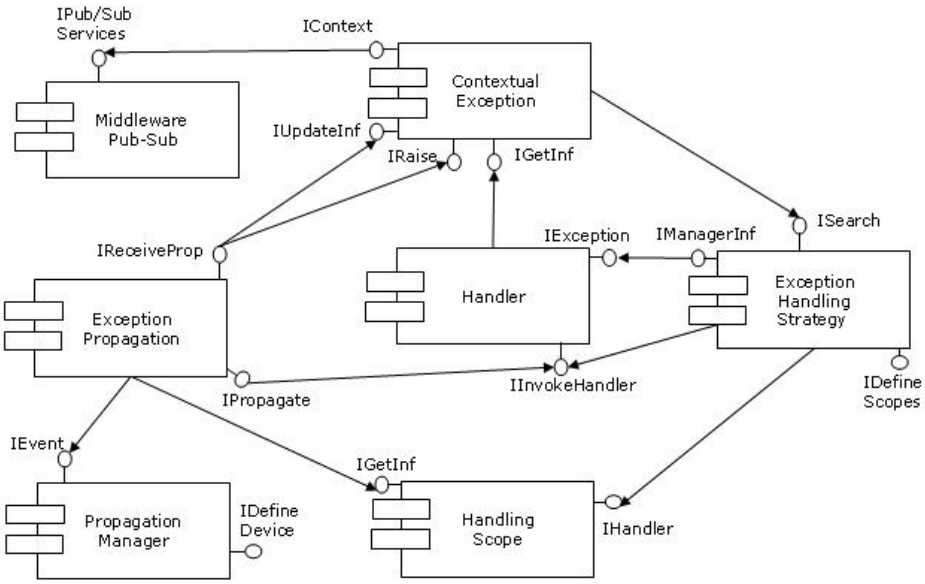


Figura 2. Arquitetura para Tratamento de Exceções Sensível ao Contexto

O componente `Handler` é responsável por especificar os tratadores e realizar as verificações de condições de contexto, necessárias para realizar um tratamento sensível ao contexto. Para isto, solicita as informações de contexto ao componente `ContextualException`. Além disso, este componente possui a tarefa de chamar os tratadores quando suas condições de contexto são satisfeitas. O componente `PropagationManager` tem a função de manter uma infra-estrutura *publish-subscribe* para permitir a propagação das exceções. Este componente permite informar se a aplicação executará o papel consumidor ou publicador de eventos, de acordo com a API do *middleware publish-subscribe* adotado.

Um componente diretamente relacionado a este é o componente `ExceptionPropagation`, que envia solicitações para realizar a propagação de exceções, seguindo a especificação do componente gerenciador. Este componente é responsável também pelo recebimento de exceções propagadas por outros dispositivos, recuperação local dos tratadores associados com a ocorrência de exceção, além de solicitar o levantamento da exceção recebida ao componente `ContextualException`. Além disso, este componente possibilita que seja realizada a propagação das exceções, assim que o tratador terminar sua execução. Considerando esta execução, existem duas possibilidades de realizar a propagação: (i) no caso de insucesso, repetição da busca por escopos de nível superior ou (ii) quando a execução é bem sucedida, verificação do escopo e propagação automática para o escopo especificado.

O componente `ExceptionHandlingStrategy` é responsável por realizar o gerenciamento das exceções e seus respectivos tratadores, mantendo um controle geral desta informação, o qual pode ser acessado por todos os outros componentes. A possibilidade de múltiplos tratadores associados a uma única exceção permite a associação dinâmica entre uma ocorrência de exceção e seu respectivo tratador, portanto é fundamental para aplicações sensíveis ao contexto. Além disso, este componente permite a especificação de uma estratégia de busca de tratadores e uma ordem de prioridade entre os tipos de escopo da aplicação. Esta seqüência de escopos será utilizada tanto na busca de tratadores, como na propagação das exceções. Este

componente tem a função de realizar as buscas de tratadores, considerando suas informações de contexto. É importante a existência de estratégias gerais, usadas no caso de nenhuma estratégia ser especificada pela aplicação. Após realizar a busca de tratadores, este componente solicita ao componente `Handler` a execução dos tratadores. Finalmente, o componente `HandlingScope` permite especificar os escopos de tratamento do mecanismo, gerenciar os tratadores associados aos escopos e recuperar dinamicamente quais os dispositivos que estão associados a um escopo.

4.2. Interfaces

As interfaces dos componentes da arquitetura podem ser utilizadas por outros componentes da arquitetura ou diretamente por aplicações que utilizem o mecanismo de tratamento de exceções. As interfaces podem ser: (i) privadas, definem serviços visíveis somente para os componentes da arquitetura; e (ii) públicas, definem serviços visíveis tanto pela arquitetura quanto pelas aplicações. A Figura 3 apresenta as interfaces publicas e privadas para cada um dos componentes da arquitetura.

Na Figura 3a o componente `ContextualException` implementa três interfaces públicas e uma privada. A interface `IRaise` possibilita que a aplicação levante e propague as exceções diretamente através dos métodos `raise` e `propagateTo`. A interface `IGetInf` permite que a aplicação e os outros componentes da arquitetura obtenham informações sobre as ocorrências de exceções e seus contextos excepcionais e a interface `IUpdateInf` permite que sejam feitas atualizações nestas informações. A interface `IContext` subscreve o interesse em receber as informações ao *middleware publish-subscribe* e recebe os contextos excepcionais enviados pelo *middleware*.

A Figura 3b ilustra as interfaces do componente `HandlingScope`. A interface pública `IHandler` permite que a aplicação associe seus tratadores com um dado escopo de tratamento e recupere todos os tratadores existentes para um dado escopo. A interface privada `IGetInf` permite recuperar dinamicamente, através do *middleware* orientado a contexto, os dispositivos que estão relacionados a um dado contexto. Já a Figura 3c ilustra as interfaces para o componente `Handler`. A interface `IException` permite a manutenção das informações sobre a exceção que está associada a um tratador. A interface `IInvokeHandler` permite que o componente `ExceptionHandlingStrategy` verifique se as condições de contexto dos tratadores são satisfeitas quando estiver realizando a busca de tratadores, além de disparar a execução dos tratadores.

A Figura 3d apresenta o componente `ExceptionHandlingStrategy`. A interface pública `IDefineScopes` permite que a aplicação especifique a seqüência desejada dos escopos que deve ser utilizada na busca de tratadores. A interface `ISearche` permite realizar a busca de tratadores. A interface `IManagerInf` permite gerenciar uma tabela de referências com a relação existente entre uma exceção e seus possíveis tratadores. Isto é feito sempre que ocorre a definição de uma nova exceção e a associação de uma exceção a um tratador. Depois que uma exceção contextual é levantada, os componentes da arquitetura interagem para realizar as atividades de gerenciamento. A informação extra sobre uma ocorrência da exceção é atualizada implicitamente pelos componentes da arquitetura, entretanto a aplicação também pode adicionar outras informações de acordo com sua necessidade.

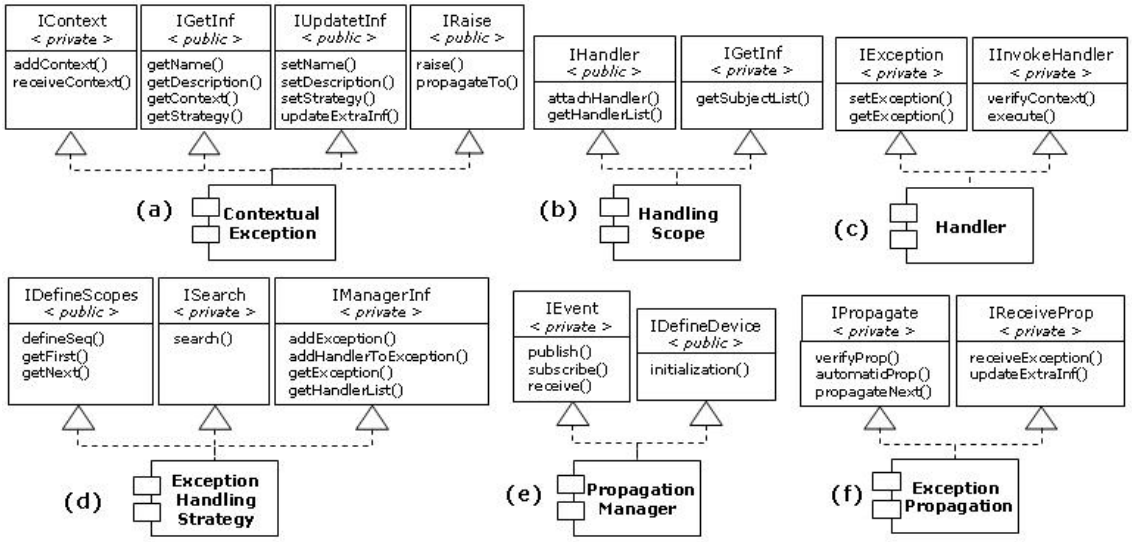


Figura 3. Interfaces dos Componentes da Arquitetura

A Figura 3e apresenta o projeto do componente `PropagationManager`. A interface privada `IEvent` é responsável por fornecer eventos com as exceções que foram propagadas por outros dispositivos e permitir a publicação de exceções. A interface pública `IDefineDevice` permite que a aplicação especifique e inicie o papel desempenhado pela aplicação na estrutura de propagação. Finalmente, a Figura 3f apresenta o projeto do componente `ExceptionPropagation`. Sempre que um tratador termina sua execução, a interface `IPropagate` verifica a necessidade de realizar a propagação automática para o escopo associado ao tratador, ou a propagação para um escopo de maior nível de granularidade, por este motivo esta interface entrecorta o fim da execução dos tratadores. A interface `IReceiveProp` permite que, após o recebimento de uma exceção via propagação, sejam atualizadas as informações de contexto necessárias para realizar o tratamento. Após a atualização destas informações, a exceção é levantada, para que então seja tratada pelo mecanismo.

5. Avaliação do Mecanismo

Esta seção apresenta a avaliação do modelo e arquitetura propostos. Devido a restrições de espaço, este artigo descreve mais detalhadamente apenas o estudo de caso baseado na aplicação *Virtual Lines* (Seção 2.3). Entretanto, outros dois estudos foram realizados, utilizando uma aplicação de *Health Care*, para suporte ao monitoramento de pacientes com doença cardiovascular sob cuidados médicos (Cacho et al., 2006) e uma aplicação de ambiente inteligente (*AmI*), um ambiente inteligente com diversos sensores para controle de temperatura, alarme de incêndio, entre outros (Damasceno et al., 2006).

5.1. Estudos de Caso

Para o estudo de caso da aplicação *Virtual Lines*, foram definidas as exceções `DeviceNotFoundException` e `AttractionOffException`. Cada uma destas exceções possui diferentes tratadores, os quais devem ser executados de acordo com suas próprias condições de contexto. Por exemplo, a exceção `DeviceNotFound Exception` ocorre quando um dispositivo móvel que deveria estar no parque não é encontrado. A subscrição de contexto excepcional que representa esta exceção contém a expressão

`((Online = false) and (DeltaT > 50000))`, que é enviada ao MoCA sempre que um novo dispositivo móvel registra sua entrada no parque. Para o domínio da aplicação *Virtual Lines*, é fundamental ter conhecimento sobre os usuários que ainda estão fisicamente no parque. Sem a existência desta exceção, mesmo após sua saída, os usuários podem ainda estar ocupando lugares nas filas, o que ocasiona uma espera desnecessária aos outros clientes presentes.

No tratamento da exceção `DeviceNotFoundException`, as condições de contexto verificadas pelos tratadores são: (i) o número de vezes que o usuário não compareceu a uma atração, mesmo após receber notificações sobre sua vez nas filas, e (ii) a existência de algum registro de problemas com o dispositivo. O tratador `ExitedUserHandler` verifica se o usuário está "desaparecido" há mais de duas atrações. Neste caso, o tratador é executado quando o mecanismo assume que o usuário saiu efetivamente do parque e procede cancelando a reserva do usuário em todas as filas nas quais está cadastrado. Por outro lado, o tratador `OffUserHandler` é executado quando o mecanismo assume que o usuário está com problemas no dispositivo móvel, mas ainda permanece no parque. Este tratador modifica o *status* do usuário a fim de assegurar que ele poderá participar das atrações mesmo depois de ter perdido sua vez nas filas. Ambos os tratadores sensíveis ao contexto estão associados ao escopo local, isto é, ao gerenciador das filas, não sendo necessário realizar a propagação da exceção.

A exceção `AttractionOffException` ocorre sempre que uma dada atração deixou de funcionar por algum defeito ou para sofrer algum tipo de manutenção urgente. Embora seja levantada diretamente pelo servidor no papel de gerenciador das filas, a exceção `AttractionOffException` também deve ser tratada pelos clientes nos dispositivos móveis, os quais podem receber sua ocorrência via propagação. O gerenciador das filas especifica dois tratadores para a exceção `AttractionOffException`: `NotifyRegionHandler` e `NotifyGroupHandler`. Estes tratadores utilizam diferentes condições de contexto para propagar a exceção aos clientes na região da atração com defeito ou para um grupo de dispositivos que possui reserva para a atração. Dependendo das condições de contexto, ambos os tratadores do gerenciador das filas podem ser executados. Por outro lado, ao receber a propagação da exceção, os dispositivos móveis da região ou grupo podem executar três tratamentos distintos, selecionados de acordo com a busca e as condições de contexto locais. O tratador `NotifyHandler`, por exemplo, é executado quando o usuário não possui uma reserva para esta atração e procede apenas informando o usuário do problema bem como a previsão de retorno da atração.

Para o segundo estudo de caso, uma aplicação para monitoramento de pacientes chamada *Health Care*, foram definidos como contextos excepcionais possíveis situações relacionadas a um ataque cardíaco. Por exemplo, a inexistência de familiares no mesmo local onde está um paciente pode implicar na necessidade de propagar a exceção para outros dispositivos localizados na mesma região. Além disso, caso não seja possível enviar a equipe de emergência ao local, pode ser necessário solicitar auxílio aos dispositivos móveis de profissionais da saúde localizados em regiões próximas ao paciente. No terceiro estudo, a aplicação AmI, a temperatura de uma região ultrapassando um valor máximo ou mínimo pode indicar uma situação excepcional; adicionalmente, se outras informações de contexto forem consideradas, tais como, a temperatura de outras regiões ou a quantidade de dispositivos, a exceção UnableToHeat pode ser propagada para a equipe de manutenção dos equipamentos.

A avaliação dos estudos levanta questões como a distinção entre fluxo normal e fluxo excepcional de eventos. O fluxo normal é representado pelo envio e recebimento de informações de contexto, enquanto o fluxo excepcional é realizado pela propagação de exceções que foram detectadas a partir das informações de contexto. Esta distinção permite atribuir prioridade de execução para o fluxo excepcional em detrimento ao normal, já que uma vez detectada, a exceção deve ser imediatamente tratada. Um segundo fator trata da possibilidade de se modificar o fluxo excepcional, uma vez que para isso é necessário apenas definir novas estratégias de propagação (Seção 4) que permitam adaptar o mecanismo aos requisitos da aplicação alvo.

5.2. Questões de Implementação, Benefícios e Limitações do Mecanismo Proposto

O modelo e a arquitetura propostos foram implementados usando a arquitetura MoCA (Sacramento et al., 2004). Detalhes de implementação encontram-se descritos em (Damasceno, 2006). Consideramos que a utilização do paradigma *publish-subscribe* e do *middleware* MoCA se mostrou como uma boa alternativa para a implementação do mecanismo de tratamento de exceções. Entretanto, pode ser interessante utilizar outros paradigmas e sistemas de *middleware* especialmente para ampliar o modelo de tratamento de exceções. Para implementar alguns componentes do mecanismo de exceções foi feito uso de programação orientada a aspectos (Kiczales et al., 1997), o que permitiu uma clara separação entre os interesses excepcionais e os outros interesses das aplicações sensíveis ao contexto. Para o estudo de caso da aplicação *Virtual Lines*, as classes e aspectos relacionados ao tratamento de exceções ficaram totalmente separados das outras classes da aplicação.

Como pode ser abstraído da seção anterior e de outros estudos de caso que realizamos (Cacho et al., 2006; Damasceno, 2006; Damasceno et al., 2006), o mecanismo proposto (Seções 3 e 4) alivia o programador de aplicações móveis de uma série de complexidades inerentes a tratamento de erros em tais aplicações. Por exemplo, o programador da aplicação não tem necessidade de implementar toda a complexidade para (i) utilização de diferentes estratégias de propagação de erros, podendo apenas definir a prioridade entre os níveis de escopo ou selecionar uma estratégia pré-definida pelo mecanismo; bem como para (ii) realizar a notificação de contextos excepcionais. Por outro lado, percebemos a necessidade da evolução do mecanismo de exceções (Seção 3) para prover suporte à resolução concorrente de condições excepcionais.

6. Trabalhos Relacionados

Não existem trabalhos na literatura que consideram os requisitos de sensibilidade ao contexto no tratamento de exceções de aplicações móveis. Existem alguns trabalhos que consideram questões gerais relacionadas ao tratamento de exceções em aplicações móveis baseadas em agentes. Entretanto, os mecanismos propostos restringem-se a características específicas de propriedades como mobilidade e autonomia de agentes. Por exemplo, na abordagem de Tripathi e Miller (2000), exceções são propagadas para agentes especiais denominados “guardiões”, os quais implementam reações gerais para as exceções. Esta solução é extremamente restritiva tendo em vista que a maior parte do código de tratamento de exceções em sistemas reais é específico de aplicação (Garcia et al., 2001). Além do tratamento não ser sensível ao contexto, cria-se um gargalo pela sua centralização em um único agente.

A abordagem de Souchon et al. (2004) não possibilita a associação dinâmica entre tratadores e ocorrências de exceção, fundamental para o tratamento de exceções no domínio de sistemas com mobilidade e sensibilidade a contexto. Em Iliasov e Romanovsky (2005), Context-Aware Mobile Agents (CAMA) é um *framework* para o desenvolvimento de aplicações móveis que suporta o conceito de escopos aninhados, que agrupam erros e os tratadores de exceção aos quais estão associados. Contudo, não existe suporte ao tratamento sensível ao contexto. Por exemplo, CAMA não trata a definição de contextos excepcionais e a busca de tratadores sensível ao contexto. Finalmente, alguns de nossos trabalhos nesta linha (Damasceno et al., 2006; Cacho et al., 2006) apresentaram apenas os estudos de caso exploratórios. Este artigo, do contrário, apresenta um modelo e uma arquitetura genéricos para Tratamento de Exceções Sensível ao Contexto.

7. Conclusões e Trabalhos Futuros

Mecanismos especializados de tratamento de exceções tem sido consistentemente uma questão central em Engenharia de Software de tal forma a promover melhor confiabilidade no desenvolvimento de sistemas de software (IEEE TSE, 2000). Este trabalho motivou a necessidade para mecanismos de tratamento de exceções sensível ao contexto que foi detectada: (i) no desenvolvimento de uma série de aplicações com o *middleware* MoCA (Seção 5), (ii) em uma análise extensiva dos vários sistemas de *middleware* existentes (Damasceno, 2006), e (iii) uma avaliação das soluções existentes de tratamento de erros em aplicações móveis (Seção 6). Isto nos permitiu obter um conjunto de requisitos e definir um modelo de tratamento sensível ao contexto com: (i) suporte explícito para especificação de "contextos excepcionais", (ii) busca sensível ao contexto por tratadores de exceção, (iii) escopo de tratamento multi-nível que fornece novas abstrações (tais como grupos), e abstrações relacionadas ao *middleware* sensível ao contexto subjacente, como dispositivos, regiões, e servidores, (iv) propagação de erros sensível ao contexto e (v) tratamento de exceções proativo.

Como trabalho futuro, pretendemos evoluir o mecanismo de exceções proposto, provendo suporte à resolução de exceções contextuais concorrentes e tratando das limitações identificadas na Seção 5. Além disso, estamos planejando avaliar a generalidade do modelo proposto (Seção 3) a partir de aplicações móveis construídas com outros paradigmas de coordenação (Seção 2.1), como o paradigma baseado em espaços de tuplas. Finalmente, pretendemos estender o mecanismo de tratamento de exceções para suportar mobilidade de código em adição a mobilidade física.

Referências Bibliográficas

Cacho, N. et al. Handling Exceptional Conditions in Mobile Collaborative Applications: An Exploratory Case Study. 4th Intl. Workshop on Distributed and Mobile Collaboration, Manchester, Jun 2006.

Capra, L. et al. CARISMA: Context-Aware Reflexive mIddleware System for Mobile Applications. IEEE Transactions on Software Engineering, v.29, n.10, Oct. 2003, p. 929-945.

Coutaz, J. et al. Context is Key. Communications of ACM. v.48, n.3, Mar. 2005, p. 49-53.

Cugola, G.; Cote, J. E. M. On Introducing Location Awareness in Publish-Subscribe Middleware. 4th International Workshop on Distributed Event-Based Systems, 2005.

Damasceno, K. et al. Context-Aware Exception Handling in Mobile Agent Systems: The MoCA Case. 5th International Workshop on Software Engineering for Large-Scale Multi-Agent Systems (SELMAS 2006) at ICSE 2006, Shanghai, China, May.

Damasceno, K. Tratamento de Exceções Sensível ao Contexto. Março 2006. Dissertação de Mestrado, Departamento de Informática, PUC-Rio, Rio de Janeiro, Brasil.

Dey, A. K.; Abowd, G. D. Towards a better understanding of context and context-awareness. Conf on Human Factors in Computing Systems, Netherlands, Apr. 2000.

Garcia, A. et al. A Comparative Study of Exception Handling Mechanisms for Building Dependable Object-Oriented Software. Journal of Systems and Software, Elsevier, v.59, n.6, Nov. 2001, p.197-222.

Garcia, A. et al. Software Engineering for Large-Scale Multi-Agent Systems. SELMAS 2005. (Post-Workshop Report) ACM Software Engineering Notes, v. 30.

Goodenough, J. B. Exception handling: issues and a proposed notation. Commun ACM v. 18, n.12, Dec. 1975, p. 683-696.

IEEE TSE (IEEE Transactions on Software Engineering), Special Issue on Current Trends in Exception Handling, v. 26, n. 9, Sep. 2000.

Iliasov, A.; Romanovsky, A. CAMA: Structured Communication Space and Exception Propagation Mechanism for Mobile Agents. ECOOP-EHWS, Glasgow, Jul. 2005.

Kiczales, G. et al. Aspect-Oriented Programming. European Conference on Object-Oriented Programming (ECOOP), LNCS 1241, Springer, Finland, Jun. 1997.

Lee, P.; Anderson, T. Fault Tolerance: Principles & Practice. Springer, 2nd ed, Wien, Austria, Jan. 1990.

Meier, R.; Cahill, V. STEAM: Event-Based Middleware for Wireless Ad Hoc Networks. Intl. Workshop on Distributed Event-Based Systems. Austria, 2002.

MQTT. http://mqtt.org/

Muthusamy, V. et al. Publisher Mobility in Distributed Publish/Subscribe Systems. In Proc. 4th Intl. Workshop on Distributed Event-Based Systems (ICDCSW'05), 2005.

Parnas, D.; Würges, H. Response to Undesired Events in Software Systems. In Proc. 2nd Intl. Conference on Software Engineering. California, USA, p. 437-446, 1976.

Pietzuch, P.; Bacon, J. Hermes: A Distributed Event-Based Middleware Architecture. In Proc. Workshop on Distributed Event-Based Systems, 2002.

Sacramento, V. et al. MoCA: A Middleware for Developing Collaborative Aplications for Mobile Users. IEEE Distributed Systems Online, v.5, n.10, Oct. 2004.

Souchon, F. et al. Improving exception handling in multi-agent systems. In Software engineering for multi-agent systems II, Springer-Verlag, LNCS 2940, Feb. 2004.

Tripathi, A.; Miller, R. Exception Handling in Agent-Oriented Systems. In Proc. Advances in Exception Handling Techniques (ECOOPW'00), Springer-Verlag, LNCS 2022, 2000.

Xu, J. et al. Fault Tolerance in Concurrent Object-Oriented Software through Coordinated Error Recovery. In Proc. 25th FTCS, 1995.

Certificação da Utilização de Padrões de Projeto no Desenvolvimento Orientado a Modelos

Maria Cristina F. Gomes[1, 2], Maria Luiza M. Campos[1], Paulo F. Pires[1, 3], Linair M. Campos[4]

[1]Programa de Pós-graduação em Informática do IM/NCE - Universidade Federal do Rio de Janeiro

[2] Petróleo Brasileiro S/A – PETROBRAS

[3] Universidade Federal do Rio Grande do Norte

[4] Núcleo de Computação Eletrônica (NCE) – Universidade Federal do Rio de Janeiro

mariacfg@posgrad.nce.ufrj.br,{mluiza,paulopires,linair}@nce.ufrj.br

Abstract. *This research presents a design patterns use certification environment at all abstraction levels of business components model driven development. The solution is based on patterns representation through specific Unified Modeling Language (UML) profiles, on Meta Object Facility (MOF) compliant patterns repository and patterns manipulation through Java Metadata Interface (JMI).*

Resumo. *Este trabalho apresenta um ambiente de certificação do uso de padrões de projeto em todos os níveis de abstração do desenvolvimento de componentes de negócio orientado a modelos. A solução se baseia na representação de padrões através de perfis específicos da Unified Modeling Language (UML), num repositório de padrões compatível com Meta Object Facility (MOF), e na manipulação desses padrões através do Java Metadata Interface (JMI).*

1. Introdução

O modelo de desenvolvimento baseado em componentes (DBC) é o resultado do investimento constante na proposição de novos métodos de trabalho e tem correspondência direta com a evolução dos níveis de abstração no desenvolvimento de software. Essa evolução passou pelas linguagens de programação estruturada com vários níveis de modulação, pela teoria e a prática da orientação a objetos, pelas tecnologias de objetos distribuídos como CORBA e a linguagem Java e pelas tecnologias específicas para o modelo DBC, como CORBA 3.0 e EJB [Herzum e Sims 1999].

Recentemente, com a abordagem de desenvolvimento orientado a modelos, pretende-se melhorar a portabilidade e interoperabilidade de sistemas, através de representações em um nível mais alto de abstração e de forma independente de soluções tecnológicas específicas. Dessas representações, através de transformações sucessivas, seria possível automatizar a geração de representações para plataformas específicas,

permitindo que o reuso seja possível nos diferentes níveis do processo [Mellor 2004]. A Arquitetura Baseada em Modelos (*Model Driven Architecture* - MDA) é uma proposta do *Object Management Group* (OMG), que enfatiza a utilização de modelos no desenvolvimento de sistemas de software.

Padrões têm um papel fundamental nos processos de desenvolvimento baseados na MDA[1]. As transformações dos modelos, nos diversos níveis de abstração até a implementação do código, necessitam que esses modelos contenham detalhe suficiente para direcionar um aplicativo de software através de todo o processo. Esses detalhes podem ser incorporados através do uso de padrões, permitindo maior automação das transformações, trazendo enorme ganho de produtividade e qualidade. Por outro lado, no modelo DBC os componentes de software possuem responsabilidades de execução de serviços de acordo com a arquitetura tecnológica utilizada. Esses diversos componentes se inter-relacionam, e para que isso aconteça de forma eficiente e coesa, é necessário seguir as melhores práticas já testadas e consolidadas, o que significa a utilização de padrões de projeto adequados. Dessa forma conclui-se que garantir a utilização correta de padrões é um aspecto do desenvolvimento de software que merece bastante atenção e destaque, tanto no modelo DBC quanto no ambiente orientado a modelos e conseqüentemente em ambientes de desenvolvimento baseado em componentes orientado a modelos.

Esse trabalho se propõe a suprir a ausência de um ambiente de certificação aberto e flexível, definindo um processo e um sistema computacional para automatizar a validação dos modelos criados quanto à utilização correta de padrões de projeto envolvidos em todas as fases do ciclo de desenvolvimento. A abordagem proposta cobre tanto os padrões de projeto independentes de plataforma empregados nas fases de requisitos e especificação, quanto os padrões dependentes de plataforma, compatíveis com a tecnologia de componentes utilizada. Na fase de implementação o sistema computacional é específico para a plataforma *Java 2 Enterprise Edition* (J2EE), podendo, no entanto, ser estendido para outros tipos de plataformas apenas com a implementação de novos serviços de reconstrução de arquitetura específicos para essas plataformas e linguagens. Foi realizada uma validação preliminar do sistema computacional desenvolvido, em ambiente controlado, que serviu para avaliar o corretismo das suas funcionalidades e pretende-se conduzir uma validação mais detalhada em ambiente real.

O trabalho encontra-se estruturado em 5 seções. A seção 2 apresenta uma introdução à abordagem da MDA e de sua relação com a utilização e certificação de padrões de projeto. Na seção 3 discutimos o ambiente de certificação proposto com as características e especificação do sistema computacional desenvolvido. Na seção 4 apresentamos um cenário de utilização desse ambiente. Na seção 5, comparamos o ambiente de certificação proposto com trabalhos relacionados e apresentamos as conclusões da pesquisa, com suas contribuições, além de possíveis trabalhos futuros.

[1] **http://www.omg.org/mda/faq_mda.htm**

2. MDA, Componentes e Certificação de Padrões de Projeto

2.1. A Abordagem da MDA

A MDA define uma abordagem para o desenvolvimento de sistemas que utiliza a especificação das funcionalidades do sistema em um modelo de alto nível, para, através de transformações sucessivas, chegar à geração da implementação dessas funcionalidades em uma plataforma tecnológica específica. Para isso a MDA propõe uma arquitetura para modelagem que inclui um conjunto de diretrizes para especificações estruturadas e formais, expressas na forma de modelos [MDA 2003].

Modelos consistem de um conjunto de elementos que descrevem algo físico, abstrato ou uma realidade hipotética. Bons modelos servem como meio de comunicação de conhecimento, projetos e idéias. O conceito central da MDA é a criação de diferentes modelos em diferentes níveis de abstração e a interligação desses modelos através de transformações até a geração do código de implementação [Mellor 2004].

Nos processos de desenvolvimento tradicionais existe uma separação bem definida nos papéis dos modelos e das linguagens de programação, onde os modelos são apenas utilizados como artefatos de projeto. Na MDA os modelos fazem parte de forma direta do processo de produção, são altamente formalizados e adquirem características de artefatos de desenvolvimento [Frankel 2003]. No processo da MDA existem três etapas principais. Na primeira etapa construímos um modelo independente de plataforma (PIM). Na segunda etapa um PIM é transformado em um ou mais modelos de plataforma específica (PSM), dependendo das plataformas tecnológicas desejadas. A etapa final do desenvolvimento é a transformação de cada PSM em código.

Nesse contexto, onde diferentes tipos de modelos são manipulados em cada etapa do processo de desenvolvimento, torna-se necessário um mecanismo comum para definição de linguagens de modelagem, chamado genericamente de metamodelagem.

As camadas de metamodelagem da estrutura da MDA são baseadas em uma arquitetura de quatro camadas [Mellor 2004], e os padrões da OMG que possuem papéis na MDA também se relacionam de acordo com essa arquitetura. O *Meta Object Facility* (MOF) reside na camada M3, a mais alta da arquitetura de metadados. É por definição, o metametamodelo comum para especificação de metamodelos da OMG. É um padrão que especifica uma linguagem abstrata para definir linguagens de modelagem, como a *Unified Modeling Language* (UML), o *Common Warehouse Metamodel* (CWM), que residem na camada M2 da arquitetura de metadados, assim como para definir o próprio MOF [Poole 2001].

Para se definir uma linguagem de modelagem específica, podemos estender uma linguagem de modelagem existente (um metamodelo existente, como por exemplo, o metamodelo da UML) ou criar uma nova (um novo metamodelo). O conceito de perfil é um mecanismo de extensão definido como parte da UML, que estabelece uma forma específica de usar a UML. A MDA se utiliza muito desse mecanismo de extensão da UML, porque necessita dar suporte a diferentes níveis de abstração [Frankel 2003]. Um perfil define uma nova linguagem simplesmente reutilizando o metamodelo da UML. Um perfil é definido por um conjunto de estereótipos, um conjunto relacionado de restrições em *Object Constraint Language* (OCL) e um conjunto de etiquetas valoradas

(*tagged values*). Outra maneira de estender o metamodelo da UML é através do MOF, aconselhável em extensões mais complexas.

2.2. A Utilização de Padrões de Projeto na MDA

A MDA unificou o paradigma do desenvolvimento baseado em padrões com a construção de modelos [Blankers 2003] e enfatizou a utilização de padrões e sua transformação através dos diferentes níveis de abstração existentes (figura 1).

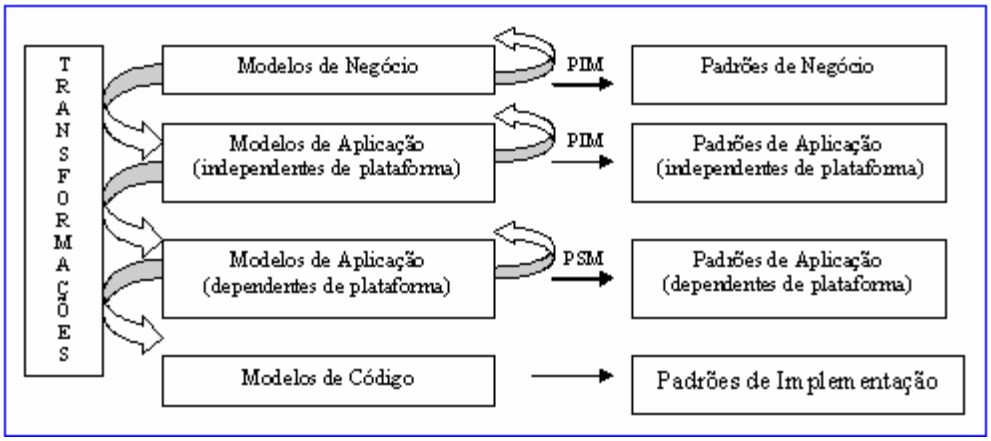


Figura 1. A utilização de padrões nos diversos níveis de abstração da MDA

Os padrões independentes de plataforma se aplicam em níveis de abstração em que ainda não estamos preocupados com a plataforma tecnológica que será utilizada. No nível dos modelos de negócio, Arlow e Neustad (2003) apresentaram uma proposta de utilização de padrões corporativos por eles chamada de padrões de arquétipos, com a descrição de alguns exemplos de padrões de arquétipos para representação de *Customer Relationship Management* (CRM), de Produto, de Inventário, de ordem (venda ou compra), de quantidade, de dinheiro, de regra, dentre outros. No nível de padrões de aplicação, utilizados no projeto lógico de uma aplicação ou componente de software, podem-se citar, entre outros, os padrões conhecidos como GoF [Gamma *et al* 1995]. Os padrões dependentes de plataforma são padrões a serem aplicados numa plataforma tecnológica específica. Cada plataforma tecnológica possui seu catálogo próprio de padrões. Esses padrões são utilizados para construções de componentes e aplicações robustas. Na plataforma J2EE temos o catálogo conhecido como Core J2EE Patterns[1].

2.3. A Certificação de Padrões de Projeto na MDA

No desenvolvimento orientado a modelos e baseado em padrões da MDA, existem padrões específicos para serem aplicados em cada nível de abstração. A formação desses níveis pode se dar de diversas formas, não existindo uma quantidade fixa de modelos de negócio e modelos de aplicação (independentes e dependentes de plataforma). A distribuição em níveis é determinada pelas características específicas de cada ambiente de desenvolvimento. A certificação da utilização de padrões nesse ambiente também necessita ser dividida em níveis, com etapas de certificação específicas para cada nível de abstração existente em um determinado processo de desenvolvimento. Cada etapa de

[1] http://java.sun.com/blueprints/corej2eepatterns/Patterns/

certificação garante a transformação correta para o próximo nível de abstração, até a implementação do código do componente que está sendo desenvolvido (figura 2).

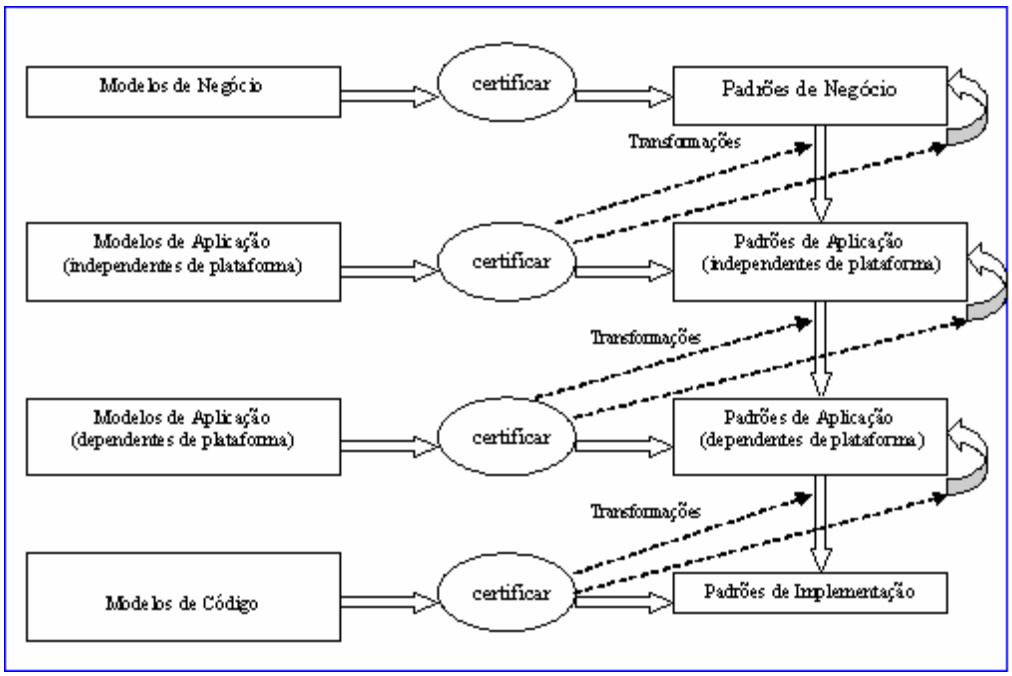


Figura 2. As certificações realizadas nos diversos níveis de abstração.

3. Ambiente de Certificação de Padrões de Projeto

Conforme descrito na seção anterior, no desenvolvimento orientado a modelos associado a uma abordagem como a da MDA, a certificação da utilização de padrões precisa ser executada em todas as etapas de transformação existentes no processo de desenvolvimento. Para atender esse requisito, esse trabalho apresenta uma proposta de um ambiente de certificação flexível, podendo ser utilizado em abordagens que contemplem diversos níveis de abstração.

3.1. Atividades e Papéis no Ambiente de Certificação

Através de pesquisas realizadas sobre certificação e sobre utilização de padrões junto a diversos projetos e na literatura, e com o levantamento das necessidades do ambiente de certificação proposto, foram identificadas as principais atividades a serem contempladas. Essas atividades foram classificadas segundo três macro-atividades, conforme representado na figura 3. Essas macro-atividades são:

- Administração de Repositório: contempla a manutenção do repositório dos padrões utilizados no ambiente de desenvolvimento;
- Reconstrução de Arquitetura: inclui a recuperação dos modelos de implementação a partir do código fonte dos componentes e aplicações;
- Análise de Modelos: envolve a comparação dos modelos dos componentes e aplicações com os modelos dos padrões utilizados e armazenados no repositório de padrões, cobrindo todos os níveis de abstração existentes no ciclo de desenvolvimento.

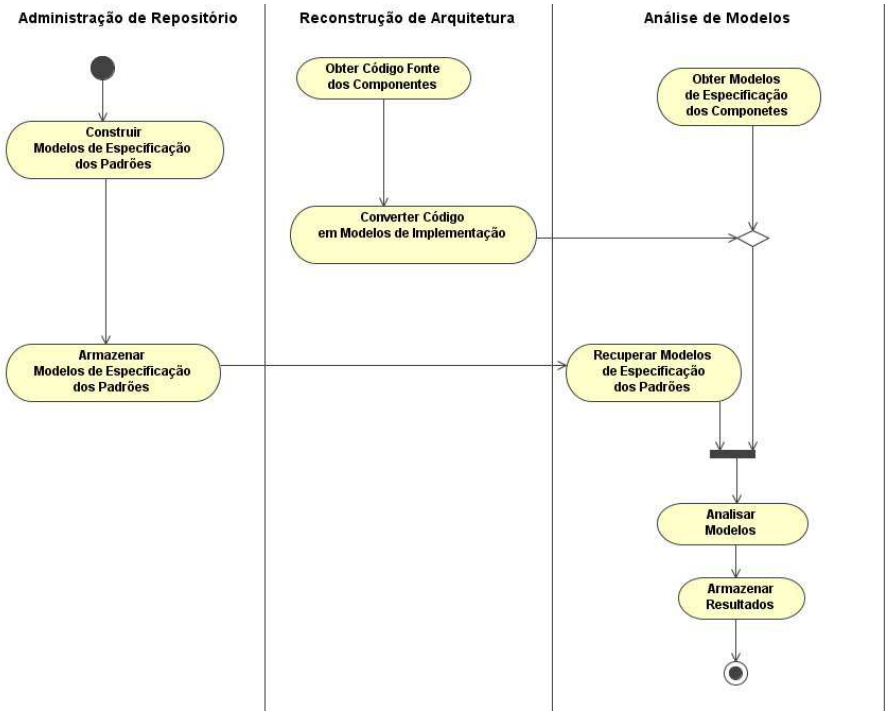


Figura 3. Diagrama de Atividades organizado por macro-atividades

Envolvidos no processo de certificação e associados a cada macro-atividade definida, foram identificados os seguintes papéis:

- Arquiteto de Negócios: responsável pela escolha dos padrões utilizados nos modelos de negócio e pela construção dos modelos das especificações desses padrões;
- Arquiteto de Aplicação: responsável pela escolha dos padrões utilizados nos modelos de aplicação dos componentes de software e pela construção dos modelos das especificações desses padrões;
- Administrador do Repositório: responsável pelo armazenamento e gerência dos modelos no repositório de padrões. É de sua responsabilidade manter atualizado o catálogo dos padrões utilizados no respectivo ambiente de desenvolvimento;
- Gerente de Reutilização: entre suas atribuições estão executar a conversão dos códigos dos componentes em modelos de implementação e analisar todos os modelos dos componentes desenvolvidos, antes da liberação desses componentes para consumo;
- Desenvolvedores: elaboram os modelos dos componentes e aplicações utilizando as especificações dos padrões existentes no repositório.

Esse modelo de atribuição de responsabilidades pode ser alterado de acordo com as características e necessidades do ambiente particular de cada organização.

3.2 A Arquitetura Conceitual do Ambiente de Certificação

A figura 4 apresenta a arquitetura do ambiente proposto, com os elementos do sistema computacional utilizado, necessário para automatizar as atividades de certificação.

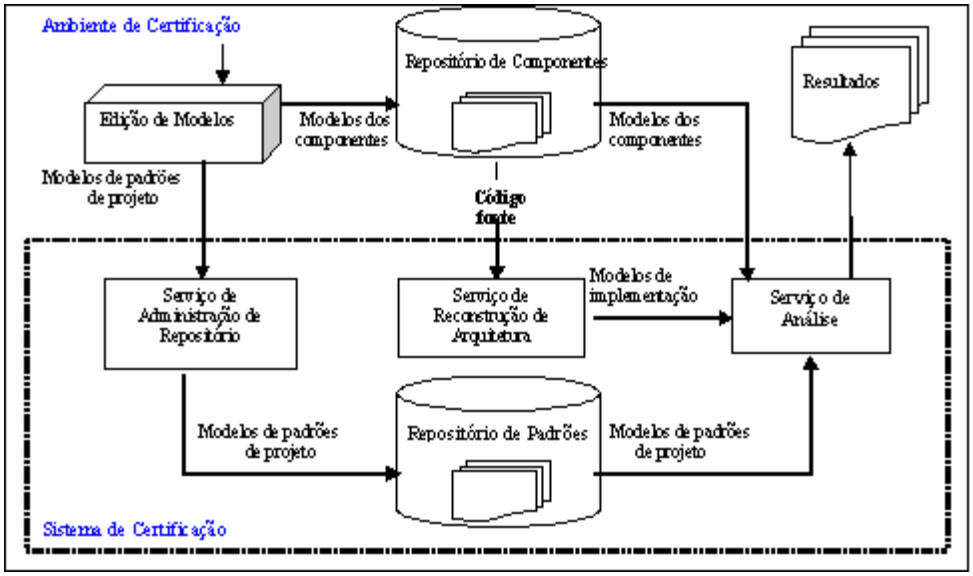


Figura 4. A Arquitetura conceitual do ambiente de certificação.

Nessa arquitetura identificamos funcionalidades existentes na abordagem da MDA que são utilizadas no ambiente de certificação proposto:

1. Edição de Modelos: utilizada na construção, através de ferramentas de modelagens específicas, dos modelos de representação dos padrões de projeto e dos modelos de desenvolvimento dos componentes e aplicações na linguagem de modelagem e formalismo de representação definido. Precisa exportar os modelos para um padrão de intercâmbio de modelos (metadados);
2. Repositório de Padrões: é utilizado para armazenar os modelos dos padrões utilizados no ambiente de desenvolvimento. Importa e exporta os modelos no padrão de intercâmbio de modelos definido. Os modelos são acessados no repositório utilizando uma forma comum de acesso a metadados;
3. Sistema de Certificação: sistema computacional de apoio, responsável pela automatização das atividades do ambiente de certificação. Sua estrutura é composta por três principais serviços. Esses serviços incorporam as características funcionais das macro-atividades identificadas no ambiente de certificação.
    a) Serviço de Administração de Repositório: responsável por manter o repositório de padrões. Os modelos de representação de padrões são importados para o repositório, e após a validação da formação correta dos modelos quanto ao formalismo utilizado. Os modelos em conformidade são aceitos e armazenados no repositório;
    b) Serviço de Reconstrução de Arquitetura: responsável pela análise e reconhecimento de código, realizando a conversão do código fonte dos componentes, obtidos do repositório de componentes, nos modelos de implementação (último nível de modelo de plataforma específica - PSM). Interpreta o código, identificando os elementos da linguagem de implementação e converte a definição desses elementos na linguagem de modelagem utilizada;

c) Serviço de Análise: responsável pela validação de modelos, analisa e certifica a correta construção dos modelos (negócio, aplicação e recuperados do código dos componentes e aplicações) com relação à utilização dos padrões de projeto. Utiliza uma forma comum de acesso a metadados para manipulação dos modelos em análise. Gera resultados com os diagnósticos da análise. Existem dois tipos de análise fornecidos pelo serviço:

- Análise dos Padrões Utilizados: Análise quanto à utilização correta dos padrões existentes no modelo. Compara a construção dos padrões identificados no modelo, com a representação desses padrões armazenados no repositório, certificando se estão em conformidade;
- Análise das Transformações de Padrões: Análise quanto às transformações de padrões existentes entre dois níveis de abstração. Nesse caso os padrões utilizados e identificados no modelo analisado, são comparados com as definições dos mapeamentos dos padrões do modelo original, certificando se as transformações entre os dois níveis estão em conformidade;

Na conexão entre os elementos da arquitetura necessitamos de mecanismos específicos de integração. Esses mecanismos estão relacionados com uma forma comum de acesso e padrão de intercâmbio de modelos. São utilizados para acesso, manipulação e intercâmbio de modelos entre as diferentes funcionalidades do ambiente.

3.3 Implementação do Ambiente de Certificação

Para a implementação da arquitetura do ambiente de certificação, foram feitas diversas escolhas tecnológicas com relação a: linguagem de modelagem e formalismo de representação, forma comum de acesso e padrão de intercâmbio de modelos, repositório de padrões e tecnologia de implementação utilizada.

Quanto à linguagem de modelagem e ao formalismo de representação, utilizados na construção dos modelos dos padrões e nos modelos de desenvolvimento dos componentes e aplicações, a escolha foi baseada nas abordagens de modelagem de papéis do *Role-Based Metamodeling Language Research Group*[1] [France *et al* 2003], e na utilização de mecanismos de extensão da UML através de perfis específicos [Dong e Yang 2003]. É necessário dispor de diversas informações sobre as características dos padrões nos seus modelos de representação. A certificação dos modelos de desenvolvimento dos componentes e aplicações que utilizam esses padrões é realizada de acordo com essas características. Da mesma forma, o sistema de certificação proposto precisa identificar os padrões existentes nos modelos analisados e todos os elementos que executam papéis nesses padrões. Assim, foi preciso acrescentar determinadas informações nos modelos de componentes e aplicações para permitir o rastreamento e identificação dos padrões presentes nesses modelos. A definição de um perfil UML específico para representação de padrões, bem como, um perfil UML específico para utilização de padrões nos modelos dos componentes e aplicações, permitem que todas as informações necessárias para o processo de certificação sejam incorporadas nos modelos através de estereótipos e etiquetas.

[1] http://www.cs.colostate.edu/~dkkim/ReuseResearch /main.html

Quanto à forma comum de acesso e padrão de intercâmbio de modelos, os padrões escolhidos foram o *Java Metadata Interface* (JMI) [JSR40 2002] e o *XML Metadata Interchange* (XMI) [XMI 2003]. Ambos são mapeamentos do MOF para a linguagem Java e XML respectivamente. Juntos, JMI e XMI fornecem uma completa estrutura de acesso e troca dinâmica de metadados independente de plataforma e de fabricante [Eckerson e Manes 1999].

O repositório de padrões escolhido foi o repositório de metamodelos MDR[1] compatível com MOF/JMI**.** Carrega qualquer metamodelo MOF e armazena suas instâncias (os metadados em conformidade com o metamodelo). Permite importação e exportação de metadados usando o padrão XMI, e manipulação desses metadados programaticamente usando interfaces Java através do JMI [Matula 2003].

A tecnologia Java [JSR244 2005] foi escolhida para a implementação do ambiente de certificação em conseqüência das escolhas do JMI e do repositório MDR. A tecnologia Java está bastante aderente à abordagem da MDA, com diversas propostas de padrões da OMG e do *Java Community Process* (JCP), relacionando o ambiente orientado a modelos da MDA com Java, a exemplo do JMI, metamodelos e perfis UML para Java e EJB [OMG 2004] [JSR026 2001].

Para o reconhecimento da linguagem Java, e conversão para UML, foram utilizadas a ferramenta ANTLR[2] e uma adaptação da classe *Modeller* da ferramenta *Java Roundtrip Engineering* (JavaRE)[3]. Essa adaptação consistiu na utilização do repositório MDR e das API's JMI do metamodelo da UML 1.4.

A utilização dessas decisões de implementação nos serviços que compõem o sistema de certificação se deu da seguinte forma:

Serviço de Administração do Repositório: Os modelos de representação de padrões são lidos pelo repositório MDR no padrão XMI. Os modelos em conformidade são armazenados no repositório pela representação de uma instância do metamodelo da UML na API JMI;

Serviço de Reconstrução de Arquitetura: Através de classes geradas pelo ANTLR, fazemos o reconhecimento da linguagem, identificamos os elementos a serem mapeados para UML e através da classe *Modeller*, adicionamos esses elementos em um atributo de representação de um modelo UML na API JMI. Após todo o código Java ser convertido, temos o modelo UML completo. Foram utilizados alguns recursos adicionados no J2SE 5.0 [JSR244 2005], para representação de determinados elementos de modelo no código implementado, como *Annotation* (para representar os estereótipos e etiquetas existentes no último nível de modelo de plataforma específica - PSM) e *Generics* (para representar as associações com multiplicidade *). O recurso de *Annotation* é resultado de uma nova proposta do JCP identificada como JSR-000175 *A Metadata Facility for the Java Programming Language* [JSR175 2004]**.**

Serviço de Análise de Modelos: Utiliza a API JMI para acessar os metadados, definições dos elementos de modelo contidos na representação JMI do modelo UML

[1] http://mdr.netbeans.org/
[2] http://www.antlr.org/
[3] http://javare.sourceforge.net/index.php

sendo analisado, identificando os padrões utilizados através dos estereótipos e etiquetas existentes. Esses padrões identificados são recuperados do repositório MDR, pela busca de uma instância do metamodelo da UML com o nome e tipo do padrão de projeto. Os modelos são comparados através das interfaces JMI, que representam os elementos existentes nos dois modelos (analisado e de representação do padrão);

4. Um Cenário de Utilização

O cenário de utilização corresponde a um exemplo de sistema de registro das vendas de uma rede de lojas de departamento. Nesse cenário temos o envolvimento dos papéis do gerente de reutilização e do desenvolvedor, descritos na seção 3.1. O desenvolver elabora manualmente todos os modelos do sistema, utilizando os estereótipos e etiquetas específicos do perfil UML para utilização de padrões, desenvolvido para o ambiente de certificação e o gerente de reutilização executa a certificação dos modelos a cada etapa de transformação no ciclo de desenvolvimento através do sistema computacional proposto.

A figura 5 apresenta o modelo de domínio correspondente ao PIM de nível mais alto de abstração, onde foi utilizado o perfil UML para utilização de padrões. Possui o estereótipo <<MNIP>> (*Modelo de Negócio Independente de Plataforma*) com a etiqueta *nivelModelo* igual a um, indicando existir mais um nível de abstração de modelo de negócio. Seus elementos de classe possuem o estereótipo <<Entidade>> indicando que representam um conceito de entidade do domínio do sistema, com a etiqueta *padroesAlvo*, indicando o padrão que será utilizado na transformação para o modelo do próximo nível de abstração.

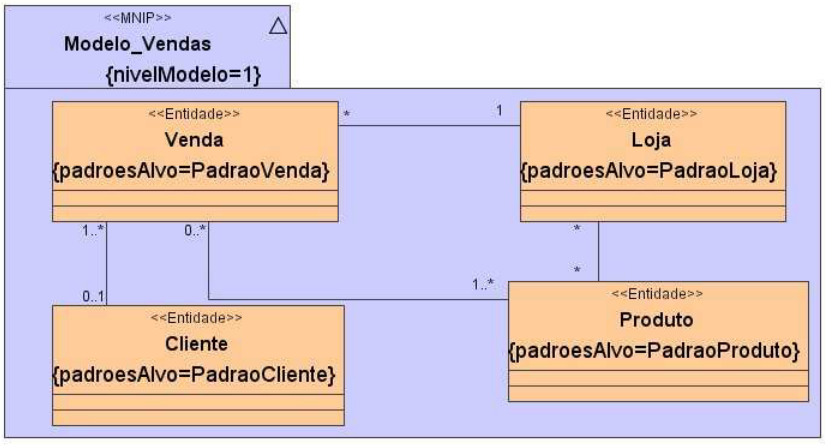


Figura 5. Modelo de Negócio Independente de Plataforma <<MNIP>> *Modelo de Vendas*

Transformamos esse modelo de negócio, aplicando exemplos simplificados de padrões corporativos de negócio, ou padrões de arquétipos de negócio, conforme denominação de Arlow e Neustadt (2003). O modelo inicial do exemplo do sistema de vendas é refinado através da substituição dos elementos Venda, Cliente, Loja e Produto, pelos padrões de negócio Venda, Cliente, Loja e Produto respectivamente.

A figura 6 apresenta o modelo do padrão de negócio Venda, que utiliza o perfil UML para representação de padrões, desenvolvido para o ambiente de certificação.

Possui o estereótipo <<PNIP>> (*Padrão de Negócio Independente de Plataforma*) com a etiqueta *nivelPadrao* igual a zero, indicando o último nível de abstração antes da transformação para um padrão <<PAIP>> (*Padrão de Aplicação Independente dePlataforma*). Todos os seus elementos com papel de classe possuem o estereótipo <<Entidade>>, indicando a representação do conceito de entidade do domínio do padrão, com a etiqueta *padroesAlvo* indicando os papéis de padrões que podem ser mapeados na transformação desses elementos para o próximo nível de abstração.

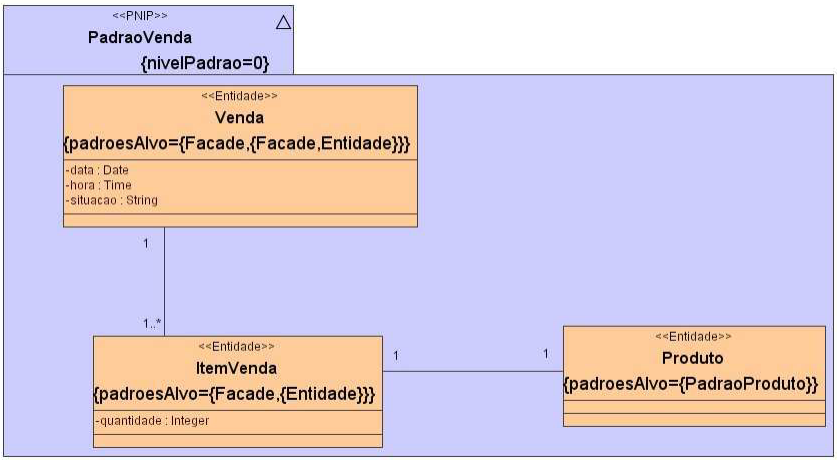


Figura 6. Padrão de Negócio Independente de Plataforma <<PNIP>> *PadrãoVenda*

No primeiro refinamento realizado no modelo de domínio do sistema, após aplicação dos padrões de arquétipos de negócio, geramos um novo modelo que não está sendo ilustrado, com estereótipo <<MNIP>>. PIM de segundo nível de abstração no ciclo de desenvolvimento corresponde a um modelo de negócio mais detalhado. Possui a etiqueta *nivelModelo* igual a zero, indicando que o próximo nível será um *Modelo de Aplicação Independente de Plataforma* <<MAIP>>, e a etiqueta *padroesAlvo* indicando o *Padrão de Aplicação Independente de Plataforma* com estereótipo <<PAIP>> que será utilizado na transformação para o próximo nível de abstração. Nesse caso o padrão utilizado será o padrão Facade do catálogo GoF.

A certificação realizada nesse modelo corresponde à certificação da primeira etapa de transformação ou refinamento. As figuras 7, 8 e 9 apresentam os resultados dessa certificação. Na figura 7, o resultado apresenta a estrutura de todos os padrões identificados no modelo analisado. Na figura 8, o resultado indica as conformidades encontradas na comparação entre as construções dos padrões existentes no modelo analisado e suas representações existentes no repositório. Outro resultado possível seria a exibição da relação de erros existentes na utilização de algum padrão no modelo. Na figura 9, o resultado relaciona as conformidades encontradas nas definições de transformação existentes no modelo original. Nesse resultado são apresentados os elementos desse modelo, os padrões de projeto utilizados na transformação de cada elemento, com os respectivos elementos e papéis gerados no modelo transformado. Ainda outro resultado possível seria a relação de erros encontrados, onde seriam apresentados os elementos dos dois modelos cujos mapeamentos de padrões não correspondem às definições existentes no modelo original. O objetivo desses resultados é auxiliar os desenvolvedores na correção das inconsistências existentes.

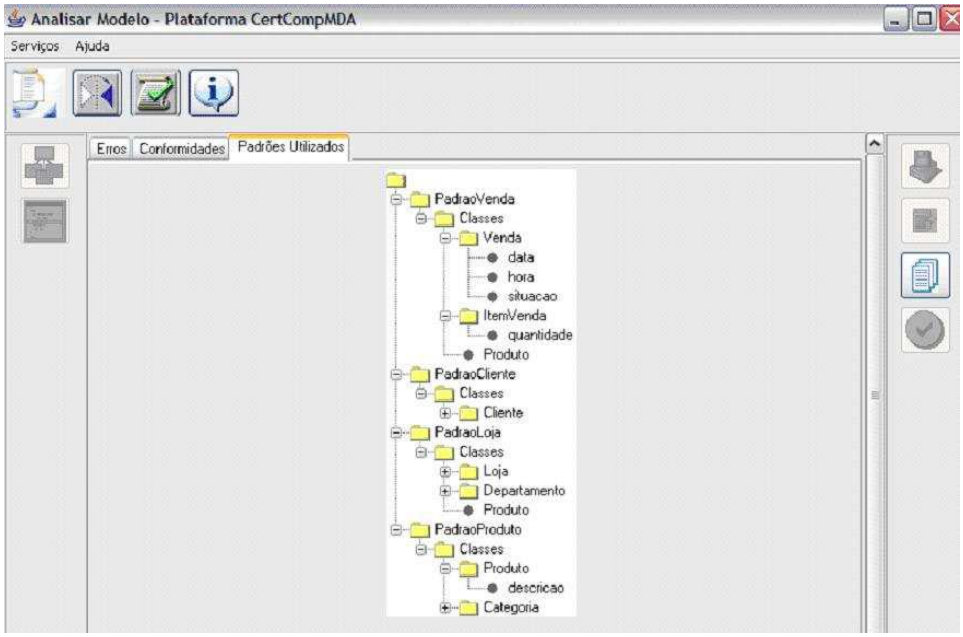


Figura 7. Padrões Utilizados

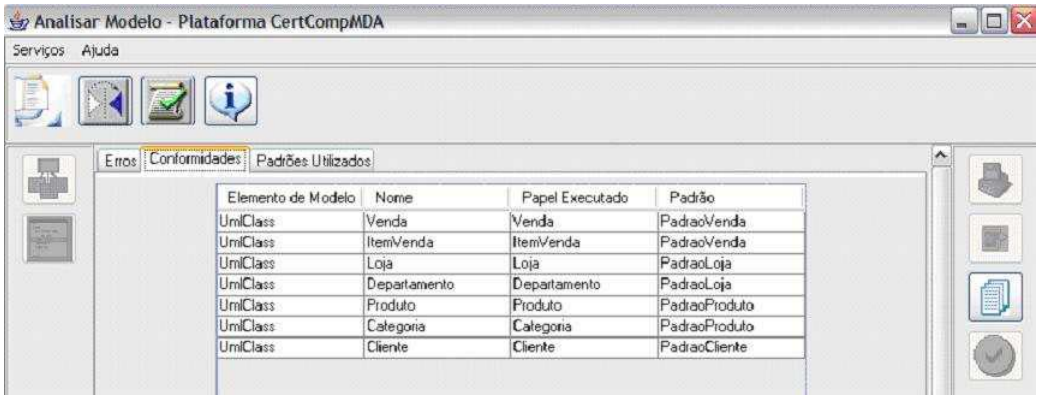


Figura 8. Padrões Utilizados – Conformidades

Figura 9. Transformação de Padrões Utilizados – Conformidades

Na segunda etapa de transformação, após a aplicação do padrão Facade, é gerado o primeiro e único modelo com estereótipo <<MAIP>> com nívelModelo igual a zero, indicando que o próximo nível será um *Modelo de Aplicação de Plataforma Específica* <<MAPE>>. A figura 10 apresenta esse modelo, onde temos a ilustração da utilização do estereótipo <<ClassePadrao>> por todos seus elementos de classe, indicando que executam um papel de padrão, especificado na etiqueta *padroesExecutados*. A etiqueta *padroesAlvo* indica a utilização do padrão

SessionFacadeValueObjet, específico da plataforma J2EE, na transformação para o próximo nível de abstração.

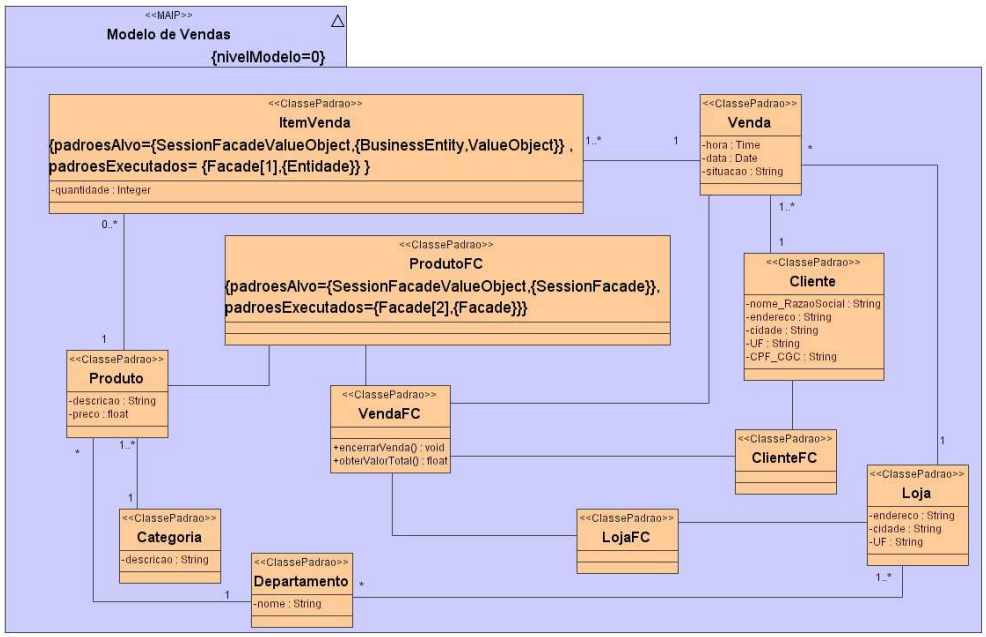


Figura10. <<MAIP>> Sistema de Vendas

A figura 11 apresenta o modelo do padrão SessionFacadeValueObject, utilizando o perfil UML para representação de padrões. Esse padrão é resultante de uma configuração nova de padrão de projeto, com a junção dos padrões SessionFacade e ValueObject do Core J2EE. Possui o estereótipo <<PAPE>> (*Padrão de Aplicação de Plataforma Específica*).

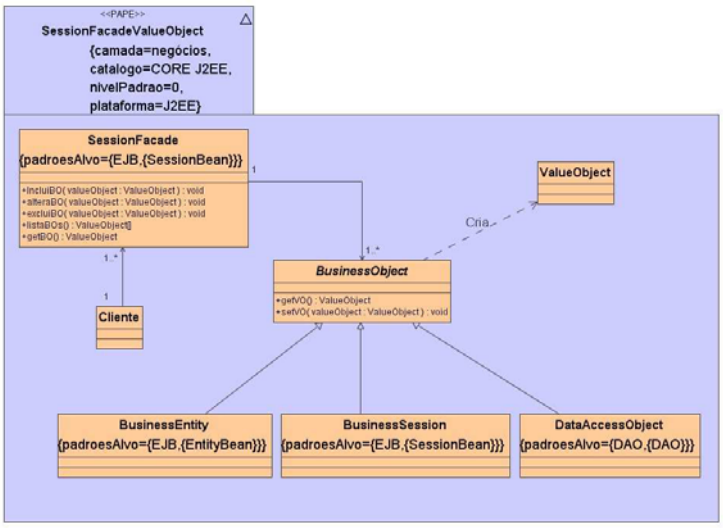


Figura 11. Padrão da Plataforma J2EE : SessionFacadeValueObject

A terceira etapa de transformação ou refinamento, após a aplicação do padrão SessionFacadeValueObject, resulta no primeiro e último modelo PSM, antes da transformação para o código J2EE. Esse modelo possui estereótipo <<MAPE>> e etiqueta *nivelModelo* igual a zero.

Além do estereótipo <<ClassePadrao>>, também podem ser utilizados os estereótipos <<OperacaoPadrao>> e <<AtributoPadrao>> nos elementos de modelo operação e atributo respectivamente, para indicar que executam papéis de padrão. Na transformação para o código, todos os estereótipos <<ClassePadrao>>, <<OperacaoPadrao>> e <<AtributoPadrao>> associados a sua respectiva etiqueta *padroesExecutados*, existentes no último nível de modelo <<MAPE>>, precisam ser transformados em anotações, para permitir a identificação dos padrões no código implementado. A figura 11 apresenta o exemplo do código gerado com a utilização dessas anotações.

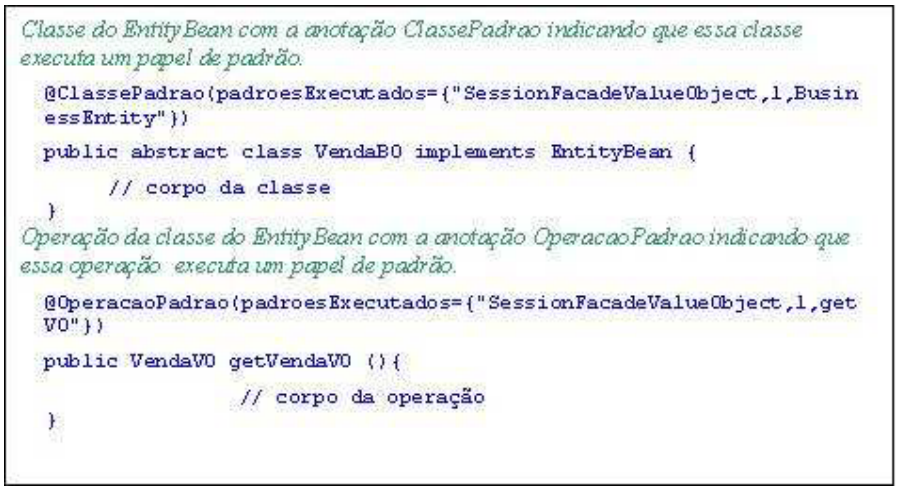


Figura 11. Estereótipos e Etiquetas como Anotações no Código

A etiqueta *padroesExecutados* é multivalorada, permitindo a definição de vários papéis de padrão para uma mesma classe, operação ou atributo, e através da informação de instância em sua lei de formação, podemos ter diversas ocorrências do mesmo padrão em um componente ou aplicação.

Em todas as etapas de transformação do ciclo de desenvolvimento do sistema, com a aplicação dos padrões de projeto específicos a cada nível de abstração, as certificações são executadas pelo gerente de reutilização, através das validações dos modelos pelo serviço de análise de modelos, conforme definido no ambiente de certificação proposto.

Os detalhes sobre a especificação dos dois perfis UML criados para o ambiente proposto, com todas as definições dos estereótipos e etiquetas utilizados, bem como a apresentação completa do cenário utilizado na validação das funcionalidades do ambiente e sistema computacional, se encontram em Gomes, M.C.F (2005).

5. Conclusão

O processo de certificação proposto inclui a definição de um formalismo de representação de padrões, com flexibilidade suficiente para representar qualquer padrão de projeto utilizado no ciclo de desenvolvimento. Define um mecanismo de mapeamento entre os padrões através dos diversos níveis de abstração, permitindo a validação das transformações de padrões na passagem de um nível para outro imediatamente adjacente. Esse sistema de certificação pode ser aplicado em qualquer

ambiente orientado a modelos, independente da quantidade de níveis de abstração e refinamentos utilizados.

Dos trabalhos e ferramentas analisados, relacionados com padrões de projeto e com o ambiente orientado a modelos da MDA, o que mais se aproxima com essa proposta é a ferramenta comercial Optimalj [Crupi e Baerveldt 2005]. Uma ferramenta que fornece um ambiente de desenvolvimento orientado a modelos baseado em padrões utilizando a plataforma tecnológica J2EE. Porém não apresenta a mesma flexibilidade para representar padrões desenvolvidos sob medida para um determinado ambiente de desenvolvimento, nem para ser utilizado em qualquer quantidade de níveis de abstração.

A utilização de perfis UML específicos para representação e utilização de padrões de projeto nos modelos das aplicações e componentes torna a solução adotada de grande alcance dentro da comunidade de desenvolvimento de software, já que nesse ambiente a UML é intensamente conhecida e adotada, sendo o padrão de fato de linguagem de modelagem de sistemas de software. Uma contribuição importante desse trabalho foi utilizar como ambiente computacional, a tecnologia Java e a plataforma J2EE, referências hoje no desenvolvimento de aplicações robustas e distribuídas. A utilização de um repositório compatível com MOF e JMI trouxe bastante flexibilidade, e a facilidade de manipulação das definições de modelos através das interfaces Java geradas tornou o desenvolvimento nesse ambiente muito mais produtivo. Como contribuições específicas podemos citar o fornecimento de uma implementação de domínio público de transformação de código Java em modelos UML (em XMI ou JMI). Esse trabalho deixa alguns problemas ainda em aberto, que poderão ser resolvidos em trabalhos futuros, como:

- Adição de restrições em OCL aos modelos de representação aumentando a precisão semântica e a precisão dos diagnósticos das avaliações realizadas;
- Desenvolvimento de extensões do sistema de certificação para outras plataformas, implementando serviços de reconstrução de arquitetura (conversão do código fonte nos modelos UML de implementação) específicos para essas plataformas;
- Incorporação dos perfis UML para Java e EJB do OMG e JCP na transformação dos modelos de plataforma específica em código;
- Implementação de um ambiente de automatização do desenvolvimento de componentes e aplicações no ambiente orientado a modelos, através da automatização das transformações de modelos utilizando os mesmos perfis UML e mecanismo de associação entre padrões definidos nesse trabalho.

Referencias

Arlow, J. e Neustadt, I. (2003) Enterprise Patterns and MDA: Building Better Software with Archetype Patterns and UML. Addison-Wesley.

Blankers, T. (2003) Combining models and patterns: delivering on the promise of increased IT productivity. COMPUWARE White Paper Series. Disponível em: http://www.compuware.com/products/optimalj/1794_ENG_HTML.htm#MDA

Crupi, J. e Baerveldt, F. (2005) Implementing Sun Microsystems Core J2EE Patterns. COMPUWARE White Paper. Disponível em: http://www.bitpipe.com/detail/RES/1095961328_371.html?src=FEATURE_GLOBAL

Dong, J. e Yang, S. (2003) Visualizing Design Patterns With A UML Profile. The Proceedings of the IEEE Symposium on Visual/Multimedia Languages (VL), pag.123-125, Auckland, Nova Zelandia, outubro, 2003. Disponível em: http://www.utdallas.edu/~syang/vl03.pdf

Eckerson, W. W. e Manes, A. T. (1999) Paving the Way for Transparent Application and Data Interchange. A Java API for Metadata. SUN JMI White Papers. Disponível em: http://java.sun.com/products/jmi/pdf/JavaMetadataAPI.pdf

France, R., *et al*. (2003) A Role-Based Metamodeling Approach to Specifying Design Patterns. In Proceeding of 27th IEEE Annual International Computer Software and Applications Conference, pag. 452- 457, Dallas, Texas, novembro, 2003. Disponível em: http://ieeexplore.ieee.org/search/wrapper.jsp?arnumber=1245379

Frankel, D. S. (2003) Model Driven Architecture. Applying MDA to Enterprise Computing. John Wiley & Sons, OMG Press.

Gamma, E., *et al*. (1995) Design Patterns: Elements of reusable object-oriented software. Addison-Wesley.

Gomes, M.C.F. (2005) Certificação da Utilização de Padrões de Projeto no Desenvolvimento Orientado a Modelos. Dissertação de Mestrado. Programa de Pós-graduação em Informática do IM/NCE – Universidade Federal do Rio de Janeiro

Herzum, P. e Sims, O. (1999) Business Component Factory. John Wiley & Sons.

JSR175 (2004) A Program Annotation Facility for the Java Programming Language. JSR175 final release. JCP. Disponível em: http://www.jcp.org/en/jsr/detail?id=175

JSR026 (2001) UML/EJB Mapping Specification. JSR026 Public Draft. JCP. Disponível em: http://www.jcp.org/aboutJava/communityprocess/review/jsr026/

JSR244 (2005). Java Platform Enterprise Edition 5 Specification. JSR244 Close of Public Review. JCP. Disponível em: http://jcp.org/aboutJava/communityprocess/pr/jsr244/index.html

JSR40 (2002). Java Metadata Interface (JMI) Specification. Versão 1.0. OMG. Disponível em: http://jcp.org/aboutJava/communityprocess/pr/jsr244/index.html

Matula, M. (2003) NetBeans Metadata Repository. SUN white Paper. Disponível em: http://mdr.netbeans.org/MDR-whitepaper.pdf

MDA (2003). Guide Version 1.0.1. Document Number: omg/2003-06-01. OMG. Disponível em: http://mdr.netbeans.org/MDR-whitepaper.pdf

Mellor, S. J., *et al*. (2004) MDA Distilled: Principles of Model-Driven Architecture. Addison Wesley.

OMG (2004) Metamodel and UML Profile for Java and EJB Specification. Version 1.0. Disponível em: http://www.omg.org/docs/formal/04-02-02.pdf

Poole, J. D. (2001) Model-Driven Architecture: Vision, Standards And Emerging Technologies. Workshop on Metamodeling and Adaptive Object Models.

XMI (2003). XML Metadata Interchange. Versão 2.0. OMG. Disponível em http://www.omg.org/docs/formal/03-05-02.pdf

Modeling Multi-Agent Systems using UML

Carla Silva[1], João Araújo[2], Ana Moreira[2], [γ]Jaelson Castro[1, 3], Patrícia Tedesco[1], [ξ]Fernanda Alencar[4] and Ricardo Ramos[1]

[1]Centro de Informática, Universidade Federal de Pernambuco, Recife, Brazil

{ctlls, jbc, pcart, rar2}@cin.ufpe.br

[2]Departamento de Informática, FCT, Universidade Nova de Lisboa, Portugal

{ja, amm}@di.fct.unl.pt

[3]Istituto Trentino di Cultura, Ist. per la Ricerca Scientifica e Tecnologica, Italy

jaelson@itc.it

[4]Departamento de Eletrônica e Sistemas, Universidade Federal de Pernambuco

fmra@ufpe.br

Abstract. *Tropos is a framework which offers an approach to guide the development of multi-agent systems (MAS). It relies on the i* notation to describe both requirements and architectural design. However, the use of i* as an architectural description language (ADL) is not suitable, since it presents some limitations to capture all the information required for designing MAS architectures. Recognizing that the Unified Modeling Language 2.0 (UML) supports software architectural description, in this work we present an extension to the UML metamodel to capture the features of agency to support MAS modeling at the architectural level. In doing so, we define a notation to model MAS architectures. Furthermore, we provide a set of heuristics to describe MAS using our UML-based notation derived from an architectural description using i*. We illustrate our approach by modeling a Conference Management System.*

Resumo. *Tropos é um framework que fornece uma abordagem para guiar o desenvolvimento de sistemas multi-agentes (SMA). Ele utiliza a notatção i* para descrever tanto os requisitos como o projeto arquitetural de SMA. Entretanto, o uso da notação i* como uma linguagem de descrição arquitetural (LDA) não é adequado porque esta notação apresenta algumas limitações para capturar toda a informação necessária para projetar a arquitetura de SMA. Reconhecendo que a Unified Modeling Language 2.0 (UML) suporta a descrição arquitetural de software, neste trabalho apresentamos uma extensão do metamodelo da UML para capturar as características de agência de forma a suportar a modelagem de SMA ao nível arquitetural. Com isto, definimos uma notação para modelar a arquitetura de SMA. Além disso, oferecemos um conjunto de heurísticas para descrever SMA usando nossa notação baseada em UML a partir da descrição arquitetural do SMA usando i*. A nossa abordagem é ilustrada usando um Sistema Gerenciador de Conferência.*

[γ] Currently on leave of absence from Universidade Federal de Pernambuco.

[ξ] Currently on leave of absence at FCT / Universidade Nova de Lisboa, Portugal.

1. Introduction

One of the most promising paradigms for developing complex software systems is the agent orientation. However, the benefits promised by the agent paradigm cannot be fully achieved yet because it lacks suitable methodologies to enable designers to clearly specify and structure their applications as agent-oriented systems. To address this issue, we are working on the improvement of Tropos [Castro et al. 2002] - a framework aimed at developing multi-agent systems. Tropos supports the four following phases of software development: Early Requirements, Late Requirements, Architectural Design and Detailed Design.

In this work, our focus is on the architectural design phase. Software architecture defines, at a high level of abstraction, the system in terms of components, the interaction between them as well as the attributes and functionalities of each component [Sommerville 2001]. Tropos relies on the i* notation [Yu 1995] to describe MAS architectural design. However, the use of i* as an architectural description language (ADL) is not suitable, since it presents some limitations to describe the detailed behaviour required for software architectural design, such as protocols, connectors, ports and interfaces. In [Silva et al. 2003] we have proposed an approach to use the UML-RT (UML-Real Time) [Selic and Rumbaugh 1998] as an ADL for Tropos. Part of the UML-RT concepts have been incorporated as architecture description constructs in UML 2.0 [OMG 2005]. Hence, in this paper we present an approach for using UML 2.0 based notation to describe MAS architecture in Tropos which is originally described using the i* notation [Yu 1995]. UML 2.0 is a standard with large tool support, which is tailored for architectural description.

This paper is organised as follows. Section 2 introduces our extension to the UML metamodel to support MAS modelling. Section 3 presents our notation to describe MAS architectural design and a process to use this notation in the context of Tropos. Section 4 illustrates our approach using a case study. Section 5 discusses related work. Finally, section 6 summarises our work and points out directions for future work.

2. Agency Profile

To enable the creation of UML profiles, a profile package has been specifically defined in the UML 2.0 specification [OMG 2005] for providing a lightweight extension mechanism to the UML standard. It contains mechanisms that allow metaclasses from existing metamodels to be extended and adapted for different purposes. This includes the ability to tailor the UML metamodel for different platforms (such as J2EE or .NET) or domains (such as real-time or business process modeling). The profile mechanism is consistent with the OMG Meta Object Facility (MOF) [OMG 2004]. This paper uses this mechanism to adapt an existing UML metamodel with constructs that are specific to the agent paradigm. Such adaptation is grouped in a profile, called *Agency Profile*.

For simplicity, the metamodel defining the agency features is divided into two categories: intentional and interaction. The intentional category concepts are described in Figure 1 while the interaction category concepts are described in Figure 2. The usage of the concepts and relationships has been motivated by a previous work [Silva et al. 2004] where we have established some agent properties required to specify MAS.

In the intentional category a MAS can be conceived as an *Organization* which is composed of a number of *Agents*. The Agent concept extends the UML metaclass Class

from the StructuredClasses package which extends the metaclass Class (from the Kernel package) with the capability to have an internal structure and ports. *Norms* are required for the *Organization* to operate harmoniously and safely. They define a policy and constraints that all the organizational members must be compliant with [Minsky and Muarata 2004]. The *Organization* is typically immersed in exactly one *Environment* that the agents may need to interact with, in order to access *Resources* according to the agent *Rights* [Zambonelli et al. 2003]. The Organization, Norm, Environment, Right and Resource concepts are extensions (and more exactly, specializations) of the UML metaclass Class.

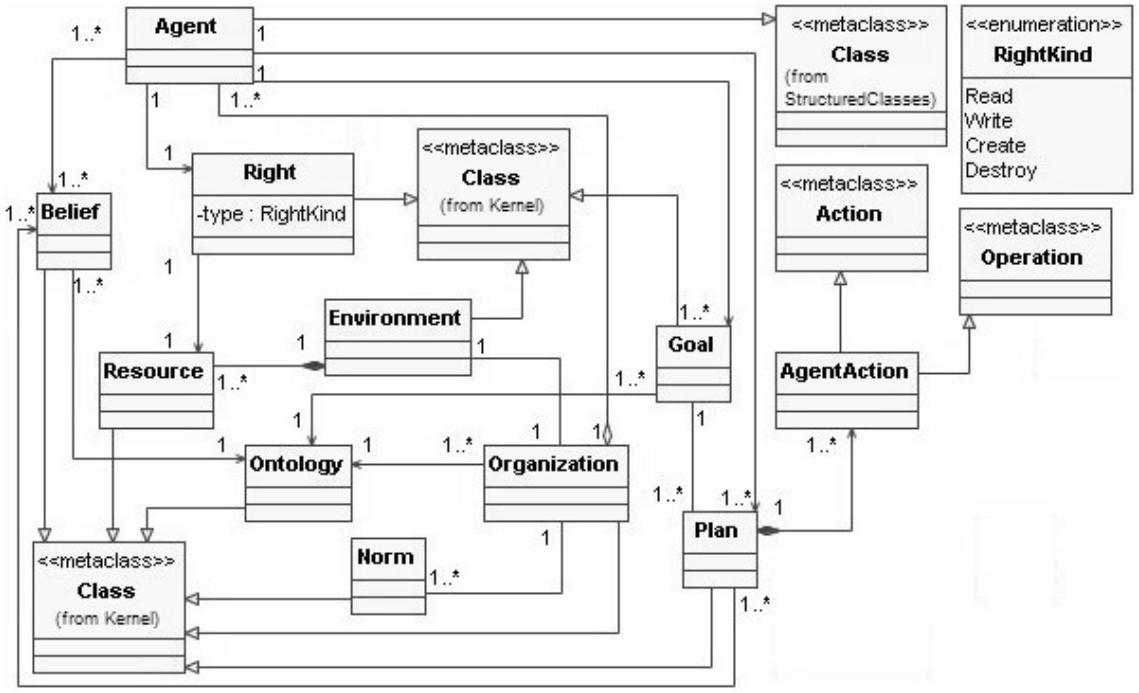


Figure 1. Agency metamodel reflecting intentional concepts

A Goal is a concrete desire of an agent [Braubach et al. 2004]. A Plan encapsulates the recipe for achieving some goal. Beliefs represent the information the agent has about its current environment and itself [Wooldridge 2002] and are conditions for executing plans. These concepts also extend the UML metaclass Class. An AgentAction determines the steps to perform a plan and extends both the Action and Operation UML metaclasses. The description of both Beliefs and Goals must comply with the Ontology used in the Organization, i.e. the vocabulary of concepts which belongs to the system domain. We define the MAS ontology by extending the UML metaclass Class. The rationale is that most of the technology available to define an ontology in MAS is based on object orientation. For example, an ontology in a specific agent platform, called JADE (Java Agent Development Framework) [Bellifemine et al. 2003], is an instance of *jade.content.onto.Ontology* class, in which a set of schemes is added to define the structure of concepts which belongs to the domain being modeled.

Since agents are going to be used in the system architectural design, we will define them by using the organizational architectural features defined in [Silva et al. 2003]. These features, defining the interaction category, are depicted in Figure 2 and include: OrganizationalPort, AgentConnector, Dependum, Dependee, Depender and

AgentConnectorEnd. They extend respectively the UML metaclasses Port, Connector, Interface, InterfaceRealization, Usage and ConnectorEnd.

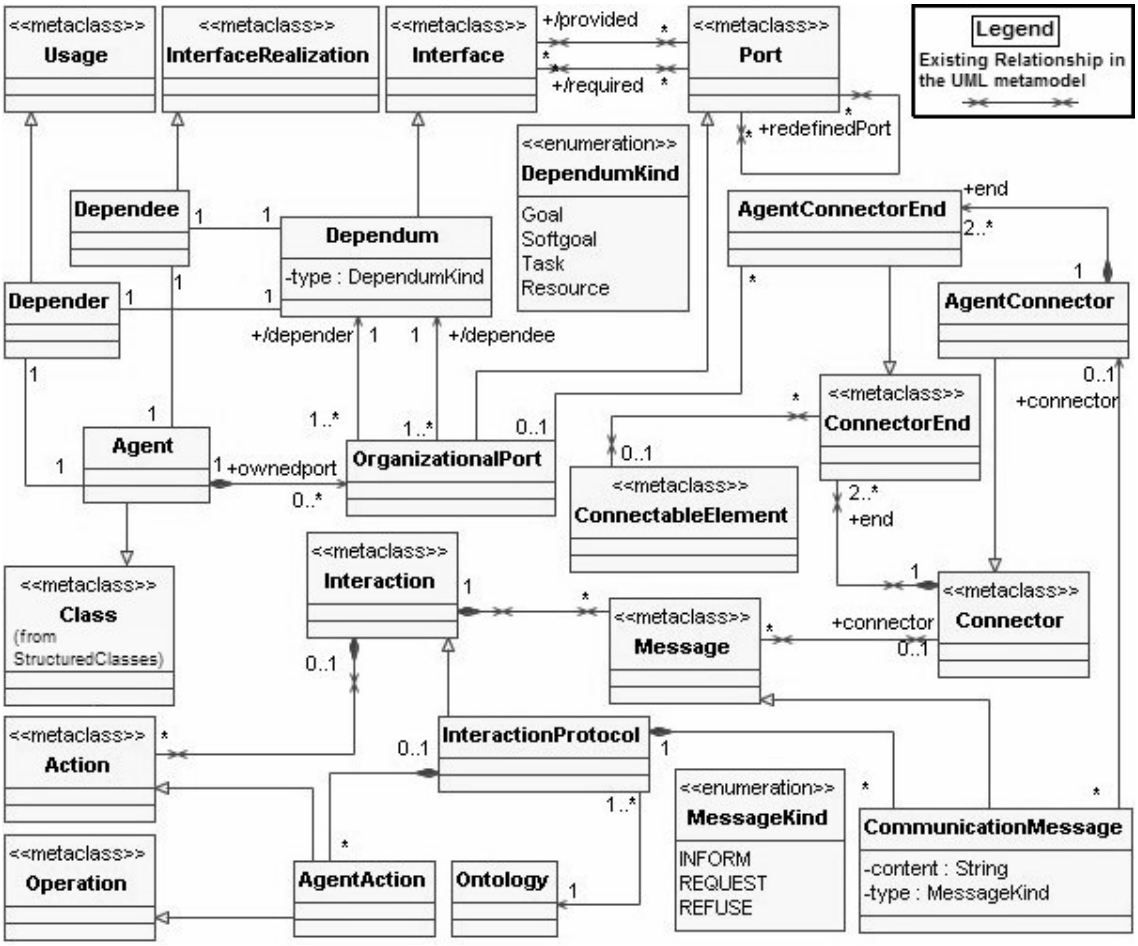


Figure 2. Agency metamodel reflecting interaction concepts

A Dependum defines an "agreement" of service providing between two agents which play the roles of depender and dependee. Thus, the agent responsible for providing the service possesses an OrganizationalPort playing the role of dependee and is related to the Dependum through a Dependee relationship. The agent which request the service possesses an OrganizationalPort playing the role of depender and is related to the Dependum through a Depender relationship. A dependum can be of four types: goals, softgoals, tasks and resources [Yu 1995], defined as the enumeration class DependumKind. Agents need to exchange signals through an AgentConnector to accomplish the contractual agreement of service providing between them. An OrganizationalPort specifies a distinct interaction point between the Agent and its environment. A AgentConnectorEnd is an endpoint of an Agentconnector, which attaches the AgentConnector to an OrganizationalPort.

Each *Agent* can interact with other agents according to an *InteractionProtocol* determined by the *AgentAction* performed by the *Agent*. An *InteractionProtocol* describes a sequence of *CommunicationMessages* that can be sent or received by agents. In addition, the *InteractionProtocol* must comply with an *Ontology,* i.e. the vocabulary of the terms used in the message contents and their meaning (both the sender and the receiver must ascribe the same meaning to symbols for the communication to be

effective). The *InteractionProtocol* concept extends the UML metaclass Interaction. The CommunicationMessage concept extends the UML metaclass Message and can be of several types including REQUEST, INFORM and REFUSE, among others (defined as the enumeration class MessageKind). These are the defined by the Foundation for Intelligent Physical Agents [FIPA 2004] which indicate what the sender intends to achieve by sending the message.

A Profile has been defined in the UML 2.0 specification as a specific meta-modeling technique in which a stereotype defines how an existing metaclass may be extended. The intention of profiles is to give a straightforward mechanism for adapting an existing metamodel with constructs that are specific to a particular domain, platform, or method. For example, in our approach we have extended some UML metaclasses to address agent-oriented concepts. An extension (a kind of association) is used to indicate that the properties of a metaclass are extended through a stereotype. In Figure 3, we present some extensions we have made, such as the stereotype Agent extending the UML metaclass Class (from StructuredClasses package), the stereotype OrganizationalPort extending the UML metaclass Port, the stereotype Dependum extending the UML metaclass Interface, the stereotype Depender extending the UML metaclass Usage, the stereotype Dependee extending the UML metaclass InterfaceRealization and the stereotype AgentConnector extending the UML metaclass Connector.

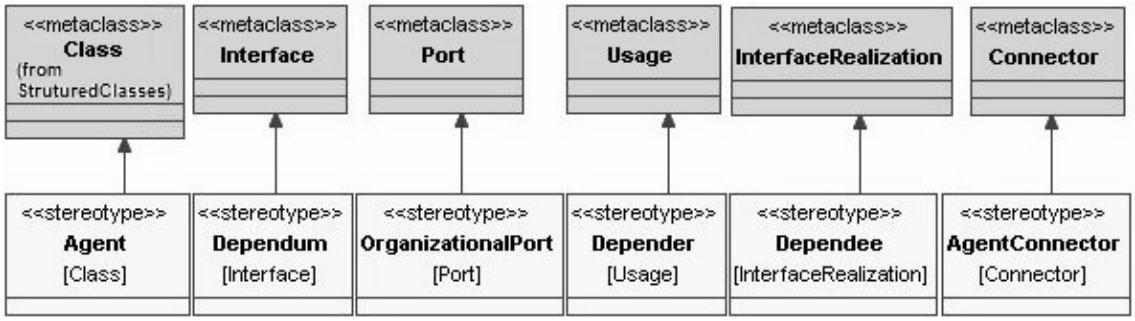


Figure 3. Agency stereotypes

3. Agent-Oriented Modeling

In this section, we present the MAS modeling diagrams specified according to our agency metamodel. These diagrams were conceived to model four views of MAS design: Architectural, Communication, Environmental and Intentional.

3.1. Architectural diagram

The architectural diagram reflects the client-server pattern [Shaw and Garlan 1996] tailored for MAS. It is defined in terms of agents which possess goals achievable by plans. Since an agent is not omnipotent, it needs to interact with other agents in order accomplish its responsibilities. An Agent possess OrganizationalPorts which enable the exchange of messages with other agents through AgentConnectors in order to accomplish some Dependum (i.e., service contract). For example, in Figure 4 we have the Provider agent which is responsible for performing the service defined in the Dependum. This agent aims at achieving the ServicePerformed goal by executing the PerformPlan plan, which, in turn, consists of performing the service() AgentAction. The Client agent aims at achieving the ServiceRequest goal by executing the RequestPlan

plan, which, in turn, consists of performing the request() AgentAction. Therefore, the Client agent is responsible for requesting the service defined in the Dependum. Both the message for requesting the service execution and the message for confirming whether the service was successfully concluded are sent through the AgentConnector.

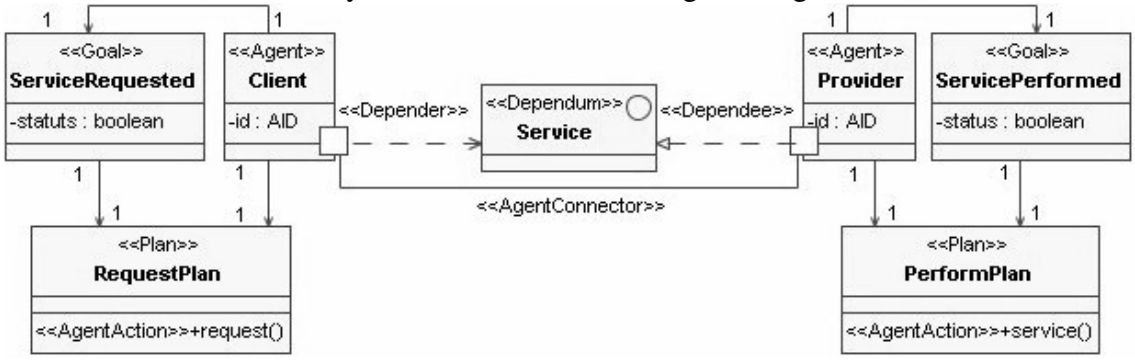


Figure 4. MAS Architectural Diagram

3.2. Communication diagram

The communication diagram is defined in terms of instances of agents and the messages exchanged between them to achieve a service providing. For example, the Figure 5 shows an interaction involving the Client and Provider agents. The Client sends a message requesting the execution of some service while the Provider sends a message informing the requested service has been performed. The interaction specified using the communication diagram is asynchronous.

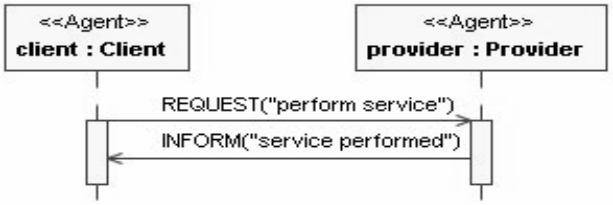


Figure 5. MAS Communication Diagram

3.3. Environmental diagram

The environmental diagram is defined in terms of agents composing an organization which is situated in an environment. This environment is composed of resources which are accessed by the agents according to their rights in order to accomplish their responsibilities. For example, in Figure 6 we have the *Provider* agent composing the *Org* organisation which is situated in the *Env* environment.

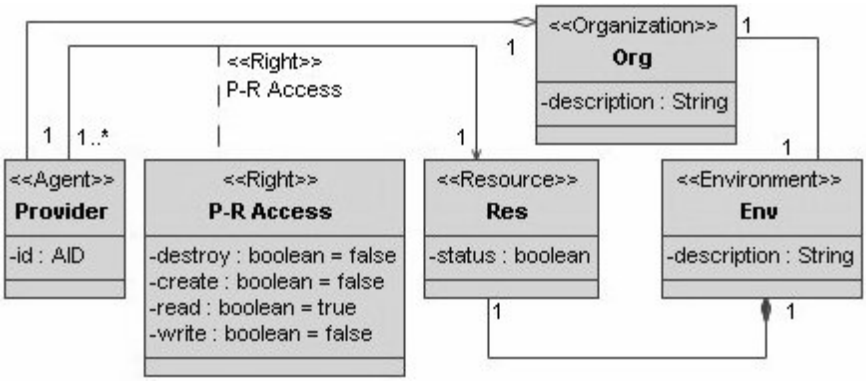


Figure 6. MAS Environmental Diagram

The Provider agent needs to access a *Res* resource available in the *Env* environment to fullfil its responsibilities. The *Provider* agent can only read the *Res* resource, according to its *P-R Access* right (read *Provider-Res Access* right)).

3.4. Intentional diagram

The intentional diagram is defined in terms of agents, their beliefs, goals, plans, as well as the norms and the ontology used in the organization. For example, in Figure 7 we have the *Provider* agent composing the *Org* organisation which must comply with the OrganizationalNorm norms. The Provider agent has a belief about if some request message has been received. Hence, the Request Received is a belief the Provider agent has.

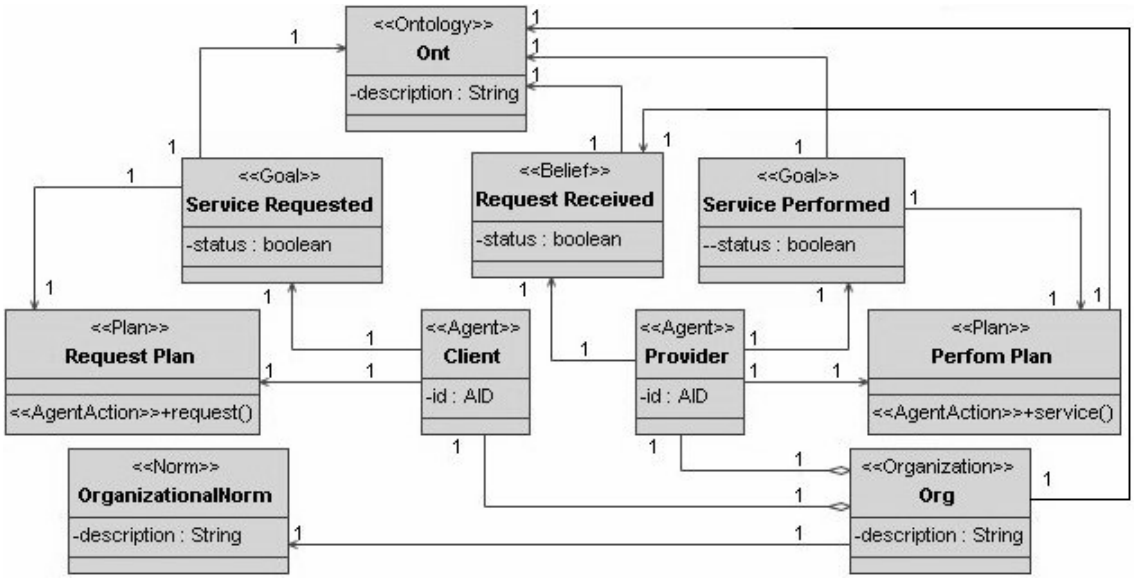


Figure 7. MAS Intentional Diagram

3.5. Mapping i* to UML at the architectural level

To support modeling and analysis during the architectural design phase, Tropos adopts the concepts and models offered by the *i** framework [Yu 1995]. However, the use of i* as an architectural description language is not suitable, since it does not support MAS architectural features, such as ports, connectors, protocols and interfaces. In section 3, we have provided a UML-based notation which supports MAS specification at the architectural level. In order to identify the information related to this notation, it is necessary to perform a means-end analysis [Yu 1995] of each actor which belongs to the MAS architecture represented in i*. This analysis helps identify the reasons/motivations associated with each dependency that an actor possesses. However, the process for performing the means-end analysis of the actors is a specialization of the original process, since we begin by establishing a main goal and from this goal we refine the actor responsibilities. The main goal is operationalized by one or more tasks through the means-end link. Each task corresponding to a means element in some means-end relationships must be further decomposed. The task can be subdivided into other tasks, (soft) goals and resources. Subtasks cannot be further decomposed. The end element in the means-end relationship can only be a (soft)goal.

In [Silva et al. 2006], we have presented some heuristics to map the i* concepts to both agency and UML-RT concepts [Selic and Rumbaugh 1998]. However, at that

stage we did not take into account the architectural concepts supported by UML 2.0. Hence, in this work, we redefine these heuristics to consider the MAS architectural concepts (Figure 4) extended from UML 2.0. The following heuristics guide the mapping of i* description of MAS architecture to our UML-based notation:

1. Each actor in the i* model becomes an «Agent» class in the architectural diagram.
2. Each dependum in the i* model becomes a «Dependum» interface in the architectural diagram. Observe that a «Dependum» can be of four types (goals, softgoals, tasks and resources) according to the agency metamodel Figure 2. This type is not provided explicitly in the model since it is a property of the model element.
3. Each depender in the i* model becomes a «Depender» dependency in the architectural diagram.
4. Each dependee in the i* model becomes a «Dependee» realization in the architectural diagram.
5. Each dependency (depender -> dependum -> dependee) in the i* model becomes a «Connector» association in the architectural diagram. Ports are added to the agents to enable the link through the connector.
6. Each resource related to the actor in the i* model becomes a «Resource» in the architectural diagram. It represents an environmental resource which the agent needs to access to perform its responsibilities. In this work we do not provide guidelines to define the agent rights to access the resources.
7. Each goal (or softgoal) in the i* models becomes a «Goal» in the architectural diagram. It represents the objectives the agent intends to accomplish.
8. Each task in the i* models becomes a «Plan» in the architectural diagram. It represents the means through which a goal is going to be achieved.
9. Each leaf task in the i* models becomes an «AgentAction» in the architectural diagram. It represents each step which composes a plan.
10. A «Belief» is some condition for performing a task (i.e, a «Plan» or an «AgentAction»). It represents the knowledge the agent has about both the environment and itself.
11. The «Organization» is the MAS the agent belongs to.

In fact, a preliminary mapping of i* concepts into agent, goal, belief, plan and resource concepts were originally proposed in [Castro et al. 2002]. However, here we did not provide a process or a notation to capture the mapped information. This new work, introduces a notation to be used in the MAS specification at the architectural level. Finally, we define a specialization of the original means-end analysis process which is appropriate for specifying the rationale of each agent before the mapping of the i* concepts to the agency concepts.

4. Case Study

To illustrate the usage of our approach, we consider the domain of conference management introduced in [Zambonelli et al. 2003] and modeled using the Tropos framework in [Silva et al. 2006]. A conference involves several individuals. During the

submission phase, *Authors* submit papers, and are informed that their papers have been received and have been assigned a submission number. In the review phase, the *Chair* has to handle the review of the papers by contacting potential *Reviewers* and asking them to review a number of papers according to their expertise. Eventually, reviews come in and are used to decide about the acceptance or rejection of the submissions. In the final phase, *Authors* need to be notified of these decisions and, in case of acceptance, will be asked to produce and submit a revised version of their papers. The *Publisher* has to collect these final versions and print the proceedings. Figure 8 presents the Conference Management System, a solution developed as an example of MAS for the conference management domain. The structure-in-5 architectural style [Kolp et al. 2002] has been chosen and applied to the MAS architectural design, but due to lack of space we do not show how we made the choice. Our focus is on the design of the MAS architecture according to agent modeling diagrams (Section 3).

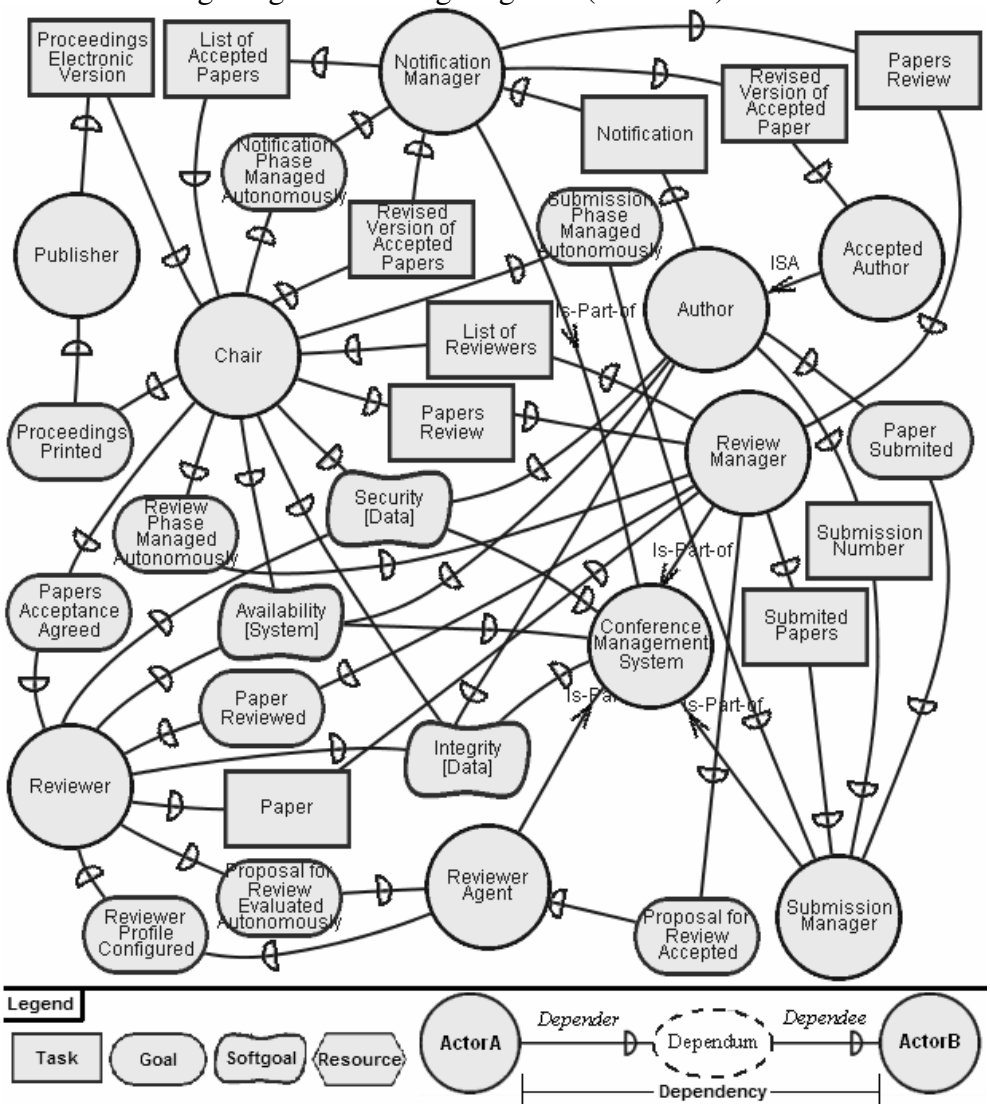


Figure 8. Conference Management System Architecture

The initial version of the Conference Management System supports the submission, review and notification phases of the conference process. The Conference Management System architecture is decomposed into four actors: Submission Manager, Review Manager, Notification Manager and Reviewer. Each of these actors is linked to

the system through an *is-part-of* relationship (see Figure 8). The Submission Manager is responsible for handling the submission phase of the process. The Review Manager is responsible for distributing the set of submitted papers to at least *n* reviewers according to their research area. The Notification Manager is in charge of handling the notification phase process. The Reviewer actor is responsible for evaluating a paper proposal according to the reviewer preferences and skills.

4.1. From i* to UML

The process presented in Section 3.5 is going to be used to produce MAS UML-based models at the architectural level in the context of Tropos. We begin by performing the means-end analysis for each actor which belongs to the MAS architecture described using the i* notation. Then, we rely on the mapping heuristics to specify each diagram presented in Section 3. For example, in our case study we perform the specialized means-end analysis of the Reviewer, Review Manager, Notification Manager and Submission Manager actors in order to capture their rationale when pursuing their goals and dependencies. The Review Manager expects to have *Papers Reviewed*. One alternative to satisfy this goal is to perform the *Manage Review Phase* task. This task is decomposed into four sub-tasks (see the refined model in Figure 9): *Collect Papers Review, Select "n" Reviewers of Paper Research Area, Propose Paper Review* and *Assign Paper Reviewer*.

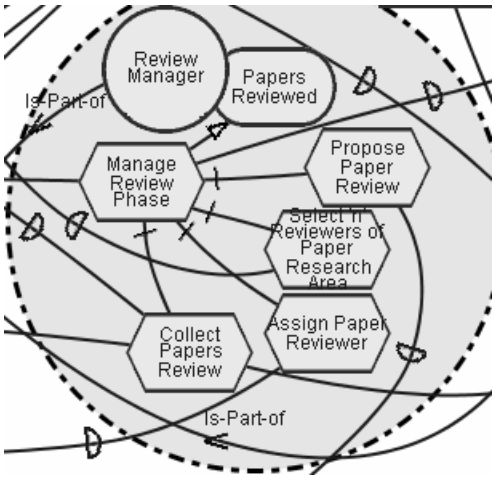


Figure 9. Means-end analysis for Review Manager

Analogously, the Reviewer actor expects to get hold of *Proposal for Review Evaluated* and, to achieve this goal, it has the alternative of performing the *Evaluate Proposal for Review* task. To carry out this task it is necessary to perform several subtasks: *Evaluate Relevance of Conference*, *Evaluate Time Availability, Evaluate Interest in Paper Subject* and *Set Personal Profile*. The Submission Manager actor is in charge of having the *Papers Submission Managed* and to accomplish this goal it has one alternative which is performing the *Manager Submission Phase* task. This task is decomposed into the *Assign Submission Number* and *Collect Paper Submission* sub-tasks. The Notification Manager actor expects to obtain the *Papers Notification Managed* and to reach this goal it has one alternative which is performing the *Manager Notification Phase* task. This task is decomposed into the *Notify Authors* and *Collect Revised Version of Accepted Papers* sub-tasks. The means-end analysis models of the Reviewer, Submission Manager and Notification Manager actors have been omitted here due to the lack of space

4.2. Architectural diagram

Having concluded the means-end analysis of Reviewer, Submission Manager, Notification Manager and Review Manager actors, we can now move on to identifying the properties that characterize that agent according to the MAS modeling diagrams (Section 3). The heuristics presented at Section 3.5 can be of some assistance to describe the Reviewer, Submission Manager, Notification Manager and Review Manager actors according to the architectural diagram (Figure 4). For example, the *Papers Reviewed* goal present in the means-end analysis of the Review Manager actor becomes a «Goal» associated to the Review Manager «Agent» class (colored area of Figure 10). The *Manage Review Phase* task becomes a «Plan» associated to both the Review Manager «Agent» class and Papers Reviewed «Goal» class. Each of the *Collect Papers Review, Select "n" Reviewers of Paper Research Area, Propose Paper Review* and *Assign Paper Reviewer* tasks becomes an «AgentAction» in the Manage Review Phase «Plan» class.

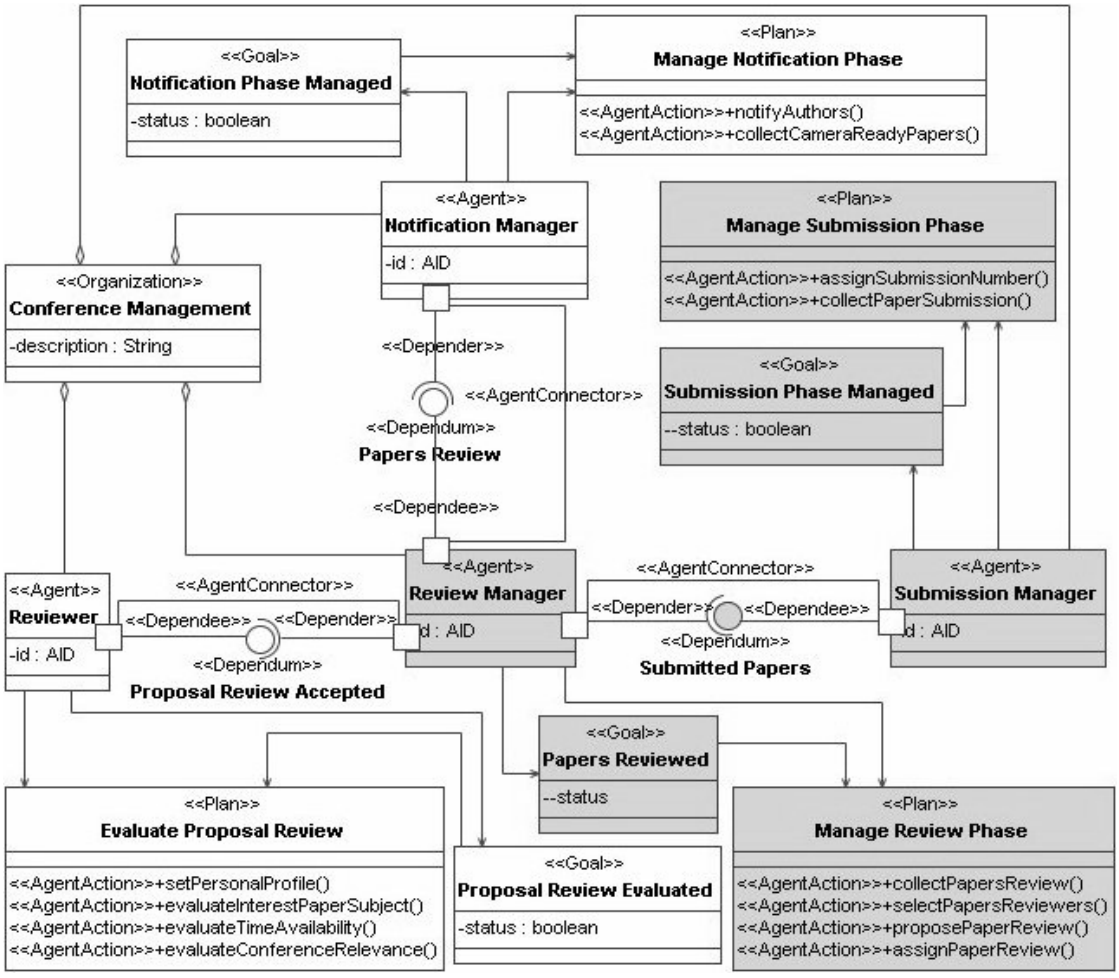


Figure 10. Conference Management Architectural Diagram

Each dependum of the Review Manager actor's dependencies becomes a «Dependum» interface. For example, the dependum of the *Submited Papers* resource dependency, between Review Manager and Submission Manager agents, becomes a *Submited Papers* «Dependum» interface, typed as a resource (colored area of Figure 10). The depender and dependee roles of the *Submited Papers* dependency become the

«Depender» dependency from Review Manager «Agent» class and the «Dependee» realization from Submission Manager «Agent» class, respectively. The *Submited Papers* resource dependency itself becomes a «Connector» between Review Manager and Submission Manager «Agent» classes. Ports are added to these «Agent» classes to enable the link through the «Connector». The same guidelines are applied to the other actors present in the i* model. As a result several classes are incorporated in the MAS architectural diagram depicted in Figure 10.

The architectural design of the Conference Management system (Figure 10) is performed by using the MAS architectural pattern depicted in Figure 4 to assign the system's responsibilities to the architectural components. The Conference Management system is composed of four agents: Submission Manager, Review Manager, Reviewer and Notification Manager. For example, in Figure 10 the colored area corresponds to the interaction between the Review Manager and Submission Manager agents to achieve the service Submitted Papers. The Review Manager agent intends to achieve the Papers Reviewed goal by means of the Manage Review Phase plan. However, to get the papers reviewed the Review Manager agent has to request the Submission Manager agent to perform the Submitted Papers service. This service provides to the Review Manager agent the submitted papers collected by the Submission Manager agent. The Submission Manager performs the requested service because it does not conflict with the achievement of the Submission Phase Managed goal. Hence, both the requested service and the goal achievement are accomplished by means of the Manage Submission Phase plan. The description of the Proposal Review Accepted and Papers Review services is achieved in a similar way.

4.3. Communication diagram

The communication diagram is defined in terms of instances of agents and the messages exchanged between them to achieve their responsibilities. For example, Figure 11 shows an interaction involving the Notification Manager, the Review Manager, the Reviewer and the Submission Manager agents.

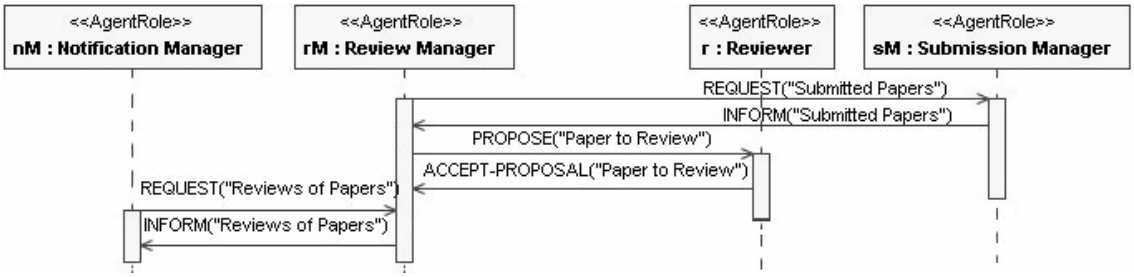


Figure 11. Conference Management Communication Diagram

The interaction specified using the communication diagram is asynchronous. Hence, the Review Manager agent sends a message requesting the submitted papers which are going to be provided through the execution of Submitted Papers service. Then, the Submission Manager answers by informing if the requested service has been performed successfully. The Review Manager agent sends a message to the Reviewer agent proposing a paper to review. If the Reviewer agent answers by accepting the proposal to review the paper, then the Proposal Review Accepted service is assumed to be achieved. The Notification Manager agent, in turn, sends a message to the Review Manager agent requesting the reviews of the papers which are going to be provided

through the execution of Papers Review service. Then, the Review Manager answers by informing if the requested service has been performed successfully.

4.4. Environmental diagram

The environmental diagram is defined in terms of agents composing an organization which is situated in an environment composed by resources which are accessed by the agents according to its rights. The heuristics presented in Section 3.5 continue to be used here to identify the properties which characterize that agent according to this diagram. Hence, all *Submited Papers*, *Papers Review*, *List of Reviewers* resource elements related to the Review Manager actor (Figure 9) become a «Resource» associated to the Review Manager «Agent» class in the environmental diagram presented in the Figure 12. In this diagram we have the *Review Manager* and *Submission Manager* agents composing the *Conference Management* organisation which is situated in the *Conference Management* environment. The Notification Manager and Reviewer agents have been omitted here due to the lack of space.

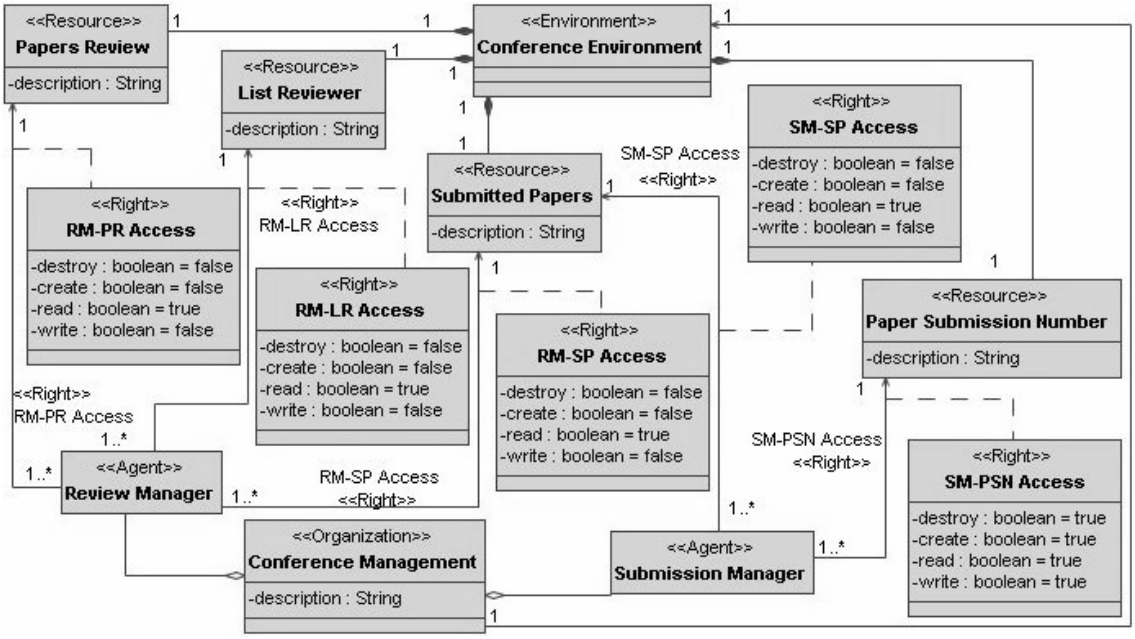


Figure 12. Conference Management Environmental Diagram

The *Submission Manager* agent needs to access the *Submitted Papers* resource available at the *Conference Management* environment to perform the Manage Submission Phase plan. The *Submission Manager* agent can only read the *Submitted Papers* resource, according to its *SM-SP Access* right (read *Submission-Manager-Submitted Papers Access* right). The *Review Manager* agent needs to access a *Papers Review* resource available in the *Conference Management* environment to perform the Manage Review Phase plan. The *Review Manager* agent can only read the *Papers Review* resource, according to its *RM-PR Access* right (read *Review Manager-Papers Review Access* right). The *Review Manager* agent also needs to access a *Submitted Papers* resource to perform the Manage Review Phase plan. His permission is for only reading the *Submitted Papers* resource, according to its *RM-SP Access* right (read *Review Manager-Submitted Papers Access* right). The description of the agent's rights to access the other resources is achieved in a similar way.

4.5. Intentional diagram

The intentional diagram is defined in terms of agents, their beliefs, goals, plans, norms and ontology. The heuristics presented in Section 3.5 are used here to identify the properties which characterize that agent according to this diagram. Hence, the condition to perform the *Collect Papers Review* task (Figure 9) is that the Review Deadline should have passed. The circumstance to perform the *Select "n" Reviewers of Paper Research Area, Propose Paper Review* and *Assign Paper to Reviewer* tasks is that the Submission Deadline should have passed. Each of these conditions becomes a «Belief» associated to the Review Manager «Agent» class. However, the intentional diagram is not shown in this paper because we have not defined yet the heuristics to derive both the MAS ontology and norms. These issues will be addressed in future work.

5. Related Work

Several languages for MAS modeling have been proposed in the last few years, such as AUML [Odell et al. 2000], MAS-ML [Silva and Lucena 2004] and SKwyRL-ADL [Mouratidis et al. 2005].

The work presented in [Mouratidis et al. 2005] proposes a metamodel to define an architectural description language (ADL) to specify secure MAS. In particular SKwyRL-ADL includes an agent, a security and an architectural model and aims at describing secure MAS, more specifically those based on the BDI (belief-desire-intention) model [Rao and Georgeff 1995]. Moreover, the Z specification language is used to formally describe SkwyRL-ADL concepts. Our notation to model MAS also supports the BDI agent model. Furthermore, we also define a process to use the proposed notation in the MAS architectural design.

The proposal of a multi-agent system modeling language called MAS-ML is presented in [Silva and Lucena 2004]. It extends the UML metamodel according to the TAO (Taming Agents and Objects) metamodel concepts [Silva and Garcia et al. 2003]. TAO provides an ontology that defines the static and dynamic aspects of MAS. The MAS-ML includes three structural diagrams – Class, Organization and Role diagrams – which depict all elements and all relationships defined in TAO. The Sequence diagram represents the dynamic interaction between the elements that compose a MAS — i.e., between objects, agents, organizations and environments. However, this approach does not provide a detailed process for guiding the use of that modeling language in MAS development as we do.

AUML [Odell et al. 2000] provides extensions of UML, including representation in three layers of agent interaction protocols, which describe the sequence of messages exchanged by agents as well as the constraints in messages content. However, AUML does not provide extensions to capture the agent's cognitive map (individual structure) or the agent's organisation (system structure). We provide UML-based diagram to capture the agent internal structure and the MAS structure, as well as a detailed process for guiding the use of these diagrams in MAS modeling.

Summarizing, we are concerned with detailing the MAS architectural design by providing a standard notation and process to guide the specification of MAS architecture. The notation we have proposed here captures both the static and dynamic architectural agent features and considers the intentions associated with each agent communication protocol.

6. Conclusions and Future Work

This paper focuses on the Tropos architectural design phase and aims at providing a notation for designing MAS as well as a process to describe MAS architecture using such notation. To achieve this, we have defined an extension of UML metamodel to support agency features to provide a notation for specifying MAS design. This notation supports the specification of architectural features in the context of multi-agent systems. Moreover, we outline a process to guide the description of agents according to our notation in the context of the Tropos framework. We also applied our approach to a Conference Management System to illustrate its feasibility.

For future work we plan to investigate whether other UML 2.0 diagrams are useful for designing MAS. For example, the work presented in [Silva et al. 2005] could be used to complement our approach to model agent plans and actions using UML 2.0 activity diagrams in Tropos. We also need to improve the heuristics to (i) derive the system ontology from i* models and (ii) describe the MAS organizational norms. Applying our approach to other case studies is required to address these open issues.

Acknowledgements

This work was supported by several research grants (CNPq Proc. 304982/2002-4, CAPES Proc. BEX 1775/2005-7, Proc. BEX 3003/05-1, Proc. BEX 3014/05-3, Proc. BEX 3478/05-0 & CAPES/ GRICES Proc. 129/05).

8. References

Bellifemine, F., Caire, G., Poggi, A., Rimassa, G. (2003) "JADE - A White Paper**",** Special issue on JADE of the TILAB Journal EXP.

Braubach, L., Pokahr, A., Lamersdorf, W. (2004) "Jadex: A Short Overview", In 5th Annual International Conf. on Object-Oriented and Internet-based Technologies, Concepts, and Applications for a Networked World (Net.ObjectDays'04), AgentExpo

Castro, J. Kolp, M., Mylopoulos, J. (2002) "Towards Requirements-Driven Information Systems Engineering: The Tropos Project", Information Systems Journal. Volume 27. Elsevier, p. 365 – 89.

FIPA (2004) FIPA (The Foundation for Intelligent Physical Agents), Available: http://www.fipa.org

Giorgini, P., Kolp, M., Mylopoulos, J., Castro, J. (2005) "Tropos: A Requirements-Driven Methodology for Agent-Oriented Software", In Henderson-Sellers, B. et al. (eds.): Agent-Oriented Methodologies. Idea Group, p. 20 – 45.

Kolp, M., Giorgini, P., Mylopoulos, J. (2002) "Information Systems Development through Social Structures", In 14th International Conference on Software Engineering and Knowledge Engineering (SEKE), Ischia, Italy.

Minsky, N and Muarata, T. (2004) "On Manageability and Robustness of Open Multi-Agent Systems", In: Lucena, C. et al. (eds.): Soft. Eng. for Multi-Agent Systems II: Research Issues and Practical App.. LNCS, Vol. 2940, Springer-Verlag, p. 189 – 206.

Mouratidis, H., Faulkner, S., Kolp, M., Giorgini, P. (2005) "A Secure Architectural Description Language for Agent Systems", In 4th Autonomous Agents and Multi-Agent Systems (AAMAS'05). Uthrecht, The Netherlands.

Odell, J., Parunak, H. V. D, Bauer, B. (2000) "Extending UML for agents", In Proc. of the 2nd Int. Bi-Conf. Workshop on Agent-Oriented Information Systems at the 17th National Conf. on Artificial Intelligence, Austin, USA. iCue Publishing, p. 3 – 17.

Rao, A. S. and Georgeff, M. P. (1995) "BDI agents: from theory to practice", Technical Note 56, Australian Artificial Intelligence Institute.

Selic, B. and Rumbaugh, J. (1998) "Using UML for Modeling Complex Real - Time Systems", Rational Whitepaper, Available: www.rational.com.

Silva, C., Castro, J., Mylopoulos, J. (2003) "Detailing Architectural Design in Requirements Driven Software Development: The Tropos Case", In Proceedings of the XVII Simpósio Brasileiro de Engenharia de Software, Manaus, Brasil, p. 85 – 93.

Silva, V., Garcia, A., Brandão, A., Chavez, C., Lucena, C., Alencar, P. (2003) "Taming Agents and Objects in Software Engineering", In: Garcia, A. et al. (eds.): Soft. Eng. for Large-Scale Multi-Agent Systems. LNCS, Vol. 2603. Springer-Verlag, p. 1 – 25.

Silva, V. and Lucena, C. (2004) "From a Conceptual Framework for Agents and Objects to a Multi-Agent System Modeling Language", In: Sycara, K. et al. (eds.): Journal of Autonomous Agents and Multi-Agent Systems. Kluwer Academic Publishers, 9, 1-2, p. 145 – 189.

Silva, C., Tedesco, P., Castro, J., Pinto, R. (2004) "Comparing Agent-Oriented Methodologies Using a NFR Approach", In Proc. of the 3rd Software Engineering for Large-Scale Multi-Agent Systems (SELMAS'04). Edinburgh, Scotland, p. 1 – 9.

Silva, V., Noya, R., Lucena, C. (2005) "Using the UML 2.0 activity diagram to model agent plans and actions", In 4th Autonomous Agents and Multi-Agent Systems (AAMAS'05). Uthrecht, The Netherlands, p. 594 – 600.

Silva, C., Castro, J., Tedesco, P., Araújo, J., Moreira, A., Mylopoulos, J. (2006) "Improving the Architectural Design of Multi-Agent Systems: The Tropos Case", In 5th Soft. Engineering for Large-Scale Multi-Agent Systems at ICSE'06 (to appear).

Shaw, M. and Garlan, D. (1996) Software Architecture: Perspectives on an Emerging Discipline. Prentice Hall.

Sommerville, I. (2001) Software Engineering – Ed.6. Addison Wesley.

OMG (2004) Meta Object Facility (MOF) 2.0 Core Specification, Available: http://www.omg.org/docs/ptc/04-10-15.pdf.

OMG (2005) Unified Modeling Language (UML): Superstructure. Version 2.0, Available: www.omg.org/docs/formal/05-07-04.pdf.

Wooldridge, M. (2002) An Introduction to Multiagent Systems. John Wiley and Sons, Ltd. England, p. 15 – 103.

Yu, E. (1995) Modelling Strategic Relationships for Process Reengineering. Ph.D. thesis. Department of Computer Science. University of Toronto, Canada.

Zambonelli, F., Jennings, N., Wooldridge, M. (2003) "Developing Multiagent Systems: the Gaia Methodology", In ACM Transactions on Software Engineering and Methodology, 12, 3, p. 317 – 370.

Specification of Real-Time Systems with Graph Grammars

Leonardo Michelon[1], **Simone André da Costa**[1 2], **Leila Ribeiro**[1]

[1]Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Porto Alegre – RS – Brazil

[2]Universidade do Vale do Rio dos Sinos (UNISINOS)
São Leopoldo – RS – Brazil

lmichelon@inf.ufrgs.br, scosta@unisinos.br, leila@inf.ufrgs.br

Abstract. *This paper presents a formal approach to specify and analyze real-time systems. We extend Object-Based Graph Grammars, a description technique suitable for the specification of asynchronous distributed systems, to be able to explicitly model time constraints. The semantics of the systems is defined in terms of Timed Automata, allowing the automatic verification of properties.*

Resumo. *Este artigo apresenta uma abordagem formal para a especificação e análise de sistemas de tempo real. Gramáticas de Grafos Baseadas em Objetos são extendidas incluindo primitivas para modelar explicitamente restrições de tempo.. A semântica é definida em termos de autômatos temporais, provendo um método para verificação automática de propriedades.*

1. Introduction

One of the goals of software engineering is to aid the development of correct and reliable software systems. Formal specification methods play an important role in accomplishing this goal [Clarke and Wing 1996]. Besides providing means to prove that a system satisfies the required properties, formal methods contribute to its understanding, revealing ambiguities, inconsistencies and incompletions that could hardly be detected otherwise.

The use of a formal specification method is even more important in the design of real-time systems, frequently used in critical security environments. A real-time system is a system in which performance depends not only on the correctness of single actions, but also on the time at which actions are executed [Stankovic et al. 1996, Stankovic 1988]. Application areas that typically need real-time models include railroad systems, intelligent vehicle highway systems, avionics, multimedia and telephony. To assure that such systems are correct, additionally to prove that they provide the required functionality, we have to prove that the time constraints are satisfied.

1.1. Formal Specification of Real-Time Systems

There are many formal approaches to model real-time systems. Timed Automata [Alur and Dill 1994, Bengtsson and Yi 2004] is one of the most proiminent methods for real-time specification. A timed automaton extends a usual automaton by adding several clocks to states and time restrictions to transitions (and states). It enables us to specify both the discrete behavior of control and the continuous behavior of time. Process calculi models including time [Lee et al. 1994, Baeten and Middelburg 2000, Bowman and Derrick 1997] have also been proposed, adding a set of timing operators to

process algebras. They offer a level of abstraction based on processes: a system is viewed as a composition of (interacting) processes. Timed Petri nets [Walter 1983] and time Petri nets [Berthomieu and Diaz 1991] are extensions of the classical Petri nets adding time values to transitions/tokens or time intervals to transitions, respectively.

Although the models discussed above may be adequate for some aspects of a system, they stress the representation of the control structure, lacking a comprehensive representation of data structure and its distribution within a system. Object-oriented models provide such abstraction by joining descriptions of data and processes within one object. Distribution and concurrency appear naturally by viewing objects as autonomous entities. Object-oriented approaches are widely accepted for specification and programming. Thus, various Unified Modeling Language (UML)-based approaches have already been proposed to model time information. HUGO/RT [Knapp et al. 2002] is an automated tool that checks if a UML state machine interacts according to the scenarios specified by a sequence diagram (extended with time constraints). For this verification, state machines are compiled into Timed Automata and sequence diagrams into Observer Timed Automaton. After the translations, the model-checker Uppaal [Behrmann et al. 2004] is called to check if the Observer Timed Automaton describes a reachable behavior of the system. [Diethers and Huhn 2004] proposes a similar approach through the Vooduu tool. Nevertheless, these tools restrict the specification of a real-time system. First of all, because they are based on UML state machines, that only support *after* and *when* time constructors. They do not offer clocks or priorities, useful concepts for real-time modeling. Besides, even though the proposals intent to be faithful to the UML informal specification, following its semantic requirements, the translation of state machines is not based on a formal semantic. In [Lavazza et al. 2001] a translation of timed state machines into a real-time specification language TRIO was proposed, but TRIO is not directly model checkable.

The approach in [Konrad et al. 2004] adds time information to UML classes. Attributes of type *Timer*, for the definition of clocks for classes, and a notation similar to timed automata, to analyze and evaluate clocks in UML state diagrams, are syntactically introduced. A translation from UML into Promela, the input language of the SPIN model-checker, is extended to give semantics to the diagrams. The main difficulty in using this approach is that, since Promela does not have built-in time constructors, clocks and time constrains have to be encoded and, since the semantics is defined by the Promela code, it might be quite difficult to understand for users not very familiar with this language.

The Omega project also aims to model and verify real-time systems. [Graf et al. 2003] proposes an extension of a UML subset with timing constructs. In [Ober et al. 2004], part of UML is mapped into Communication Extended Timed Automata (input language for the validation tool). Static properties are described as Observer Automata and dynamic properties by UML Observers. This is a very interesting approach, although the user has to learn a variety of different languages and diagrams to completely specify and verify a system.

1.2. Our Contribution

In this paper, we propose extending the formal description technique Object-Based Graph Grammars (OBGGs) [Dotti and Ribeiro 2000, Ribeiro et al. 2005] to specify real-time systems. OBGG is a visual formal specification language suitable for the specification

of asynchronous distributed systems. The basic idea of this formalism is to model the states of a system as graphs and describe the possible state changes as rules (where the left- and right-hand sides are graphs). The behavior of the system is then described via applications of these rules to graphs describing the actual states of a system. Rules operate locally on the state-graph, and therefore it is possible that many rules are applied at the same time. OBGGs are appealing as specification formalism because they are formal, they are based on simple but powerful concepts to describe behavior, and at the same time they have a nice graphical layout that helps non-theoreticians understand an object-based graph grammar specification. Due to the declarative style (using rules), concurrency arises naturally in a specification: if rules do not conflict (do not try to update the same portion of the state), they may be applied in parallel (the specifier does not have to say explicitly which rules shall occur concurrently).

OBGGs can be analyzed through simulation [Duarte et al. 2002] and verification (using the SPIN model-checker) [Dotti et al. 2003]. Compositional verification (using an assume-guarantee approach) is also provided [Ribeiro et al. 2006]. Moreover, there is an extension of OBGGs to model inheritance and polymorphism [Ferreira 2005]. However, OBGGs do not provide explicit time constructs, and therefore are not suited to model and analyze real-time systems. Here we propose a mapping from a timed extension of OBGG specifications to Timed Automata. This way, we can use the available (Timed Automata) verification tools to check properties of timed OBGGs.

Our approach adds time stamps to the messages (allowing to program certain events to happen in the future), extends the appealing formal description technique OBGG, and supports verification of properties written in temporal logic. The main contributions of this paper are: (i) the proposal of a timed extension to OBGG; and (ii) the translation of the extended OBGG to Timed Automata, leading to a timed semantics of OBGGs. The paper is organized as follows: Section 2 presents OBGGs and its timed extension; Section 3 reviews Timed Automata; the semantics of timed OBGGs is described in Section 4; Section 5 analyzes the example; and final remarks are in Section 6.

2. Object Based Graph Grammars

Object based graph grammar (OBGG) is a formal visual language suited to the specification of object-based systems. We consider object-based systems with the following characteristics: (i) a system is composed of various objects. The state of each object is defined by its attributes, which may be pre-defined values or references to other objects. An object cannot read or modify the attributes of other objects; (ii) objects are instances of classes. Each class includes the specification of its attributes and of its behavior; (iii) objects are autonomous entities that communicate asynchronously via message passing. An OBGG specifies a system in terms of states and changes of states, where states are described by graphs and changes of states are described by rules.

Following, the definitions used for the description of Timed Object-Based Graph Grammars (TOBGGs) are presented. Each formal definition is preceded by an informal description of its meaning. The formal definitions are necessary to follow the translation of TOBGG to Timed Automata, described in Section 4. For the comprehension of the other contributions of this paper, it is possible to skip the formal definitions. Due to space limitations, the examples are in Subsection 2.2.

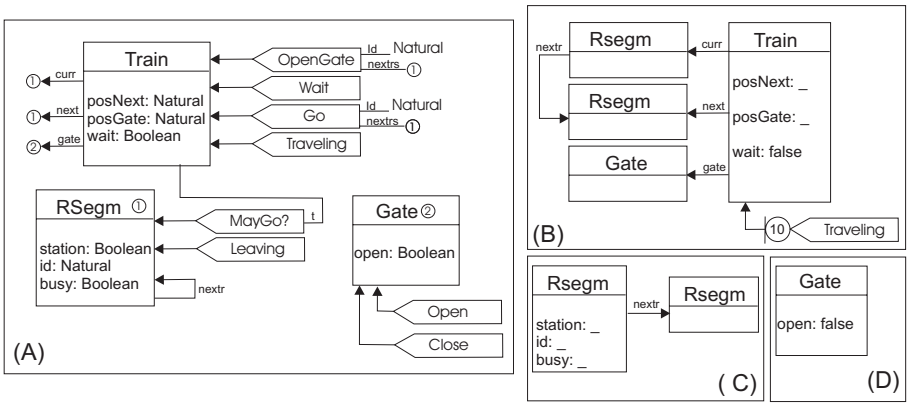


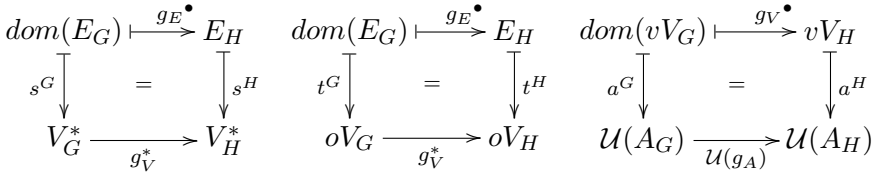
Figure 1. (A) Type Graph, (B) Train Initial Graph Template, (C) RSegm Initial Graph Template and (D) Gate Initial Graph Template

Graph, Graph Morphism. A *graph* consists of a set of vertices partitioned into two subsets, of objects and values (of abstract data types), and a set of edges partitioned into sets of message and attribute edges. Values are allowed as object attributes and/or message parameters. Messages, modeled as (hyper)edges, may also have other objects as parameters and must have a single target object. These connections are expressed by total functions assigning to each edge its source and target vertices. Figure 1(A) illustrates a graph for an object-based system. Values of abstract data type Natural are allowed, for example, as attributes of the Train object and as parameters of the OpenGate message. The Train object also has references to Gate and RSegm objects as attributes. Message OpenGate has the Train object as target and as sources a reference to RSgm object and Natural values. A *graph morphism* expresses a structural compatibility between graphs: if an edge is mapped, the corresponding vertices, if mapped, must have the same sources/target vertex; if a vertex is mapped, the attributes must be the same.

Let $f : A \rightarrow B$ be a partial function , $f^\bullet$ denotes the function $f^\bullet : dom(f) \rightarrow B$, s.t, $f(x) = f^\bullet(x),\ \forall x \in dom(f)$. If $Spec$ is an algebraic specification, $\mathcal{U} : \mathbf{Alg(Spec)} \rightarrow \mathbf{Set}$ is the forgetful functor that assigns to each algebra the disjoint union of its carrier sets. It is assumed that the reader is familiar with basic notions of algebraic specification (see, e.g., [Ehrig and Mahr 1985]).

DEFINITION 1 (GRAPH, GRAPH MORPHISM) *Given an algebraic specification $Spec$, a* ***graph*** $G = (V_G, E_G, s^G, t^G, A_G, a^G)$ *consists of a set V_G of* vertices *partitioned into sets oV_G and vV_G (of objects and values, respectively), a set E_G of* (hyper)edges *partitioned into sets mE_G and aE_G (of messages and attributes, respectively), a total* source *function $s^G : E_G \rightarrow V_G^*$, assigning a list of vertices to each edge, a total* target *function $t^G : E_G \rightarrow oV_G$ assigning an object-vertex to each edge, an attribution function $a^G : vV_G \rightarrow \mathcal{U}(A_G)$, assigning to each value-vertex a value from a carrier set of A_G.*

A ***(partial) graph morphism*** *$g : G \rightarrow H$ is a tuple (g_V, g_E, g_A), where the first components are partial functions $g_V = g_{oV} \cup g_{vV}$ with $g_{oV} : oV_G \rightarrow oV_H$ and $g_{vV} : vV_G \rightarrow vV_H$ and $g_E = g_{mE} \cup g_{aE}$ with $g_{mE} : mE_G \rightarrow mE_H$ and $g_{aE} : aE_G \rightarrow aE_H$; and the third component is a total algebra homomorphism such that the diagrams below commute. A morphism is called total if all components are total. This category of graphs and partial graph morphisms is denoted here by* ***GraphP*** *(identities and composition are defined componentwise).*



OB-Graphs. An *OB-graph* is a graph equipped with a morphism $type$ to a fixed graph of types [Corradini et al. 1996]. Since types constitute the static part of the definition of a class, we call the graph of types as *class graph*. Two restrictions are imposed to a *class graph* to guarantee that it corresponds to a class in the sense of the object paradigm: the first is that there are no data values in a class graph (they are represented by the name of data types); and the second imposes that each class can have exactly one list of attributes. A morphism between OB-graphs is a partial graph morphism that preserves the typing.

DEFINITION 2 (OB-GRAPHS) *Let $Spec$ be a specification. A graph C is called a* ***class graph*** *iff (i) A_C is a final algebra*[1] *over $Spec$, (ii) for each object vertex $v \in oV_C$ there is exactly one attribute hyperedge ($ae \in aE_C$) with target v. An* ***OB-graph over*** *C is a pair $OG^C = (OG, type^{OG})$ where OG is an graph called* ***instance graph*** *and $type^{OG} : OG \to C$ is a total graph morphism, called the* ***typing morphism****. A morphism between OB-graphs OG_1^C and OG_2^C is a partial graph morphism $f : OG_1 \to OG_2$ such that for all $x \in dom(f)$, $type^{OG_1}(x) = type^{OG_2} \circ f(x)$. The category of OB-graphs typed over a class graph C, denoted by* ***OBGraph(C)****, has OB-graphs over C as objects and morphisms between OB-graphs as arrows (identities and composition are the identities and composition of partial OB-graph morphisms).*

The operational behavior of the system described by a graph grammar is determined by the application of grammar rules to the graphs that represent the states of the system (starting from an initial state).

Rule. A rule of an object-based grammar consists of (the numbers in parenthesis at the end of each item correspond to conditions in Definition 3):

- *a finite left-hand side L*: describes the items that must be present in a state to enable the application of the rule. The restrictions imposed to left-hand sides of rules are:
  - There must be exactly one message vertex, called trigger message – this is the message handled/deleted by this rule (cond. 2).
  - Only attributes of the target object of the trigger message should appear – not all the attributes of this object should appear, only those necessary for the treatment of this message (cond. 3).
  - Values of abstract data types may be variables that will be instantiated at the time of the application of the rule. Operations defined in the abstract data type specification may be used (cond. 6 and 7).
- *a finite right-hand side R*: describes the items that will be present after the application of the rule. It may consist of:
  - Objects and attributes present in the left-hand side of the rule as well as new objects (created by the application of the rule). The values of attributes may change, but attributes cannot be deleted (cond. 4 and 5).
  - Messages to all objects appearing in R.

[1] An algebra in which each carrier set is a singleton.

- a condition: this condition must be satisfied for the rule to be applied. This condition is an equation over the attributes of left- and right-hand sides.

DEFINITION 3 (RULE) *Let C be a class graph, $Spec$ be a specification and X be a set of variables for $Spec$. A rule is a pair (r, Eq) where Eq is a set of equations over $Spec$ with variables in X and $r = (r_V, r_E, r_A) : L \rightarrow R$ is a C-typed OB-graph morphism s.t.*

1. *L and R are finite;*
2. *a message is deleted: $\exists! e \in mE_L$, called* trigger(r)*, $trigger(r) \notin dom(r_E)$;*
3. *only attributes of the target of the message may appear in L: $(aE_L = \emptyset) \vee ((\exists! e \in aE_L) \wedge t^L(e) = t^L(trigger(r)))$;*
4. *attributes of existing objects may not be deleted nor created: $\forall o \in oV_L.(\exists e \in aE_L.t^L(e) = o \Rightarrow \exists e' \in aE_R.t^L(e') = r_V(o))$;*
5. *objects may not be deleted: $\forall o \in oV_L.o \in dom(r_V)$;*
6. *the algebra of r is a quotient term algebra over the specification $Spec^{Nat}$ including a set of equations Eq and variables in X;*
7. *attributes appearing in L may only be variables of X: $\forall v \in vV_L.a^L(v) \in X$;*
8. *the algebra homomorphism component of r (r_A) is the identity (rules may not change data types).*

We denote by $Rules(C)$ the set of all rules over a class graph C.

Object-Based Graph Grammar. An object-based system is composed of:

- a *type graph*: a graph containing information about all the attributes of all types of objects involved in the system and messages sent/received by each kind of object.
- a *set of rules*: these rules specify how the objects will behave when receiving messages. For the same kind of message, we may have many rules. Depending on the conditions imposed by these rules (on the values of attributes and/or parameters of the message), they may be mutually exclusive or not. In the latter case, one of them will be chosen non-deterministically to be executed. The behavior of an object when receiving a message is specified as an atomic change of the values of the object attributes together with the creation of new messages to any objects.
- an *initial graph*: this graph specifies the initial values of the attributes of the objects, as well as messages that must be sent to these objects when they are created. The messages in this graph can be seen as triggers of the execution of the object.

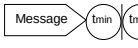
DEFINITION 4 (OBJECT-BASED GRAPH GRAMMAR (OBGG)) *An **object-based graph grammar** is a tuple $OBGG = (Spec, X, C, IG, N, n)$ where $Spec$ is an algebraic specification, X is a set of variables, C is a class graph, IG is a C-typed graph, called **initial or start graph**, N is a set of rule names, $n : N \rightarrow Rules(C)$ assigns a rule to each rule name.*

2.1. Timed OBGG

Originally, the Graph Grammars formalism does not include the concept of time. Here we will incorporate time to the model in order to model real-time systems. There are several choices of where to put the time in a graph grammar: rules, messages, and objects. We have decided to put time stamps on the messages describing when they are to be delivered/handled. In this way, we can program certain events to happen at some specific

time in the future. Rules do not have time stamps, that is, the application of a rule is instantaneous. Moreover, we adopt relative time: time stamps are not to be understood as absolute time specifications of when an event should occur, but rather as an interval of time relative to the current time in which the event should occur.

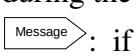
Syntactically a Timed Object Based Graph Grammar (TOBGG) is an OBGG with an additional time representation at the messages. The time stamps of the messages have the format: $\langle tmin, tmax \rangle$, with $tmin \leq tmax$, where $tmin$ and $tmax$ are the minimum/maximum number of time units, relative to the current time, within which the message should be handled. The possible time-stamps are:

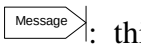
- 

: this message must be handled in at least $tmin$ time units and at most in $tmax$ time units, i.e., in interval $[tmin + current\ time,\ tmax + current\ time]$.
- 

: if $tmin$ is omitted, the current time is assumed as the minimum time for this message, i.e., the message must be handled in interval $[0 + current\ time,\ tmax + current\ time]$.
- 

: if $tmax$ is omitted, infinite is assumed (i.e., this message has no time limit to be delivered). It must be handled in interval $[tmin + current\ time,\ +\infty)$.
- 

: if $tmax = tmin$, this message must be handled in a specific time during the simulation, i.e., in interval $[t + current\ time,\ t + current\ time]$.
- 

: if $tmin$, $tmax$ and the bar $|$ are omitted, the message can be delivered from the current time on, and has no time limit to be handled, i.e., it must be handled in interval $[0 + current\ time,\ +\infty)$.
- 

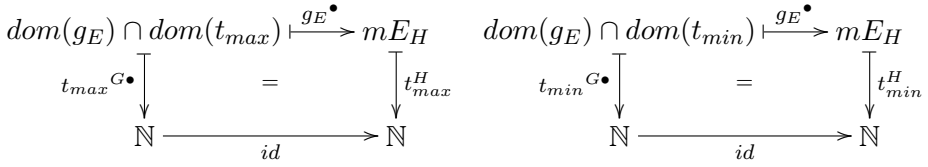
: this notation is equivalent to having $tmin = tmax = t$, with t being the current time. This means this message must be handled immediately, i.e., in interval $[current\ time,\ current\ time]$.

Now, we will define timed OBGGs, short TOBGGs using (typed and attributed) hypergraphs. Let $Spec^{Nat}$ denotes an algebraic specification including sort Nat and the usual operations and equations for natural numbers.

Timed Graph, Timed Graph Morphism. A *timed graph* consists of a graph with two partial functions, which assign a minimum/maximum time to each message. A *timed message* has the minimum/maximum time defined. A *(partial) timed graph morphism* is a graph morphism that is compatible with time, that is, if a timed message is mapped, the time of the target message must be the same.

DEFINITION 5 (TIMED GRAPH, TIMED GRAPH MORPHISM) *Let $Spec^{Nat}$ be a specification, a* ***timed graph*** *$TimG = (G, t^G_{min}, t^G_{max})$ consists of a graph G and partial functions $t^G_{min}, t^G_{max} : mE_G \to \mathbb{N}$, assigning a minimum/maximum time to each message edge of G. A* ***timed message*** *is a message $m \in mE_G$ such that $t^G_{min}(m)$ and $t^G_{max}(m)$ are defined. For timed messages m, we require that $t^G_{min}(m) \leq t^G_{max}(m)$.*

A ***(partial) timed graph morphism*** *$g : G \to H$ is a graph morphism $g = (g_V, g_E, g_A)$, such that, the diagrams below commute. A morphism is called total if both components are total. The category of timed graphs and partial timed graph morphisms is denoted by* ***TimGraphP*** *(identities and composition are defined componentwise).*



Timed OB-Graph. The definition of timed OB-graph is analogous to the untimed case.

Timed Rule. A *timed rule* is a rule with an additional requirement: the message in the left-hand side should not have a specific time stamp, i.e., t_{min} and t_{max} are undefined. This means that the aim of the rules shall be to specify how to handle a message, and not when it should be delivered.

DEFINITION 6 (TIMED RULE) *Let $timed_{rule}(r : L \rightarrow R, Eq)$ be a rule according to Definition 3. Then $timed_{rule}$ is a (timed) rule if the following condition is satisfied:*

1. *$t^L_{min}(trigger(r))$ and $t^L_{max}(trigger(r))$ are undefined.*

Timed Object-Based Graph Grammar (TOBGG). The definition of TOBGG is analogous to Definition 4, considering timed graphs and timed rules.

2.2. Example: Railroad System

In this section we model a simple railroad system using graph grammars. The railroad system is composed of instances of three entities: Train, Gate and RSegm.

Train Entity: the graph grammar of the Train Entity is depicted in Figures 1 (A) and (B) (type graph and initial graph template) and 2 (rules). The type graph shows that each train has six attributes: its current position (curr), a reference to the position the train can move to (next), a reference to a gate (gate), the identifier of the next position (posNext), the identifier of the gate (posGate), and a state attribute that describes whether the train is moving or waiting (wait).

Trains may receive four kinds of messages: Wait, Go, OpenGate and Traveling. Messages OpenGate and Go have parameters. The initial graph template in Figure 1 (B) describes that initially a train must be connected to two instances of RSegm, one of Gate, and has a pending Traveling message (this message will trigger the movements of the train). The attribute wait is initialized to false. To create an object of this class, four arguments are need to instantiate this template: a natural number (posGate), two railroad segments (curr and next) and one gate (gate).

The behavior of the Train entity is modeled by the rules shown in Figure 2. Rule R1-T describes that, when receiving a message Traveling, a train tries to travel to the railroad segment pointed by its next attribute. This is modeled by the message MayGo? at the right-hand side of the rule asking permission to enter this segment. Rule R2-T chooses to wait at least 10 time units before trying to travel. Since both rules delete the same message, for each message, one will be non-deterministically chosen to be applied. Rule R3-T models that, when receiving a Wait message, the train asks permission to enter the next railroad segment. Rule R4-T describes the movement of a train: it updates its attributes, sends a Traveling message to itself to be received in (at least) 10 time units (simulating the time needed to reach the end of this segment) and sends a message to the

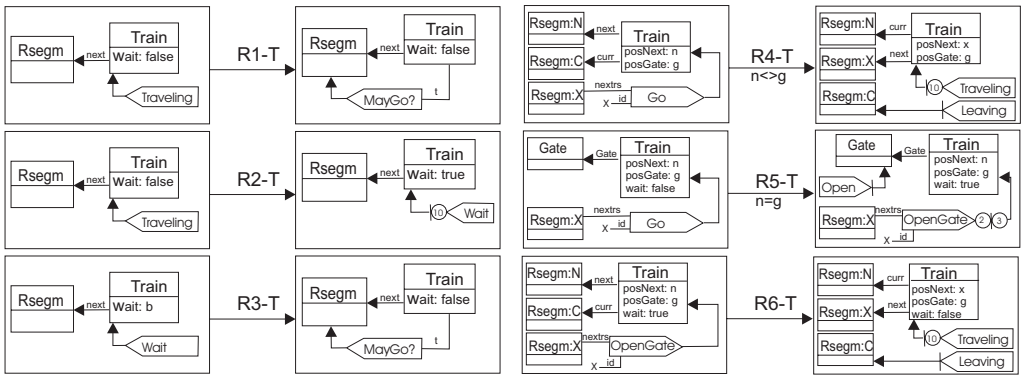


Figure 2. Train Rules

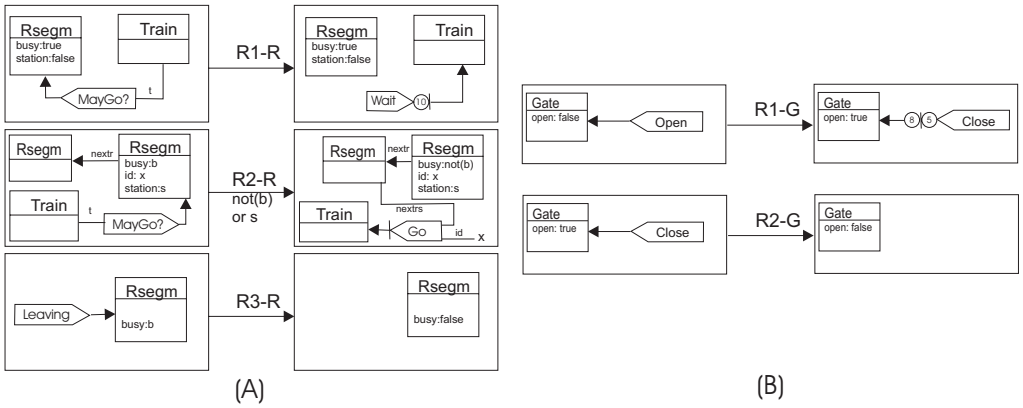


Figure 3. (A) Railroad Segment Rules, (B) Gate Rules

segment it was in to inform that this train is leaving. Note that this rule has a condition n<>g, expressing that this movement may only occur if there is no gate g to enter the next position n. If there is a gate, message Go will be treated by rule R5-T, that requires the gate to open immediately (the Open message is scheduled to arrive exactly in the next time unit (without delay)). The application of rule R5-T also generates a message OpenGate, that shall arrive between 2 and 3 time units (the time needed for the gate to open), and will trigger rule R6-T, that will then move the train to the next position.

Railroad Segment Entity: this graph grammar is depicted in Figures 1(A), 1(C) and 3 (type graph, initial graph template and rules, respectively). The type graph describes that each railroad segment keeps the information about its identifier (attribute id: Natural), its neighbour (the reference nextr), its state (busy:Boolean) and the knowledge whether it is a station or not (station:Boolean). The initial graph is given by two consecutive RSegm instances. Instances of RSegm can react to messages MayGo?, telling a train that it can either move to it (rule R2-R) or that it should wait at least 10 time units (rule R1-R), and to messages Leaving, updating its busy attribute (rule R3-R).

Gate Entity: the specification of the Gate Entity is shown in Figure 1 (A) and (D) and rules in Figure 3 (B). The type graph indicates that gates have one attribute open that describes whether the gate is opened or closed. By rule R1-G, if the gate is requested to open, its closure is scheduled to occur between 5 and 8 time units, i.e. all open requests cause a delay in closing the gate. By rule R2-G, if the gate is opened and there is a close request, attribute open is modified to false.

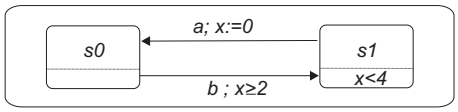


Figure 4. Timed Automaton

3. Timed Automata

Timed Automata ([Alur and Dill 1994], [Bengtsson and Yi 2004]) are used to specify and verify real-time systems. To express the behavior of a system with time restrictions, Timed Automata extend Nondeterministic Automata with a finite set of clocks. In this model states and transitions are associated to clock constraints. A clock constraint is a conjunction of atomic constraints, which compare clock variables with a constant value (a nonnegative rational value). Formally, let x be a clock in a set X of clock variables and c be a constant in $\mathbb{Q}$, then the set $\phi(X)$ of *clock constraints* φ is defined by the grammar:

$$\varphi := x \leq c \mid c \leq x \mid x < c \mid c < x \mid \varphi_1 \wedge \varphi_2,$$

A clock constraint associated to a state (named invariant) indicates how many time units the system may remain on a certain state. The constraint of a transition represents its activation conditions. Moreover, each transition is associated to a set (possibly empty) of clocks that are reset with the occurrence of this transition.

DEFINITION 7 (TIMED AUTOMATON) *A timed automaton TA is a tuple $(L, L^0, \Sigma, X, I, E)$, where:*

- *L is a set of states;*
- *$L^0 \subseteq L$ is a set of initial states;*
- *Σ is a set of labels;*
- *X is a finite set of clocks;*
- *I is a mapping that labels each state s in L with some clock constraint in $\phi(X)$;*
- *$E \subseteq L \times \Sigma \times 2^X \times \phi(X) \times L$ is a set of transitions. Each tuple $(s, a, \varphi, \lambda, s')$ represents a transition from state s to a state s' labeled with a. φ is a clock constraint over X that specifies when the transition is enabled (it may be the empty constraint ε), and the set $\lambda \subseteq X$ gives the clocks to be reset with this transition.*

Figure 4 shows an example of a timed automaton where $s0$ and $s1$ represent the states of the system. The clock constraint $x < 4$ in state $s1$ means that the system can remain in this state while the clock value x is less than four. The transitions are $(s0, b, x \geq 2, \{\ \}, s1)$ and $(s1, a, \varepsilon, \{x\}, s0)$. To each timed automaton TA we can associate a corresponding transition system [Alur and Dill 1994]. The possible transitions are the ones specified in TA, and transitions that increment the clocks (all clocks are incremented simultaneously). All transitions and reachable states must satisfy the time restrictions.

4. Semantics of Timed Object Based Graph Grammars

As discussed in Section 2, the semantics of graph grammars is based on rule applications. Due to space limitations, we will not give the formal definitions of how these rule applications are obtained (via pushouts in the corresponding category). Instead, we will explain how this application is done and how time is handled. Then, we will define formally how to obtain a timed-automata that grasps this idea of behavior of timed OBGGs.

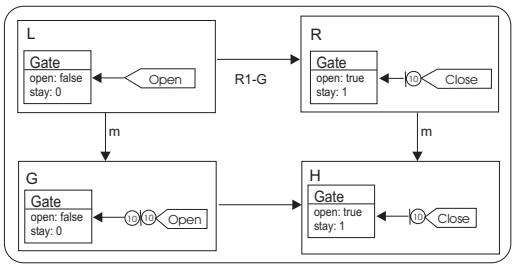


Figure 5. Example of a Rule Application

Given a rule r and a state G, we say that this rule is applicable in this state if there is a match m, that is, an occurrence of the left-hand side of the rule in the state (formally, this is modeled by the existence of a total graph morphism from the left-hand side of the rule to the state graph). We denote such *rule application* by $G \overset{(r,m)}{\Longrightarrow} H$. An example of a rule application is shown in Figure 5. Graphs G and H are called *input* and *output* graphs of this rule application. Since in the left-hand side graph L the t_{min} and t_{max} attributes are undefined (see Definition 6), there are no time constrains on when this rule is applicable. These constrains will be handled by the timed automata (that will define the semantics of a TOBGG), rule applications will be used just to obtain the states of the automata.

In a graph, there may be various vertices/edges with the same types. This means that the same rule may be applicable to a graph using different matches. A *computation of a graph grammar* is a sequence of rule applications starting with the initial graph of the grammar, and in which the output graph of one rule application is the input graph of the next one. We say that a graph (or state) G is *reachable* if there is a computation in which the output graph of the last rule application is G. Typically, the semantics of a system described using a graph grammar is a transition system where the states are graphs and the transitions describe the possible rule applications. This semantics does not take into consideration any time restrictions. In order to define the transition system that gives semantics to a timed object-based graph grammar, we will show how to build a timed-automata that has the desired behavior and respects the required time constraints. If the initial state, the set of rules and the set of reachable states of a grammar are finite[2], a finite timed automata will be generated.

Time will be handled in the following way: a clock will be assigned to each timed message, that is, to each message that is generated with some time constraint (minimum and/or maximum delivery time); this clock is initialized with zero, and, as time advances, this clock will eventually reach the minimum/maximum time. Since all clocks advance simultaneously in timed automata, the relations among the delivery times of messages in a state are preserved in the subsequent states, and this assures that the time constraints of the system are adequately modeled. However, this means that we can only define a timed automata for grammars that reach states with an arbitrary number of messages (because the number of clocks in an automata is fixed). If we consider finite-state systems (that are the ones that are possible to automatically verify), this imposes no restriction.

To define this automata, states will be described by pairs $(G, msgclock^G)$, where

[2]We assume that we have only one representative for each isomorphism class of graphs in the set of reachable states.

G is a timed object-based graph and $msgclock^G$ is defined as follows:

Clock assignment function (msgclock) : Given a graph G and a set of clocks X, the clock assignment function $msgclock^G : mE_G \rightarrow X$ is a partial injective function that assigns a clock to each timed message of G;

x-tmessage bounded graph grammar : Grammar GG is x-tmessage bounded, where x is a natural number, if there is no reachable state of GG in which there are more than x timed messages.

DEFINITION 8 (SEMANTICS OF A TOBGG) *Let $TOBGG = (Spec, X, C, IG, N, n)$ be an x-tmessage bounded timed object-based graph grammar. The **semantics** of $TOBGG$ is the timed automata $TA = (L, L^0, \sum, X, I, E)$ where:*

- $L = \{(G, msgclock^G) | G$ *is reachable in* $TOBGG$ *and* $msgclock^G$ *is a clock assignment function*$\}$;
- $L^0 = (IG, msgclock^{IG})$;
- $\sum = N$;
- $X = \{clock_1, ..., clock_x\}$;
- $I(G, msgclock^G)$ *is the conjunction of all formulas* $msgclock^G(msg) \leq t_max^G(msg)$, *with* $msg \in mE_{IG}$ *and* $t^G_{max}(msg) \in \mathbb{N}$;
- E *is the set of all transitions* $((G, msgclock^G), a, \varphi, \lambda, (H, msgclock^H))$ *such that*
  - $\exists$ *a rule application* $G \stackrel{(r,m)}{\Longrightarrow} H$, *let* $trigger(r) = tr$;
  - $a = (r, m)$,
  - $\varphi = \begin{cases} (msgclock^G(tr) \geq t^G_{min}(tr)) & \text{, if } min = t^L_{min}(tr) \\ \varepsilon & \text{, if } t^L_{min}(tr) \text{ is undefined} \end{cases}$
  - λ *is the set of clocks assigned to the timed messages created by the rule application.*

The clock constraint (invariant) $(msgclock(msg) \leq t_{max}(msg))$ on states $(I(s))$ assures that the system may stay in state s at most until the clocks of all timed messages of s are less than their respective maximum time limits (because after this time, there will be at least one message in s that was not delivered in time). The clock constraint on transitions φ assures that, if a message has a minimum time to be delivered ($t^L_{min}(trigger(r))$), it will not be processed before this time, if a message does not have such restriction, it may be processed at any time (the restrictions on maximum times for delivery are modeled by function I, as described above). Moreover, λ indicates which clocks shall be reset with the transitions, these are all the clocks used in the newly created timed messages in the target state of the transition.

Now, this construction will be illustrated by an example. Figure 6 shows a (partial) timed automata (TA) that was obtained from translation of example in Section 2.2. The initial state of TA is the state IG. G3, G4, G6, G7, G13, G14, G15 and G16 are some states of the automata TA (that are obtained with rules application from IG). The time constraints were inserted following Definition 8. The complete timed automata for this example has 36 states and 40 transitions.

Due to the construction of the timed automata based on rule applications over reachable states of $TOBGG$, this semantics would be compatible with a traditional semantics based on sequences of rule applications in the following sense: whenever there is

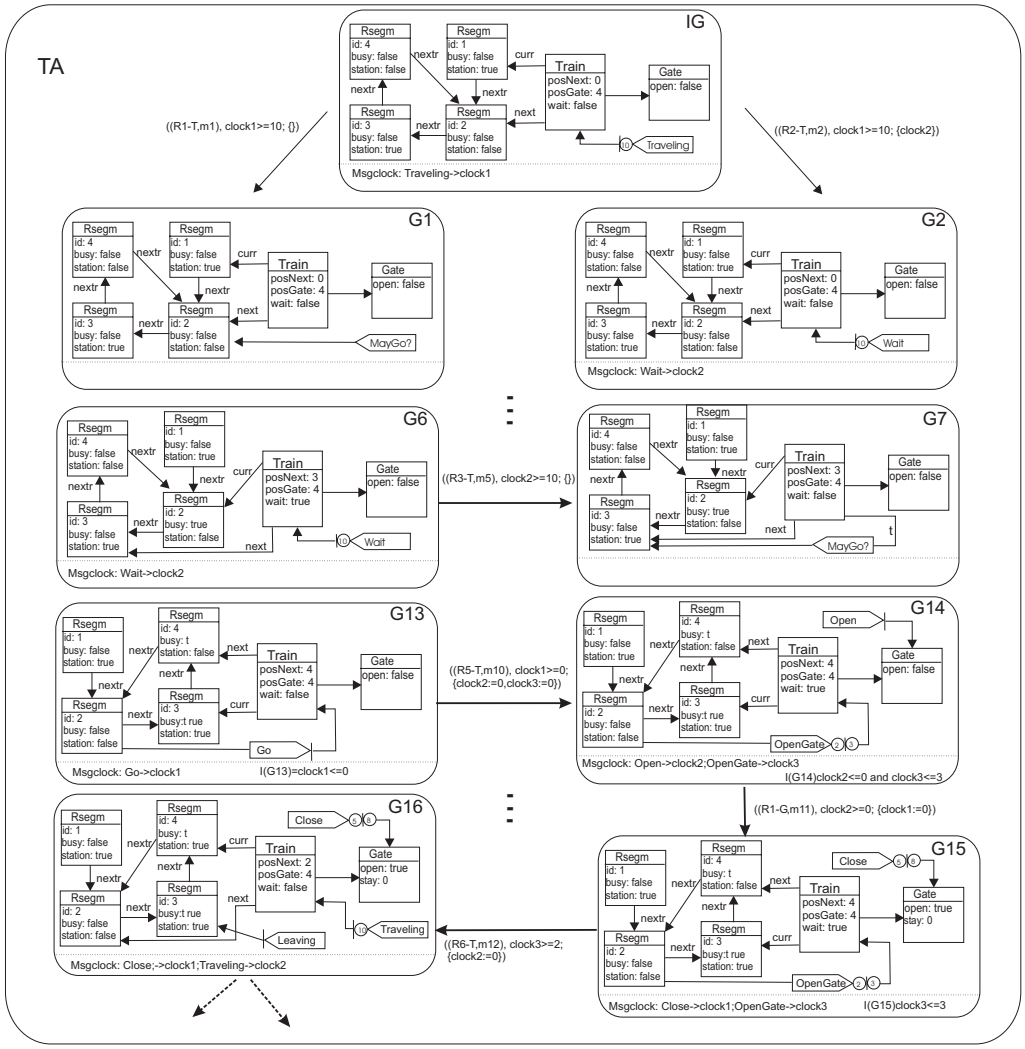


Figure 6. (Part of the) Timed Automata for the Railroad System

a transition $((G, msgclock^G), (r, m), \varphi, \lambda, (H, msgclock^H))$ in the timed automata, there is a corresponding transition from graph G to graph H corresponding to applying rule r using m to graph G. The converse might not be true because there may be rule applications that do not obey the time restrictions (violate either minimum or maximum delivery time of messages). Thus, some graphs (states) that are reachable in computations of a grammar will not be reachable in the corresponding timed automata. Note that there is no incompatibility of semantical definitions here, since we have defined the semantics of timed object-based graph grammars in terms of timed automata. The definition of a semantics purely based on rule applications, and corresponding proof of equivalence, is left for future work.

In Figure 7 we present two other possibilities of initial states (IG1, IG2) for the example in Section 2.2. The timed automaton that represents the TOBGG with the initial graph IG1 has 21 states and 24 transitions. For the TOBBG with initial graph IG2, the timed automaton has 123 states and 264 transitions. These numbers do not consider unreachable states and transitions.

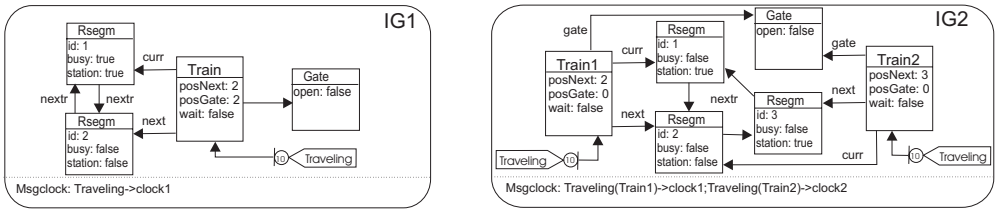


Figure 7. Examples of initial states

5. Verification

For simulation and verification of properties of real-time systems we chose to use Uppaal [Behrmann et al. 2004], a toolkit developed by Uppssala University and Aalborg University. Uppaal is a tool for validation (via graphical simulation) and verification (via automatic model-checking) that has timed automata as the input language and a subset of CTL as the specification language. The simulator can be used in three ways: the user can run the system manually and choose which transitions to perform, the random mode can be toggled to let the system run on its own, or the user can go through a trace (saved or imported from the verifier) to see how certain states are reachable. The verifier is designed to check a subset of CTL (Computation Tree Logic). The formulas to be checked must be in one of the following forms:

- $A[]\phi$: for all paths, ϕ always hold;
- $E <> \phi$: there exists a path where ϕ eventually holds;
- $A <> \phi$: for all paths, ϕ will eventually hold;
- $E[]\phi$: there exists a path where ϕ always holds;
- $\phi - - > \psi$: whenever ϕ holds ψ will eventually hold.

where ϕ e ψ are Boolean expressions that can refer to states, integer variables and clocks constraints. The word deadlock can be used to verify deadlocks.

Safety properties mean that something bad will never happen. To check these properties in Uppaal, we use forms $A[]\phi$ or $E[]\phi$. For example, the property $A[]$ not deadlock was checked for the railroad system of section 2.2 (with initial graph IG (Figure 6)) and was satisfied. This indicates the absence of deadlock for all paths of the system.

Reachability properties are the simplest form of properties. They specify whether a given property ϕ can be satisfied in some reachable state. The form $E <> \phi$ is used to check these properties. For example, we can check if there is one path in which state G7 of automata TA in Figure 6 (that represents the situation when train asks permission to pass to the railroad segment identified with id 3) is reachable and clock2 has a value less than 10 time units ($E <>$ TA.G7 and clock2< 10). This property was checked and was not satisfied because when the system reaches state G7, the value of clock2 must be already bigger than 10 (this is the condition of the transition to reach this state).

Liveness properties are used to check if something good will eventually happen. These properties are expressed by the forms $A <> \phi$ and $\phi - - > \psi$. For example, TA.G13$- - >$TA.G16 that means if state G13 of TA occurs, then G16 will occur, too. This means that if the train can travel to a railroad segment that have a gate (situation represented by state G13) the gate will eventually open and the train will travel to this segment (represented by state G16). This property is satisfied for the example.

For the automaton that represents the TOBGG with IG1 as initial graph (Figure 7, with one train, two railroad segments and one gate), we verified the following properties: the system never deadlocks; the train always requests the opening before passing a gate; and the unreachability of same states.

For the automaton that represents the TOBGG with IG2 as initial graph (Figure 7, with two trains and three railroad segments without gate), we verified the following properties: the system never deadlocks; a train never enters in a railroad segment with another train; when a train leaves a railroad segment, the segment is released.

6. Final Remarks

In this paper we introduced Timed Object-Based Graph Grammars (TOBGGs), an extension of Object-Based Graph Grammars that includes the notion of time, allowing the specification and analysis of real-time systems. Time stamps on the messages describe when they are to be delivered/handled. The semantics of TOBGG was defined in terms of Timed Automata. This provided a way to verify properties over TOBGG specifications using the Uppaal model-checker. The main reason for extending Object-Based Graph Grammars is that, besides being formal, they are quite intuitive even for people not used to formal description languages. This is an advantage of graph grammars comparing to other specification methods such as real-time process algebras and Timed Petri Nets.

Since the usual semantics of graph grammars is the set of computations generated by rule applications, we plan to define a semantics for TOBGG analogously, without a translation to timed automata. This would involve including clocks in the state graphs and using conditions for rule applications to assure that time restrictions are not violated. In this case, to be able to use the Uppal verification tool, we would have to provide a proof of the equivalence of the two semantics. Another topic of future work is how to lift constructions like the compositional verification method introduced in [Ribeiro et al. 2006] and the inheritance and polymorphism concepts [Ferreira 2005] to TOBGGs.

References

Alur, R. and Dill, D. L. (1994). A theory of timed automata. *Theoretical Computer Science*, 126(2):183–235.

Baeten, J. and Middelburg, C. (2000). Process algebra with timing: Real time and discrete time. In *Handbook of Process Algebra*, chapter 4. Elsevier.

Behrmann, G., David, A., and Larsen, K. G. (2004). Tutorial on UPPAAL. In *Formal Methods for the Design of Real-Time Systems*, volume 3185 of *LNCS*, pages 200–236. Springer.

Bengtsson, J. and Yi, W. (2004). Timed automata: Semantics, algorithms and tools. In *Lecture Notes on Concurrency and Petri Nets*, volume 3098 of *LNCS*, pages 87–124. Springer.

Berthomieu, B. and Diaz, M. (1991). Modeling and verification of time dependent systems using time petri nets. *IEEE Trans. Softw. Eng.*, 17(3):259–273.

Bowman, H. and Derrick, J. (1997). Extending lotos with time: A true concurrency perspective. In *AMAST Workshop on Real-Time Systems, Concurrent and Distributed Software*, volume 1231 of *LNCS*, pages 382–399. Springer.

Clarke, E. M. and Wing, J. M. (1996). Formal methods: state of the art and future directions. *ACM Computing Surveys*, 28(4):626–643.

Corradini, A., Montanari, U., and Rossi, F. (1996). Graph processes. *Fundamenta Informaticae*, 26(3/4):241–265.

Diethers, K. and Huhn, M. (2004). Vooduu: Verification of object-oriented designs using uppaal. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988 of *LNCS*, pages 139–143. Springer.

Dotti, F. L., Foss, L., Ribeiro, L., and Santos, O. M. (2003). Verification of distributed object-based systems. In *6th IFIP International Conference on Formal Methods for Open Object-based Distributed Systems*, volume 2884 of *LNCS*, pages 261–275. Springer.

Dotti, F. L. and Ribeiro, L. (2000). Specification of mobile code systems using graph grammars. In *Formal Methods for Open Object-Based Distributed Systems*, pages 45–64. Kluwer.

Duarte, L. M., Dotti, F. L., Copstein, B., and Ribeiro, L. (2002). Simulation of mobile applications. In *Proc. of Communication Networks and Distributed Systems Modeling and Simulation Conference*, volume 1, pages 1–15.

Ehrig, H. and Mahr, B. (1985). Fundamentals of algebraic specification: Equations and initial algebra semantics. In *EATCS Monographs on Theoretical Computer Science*, volume 6. Springer.

Ferreira, A. P. L. (2005). *Object-Oriented Graph Grammars*. PhD thesis, Universidade Federal do Rio Grande do Sul.

Graf, S., Ober, I., and Ober, I. (2003). Timed annotations with UML. In *Workshop on Specification and Validation of UML models for Real Time and Embedded Systems*.

Knapp, A., Merz, S., and Rauh, C. (2002). Model checking - timed uml state machines and collaborations. In *Proceedings of the 7th International Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, pages 395–416. Springer.

Konrad, S., Campbell, L. A., and Cheng, B. H. C. (2004). Automated analysis of timing information in uml diagrams. In *Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 350–353. IEEE Computer Society.

Lavazza, L., Quaroni, G., and Venturelli, M. (2001). Combining uml and formal notations for modelling real-time systems. *SIGSOFT Softw. Eng. Notes*, 26(5):196–206.

Lee, I., Brémond-Grégoire, P., and Gerber, R. (1994). A process algebraic approach to the specification and analysis of resource-bound real-time systems. *Proc. of the IEEE*, 82(1):158–171.

Ober, I., Graf, S., and Ober, I. (2004). Validation of uml models via a mapping to communicating extended timed automata. In *SPIN*, volume 2989 of *LNCS*, pages 127–145. Springer.

Ribeiro, L., Dotti, F., and Bardohl, R. (2005). A formal framework for the development of concurrent object-based systems. In *Formal Methods in Software and Systems Modeling*, volume 3393 of *LNCS*, pages 385–401. Springer.

Ribeiro, L., Dotti, F. L., Santos, O., and Pasini, F. (2006). Verifying object-based graph grammars: An assume-guarantee approach. Accepted for publication in Software and Systems Modeling.

Stankovic, J. A. (1988). A serious problem for next-generation systems. *IEEE Computer*, 21(10):10–19.

Stankovic, J. A. et al. (1996). Strategic directions in real-time and embedded systems. *ACM Computing Surveys*, 28(4):751–763.

Walter, B. (1983). Timed petri-nets for modelling and alalyzing protocols with real-time characteristics. In *Protocol Specification, Testing, and Verification*, pages 149–159. North-Holand.

Uma Extensão do RUP para Modelagem Rigorosa de Sistemas Concorrentes

Robson Godoi, Rodrigo Ramos, Augusto Sampaio

Centro de Informática – Universidade Federal de Pernambuco
Caixa Postal 7851 – CEP 50.740-540 – Recife – PE – Brasil

{rgam,rtr,acas}@cin.ufpe.br

Abstract. *Despite the great success of traditional software development processes on defining guidelines to be used in practice, many development aspects of critical and concurrent systems are still neglected. These problems are even more serious in processes that support emerging paradigms, such as model driven development. In this context, we propose a rigorous development strategy to model concurrent systems in UML-RT, based on the Rational Unified Process. In this strategy, although each analysis and design task is consistent and justified by model transformations that preserve the system behaviour, the underlying formalism is transparent for the developer.*

Resumo. *Apesar do grande sucesso dos processos de desenvolvimento de software tradicionais na criação de guias e boas práticas de desenvolvimento, diversos aspectos do desenvolvimento de sistemas críticos concorrentes são negligenciados. Estes problemas são ainda mais acentuados em processos que suportam paradigmas emergentes, como o desenvolvimento dirigido a modelos. Neste contexto, propomos uma estratégia rigorosa de modelagem de sistemas concorrentes em UML-RT, baseada no Rational Unified Process. Nesta estratégia, apesar de cada passo de análise e projeto ser consistente e ser justificado por transformações de modelos que preservam o comportamento do sistema, todo o formalismo que suporta estas atividades é transparente para o desenvolvedor.*

1. Introdução

Com a crescente demanda por aplicações concorrentes, um grande número de novos princípios e paradigmas de desenvolvimento de software vem surgindo para lidar com a complexidade inerente a estas aplicações. Alguns processos e metodologias têm se destacado por incluírem guias dos passos e de boas práticas a serem seguidos pelos desenvolvedores. Porém, devido à complexidade destes domínios, a execução segura e consistente destes passos constitui um grande desafio na prática.

Estes problemas são ainda mais evidentes quando lidamos com aplicações não triviais, que possuem aspectos de concorrência que devem ser precisamente especificados e verificados. Isto gera a necessidade de estratégias de desenvolvimento com passos que garantam a consistência do desenvolvimento; estes passos devem ser incorporados a um processo bem definido que, preferencialmente, torne transparente ao desenvolvedor formalismos possivelmente utilizados. Na literatura, algumas estratégias de desenvolvimento [7, 11, 12] têm adotado Métodos Formais, porém estas têm se

concentrado, essencialmente, na preservação de comportamento e consistência das diferentes visões da arquitetura durante o desenvolvimento, não incorporando os aspectos formais a um processo de desenvolvimento mais amplo.

Neste trabalho, propomos uma estratégia de desenvolvimento de sistemas concorrentes para UML-RT [15], um *profile* para UML utilizado para descrever sistemas concorrentes e distribuídos, através de uma adaptação e extensão do Rational Unified Process (RUP) [6]. Focamos na disciplina de Análise e Projeto, mas analisamos também os impactos em outras disciplinas do RUP para suportar um processo de desenvolvimento rigoroso baseado em modelos, que garanta a consistência e preservação de propriedades do sistema durante cada passo da modelagem. Cada passo é justificado por transformações de modelos e noções precisas baseadas em uma linguagem formal. Tais noções agem como uma interface formal para práticas de desenvolvimento, fazendo com que o desenvolvedor possa exercer suas atividades de forma consistente, porém sem se preocupar com o formalismo que as suporta.

A base formal da estratégia proposta é construída a partir de resultados com uma base semântica sólida. Em [11], atribuímos uma semântica formal e unificada à UML-RT através de seu mapeamento para *Circus* [17], uma linguagem formal que combina Z [16], CSP [13] e suporta o cálculo de refinamento de Morgan [9]. Em [10, 12, 14] apresentamos e provamos um conjunto abrangente de leis algébricas para UML-RT, que capturam tanto transformações simples no modelo, como *refactorings* precisos, preservando tanto o comportamento estático quanto o dinâmico do modelo.

Na próxima seção, introduzimos brevemente a linguagem de modelagem UML-RT. Nas seções 3 e 4 apresentamos as adaptações necessárias para as disciplinas de Requisitos e Análise e Projeto do RUP, respectivamente. No decorrer destas seções, ilustramos nossa estratégia através da modelagem rigorosa de parte de um sistema de automação industrial. Finalmente, nossas conclusões e trabalhos relacionados são apresentados na Seção 5.

2. Introdução à UML-RT

UML-RT, como outras linguagens de descrição arquitetural (ADLs), modela sistemas reativos com componentes arquiteturais ativos que interagem entre si e operam concorrentemente. A comunicação é modelada através da troca de mensagens de entrada e saída, que podem ser síncronas ou assíncronas. Apesar destes conceitos terem influenciado também o modelo de componentes na recente versão UML 2.0, aqui, utilizamos o *profile* UML-RT porque consideramos que seu modelo de componentes ativos é mais consolidado que o proposto para UML 2.0. Além disto, UML-RT conta com um suporte ferramental mais consolidado.

Os conceitos de UML-RT são adicionados à UML através de quatro novos elementos de modelagem: cápsula, protocolo, porta e conector. Cápsulas descrevem componentes arquiteturais cujos únicos pontos de interação são chamados de portas. Tais portas são interligadas por conectores e comunicam sinais declarados previamente em protocolos.

Para ilustrar a notação de UML-RT e nossa estratégia de desenvolvimento, apresentamos a modelagem simplificada de um sistema de automação industrial, responsável por processar peças. O sistema consiste dos seguintes dispositivos: um repositório de entrada com peças não processadas, um repositório de saída com peças já

processadas, algumas máquinas que processam as peças e alguns agentes de transporte. Cada peça não processada deve ser retirada do repositório de entrada, passar por todas as máquinas em um trajeto pré-definido e ser armazenada no repositório de saída. As máquinas e repositórios encontram-se fisicamente distantes, e solicitam peças a um agente autônomo de transporte, chamado *holon*.

Um modelo inicial do sistema é apresentado na Figura 2.1. No diagrama de estrutura Str_M da figura (retângulo inferior), é possível ver que a cápsula Main, que representa todo o sistema, é, na realidade, composta de três outras instâncias de cápsula: sys, sin e son, que interagem entre si; estas instâncias são dos tipos ProdSys e Storage. No diagrama Str_M, Main delega todas mensagens comunicadas pelas portas mi e mo para sin e son.

A declaração das cápsulas (classes ativas) é apresentada no diagrama de classes Cls_M (retângulo superior), mostrando as relações entre cápsulas, protocolos e classes. Por exemplo, a relação son entre Main e Storage indica que existe uma instância da cápsula son dentro da estrutura de Main, e a relação so entre Storage e o protocolo STO indica que a porta so é do tipo do procolo STO. Outra parte importante do comportamento do sistema é descrito através de *Statecharts*. Através destes diagramas (lado direito da figura), protocolos podem estabelecer quais são as seqüências de sinais possíveis na interação entre cápsulas (como o *Statechart* de STO que estabelece que um sinal req deve sempre ser seguido por um output), e cápsulas podem definir como seu comportamento reativo será disparado por sinais externos.

Nos *Statecharts* da Figura 2.1 utilizamos a linguagem de especificação *Circus* para facilitar a verificação dos modelos durante um desenvolvimento rigoroso. Por exemplo, no *Statechart* de Storage, existem duas transições de saída no estado Sa. A transição à direita é disparada caso um sinal req seja recebido através da porta so e buffer não esteja vazio. A ação correspondente declara uma variável x para capturar o resultado do método remove. Esta é a maneira como é feito em *Circus*, já que remove é na realidade interpretado por um esquema em Z. O valor de x é então enviado através da porta so. A sintaxe para escrever estas ações de comunicação é a mesma que em CSP.

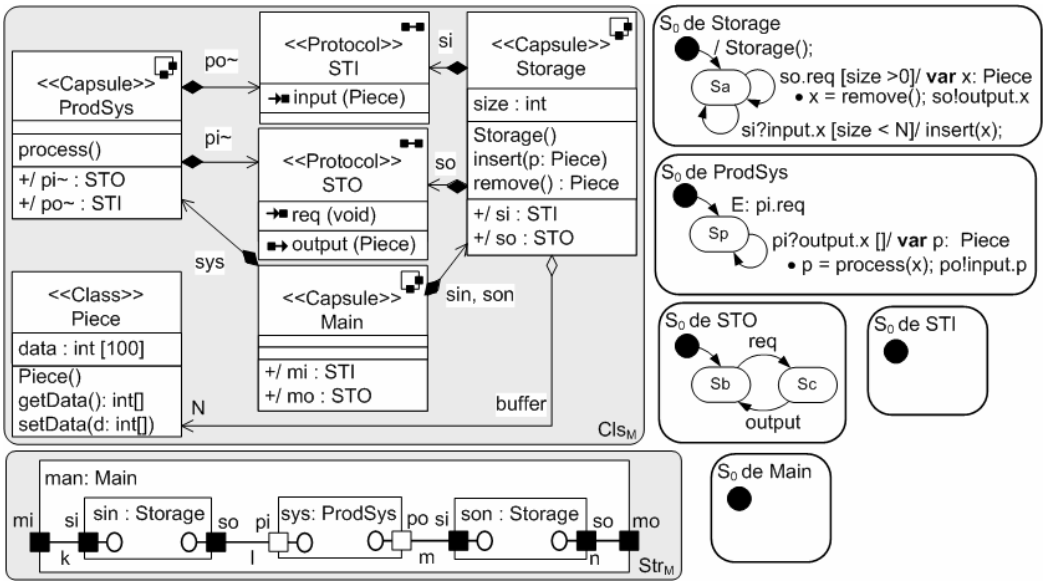


Figura 2.1. Modelo Abstrato de Análise

3. Modelagem de Requisitos

O RUP é um framework para processos incrementais e iterativos de desenvolvimento de software de propósito geral, com características, entre outras, de ser guiado por casos de uso e ser focado na criação de uma arquitetura robusta. Neste processo, atividades são agrupadas nas seguintes disciplinas: Modelagem do Negócio, Requisitos, Análise e Projeto, Implementação, Teste, Distribuição, Gerenciamento de Configuração e Mudanças, Gerenciamento de Projeto e Ambiente.

Apesar de nosso foco ser na disciplina de Análise e Projeto do RUP, mudanças são necessárias em outras disciplinas do processo, visando, principalmente, antecipar e mitigar potenciais riscos no desenvolvimento de aspectos relativos à concorrência. Nossa proposta inclui ou altera detalhes de fluxo e atividades das disciplinas de Requisitos e Análise e Projeto. Outras disciplinas, como Gerenciamento de Projeto e a Implementação, são afetadas somente em termos da organização da execução de suas atividades, considerando principalmente impactos na visão dinâmica (fases e iterações) do RUP; maiores detalhes sobre o impacto nestas disciplinas podem ser encontrados em [5]. No restante desta seção, apresentamos nossa proposta com relação à disciplina de Requisitos.

Na disciplina de Requisitos, no detalhe de fluxo *Definição do Sistema*, inicia-se a convergência dos requisitos de alto nível para um esboço mais detalhado das exigências do sistema. O foco é a identificação completa de atores e casos de uso e a extensão da modelagem dos requisitos não-funcionais, como definido no artefato *Especificações Suplementares*, bem como a criação de matrizes de rastreabilidade constantes no artefato *Atributos de Requisitos*. Neste trabalho, duas matrizes são utilizadas: uma relacionando os *Elementos Ativos* aos *Requisitos* e outra os *Casos de Uso* aos *Requisitos*.

Propomos a extensão do fluxo da disciplina, introduzindo um desvio condicional entre o detalhe de fluxo *Definição do Sistema* e o *Gerenciamento do Escopo do Sistema* (Figura 3.1), onde poderá ser identificada a existência de um comportamento ativo. Esta identificação é de vital importância para antecipar o tratamento de concorrência, conforme ilustrado a seguir. Criamos o detalhe de fluxo de *Identificação de Atividades Autônomas*, com objetivos de: nivelar a compreensão sobre concorrência no sistema entre toda equipe de projeto; detalhar o documento de *Especificações Suplementares* com a representação de concorrência no sistema; e refinar o *Modelo de Casos de Uso* para refletir os casos de uso relacionados à concorrência no sistema. Para tanto, foi criada a atividade *Identificação de Elementos Ativos* dentro deste detalhe de fluxo para se definir que partes do software serão passivas ou ativas, através do exame dos

Figura 3.1 – Contexto com o Fluxo de Requisitos adaptado

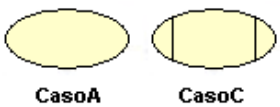


Figura 3.2 – Casos de Uso padrão (A) e com identificação de concorrência (C)

requisitos funcionais (**RF)** e não-funcionais **(RNF**).

Dois passos importantes são descritos nesta atividade: *Descoberta de Elementos Ativos* e *Análise de Requisitos Não-Funcionais de Concorrência*. Estes são responsáveis por analisar as descrições dos **RF** e **RNF**, respectivamente, do sistema, onde possíveis elementos ativos são identificados na descrição destes requisitos. Estes elementos ativos são sinalizados no artefato *Especificações Suplementares*. Consultando as matrizes de rastreabilidade, identificamos os casos de uso relacionados a estes elementos e os sinalizamos no artefato *Modelo de Casos de Uso*, através de uma representação gráfica para expressar concorrência similar à encontrada em [4], como o CasoC na Figura 3.2.

Tabela 3.1 Enumeração dos requisitos do sistema

Requisitos	Descrição
[**RF01**] Inserir Peças	O cliente pode inserir peças no repositório de entrada a qualquer momento.
[**RF02**] Recuperar Peças	O cliente pode recolher peças processadas pelo sistema no repositório de saída.
[**RF03**] Processar Peças	O sistema dever garantir que o processamento das peças ocorrerá obedecendo à seqüência de processadores descrita no programa de produção.
[**RF04**] Programar Produção	O operador está apto a mudar a configuração das máquinas.
[**RNF02**] Controle de Acesso (**RNF** de segurança)	O sistema controla o acesso às funcionalidades mediante a identificação do usuário através de uma senha de acesso e o nível de privilégio que ele tenha.
[**RNF04**] Processamento Simultâneo de Peças (**RNF** de concorrência)	O sistema pode processar diversas peças simultaneamente, obedecendo-se a programação de produção elaborada pelo operador.

Em nosso estudo de caso, nos passos de *Descoberta de Elementos Ativos* e *Análise de Requisitos Não-Funcionais de Concorrência,* a partir da análise dos requisitos do sistema (apresentados na Tabela 3.1), antecipamos a descoberta dos elementos ativos contidos na Tabela 3.2, que representam dispositivos físicos e autônomos neste sistema.

Tabela 3.2 Elementos ativos identificados

Nome	Descrição
Repositório de entrada	Responsável por armazenar peças a serem processadas e encaminha-las ao primeiro processador do programa de produção.
Processadores	Responsáveis pela recepção, processamento e encaminhamento de peças na linha de produção.
Holons	Responsável pelo transporte de peças entre os repositórios e processadores.
Repositório de saída	Responsável por armazenar peças processadas, e entregá-las aos clientes.

Tabela 3.3 - Matriz de rastreabilidade relacionando elementos ativos e requisitos

	RF01	RF02	RF03	RF04	RNF02	RNF04
Repositório de entrada	x		x			
Processadores			x			x
Holons			x			
Repositório de saída		X	x			

Tabela 3.4 - Matriz de rastreabilidade entre requisitos e casos de uso

	RF01	RF02	RF03	RF04	RNF02	RNF04
Inserir Peças	x					
Recuperar Peças		x				
Processar Peças			x			x
Programar Processamento				x	x	

Como saída desta atividade temos a atualização das *Especificações Suplementares*, onde os elementos com comportamento ativo são sinalizados (Tabela 3.2), e dos *Atributos de Requisitos*, onde são registradas as matrizes de rastreabilidade relacionando *Elementos Ativos* aos *Requisitos* (Tabela 3.3) e os *Casos de Uso* aos *Requisitos* (Tabela 3.4).

Consultando-se as matrizes que relacionam os elementos ativos, requisitos e casos de uso (Tabelas 3.3 e 3.4), identificamos os casos de uso (Figura 3.3) relacionados aos elementos ativos e a concorrência, que são: *Inserir*, *Recuperar* e *Processar Peças*.

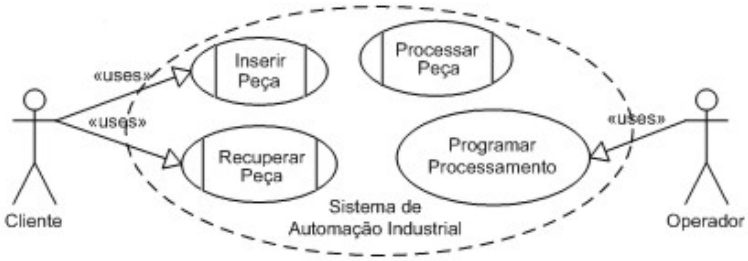


Figura 3.3 Casos de Uso do Sistema de Automação Industrial

4. Uma disciplina para Análise e Projeto (A&P)

Em nossa extensão da disciplina de A&P (Figura 4.1), adicionamos os detalhes de fluxo: *Projetar Componentes Ativos*, onde são refinados os elementos de projeto ativos

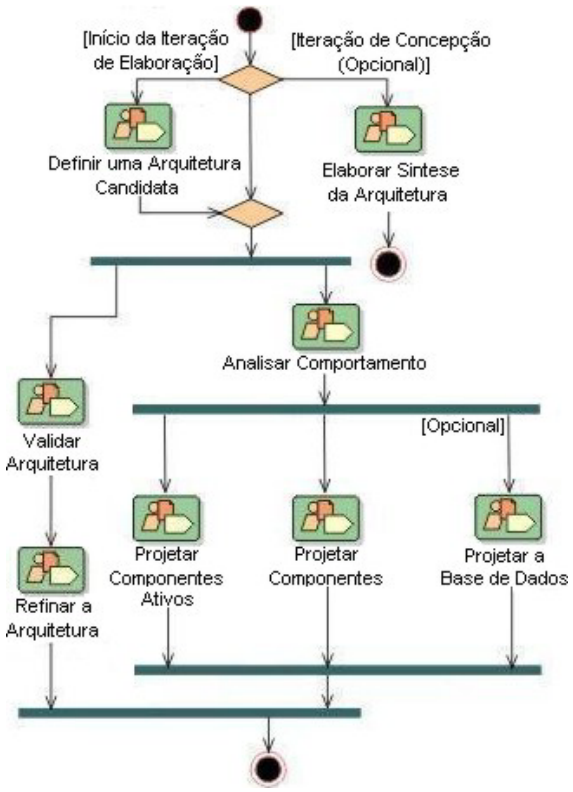


Figura 4.1 – Novo fluxo de Análise e Projeto

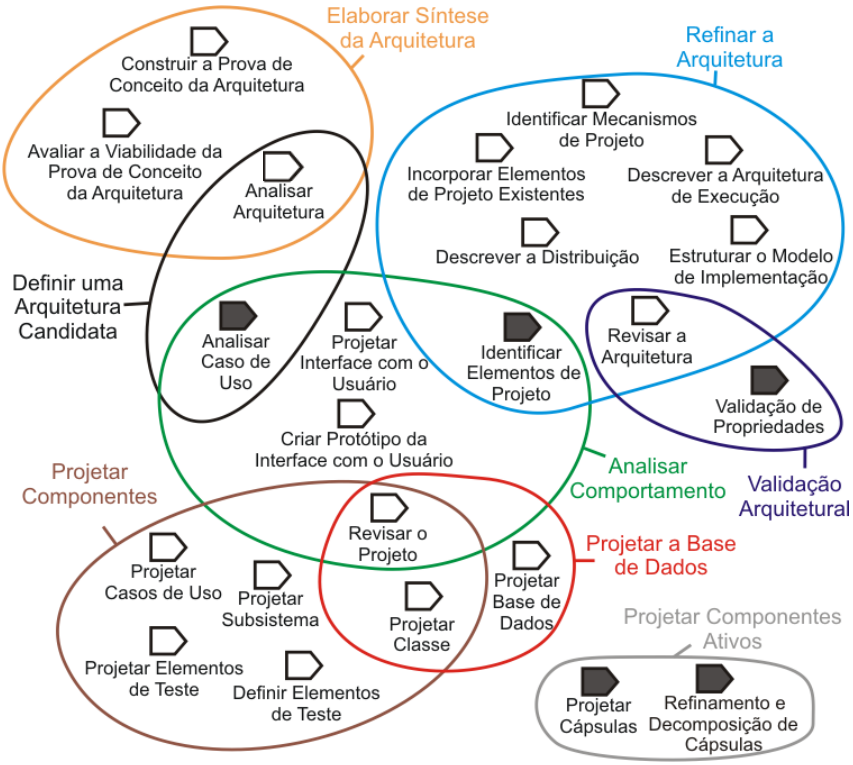


Figura 4.2 – Visão consolidada das Atividades x Detalhe de fluxo

e atualizadas as realizações de caso de uso, com base em leis para transformação de modelos UML-RT [10, 11,14]; e *Validar Arquitetura*, onde a arquitetura do sistema é validada utilizando-se a semântica de UML-RT proposta em [11]. Além disto, os detalhes de fluxo *Definir uma Arquitetura Candidata* e *Analisar Comportamento* foram alterados. Na Figura 4.2, enfatizamos quais atividades são alteradas ou adicionadas em cada um destes detalhes de fluxo, os quais serão detalhados no restante desta seção.

4.1. Definir uma Arquitetura Candidata

Um dos principais objetivos deste detalhe de fluxo, conduzido no início da fase de Elaboração, é criar um esboço inicial da arquitetura do software, onde são identificadas as classes de análise para os casos de uso relevantes da arquitetura. Este detalhe de fluxo é afetado em nossa estratégia pela alteração da atividade *Analisar Caso de Uso*, conforme apresentamos a seguir. O restante deste detalhe de fluxo não foi alterado, entretanto sugerimos a aplicação de leis de transformação de modelos consistentes [10], um dos focos de nosso trabalho em todas as atividades.

4.1.1. Analisar Caso de Uso

Um novo objetivo foi adicionado à atividade original: *Identificar as cápsulas que participam do fluxo de eventos do Caso de Uso*, bem como um novo passo: *Encontrar Cápsulas e Classes de Análise a partir do comportamento de um Caso de Uso*. A identificação de cápsulas deve ocorrer para todo caso de uso com indicação de concorrência, fornecida pela disciplina de Requisitos; à medida que cápsulas são identificadas, protocolos que regem sua interação são também definidos.

Para atingir este objetivo os seguintes artefatos são gerados e atualizados: *Classes de Análise*, representando um modelo inicial das responsabilidades e do comportamento do sistema; *Cápsula* e *Protocolo*, identificando parte dos elementos de projeto criados para representar concorrência; e *Realização de Caso de Uso*, representando a realização dos casos de uso em termos de colaborações de objetos.

Nossa estratégia assume a seguinte premissa: sistemas ativos ou concorrentes necessitam incluir pelo menos uma cápsula, que representa um fluxo de execução no sistema, e, desta forma, é parte inerente de qualquer sistema concorrente. No passo *Encontrar Cápsulas e Classes de Análise a partir do comportamento de um Caso de Uso*, é identificado o conjunto dos possíveis elementos do modelo (cápsulas e classes de análise) que serão capazes de representar o comportamento descrito nos casos de uso. Assim, cada caso de uso com concorrência é representado ao menos por uma cápsula de fronteira, que representa a interface do caso de uso, juntamente com seus protocolos. Caso um elemento de controle seja identificado também como ativo, este também será representado como uma cápsula.

Em nosso estudo de caso, executamos o detalhe de fluxo *Definir uma Arquitetura Candidata* em uma das iterações iniciais na fase de Elaboração; por conseguinte, as atividades de *Analisar Caso de Uso* e *Analisar Arquitetura* são exploradas. Destacamos na primeira atividade, a identificação das cápsulas que participam do fluxo de eventos do caso de uso. De acordo com o *guideline* do RUP, estas são classificadas como cápsulas de fronteira ou controle, dependendo dos elementos ativos identificados. Baseado neste *guideline*, construímos nosso modelo da Figura 2.1 sistematicamente a partir da visão de casos de uso. As instâncias de cápsula sys, sin e son representam as cápsulas de fronteiras dos casos de uso *Inserir*, *Recuperar* e *Processar Peças*, respectivamente; neste contexto interpretamos cada cápsula como subsistemas de todo o sistema, representado aqui pela cápsula Main, e assumimos que possíveis classes de controle estão contidas dentro destes subsistemas. Desta forma deixamos as extrações destes elementos de controle para um passo subseqüente do desenvolvimento.

A partir do modelo de análise na Figura 2.1, prosseguimos na *Definição de uma Arquitetura Candidata*. Adotamos uma arquitetura em camadas simples, onde a manipulação de dados é isolada das regras de negócio. Conseqüentemente, explicitamos a introdução de uma classe de coleção de dados, PieceCollection, para armazenar peças de trabalho; na realidade, esta classe é extraída de Storage utilizando-se uma regra de *Extração de Classes* a partir de cápsulas [10]. O resultado deste passo de projeto é apresentado na Figura 4.3.

4.2. Analisar Comportamento

Neste detalhe de fluxo, transformamos as descrições comportamentais de um caso de uso em elementos nos quais o projeto pode ser baseado. Alteramos a atividade *Identificar Elementos de Projeto*.

4.2.1. Identificar Elementos de Projeto

Nesta atividade, as interações entre as classes de análise são examinadas para identificar elementos de projeto do modelo. Estas classes evoluem para diversos elementos de projeto: classes, cápsulas, subsistemas, interfaces e protocolos. A decisão de quais classes de análise devem ser transformadas em cápsulas pode ser realizada através de

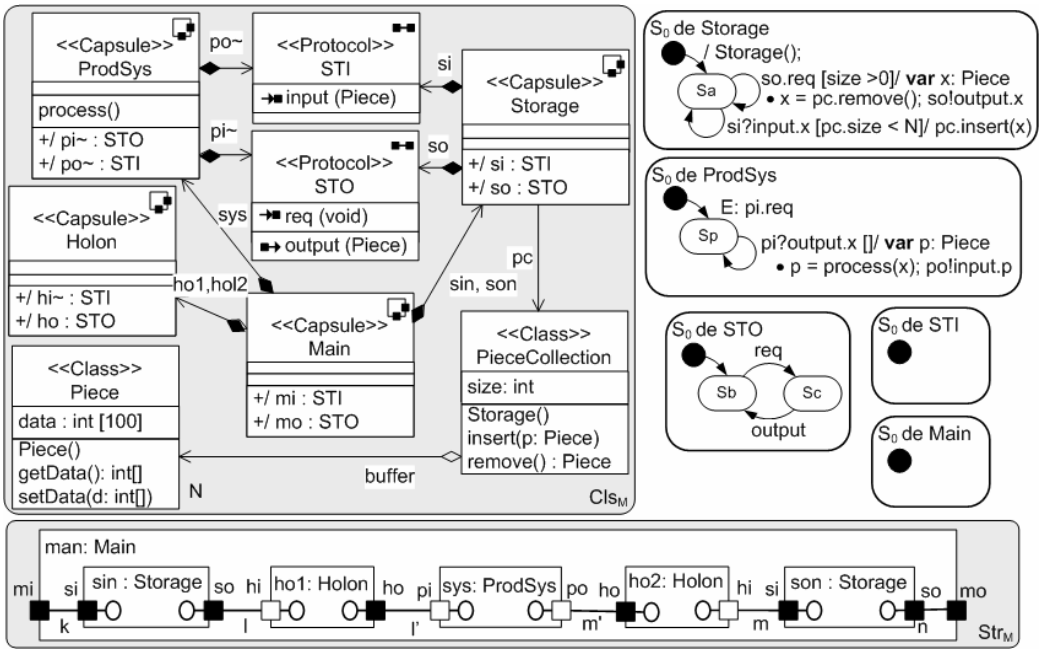


Figura 4.3. Extração de PieceCollection e identificação de agentes de transporte

diversas técnicas. Neste trabalho, utilizamos uma árvore de decisão (Figura 4.4), onde sugerimos que: uma classe de análise, cujo comportamento pode ser disparado por eventos externos, deve ser representada como uma cápsula. Analogamente, se a classe de análise controla ou delega ações a um elemento de projeto já identificado como cápsula, então esta classe deve-se tornar também uma cápsula. A razão é que, conceitualmente, uma classe passiva não pode ter um elemento ativo (cápsula) como atributo e invocar seus serviços diretamente; serviços de uma cápsula devem ser requeridos através de portas que implementam um protocolo de comunicação. Tais serviços devem ser chamados a partir de uma outra cápsula. Durante esta atividade, transformações de modelo consistentes [10], como *Transformar Classe em Cápsula* ou *Extrair Classe,* podem ser aplicadas.

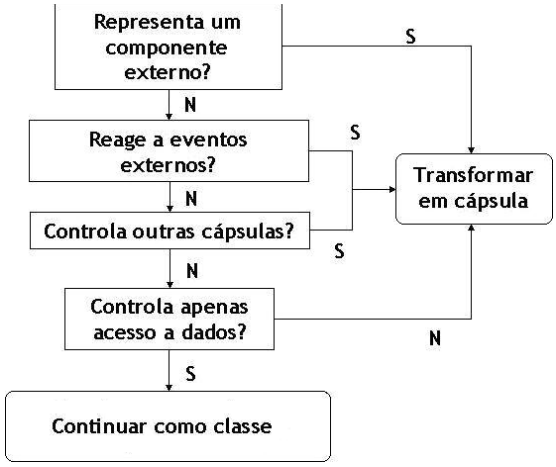


Figura 4.4 – Árvore de decisão para transformar classes em cápsulas

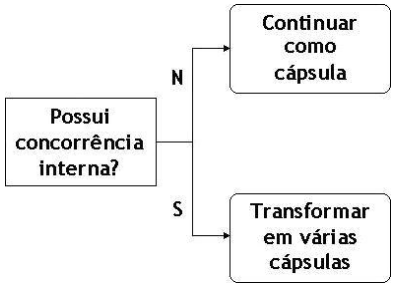


Figura 4.5 – Árvore de decisão sobre cápsulas

Agentes de transporte não são necessários somente para intermediar a comunicação entre processadores fisicamente separados, mas também para liberar processadores de

se preocuparem com o plano de processamento global. Para criar um agente de transporte (*Holon*), precisamos executar as atividades *Identificar Elementos de Projeto*. Para isto, introduzimos uma cápsula intermediaria entre as cápsulas que se comunica com os processadores, como mostra a Figura 4.3.

4.3. Projetar Componentes Ativos

Este novo detalhe de fluxo busca, de maneira complementar e especializada, atingir os mesmos objetivos do detalhe de fluxo original (*Projetar Componentes*), mas enfatizando componentes ativos. O fluxo é conduzido principalmente durante as fases de Elaboração e Construção, onde são refinados os elementos de projeto ativos e atualizadas as realizações de casos de uso, com base em leis de refinamento de modelos apresentados em [10, 12, 14]. Seus objetivos são: elaborar, refinar e decompor a definição dos elementos ativos do projeto, a fim de detalhar como os mesmos realizarão o comportamento no projeto; e refinar e atualizar as realizações de casos de uso com base nos novos elementos ativos de projeto identificados. A seguir apresentamos suas atividades.

4.3.1. Projetar Cápsulas

Nesta atividade, apresentada na Figura 4.6, são elaboradas e refinadas as descrições das cápsulas, devendo ser executada para cada cápsula, incluindo as identificadas no escopo desta atividade. Apesar de ser mencionada no RUP no detalhe de fluxo *Projetar Componentes*, neste trabalho introduzimos novos passos, como exposto a seguir.

Atividade: Projetar Cápsula

Propósito:
- Elaborar e refinar as descrições de uma cápsula.

Papel: Projetista de Cápsula

Freqüência: Quando requerido, tipicamente várias vezes em uma iteração, e mais frequentemente nas iterações de elaboração e construção.

Passos:
- Introdução de Novos Elementos de Projeto
- Criar Portas e Protocolos
- Validar Interações da cápsula
- Definir Máquina de estado da Cápsula
  - Definir Estados
  - Definir Transições
- Definir necessidade de Classes Passivas
- Introduzir Herança à Cápsula
- Validar Comportamento da Cápsula

Artefatos de Entrada	Artefatos de Saída
▪ Cápsula ▪ Eventos ▪ Protocolos	▪ Cápsula ▪ Classes de Projeto ▪ Protocolos

Detalhes do Fluxo
- Análise e Projeto
- Projeto de Componentes Ativos

Figura 4.6 – Atividade de Projetar Cápsulas

Atividade: Refinamento e Decomposição de Cápsula

Propósito:
- Refinar e decompor a descrição de uma cápsula.

Papel: Projetista de Cápsula

Freqüência: Quando requerido, tipicamente várias vezes em uma iteração, e mais frequentemente nas iterações de elaboração e construção.

Passos:
- Identificar Oportunidades de Refinamento
- Realizar Refinamento de Dados
- Realizar Refinamento de Controle
- Realizar Decomposição

Artefatos de Entrada:	Artefatos de Saída:
▪ Cápsula ▪ Eventos ▪ Protocolos	▪ Cápsula ▪ Classes de Projeto ▪ Protocolos

Detalhes do Fluxo:
- Análise e Projeto
- Projeto de Componentes Ativos

Figura 4.7. Atividade de Refinamento e Decomposição de Cápsulas

No passo *Introdução de Novos Elementos de Projeto*, introduzimos elementos de projeto a uma cápsula, como atributos, cápsulas internas, métodos, classes, portas ou relacionamentos. Tais elementos podem ser introduzidos através de leis básicas de UML-RT, como *Introduzir Método* [10]. No passo *Definir Máquina de Estados da Cápsula*, os aspectos comportamentais das cápsulas são definidos em termos de máquinas de estados, onde são definidas as respostas aos eventos e o ciclo de vida dos objetos; desta forma, podemos atualizar o modelo com a inclusão dos novos elementos. No último passo desta atividade, *Validar Comportamento da Cápsula*, o comportamento da cápsula deve ser avaliado e validado, usando a técnica manual *Walk-Through* ou uma ferramenta de simulação automática. Neste momento, o comportamento da cápsula é validado isoladamente do resto do sistema, de maneira similar à execução de teste unitário. Uma validação de todo o sistema será executada no detalhe de fluxo *Validar Arquitetura*.

4.3.2. Refinamento e Decomposição de Cápsulas

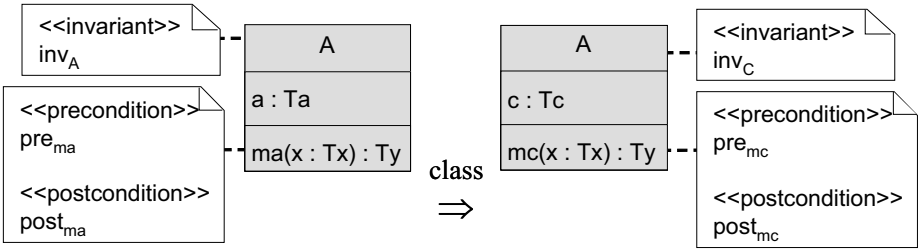
Nesta atividade, apresentada na Figura 4.7, faz-se a evolução das cápsulas através de refinamentos e decomposições, gerando e atualizando classes, cápsulas e protocolos do modelo; este é um processo iterativo que pode incluir, inclusive, cápsulas decompostas anteriormente. Neste processo, cápsulas que representam mais de uma abstração são decompostas (Figura 4.5), quer seja para aumentar a qualidade do modelo, representar entidades identificadas nos requisitos do sistema, ou atender requisitos não funcionais do sistema (como a execução de fluxos de controle concorrentes).

No passo *Identificar Oportunidade de Refinamento*, cada cápsula ou classe deve ser analisada em busca de oportunidades de refinamento. Podemos refinar um componente, realizando: refinamento de dados, onde a estrutura interna dos dados é refinada, buscando uma melhor forma de representação em termos de desempenho ou aderência a padrões arquiteturais e de projeto; um refinamento de controle, onde os elementos comportamentais (máquinas de estados, código, etc.) são refinados, objetivando melhorar o desempenho ou uma melhor distribuição de responsabilidades; e uma decomposição, onde cada componente é decomposto em dois ou mais componentes, buscando aumentar a possibilidade de reuso ou distribuição e diminuir a complexidade. Neste passo, podemos utilizar diversos *guidelines*, dentre eles a análise do comportamento interno das cápsulas para identificar uma possível concorrência interna, indicando a necessidade de transformar em várias cápsulas, conforme Figura 4.5. Este passo é vital para a evolução desta atividade, pois a partir dele identificamos quais passos adicionais serão executados.

Em todos estes passos, o refinamento ou a decomposição deve ser executada através da aplicação de leis de transformação que garantam a preservação do comportamento. Uma vez identificada a necessidade de refinamento de dados, o passo *Realizar Refinamento de Dados* é executado, sendo obtido através da substituição da estrutura interna dos dados, mantendo a interface externa. Para garantir a preservação do comportamento externo observável, propomos a utilização da lei *Refinamento de Dados* [14], Figura 4.8.

Nesta lei, a classe A é refinada mantendo-se a interface original; entretanto, para cada método, é possível enfraquecer a pré-condição ou fortalecer a pós-condição, tornando A mais especializada. Esta lei pode ser aplicada também a cápsulas. Precisamos destacar

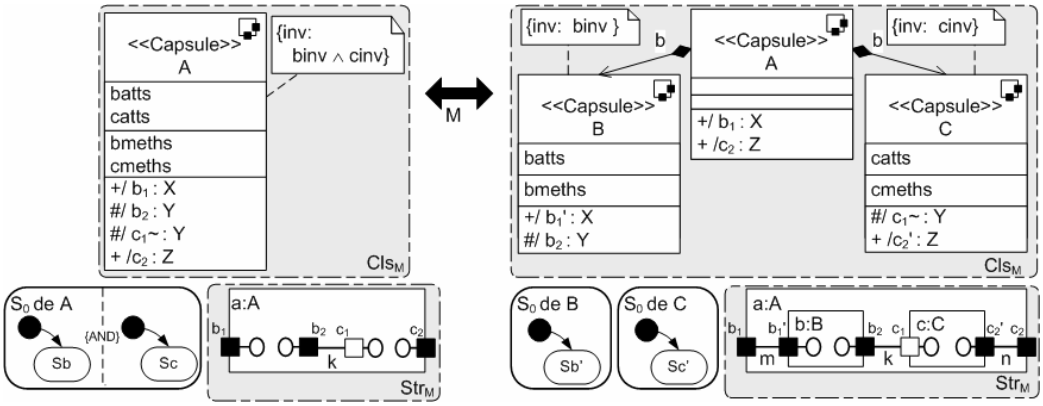
que quando um refinamento de dados ocorre, automaticamente as operações que manipulam os atributos devem ser ajustadas à nova representação.



provided 1. $\forall$ a: Ta; c : Tc; x : Tx | $inv_A \wedge inv_C \bullet pre_{ma} \wedge CI \Rightarrow pre_{mc}$
2. $\forall$ a : Ta; c, c' : Tc; x : Tx; y : Ty | $inv_A \wedge inv_C$ ^ $inv_C' \bullet$
$pre_{ma} \wedge CI \wedge post_{mc} \Rightarrow (\exists a' : Ta \mid inv_A' \bullet CI' \wedge post_{ma})$

Figura 4.8 Lei de Refinamento de Dados

Uma vez identificada a necessidade de refinamento de controle, o passo *Realizar Refinamento de Controle* é executado, através do refinamento dos elementos comportamentais, podendo ser utilizada, por exemplo, a lei *Criar Região* [10], que cria uma nova região no *Statechart*. No passo *Realizar Decomposição*, cada elemento é decomposto em dois outros elementos, aplicando-se, por exemplo, a lei *Decomposição Paralela de Cápsula*, conforme a Figura 4.9, onde uma cápsula A é decomposta no paralelismo de instâncias de cápsulas (B e C) com o propósito de diminuir complexidade e de, potencialmente, aumentar reuso.



Condições: (→) { batts, binv, bmeths, (b_1, b_2), Sb} e { catts, cinv, cmeths, (c_1, c_2), Sc} particionam A;
(↔) As máquinas de estado Sb e Sc são isomórficas à Sb' e Sc', respectivamente, exceto que todas as ocorrências de b_1 e c_2 são substituídas respectivamente, por b_1' e c_2'; os protocolo X e Z possuem uma máquina de estados determinística.

Figura 4.9 Lei de Decomposição Paralela de Cápsula

No lado esquerdo da lei da Figura 4.9, existe a condição de que A deve estar *particionada*. Informalmente, uma cápsula *particionada* é formada por dois grupos disjuntos de variáveis, métodos, portas e regiões do *Statechart*, que se comunicam somente através de portas internas de A; cada partição é proibida de acessar diretamente qualquer elemento da outra parte. O efeito da decomposição é criar duas novas cápsulas B e C, uma para cada partição, e reestruturar a cápsula original (A) para agir como um mediador, preservando sua assinatura e comportamento externo.

A arquitetura de nosso estudo de caso é incrementalmente enriquecida por meio da execução dos detalhes de fluxos de *Analisar Comportamento* e *Projetar Componentes Ativos*. Em particular, durante a atividade *Refinamento e Decomposição de Cápsulas*, decompomos o processador ProdSys nas duas cápsulas ProcessorA e ProcessorB, usando principalmente a *Lei de Decomposição Paralela* (Figura 4.9). Com a introdução destas duas cápsulas, as responsabilidades de ProdSys são divididas entre estes dois novos processadores, e o seu comportamento original é representado pela interação deles. O papel final de ProdSys é somente o de mediar a comunicação entre ProcessorA e ProcessorB através de suas portas com o mundo externo. Como este papel de ProdSys passa a ser irrelevante em nossa arquitetura, ela é eliminada através de leis para remoção de cápsulas [10]. O modelo resultante é apresentado na Figura 4.10.

Mostramos também, na Figura 4.10, a reestruturação do agente de transporte Holon para que este se conecte a todos os elementos do processamento (repositórios e processadores), e possa, assim, gerenciar a programação do processamento. Para realizarmos esta reestruturação, primeiramente, observamos a necessidade de um outro agente de transporte entre os processadores, como na atividade *Identificar Elementos de Projeto* e, depois, compomos os agentes dois a dois (*refactoring* através da aplicação inversa da *Lei de Decomposição Paralela*, Figura 4.9). Como, neste momento, *Holon* possui um papel de um simples *proxy* de comunicação, um refinamento de controle e de dados (durante a atividade *Refinamento e Decomposição de Cápsulas*) poderá ser necessário em uma iteração posterior com o objetivo de criar agentes mais complexos e

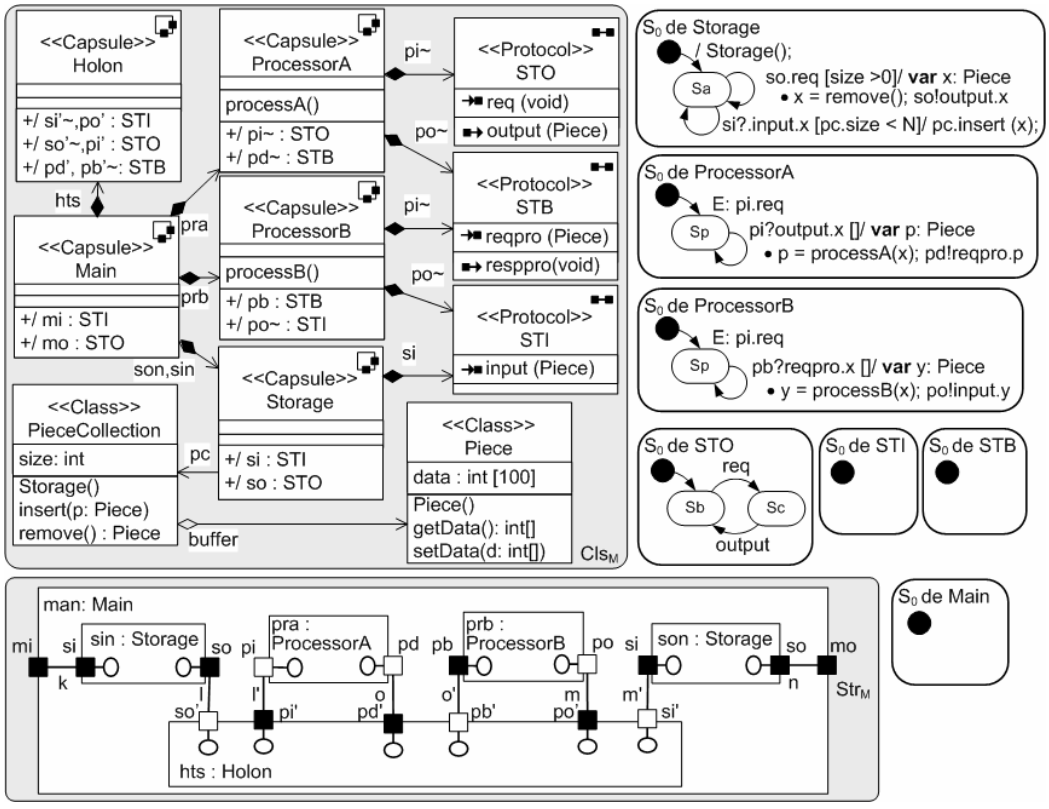


Figura 4.10. Decomposição do processador, projeto de cápsulas e *refactoring* do agente de transporte

que possam lidar com possíveis mudanças na programação do processamento.

Note que a decomposição de cápsulas pode acontecer em diferentes atividades da modelagem, diferenciando-se basicamente pelo papel da cápsula no sistema. Uma cápsula identificada em um **RF**, durante a atividade *Descoberta de Elementos Ativos,* geralmente será considerada na atividade *Identificar Elementos de Projeto*, pois ela representa um elemento necessário na arquitetura para a realização do comportamento do caso de uso. Já uma cápsula identificada em um **RNF**, durante a atividade *Análise de Requisitos Não-Funcionais de Concorrência,* é considerada durante a atividade de *Refinamento e Decomposição de Cápsulas*.

4.5. Validar Arquitetura

Neste novo detalhe de fluxo busca-se garantir que a arquitetura desenvolvida seja robusta, livre de problemas inerentes a sistemas concorrentes como *livelock* e *deadlock,* e que de fato implementa os **RFs** da aplicação. Neste fluxo, todas as cápsulas projetadas são validadas em conjunto, sendo realizado durante as fases Elaboração e Construção. Seus objetivos são: Avaliar e validar a arquitetura considerando propriedades clássicas de concorrência e propriedades específicas da aplicação; e Revisar a arquitetura para refletir possíveis problemas encontrados na validação, possivelmente provocando novas iterações para tratar concorrência.

4.5.1. Validação de Propriedades

Esta atividade foi criada para avaliar o projeto das cápsulas, onde as propriedades clássicas de concorrência são validadas, possibilitando também a validação de propriedades específicas da aplicação, obedecendo ao mesmo formato de outras atividades apresentadas (Figura 4.6 e 4.7).

O RUP indica que o comportamento de cada elemento ativo deve ser avaliado e validado, entretanto, a validação do sistema como um todo (todas as cápsulas), em termos das propriedades clássicas de concorrência não é explorado, muito menos com relação à validação de propriedades específicas da aplicação. Desta forma, consideramos de grande relevância que esta validação ocorra para garantir a robustez da arquitetura, um marco, no próprio RUP, que caracteriza o fim da fase de Elaboração.

Particularmente, propomos o mapeamento dos elementos de UML-RT em uma notação formal, *Circus* [11], uma linguagem formal que combina Z e CSP, e que formaliza todos os passos que envolvem transformações de modelo neste trabalho [10,12,14]. O comportamento das cápsulas de UML-RT, quando desconsideramos ações do *Statechart*, pode ser representado através do modelo de *traces* de CSP, assim como a interação entre cápsulas. Este mapeamento para CSP é uma simplificação do definido para *Circus* em [11].

No passo *Executar a Tradução de Statechart para CSP* são executadas as traduções de todas as máquinas de estados para CSP. No passo *Validar Propriedades Clássicas*, após a tradução de todas as máquinas de estados (de cápsulas e protocolos) para CSP, propriedades clássicas do sistema devem ser validadas, através da ferramenta para verificação de modelos FDR [3], obtendo-se uma verificação parcial da robustez da arquitetura. Para uma verificação mais abrangente, foi criado o passo *Validar Propriedades Específicas*, onde podemos validar todas as regras de negócio que

considerarmos relevantes. Por exemplo, considere uma regra de negócio (propriedade da aplicação) codificada em CSP como um processo RN, assumindo que a aplicação em si está codificada como o processo P, a verificação de que P satisfaz RN pode ser realizada em FDR pelo refinamento: $RN \sqsubseteq P$. Em nosso estudo de caso, por exemplo, pode-se verificar formalmente se toda a peça entregue ao sistema será eventualmente processada ou se obedecerá sempre à programação de processamento.

5. Conclusão e Trabalhos Futuros

Apresentamos uma estratégia rigorosa de modelagem de sistemas concorrentes para UML-RT, através da adaptação e extensão das disciplinas do *Rational Unified Process* (RUP), com foco em Análise e Projeto [5]. Apesar de aspectos de concorrência já serem considerados informalmente no RUP, nossa abordagem propõe e detalha passos consistentes de desenvolvimento que conservam o comportamento do sistema e permite a análise formal de propriedades da aplicação. Estes passos são descritos no contexto de uma estratégia sistemática que integra princípios de um processo de desenvolvimento prático, como o RUP, com resultados consolidados na área de Métodos Formais. Apesar do foco prático deste trabalho, aspectos formais são usados para garantir, por exemplo, que regras de transformação de modelos preservam o significado. Uma vez provadas, estas regras (por exemplo, para decompor cápsulas, Figura 4.7) podem ser usadas pelo desenvolvedor sem necessidade do conhecimento do formalismo utilizado.

Nosso trabalho inclui uma análise de como a concorrência deve ser abordada desde o inicio do desenvolvimento, disciplina de Requisitos, até a disciplina de Análise e Projeto, onde as decisões arquiteturais de um sistema concorrente sofrem seus maiores impactos. Devido ao uso integrado de transformações de modelo consistentes, acreditamos que este trabalho é também uma contribuição original para o desenvolvimento baseado em modelos UML-RT, através da extensão e adaptação de detalhes de fluxo, atividades e *guidelines*.

Relacionado ao desenvolvimento de sistemas concorrentes, podemos citar vários trabalhos [1, 7, 2]. Em [7], uma metodologia de desenvolvimento de modelos UML-RT é apresentada através de transformações de modelo e passos de validação de modelo. Apesar das similaridades deste trabalho com o nosso, ele foca na definição de passos que preservam a integridade entre várias visões do modelo, mas negligenciam a integração destes passos com um processo de desenvolvimento prático, como o RUP [5]. Atividades de decomposição e refinamento de componentes, como definidas na Seção 4, também são relacionadas na literatura em processos baseados em componentes, como Kobra [1], mas este não utiliza uma estratégia formal de refinamentos, como definida em nossa abordagem. Uma extensão do RUP para o desenvolvimento de sistemas baseados em aspectos pode ser encontrada em [2]. Apesar das vantagens deste paradigma para o desenvolvimento de sistemas concorrentes, nenhuma integração com Métodos Formais, capaz de guiar um desenvolvimento rigoroso é apresentada. Além disto, nenhum destes trabalhos apresenta e prova formalmente um conjunto de leis para UML-RT abrangente como o nosso [10].

Um tópico importante para trabalho futuro é a apresentação detalhada de uma atividade de implementação, que integre aspectos práticos e formais, como realizamos para a disciplina de Análise e Projeto. Uma breve discussão desta disciplina é realizada em [5], sugerindo, principalmente, a geração automática de código a partir de diagramas

comportamentais do modelo (por exemplo, *Statecharts*), seguindo a filosofia do desenvolvimento dirigido a modelos. Uma discussão sobre os impactos na visão dinâmica (fases e iterações) do RUP também é realizada em [5], utilizando uma abordagem semelhante a [8].

Referências

[1] Atkinson, C., Bayer, J., Laitenberger O., Zettel, J. Component-Based Software Engineering: The KobrA Approach. International Workshop On Component-Based Software Engineering, 2000.

[2] Cole, L. , Piveta, E., Sampaio, A. RUP Based Analysis and Design with Aspect. XVIII Brazilian Symposium on Software Engineering – SBES, 2004.

[3] Formal Systems (Europe) Ltd. Failures-Divergences Refinement: FDR2 User Manual, 1997. Disponível em http://www.fsel.com.

[4] Fowler, M. UML Essencial - 3ª edição, Porto Alegre, Bookman, 2005.

[5] Godoi, R. Uma disciplina de Análise e Projeto para Aplicações Concorrentes, baseada no RUP. Dissertação de Mestrado. Centro de Informática, Universidade Federal de Pernambuco, Recife, Brasil, 2005.

[6] Kruchten, P.. The Rational Unified Process - An Introduction. Addison-Wesley, 2000.

[7] Küster, J., Engels, G. Consistency Management within Model-Based Object-Oriented Development of Components. 2nd International Symposium on Formal Methods for Components and Objects, LNCS 3188, p. 157-176, 2004.

[8] Tiago Massoni, Augusto Sampaio and Paulo Borba. A RUP-based Software Process Supporting Progressive Implementation. UML and Unified Process. p. 375-387 IRM Press, April 2003

[9] Morgan, C. Programming From Speci_cations. Second edn. Prentice Hall, 1994.

[10] Ramos, R. Desenvolvimento Rigoroso com UML-RT. Dissertação de Mestrado. Centro de Informática, Universidade Federal de Pernambuco, Recife, Brasil, 2005.

[11] Ramos, R., Sampaio, A., Mota, A. A Semantics for UML-RT Active Classes via Mapping into Circus. 7th IFIP International Conference on Formal Methods for Open Object-Based Distributed Systems, Volume 3535 of Lecture Notes in Computer Science, pp. 99-114, Springer (2005)

[12] Ramos, R., Sampaio, A., Mota, A.. Transformation Laws for UML-RT. 8th IFIP international Conference on Formal Methods for Open Object-Based Distributed Systems, Bologna, Italy, 14 - 16 June (2006). To Appear.

[13] Roscoe A. W. The Theory and Practice of Concurrency. Prentice Hall, 1998.

[14] Sampaio, A., Mota, A., Ramos, R. Class and Capsule Refinement in UML for Real Time. Invited Paper. Brazilian Workshop on Formal Methods. Volume 95 of Electronic Notes in Theoretical Computer Science, pp. 23–51, Elsevier Science, 2004

[15] Selic, B., Rumbaugh, J.. Using UML For Modeling Complex RealTime Systems. Rational Software Corporation, 1998. Disponível em http://www. rational.com.

[16] Spivey, M.. The Z Notation: A Reference Manual. Second edn. Prentice Hall, 1992.

[17] Woodcock, J., Cavalcanti, A. Semantics of circus, the. In the Formal Specification and Development in Z and B Conference , Volume 2272 of Lecture Notes in Computer Science, pp. 184–203. Springer-Verlag. 2002.

Component-Based Groupware Development Based on the 3C Collaboration Model

Marco Aurélio Gerosa[1], Alberto Barbosa Raposo[2],
Hugo Fuks[1], Carlos José Pereira de Lucena[1]

[1]Laboratório de Engenharia de Software (LES), Departamento de Informática
[2]Grupo de Tecnologias em Computação Gráfica (Tecgraf), Departamento de Informática
Pontifícia Universidade Católica do Rio de Janeiro – PUC-Rio
R. Marquês de São Vicente, 225, Rio de Janeiro, RJ, 22453-900, Brasil

{gerosa, hugo, lucena}@inf.puc-rio.br, abraposo@tecgraf.puc-rio.br

Abstract. *Groupware is evolutionary and has specific difficulties of development and maintenance. Its code normally becomes unstructured and difficult to evolve. In this paper, a groupware development approach based on components organized according to the 3C collaboration model is proposed. In this model, collaboration is analyzed based on communication, coordination and cooperation. Collaboration requirements, analyzed based on the 3C model, are mapped onto software components, also organized according to the model. The proposed approach is investigated as a case study to the development of the new version of the AulaNet environment. The environment's code currently suffers from the aforementioned problems. In order to instantiate the environment's communication services, 3C based component kits were developed for the case study. The components allow composition, re-composition and customization of services to reflect changes in the collaboration dynamics.*

Resumo. *Groupware é evolucionário e possui dificuldades específicas de desenvolvimento e manutenção. Seu código normalmente se torna desestruturado e difícil de evoluir. Neste artigo, é proposta uma abordagem de desenvolvimento de groupware baseado em componentes organizados em função do modelo 3C de colaboração. Neste modelo, a colaboração é analisada a partir da comunicação, coordenação e cooperação. Os requisitos de colaboração do grupo, analisados em função do modelo 3C, são mapeados em componentes de software, também organizados em função do modelo. A abordagem proposta é investigada num estudo de caso no desenvolvimento da nova versão do ambiente AulaNet. O código do ambiente atualmente sofre dos problemas mencionados. Para instanciar os serviços de comunicação do AulaNet, kits de componentes foram desenvolvidos. Os componentes possibilitam a composição e customização dos serviços para refletir mudanças na dinâmica da colaboração.*

1. INTRODUCTION

Douglas Engelbart [1968] pointed out the relevance of applications for office automation, hypertext and groups. Today the first two are widely available, used and commercially accepted, while groupware technology is still perceived to be unstable and commercially risky, generating few products [Greenberg 2006]. In most companies, computational support for collaboration is limited to systems for exchanging messages or filing documents.

Groupware technology has not attained its potential yet. Research on CSCW is now at a fairly advanced stage, although it lacks a manner of simplifying the programming of collaborative applications and of promoting a critical mass of users. Groupware development still requires qualified programmers trained to deal with protocols, connections, resource sharing, distribution, rendering, session management, etc. This limits the number of developers active in the area and misplaces the creativity and efforts of these developers, taking their attention out from the creation of solutions to the solving of low-level technical problems, disrupting the investigation of collaboration support.

This kind of problems in developing groupware are experienced in the development and maintenance of the AulaNet environment. AulaNet is a web-based environment for teaching and learning. AulaNet has been under development since 1997 and is widely used. The AulaNet development group is made up of doctoral, masters and graduate students who, as well as maintaining the software, use it in their theses, dissertations and monographs, implementing and testing the concepts produced in their work. The system has grown through prototyping, while its functionalities have been implemented in an evolving fashion. The constant changes in the collaboration support and the evolution of the technologies used has made the application's code strongly linked and with a low level of cohesiveness. Technical aspects permeate the entire code, getting mixed with the collaboration support, diverting the developer's attention. Changes in the environment are reflected in various parts of the code and cause undesirable collateral effects, hindering the evolution of the environment, integration of new members to the development team and integration with the company responsible for distributing and customizing the environment.

This scenario illustrates the need to support groupware development, allowing developers to build an extendable system, in a way more suited to accompanying the evolution of collaboration support and the characteristics of tasks and groups. The low-level complexities should be encapsulated and the investigation of interaction through prototyping should be better supported. Non-specialized developers should be able to adapt and reconfigure the solution for their specific needs – a desirable aim, given that there is no way of foreseeing all collaboration demands [Pumareja et al. 2004].

This article proposes the use of 3C based components as a means of developing extendable groupware whose assembly is determined by collaboration needs. By analysing the problem from the viewpoint of the 3C model and using a component structure designed for this model, changes in the collaboration dynamics are mapped onto the computational support. This way, the developer has a workbench with a

component-based infrastructure designed specifically for groupware, based on a collaboration model.

By designing and developing collaboration tools in the form of software components, the developer is given the means to assemble a specific groupware for the collaboration needs of each group. Tools are selected from a component kit based on the 3C model for the support of the established dynamics. The developer selects the most suited components from those of the same family.

The proposed approach is being applied to the re-development of the AulaNet environment. A layered architecture is defined comprising component frameworks and collaboration components.

2. COMPONENT BASED GROUPWARE

Szyperski [2003] lists four main reasons for using component software. The first and oldest one is related to the idea of a *components market* in which companies search and purchase components. The second reason is related to *product line*. Components are developed with the aim of reusing them in various systems, reducing the total investment and maintenance costs. The third is related to the idea of assembly by the final user (*tailorability*). The fourth reason is related to the use of *dynamic services*, discovered and installed as they become necessary. Some component based groupware are presented bellow.

The LIVE platform [Banavar et al. 1998] provides support for the construction of synchronous groupware. The component model used by LIVE is based on the JavaBeans specification and supplies a high-level interface for developing groupware. DISCIPLINE [Marsic 1999] is a platform designed for the development of synchronous groupware for the educational domain. Its architecture comprises replicated components and resources centralized on the server.

FreEvolve [Won et al. 2005], previously called EVOLVE [Stiemerling et al. 1999] (before its release under the GPL license), is a component-based system developed on a client-server architecture on the Internet. FreEvolve was designed to enable adaptation and assembly by final users of the application. The component model used in FreEvolve is called FlexiBeans, an extension of JavaBeans.

The DACIA platform (Dynamic Adjustment of Component InterActions) [Litiu & Prakash 2000] is aimed towards the development of groupware for mobile devices. The platform defines its own component model. In this model, a component is called PROC (Processing and ROuting Component).

DreamTeam [Roth & Unger 2000] is a component-based platform for assembling synchronous groupware. The platform provides a development environment, with specific tools, an execution environment and a simulation environment. The components are called TeamComponents and are associated with user interface and data manipulation components. The developer is supplied with groupware components, user interface components and data manipulation components.

The CoCoWare platform [Slagter & Biemans 2000] offers final users the capability to assemble the application according to their needs and extend it to follow the evolution of work processes. CoCoWare offers components for dealing with work sessions and

managing collaborative tools, providing information on what can be modified, how it can be done and the impact of the modifications.

GroupKit [Roseman & Greenberg 1996] is a toolkit containing components and an execution platform. GroupKit uses Tcl/Tk and is aimed towards the development of synchronous groupware. The toolkit encapsulates various complexities inherent in this type of application, meaning developers can focus their attention on the interaction.

The approaches found in the literature concerning the use of software components in groupware development basically focus on the second and third reasons previously identified by Szyperski [2003] (*product line* and *tailorability*). Some systems provide tools for building collaborative applications based on interoperable components, while some others offer final users assembly. There is no standard component model.

The proposed approach focuses on the second reason (*product line*), aiming to provide groupware developers with the means to assemble collaborative systems based on the 3C collaboration model. The literature presents many proposals for using software components for groupware development. However, none of these use the 3C collaboration model as a basis for designing and organizing software components and the development process.

3. THE 3C COLLABORATION MODEL

The 3C collaboration model is based on the idea that to collaborate, members of a group communicate, coordinate and cooperate. The 3C model derives from the seminal article by Ellis et al. [1991]. The model proposed by Ellis et al. is used to classify computational support for collaboration. In this article, the 3C model is used as a basis for modeling and developing groupware. There is also a difference in terminology; the joint operation in the shared workspace is denominated collaboration by Ellis, while it is denominated cooperation in the 3C model.

The 3C model is equivalent to the Clover model [Laurillau & Nigay 2002], with a slight denomination difference: communication, coordination and production. Differently from the Clover model, in this article the 3C model guides the development of component groupware and serves as a basis for a component-based architecture.

The 3C model is widely used in the literature. Bandinelli et al. [1996] use the three dimensions of the 3C model to improve the computational support of software processes, especially communication and cooperation, which, according to them, are not adequately treated by traditional processes, which privilege coordination. Bretain et al. [1997] use the three Cs as a basis to analyze and interview groups whose activities are conducted outdoors, such as firefighters, plumbers, reporters and sales representatives. Sauter et al. [1995] and Borghoff & Schlichter [2000] use the three Cs to classify collaborative tools. Marsic & Dorohoceanu [2003] use the three Cs to analyze user interface elements.

A diagram of the 3C model is shown in Figure 1. Communication involves the exchange of messages and the negotiation of commitments. Coordination enables people, activities and resources to be managed so as to resolve conflicts and facilitate communication and cooperation. Cooperation is the joint production of members of a group within a shared space, generating and manipulating cooperation objects in order to complete tasks [Fuks et al. 2005]. Despite their separation for analytic purposes,

communication, coordination and cooperation should not be seen in an isolated fashion; there is a constant interplay between them.

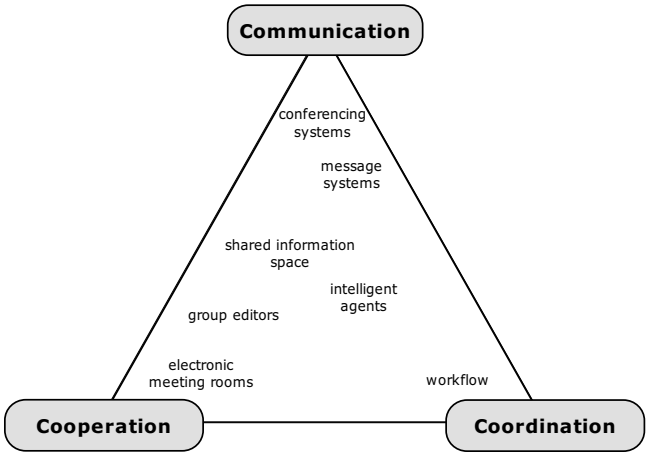


Figure 1. Diagram of the 3C collaboration model [Borghoff & Schlichter 2000]

Groupware such as chat, for example, which is a communication service, requires communication (exchange of messages), coordination (access policies) and cooperation (registration and sharing).

4. ASSEMBLY OF GROUPWARE AND COLLABORATIVE SERVICES

A collaboration environment normally offers its participants a set of collaborative services to be used in the different moments of collaboration. Table 1 shows the collaboration services found in the following environments: AulaNet (http://www.eduweb.com.br), TelEduc (http://teleduc.nied.unicamp.br), AVA (http://ava.unisinos.br), WebCT (http://www.webct.com) and Moodle (http://www.moodle.org), all from the educational domain, and GroupSystems (http://www.groupsystems.com), YahooGroups (http://groups.yahoo.com), OpenGroupware (http://www.opengroupware.org) and BSCW (http://bscw.fit.fraunhofer.de), designed for group work.

	Communication Services							Coordination Services								Cooperation Services											
	Mail	Discussion List	Forum	Mural	Brainstorming	Chat	Messenger	Agenda	Activities Report	Participation Monitoring	Questionnaire	Tasks	SubGroups	Resource	Guidance	Voting	Repositories	White Board	Search	Glossary	Links	Cooperative Journal	Classifier	Wiki	Contact Manager	Peer Review	FAQ	Notes	RSS
AulaNet	X	X	X			X	X		X	X	X	X	X			X	X				X							X	
TelEduc	X		X	X		X		X	X	X	X	X	X				X				X						X	X	
AVA	X		X	X		X		X	X	X	X	X			X		X			X	X						X	X	
WebCT	X		X			X		X	X		X	X				X	X	X	X	X							X		
Moodle			X			X	X	X	X	X	X	X	X			X	X		X	X	X	X		X		X	X		X
GroupSystems			X		X			X		X	X	X	X			X							X					X	
YahooGroups		X				X		X	X		X					X	X		X	X	X								X
OpenGroupware	X			X				X				X		X											X				
BSCW			X					X	X			X	X			X	X		X		X				X				

Table 1. Collaborative services

As the table shows, there are many similar services normally found in a variety of environments. For example, most of the analyzed systems provide forums, chat, agenda, activity reports, questionnaires, task management, voting, repository and links. Additionally, each service may be seen as an independent unit. These characteristics propitiate the application of component-based development techniques, where the collaborative services are the groupware components.

Services may be classified according to their purposes and characteristics according to the 3C model. The services in Table 1 are organized into communication, coordination and cooperation. A unified component model would let the developer to select from the most suitable service from all those belonging the same family. Moreover, as can be noted in Table 1, there are services that are provided by a groupware solution, but given that they are not in a component form they cannot be made available for use in other solutions. For users, it is interesting to interchange services in order to complement environments and reuse experiences.

The same rationale that was used for environment and its services, may be used for services and their functionalities. For example, almost every chat has a shared area where messages are displayed, a list of connected participants and an area for writing messages. By using a component-based architecture, these functionalities may be reused.

Moodle's chat service displays the time that the participant remained inactive, a beep for attracting a participant's attention and each participant's photo. WebCT provides support for private messages and notifies when someone enters the chat room. Encapsulating these functionalities into components would also allow other developers to use them in their projects. It also becomes possible to evolve, adjust and build services by varying and reconfiguring the collaboration components.

This analysis leads to the adoption of software components at two levels, as illustrated in Figure 2. The first level comprises the components that implement the communication, coordination and cooperation services, used to offer computational support to the collaboration dynamics as a whole. The second level comprises the components used to assemble the aforementioned services, providing specific support to communication, coordination and cooperation within the dynamics of a particular service. The components that implement the collaborative services are called *services* and the components used to implement the computational support for service collaboration are called *collaboration components*.

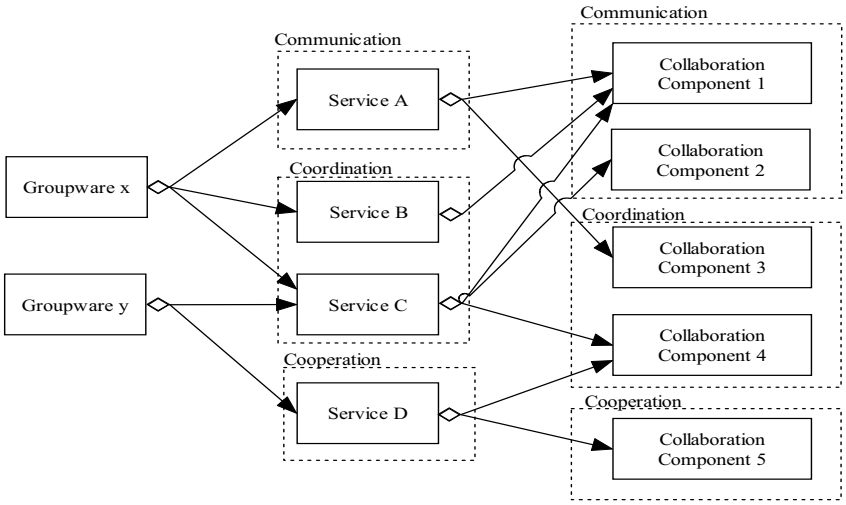


Figure 2. Groupware composition

A component-based groupware environment comprises communication, coordination and cooperation services, which may be reused in many other environments. The services share collaboration components that implement collaboration support, modeled in this article based on the 3C model. Based on component kits organized according to the 3C model, the developer assembles an application to provide support to the collaboration dynamics. The collaboration components of a C are reused in services of the other Cs.

4.1. The Collaboration Component Kit

Domain engineering aims to provide components that implement the concepts of a software domain and may be reused to implement new applications on this domain. By mapping domain concepts onto components, the chances of reusing components in all development phases increases. Using this approach, the components are coded (space of the solution) according to the needs of the domain (space of the problem) [D'Souza & Wills 1998]. Domain engineering is well suited to use in domains presenting complex processes and characteristics, where there are modeling difficulties when using traditional processes, which is the case of CSCW and groupware.

In this work, the domain analysis, the first step of domain engineering, was based on the literature and on the knowledge accumulated by the AulaNet development group, which has nine years of experience in developing tools for collaboration. The domain analysis was restricted to communication tools, which in addition to their communication elements present a representative cross-section of coordination and cooperation elements.

A communication service deals with messages, which use textual, video, audio or pictorial media [Daft & Lengel 1986]. The media sometimes present a degree of variability, such as limits on text size or vocabulary, in the case of textual media, and the rate of data capture and transmission, in the case of audio and video. Some services send email to recipients, enable attachment of files and also provide a spell-checker to help write text. Some functionalities such as message categorization, conversation paths and commitment stores appear in systems, namely AulaNet Conferences [Fuks et al. 2002], ACCORD [Laufer & Fuks 1995] and Coordinator [Winograd & Flores 1987].

Messages are organized in a linear, hierarchical or network dialogue structure [Stahl 2001] and may be transmitted in blocks or continuously.

Support for coordination in a communication service is related to channel access policies, task and participant management and participation monitoring. Communication services present management of access permissions associated with participants or roles. Roles are associated to tasks within the scope of an activity. Sometimes, tasks are enacted by a workflow engine. Message assessment provides information for evaluating the competency of participants, used to define the dynamics of the activities, the association of roles and the definition of subgroups. Participation takes place in the context of a session and is based on awareness information, such as the blinking that informs that a participant is writing a message. Some services provide information on the presence and availability of participants.

Support for cooperation in a communication service is related to the archiving and handling of information. The content of communication sessions is registered in repositories in the form of cooperation objects. These objects are associated with a version manager, access registration, statistical analysis, trash bin, recommendation system, search mechanism and ranking mechanism.

A component kit is a collection of components designed to work as a set [D'Souza & Wills 1998]. A family of applications can be generated from a component kit, using different combinations and sometimes developing other components on demand. A component kit does not need to be exhaustive. Component kits are extendable, allowing new components to be included as necessary. To be reusable, software components should be refined repeatedly until they reach the desired maturity, reliability and adaptability [D'Souza & Wills 1998].

Collaboration components are used to assemble services implementing the collaboration aspects. The collaboration component kit that was obtained from the domain analysis is shown in Table 2.

COMMUNICATION	COORDINATION	COOPERATION
MessageMgr	AssessmentMgr	CooperationObjMgr
TextualMediaMgr	RoleMgr	SearchMgr
VideoMediaMgr	PermissionMgr	VersionMgr
AudioMediaMgr	ParticipantMgr	StatisticalAnalysisMgr
PictorialMediaMgr	GroupMgr	RankingMgr
DiscreteChannelMgr	SessionMgr	RecommendationMgr
ContinuousChannelMgr	FloorControlMgr	LogMgr
MetaInformationMgr	TaskMgr	AccessRegistrationMgr
CategorizationMgr	AwarenessMgr	TrashBinMgr
DialogStructureMgr	CompetencyMgr	
ConversationPathsMgr	AvailabilityMgr	
CommitmentMgr	NotificationMgr	

Table 2. Collaboration Component Kit

Component frameworks [Syzperski 1997] are used to provide support to the management and execution of the components. In the proposed architecture, a component framework is used for each proposed component type (service, collaboration), allowing the peculiarities of each one to be met. Services are plugged

into the Service Component Framework for the assembling of the groupware environment, and collaboration components are plugged into the Collaboration Component Framework for the assembling of the services. Component frameworks are responsible for handling the installation, removal, updating, deactivation, localization, configuration, monitoring, and import and export of components. The Service Component Framework manages the instances of the services and their links to the corresponding collaboration components. The Collaboration Component Framework manages the instances of collaboration components derived from the Collaboration Component Kit.

Most of the functionalities of the component frameworks are recurrent and reusable. A framework can be used for the instantiation of a family of systems. In this article, a framework is used to instantiate the component frameworks. This type of framework is called a *component framework framework* (CFF) [Szyperski 1997, p.277]. A component framework framework is conceived as a second-order component framework whose components are component frameworks. Just as a component interacts with others directly or indirectly via the component framework, the same applies to component frameworks, whose highest level support is the component framework framework.. Extending the notion used by Szyperski [1997], Figure 3 illustrates the application architecture, including the Groupware Component Framework Framework, as a second-order component framework.

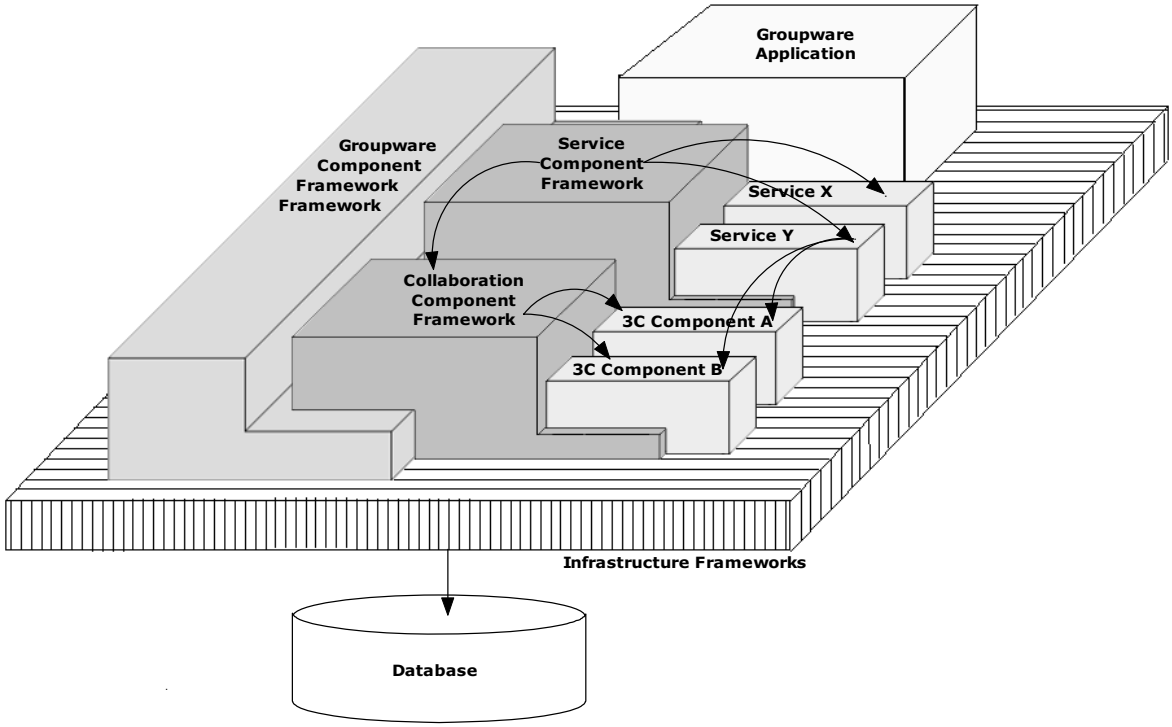


Figure 3. The proposed architecture

The proposed architecture is divided in layers, comprising the presentation layer (not shown in Figure 3), for the capture and presentation of data and for user interaction; the business layer, which captures the model of the business logic of the application's domain; and the infrastructure layer, which implements the low-level technical services.

The division in layers is important in responding to the complexity of component-based systems [Szyperski 1997].

The same infrastructure developed for the business layer can be used for more than one presentation. When the business layer services need remote access to a PDA client, for example, web services are made available to encapsulate the façade of the business layer. In other cases, the presentation directly accesses the business façade.

The application's architecture reflects the structure of the domain's components, representing a high level logical project independent of the support technology [D'Souza & Wills 1998]. The components plugged in the business layer implement the concepts of the 3C collaboration model.

The same service may have many independent instances. For example, in the case of the AulaNet environment, a component instance is created for each course that uses the same service. The Service Component Framework manages the component instances and keeps the current state of each of them, enabling restoration at a later date. Whenever a new instance is created, the standard values defined in the descriptor file are used.

The instantiation of the collaboration components used in assembling the service follows the service instantiation. The Service Component Framework interacts with the Collaboration Component Framework to enable the instantiation and the association between the instances of the components. In order to reduce the coupling between the two component frameworks, a contract interface is used.

Installation and management of the collaboration components follows a procedure similar to that described for the services. The collaboration components have descriptor files that define standard configurations, used in the instantiation of the components. The Collaboration Component Framework manages the configuration of the collaboration components.

5. CASE STUDY

This section presents some case studies conducted in order to evaluate the proposed approach.

5.1. The AulaNet environment

The new version of AulaNet is being completely rewritten, using the approach proposed in this article. This version makes use of components based on the 3C model to encapsulate a cohesive set of data and functions. The component frameworks encapsulate and provide low-level services, providing support towards the development and maintenance of groupware.

In designing the computational support for collaboration, services are selected for each of the activities. Based on the feedback obtained from the use of the services, the computational support for collaboration is continually adjusted. If there is a change to the course's dynamics, the environment is reassembled by adding, replacing or removing services. As well as the inclusion of new services, the feedback obtained from the use of the environment may lead to the replacement of existing services. For example, if the coordinators of a course conclude that learners are having difficulties

with the Debate service, they may replace it with a regular Chat, which presents a simpler interface with less functionality. A problem that may arise with this substitution is that sessions, assessments, participations, etc. remain registered from the use of the previous service. If the two services use the same components for message recording, then the environment's import/export functionality can be used to transfer data.

Encapsulating services into components enables the same service to be deployed with different configurations and characteristics for handling distinct tasks. For example, in one of the environment's courses, the Conference is used for the argumentation on the weekly argumentation and for peer evaluation. Each installed component is specifically named and configured. Thus, each service's sessions are independent, enabling more detailed reports and statistics, more precision in searches and the adoption of different categories, roles, permissions and evaluation criteria. Any service may be duplicated by deploying it twice (duplicating the corresponding file in the directory structure).

In order to encapsulate a service not originally developed for the environment, it is necessary to create a package that supports its installation. It is necessary to create the descriptor file, the scripts and the directory structure defined in the AulaNet component model.

Some modifications may be solved by customizing rather than replacement of services. For example, to make it possible the deactivation of the Conferences service, the corresponding property in the component's descriptor file must be available as a parameter.

5.1.1 The Debate Service

An early version of the Debate service was implemented using a communication component, tailored for synchronous communication protocols, and a cooperation component, which implements a plain shared space. This version of Debate is a typical chat service, containing an expression element, where learners type their messages, and awareness elements, where messages from learners taking part in the chat session are displayed, as shown in Figure 4.

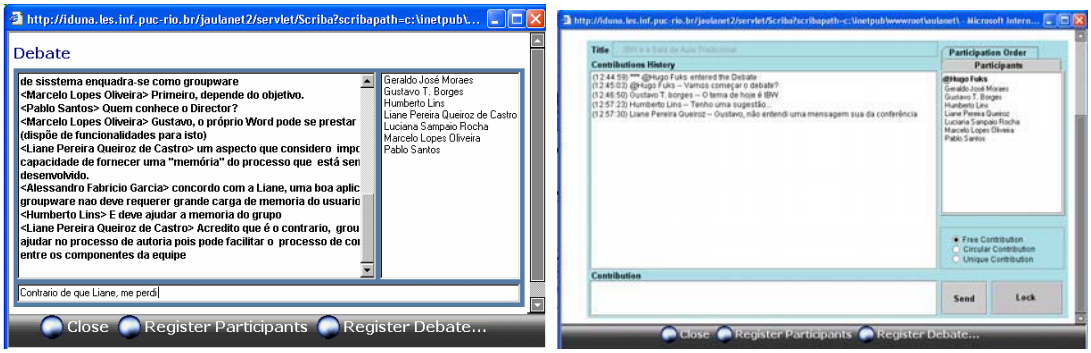


Figure 4. Early Debate interface (left) and current Debate interface (right)

The early version provided no support for coordination, leaving it to the standing social protocol. However, some courses that use a well-defined procedure for the debate activity, such as the one shown in Figure 5, need effective coordination support. Floor control, participation order and shared space blocking ability were added to the service.

The shared space was also enhanced with new awareness elements, like session title, timestamp and identification of mediators.

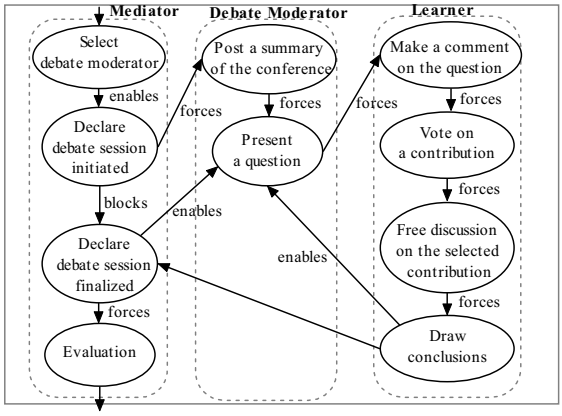


Figure 5. Expanded debate workflow

The same communication component was used for the new version of Debate, given that the synchronous communication protocols and the message characteristics remained the same. The cooperation component, which implements the shared space, was also enhanced with new awareness elements.

This example illustrates the usage for a component-based architecture capable of dealing with the three Cs of the collaboration model. Table 3 presents the composition of both versions of the tool.

Early version	**Current version**
MessageMgr	MessageMgr
DiscreteChannelMgr	DiscreteChannelMgr
AssessmentMgr	AssessmentMgr
RoleMgr	RoleMgr
PermissionMgr	PermissionMgr
ParticipantMgr	ParticipantMgr
SessionMgr	SessionMgr
CooperationObjMgr	AwarenessMgr
	CooperationObjMgr
	FloorControlMgr

Table 3. 3C components used in the Chat and Debate services

The collaborative service was extended to follow the evolution of the work dynamics. The use of the 3C model allowed an isolated analysis of the necessities and difficulties of each collaboration aspect. Based on this analysis a more suitable service was assembled, mapping collaboration necessities onto software components, both of them organized according to the 3C collaboration model.

5.2 Case Study on a Course

The proposed approach was also used on the Groupware Engineering course at PUC-Rio's Computer Science Department. The case study was conducted on the second semester of 2005, with two undergraduate, 3 masters and 2 doctoral students. The result

of the case study was evaluated via direct observation by course lecturers and the application of individual questionnaires.

Each student selected an application and analyzed its functionalities, classifying them as communication, coordination or cooperation. The student also presented an architecture and a prototype that offers support to a extension of the system, using the infrastructure and the 3C components. They succeeded in using the components for designing groupware.

The students also received and replied to a questionnaire. Most of them evaluated as moderate the level of difficult of using the 3C model for the analysis of the chosen application, on a scale from very difficult to very easy. The understanding of the 3C model was also considered moderate. These results were considered satisfactory, given that the students had their first contact with the 3C model during the course and that they are not specialists in groupware. Regarding the reach of the 3C model in system modeling, 5 students evaluated it as sufficient and 2 as fair. In relation to the use of 3C components, 5 students identified the solution as complex and 2 as normal, in a scale ranging from very simple to very complex. This result was also considered satisfactory, given that, in addition to not being specialists in groupware, the students are not specialists in software components either. Although the majority classified the solution as complex, all of them evaluated the utilization of 3C components in the assembly of groupware as good or very good. In relation to the encapsulation of low-level complexities, 3 students evaluated the solution as neutral, 2 as good and 2 as very good. Finally, in evaluating the computational support for groupware using 3C components, 2 students evaluated the solution as neutral and 5 as good. The results obtained in the questionnaire were in general positive.

6. CONCLUSION

This research proposes the structuring of a collaborative system using components that encapsulate the technical difficulties of distributed and multi-user systems and reflect the concepts of collaboration modeled by the 3C model. In the context of this work, domain engineering is based on the 3C collaboration model. The transition between development activities and the mapping of analytic concepts onto the structures of the code is narrowed, facilitating iterative development and future maintenance of the application. Component based development together with specific domain concepts has shown itself as a feasible approach to groupware development.

In developing groupware, the requirements are rarely clear enough to allow for a precise specification of the system's behavior in advance. It is difficult to predict how a particular group will collaborate and each group has highly distinct characteristics and objectives [Gutwin & Greenberg 2000]. By involving a group, the possibilities of interactions multiply and the demand for synchronism and solving deadlocks increases, posing problems in the construction of suitable interaction mechanisms and conducting tests. Using a set of components that are reused in diverse situations increases the system's reliability and stability, as well as allows the replacement of components with limited impact. The developer can also prototype different configurations in order to refine system's requirements and collaboration support. Providing a component kit mitigates the need to anticipate and provide support for all the potential uses. Medium

granularity blocks are provided, which the developer uses to assemble the application [Szyperski 1997]. The reuse provided by the components also reduces the amount of lines of code. For example, the AulaNet conference service has 4279 exclusive lines of code (not used in other services). In the new version, it has 2905 lines of code, where just 317 are related to the business layer. The rest is related to the presentation layer, which remained the same.

"Without an adequate architecture, the construction of groupware and interactive systems in general is difficult to maintain and iterative refinement is hindered" [Calvary et al. 1997]. A component-based architecture allows components to be selected to assemble a groupware solution meeting a group's specific interests. The components are customized and combined as required, keeping in mind future maintenance. The use of this approach enables prototyping and experimentation, which are fundamental in CSCW, given that the success cases are very few and poorly documented. The use of components improves the dynamics adaptation of the environment and the support for collaboration through the system's reassembly and reconfiguration.

However, it is worth stressing that the proposed solution does not eliminate the need for an aware developer who is knowledgeable about the subject in question. It is not enough to link the components randomly to produce an effective collaborative system.

References

Banavar, G., Doddapaneti, S., Miller, K. & Mukherjee, B. (1998) Rapidly Building Synchronous Collaborative Applications by Direct Manipulation. In Proceedings of the 1998 ACM Conference on Computer Supported Cooperative Work (CSCW'98), pp. 139-148.

Bandinelli, S., Nitto, E.D. & Fuggetta, A. (1996) "Supporting cooperation in the SPADE-1 Environment", IEEE Transactions on Software Engineering, V 22, N 12, pp. 841-865

Borghoff, U.M. & Schlichter, J.H. (2000) Computer-Supported Cooperative Work: Introduction to Distributed Applications. Springer, USA.

Bretain, I., Fredin, L., Frost, W., Hedman, L.R., Kroon, P., McGlashan, S., Sallnas, E.L. & Virtanen, M. (1997) Leave the Office, Bring Your Colleagues: Design Solutions for Mobile Teamworkers. Proc. CHI'97, ACM Press, pp.335-336

Calvary, G., Coutaz, J. & Nigay, L. (1997) From Single-User Architectural Design to PAC*: a Generic Software Architectural Model for CSCW. Conference on Human Factors in Computing Systems (CHI'97), pp 242-249.

D'Souza, D.F. & Wills, A.C. (1998) Objects, Components and Frameworks with UML: The Catalysis Approach. Addison Wesley, ISBN 0-201-31012-0, 1998.

Daft, R.L. & Lengel, R.H. (1986). Organizational information requirements, media richness and structural design. Management Science 32(5), 554-571.

Ellis, C.A., Gibbs, S.J. & Rein, G.L. (1991) Groupware - Some Issues and Experiences. Communications of the ACM, Vol. 34, No. 1, pp. 38-58.

Engelbart, D. & English, W. (1968) Research Center for Augmenting Human Intellect, Proc. Fall Joint Computing Conference, AFIPS Press, 395-410

Fuks, H., Gerosa, M.A. & Lucena, C.J.P. (2002), "The Development and Application of Distance Learning on the Internet", Open Learning Journal, V. 17, No. 1, February 2002, ISSN 0268-0513, Cartafax Pub, pp. 23-38.

Fuks, H., Raposo, A.B., Gerosa, M.A. & Lucena, C.J.P. (2005) Applying the 3C Model to Groupware Development. International Journal of Cooperative Information Systems (IJCIS), v.14, n.2-3, Jun-Sep 2005, World Scientific, ISSN 0218-8430, pp. 299-328.

Greenberg, S. (2006) "Toolkits and Interface Creativity", Journal of Multimedia Tools and Applications, Special Issue on Groupware, Kluwer. In Press. Disponível em http://grouplab.cpsc.ucalgary.ca/papers

Gutwin, C. & Greenberg, S. (2000) The Mechanics of Collaboration: Developing Low Cost Usability Evaluation Methods for Shared Workspaces. IEEE 9th Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises -WETICE (2000), p. 98-103.

Laufer, C. & Fuks, H. (1995) "ACCORD: Conversation Clichés for Cooperation", Proceedings of The International Workshop on the Design of Cooperative Systems, France, pp 351-369.

Laurillau, Y. & Nigay, L. (2002) "Clover architecture for groupware", Proceedings of the Conference on Computer-Supported Cooperative Work (CSCW 2002), pp. 236 - 245

Litiu, R. & Prakash, A. (2000) "Developing Adaptive Groupware Applications Using a Mobile Computing Framework", Proceedings of the ACM Conference on Computer Supported Cooperative Work (CSCW'00), pp. 107-116.

Marsic, I. & Dorohonceanu, B. (2003) "Flexible User Interfaces for Group Collaboration". International Journal of Human-Computer Interaction, Vol.15, No.3, pp. 337-360

Marsic, I. (1999) DISCIPLE: a framework for multimodal collaboration in heterogeneous environments. ACM Computing Surveys, 31 (2es), Article No. 4.

Pumareja, D., Sikkel, K. & Wieringa, R. (2004) "Understanding the dynamics of requirements evolution: a comparative case study of groupware implementation", REFSQ 2004, Essener Informatik Beiträge 9, pp. 177-194.

Roseman, M. & Greenberg, S. (1996) "Building real time groupware with GroupKit, a groupware toolkit". ACM Transactions on Computer-Human Interaction, 3, 1, p. 66-106.

Roth, J. & Unger, C. (2000) Developing synchronous collaborative applications with TeamComponents. In Designing Cooperative Systems: the Use of Theories and Models, 5th International Conference on the Design of Cooperative Systems (COOP'00), pp. 353-368.

Sauter, C., Morger, O., Muhlherr, M., Thutchytson, A. & Teusel, S. (1995) CSCW for Strategic Management in Swiss Enterprises: an Empirical Study. Proceedings of the 4th European Conference on Computer Supported Cooperative Work (ECSCW'95), Sweden, 117-132

Slagter, R.J. & Biemans, M.C.M. (2000) "Component Groupware: A Basis for Tailorable Solutions that Can Evolve with the Supported Task", in Proceedings of the International ICSC Conference on Intelligent Systems and Applications (ISA 2000), Australia.

Stahl, G. (2001) WebGuide: Guiding collaborative learning on the Web with perspectives, Journal of Interactive Media in Education.

Stiemerling, O., Hinken, R. & Cremers, A.B. (1999) The EVOLVE Tailoring Platform: Supporting the Evolution of Component-Based Groupware. In Proceedings of the 3rd International Enterprise Distributed Object Computing Conference (EDOC'99), pp. 106-115.

Szyperski, C. (1997) Component Software: Beyond Object-Oriented Programming, Addison-Wesley, ISBN 0-201-17888-5

Szyperski, C. (2003) Component technology – what, where, and how? Procedings of the 25th International Conference on Software Engineering (ICSE'03), IEEE, pp 684-693.

Winograd, T. & Flores, F. (1987) Understanding Computers and Cognition. Addison-Wesley, USA, 1987.

Won, M., Stiemerling, O. & Wulf, V. (2005) "Component-Based Approaches to Tailorable Systems", End User Development, Kluwer, pp. 1-27.

A Component Based Infrastructure to Develop Software Supporting Dynamic Unanticipated Evolution

Hyggo Almeida [1], Angelo Perkusich[1], Evandro Costa[2], Glauber Ferreira[1], Emerson Loureiro [1], Loreno Oliveira[1], Rodrigo Paes[3]

[1]Embedded Systems and Pervasive Computing Lab
Electrical Engineering and Informatics Center
Federal University of Campina Grande – Campina Grande – PB – Brazil

[2]Institute of Computing – Federal University of Alagoas
Maceió – AL – Brazil

[3]Pontifical Catholic University of Rio de Janeiro
Rio de Janeiro – RJ – Brazil

{hyggo,perkusic,glauber,emerson,loreno}@dee.ufcg.edu.br

Abstract. *This paper presents a component based infrastructure for developing software supporting dynamic unanticipated software evolution. We propose a component model providing mechanisms for managing unpredicted software changes, even at runtime. A Java implementation of the proposed model is also presented. Moreover, a performance evaluation model and an Eclipse-based tool to support composition activities are described. Finally, a pervasive computing middleware application developed using the proposed infrastructure is presented.*

Resumo. *Neste artigo apresenta-se uma infra-estrutura baseada em componentes para o desenvolvimento de software com suporte à evolução dinâmica não antecipada. Propõe-se um modelo de componentes que provê mecanismos para gerenciar alterações no software não previstas em projeto, inclusive em tempo de execução. Apresenta-se uma implementação em Java do modelo de componentes e um modelo analítico para avaliação de desempenho. Para dar suporte às atividades de desenvolvimento, propõe-se uma ferramenta baseada na plataforma Eclipse. Por fim, apresenta-se uma aplicação da infra-estrutura proposta para o desenvolvimento de um middleware para computação pervasiva.*

1. Introduction

Various studies have pointed evolution as responsible for up to 90% of the total cost of software development [Ebraert et al. 2005]. The impact of evolution on existing design and code is more significant when software requirement changes have not been anticipated. Unanticipated software evolution is a key issue on software engineering, since it is strongly related to software development and maintenance cost and time.

By definition, "unanticipated software evolution is not something for which we can prepare during the design of a software system" [1]. This way, any support for unanticipated software evolution must not require developers to specify which part of the software could evolve. Any part of the software must inherently support evolution, and developers should not bother themselves about the mechanisms that allow this evolution. Moreover, in the case of systems with frequent requirement changes or when the execution of such systems cannot be interrupted, evolution must be managed at runtime.

This seamless evolution support is not provided by existing techniques of Software Engineering, such as application frameworks [Fayad et al. 2000], component based systems [Crnkovic 2001], service oriented architectures [Papazoglou 2003], and plug-in based development [Mayer et al. 2003]. Although some of these approaches provide flexibility

[1]FUSE Workshop - http://www.informatik.uni-bonn.de/~gk/use/fuse2004/

for changes even at runtime, they have not been conceived to support unanticipated changes. Developers have to point out potential parts to be changed in the future through extension points, services interfaces, framework hot spots, plug-in interfaces, etc.

The major problem arises when a part of the software which was thought to be fixed has to change. Unanticipated changes force developers to extensively modify existing software architecture, design, and code. A change is considered "unanticipated" when its implementation does not depend on hooks encoded in previous versions of the changed software [Kniesel et al. 2002].

In this paper we introduce a component based infrastructure to develop software supporting dynamic unanticipated evolution. We propose a component model, named COMPOR COMPONENT MODEL SPECIFICATION (CMS), which allows changing any part of the software, by removing and/or adding components, even at runtime. CMS promotes dynamic unanticipated software evolution through a simple and lightweight design model. Component based concepts, such as containers and components, are used to build applications based on a hierarchical composition. We present a Java implementation of the CMS, called JAVA COMPONENT FRAMEWORK (JCF). Due to the simplicity of the object oriented framework model, it can be implemented using other programming languages. An implementation based on Python and another one in C++ are discussed in the concluding remarks of this paper. By using JCF, it is possible to develop Java applications that can be changed during runtime, even for unpredicted changes. A performance evaluation model for CMS-based architectures is also described. This performance model is very useful to define which CMS architecture is more suitable for a specific application. Moreover, we present a set of Eclipse-based tools (`http://www.eclipse.org`) for supporting the software composition activities, called COMPONENT COMPOSITION TOOLS (CCT). Finally, a case study for the proposed infrastructure is described: a pervasive computing middleware.

The remainder of this paper is organized as follows. Section 2 describes the CMS. Section 3 introduces the JCF. In Section 4 we describe a performance evaluation model for the CMS. Section 5 introduces the CCT. The pervasive computing middleware developed using CMS/JCF is presented in Section 6. Related works are discussed in Section 7. Finally, concluding remarks are presented in Section 8.

2. Component Model Specification

According to the COMPOR COMPONENT MODEL SPECIFICATION (CMS), a component based system is described as a composition of two kinds of entities: functional components and containers. Functional components are software entities implementing application-specific functionalities, making them available by means of services and events. A functional component is not composed of other components, that is, it has no child components. Functional components represent the atomic architectural elements of the software application according to the CMS.

Containers are software entities that implement no application-specific functionalities. A container controls the access to the services and events provided by its child components, which may be functional components or other containers. Functional components are made available by inserting them into containers. It is necessary to have at least one container, named *root container*, in order to insert functional components into it, and then run the application. Each container has tables of services provided by and events of interest to its child components.

2.1. Component Deployment Model

After inserting a new component into a given container, the tables of services and events for each container up to the root of the hierarchy must be updated accordingly. This is necessary in order to make available the services provided by the inserted component as well as to allow the notification of the occurrence of events that are of interest to the component. Figure 1 depicts the component deployment process and its steps are detailed as follows.

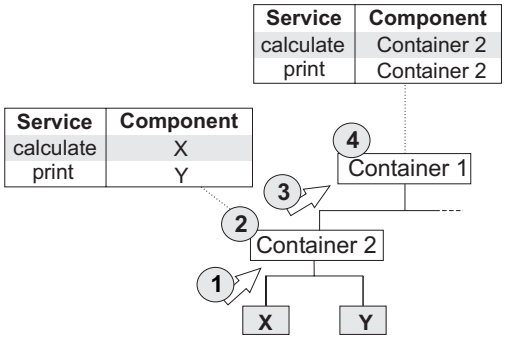


Figure 1. Component deployment - updating service and event tables.

1. A component *X* implementing the "calculate" service is added to *Container 2*.
2. *Container 2* updates the service table adding the services provided by the component *X* (the same occurs for the event table).
3. *Container 2* asks its parent container, *Container 1* in this case, to update its service table, by adding the services provided by *Container 2*.
4. *Container 1* updates its service table.

After the execution of these steps, the services provided by the component *X* can be accessed from any component in the hierarchy without an explicit reference to it. Note that a component has only reference to its parent container. In the case of two or more components having services or events with the same identifier, an *alias* is used. Thus, it is possible to register a nickname for each service, allowing services providers to coexist within the same application and be diversified in terms of non-functional features. The remove operation is similar to the deployment, but after removing a component the next invocations for its services will not work.

2.2. Interaction Model

The interaction model is based on services and events. In the first case any component may invoke a service of another component, even when the component belongs to other container. The interaction based on events focuses on the announcement of a state change in a given component to all the interested components. In both cases there is no explicit reference among components.

2.2.1. Service based interaction

As said before, after inserting a component into a given container, its services are made available to any other component in the application. Therefore, assuming the existence of a service named "save" implemented by a component *K* of the application, the execution of this service can be requested by a component *X*, without an explicit reference to *K*. In Figure 2 such an interaction process is illustrated and detailed as follows.

1. Component *X* requests the execution of the "save" service to its parent container.
2. Based on its service table, *Container 2* verifies that no child component implements the "save" service.
3. *Container 2* forwards the request to its parent container, in this case *Container 1*.

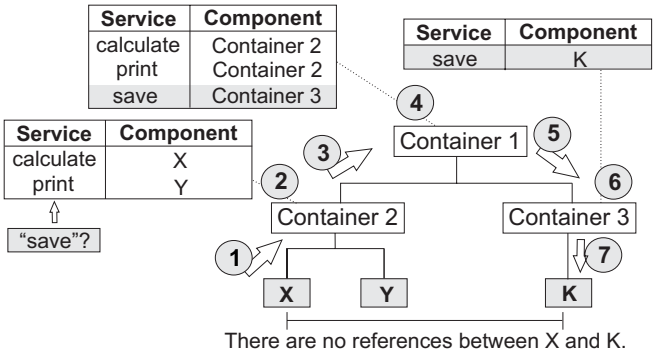


Figure 2. Service based interaction - No explicit references.

4. *Container 1*, according to its service table, verifies that one of its children implements the "save" service, *Container 3* in this case. The *Container 1* sees *Container 3* as the component that implements the requested service.

5. *Container 1* then forwards the service request to the *Container 3*.

6. *Container 3* does not implement the service but has a reference to the component that implements the requested service, and forwards the request to it, in this case component *K*.

7. Component *K* executes the "save" service and returns the result following the reverse path, back to the requester.

It is important to point out that there are no references between the component requesting the service (*X*) and the component that provides it (*K*). Thus, it is possible to change the component that provides the "save" service without modifying the rest of the structure.

2.2.2. Event based interaction

When an event is announced by a given functional component, all the components in the hierarchy of the application that are interested in the event must be notified. The interaction based on events is also controlled by containers, and thus there are no direct references among functional components. This process is shown in Figure 3 and the steps are detailed as follows.

1. The component *X* announces an event named "Event A".

2. The announcement is directly received by its parent container (*Container 2*), which verifies if any of its child components have to be notified about the event, by inspecting the event table.

3. *Container 2* forwards the event to the interested components, in this case only the component *Y*.

4. *Container 2* then forwards the event to its parent container (*Container 1*).

5. *Container 1*, according to its event table, forwards the event to those interested on it, except the one that announced the event (*Container 2*). Since *Container 1* is the root of the hierarchy, there is no parent container to forward the event. Thus, the event is only forwarded to *Container 3*.

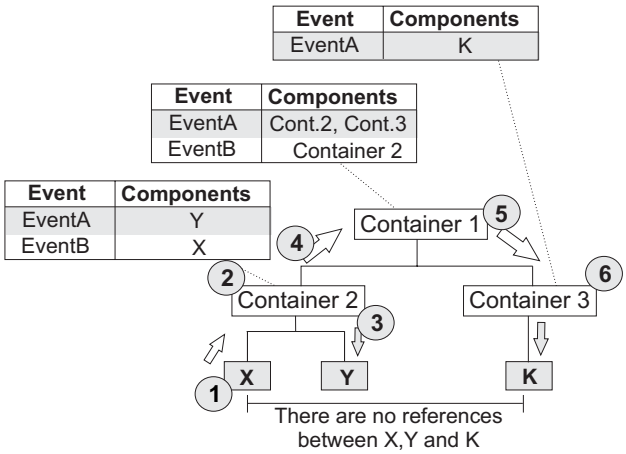


Figure 3. Event based interaction - event notification without explicit references.

6. *Container 3* forwards the event according to its event table that in this case is component *K*.

As can be seen in Figure 3, there are no references between the component that announced the event (*X*) and those interested on it (*Y* and *K*). Thus, the component that announces the event can be changed without modifying the rest of the structure.

2.3. Overriding Services: the Black-Box Inheritance Mechanism

Due to the container-mediated deployment and interaction models, CMS provides mechanisms to override services via "black-box inheritance". In other words, it is possible to reuse component services without extending the component code, even at runtime. This is possible because a component can require a service implemented by itself. Since the services provided by functional components are accessed via the container, it is only necessary to publish internal component functionalities as external services and access them via a container. Figure 4.a illustrates an example of this process. Consider the "readFile" service, which is implemented by the sequence of functionalities: "buffering" and "IOoperation". If the internal functionalities are published as services, *Component1* can access them via container. Then, the "readFile" service is decoupled from "buffering" and "IOoperation" functionalities.

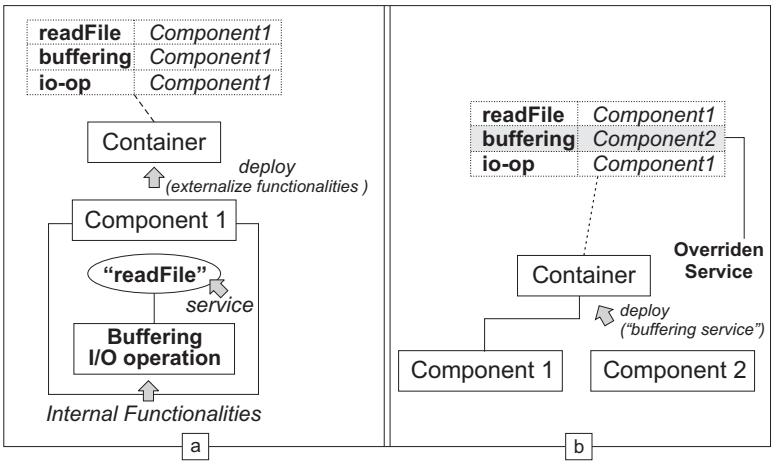


Figure 4. Service overriding - internal functionalities as component services.

To override the "buffering" internal functionality, for example, it is only necessary to deploy a component that provides a service with the same name. Figure 4.b presents

the service overriding process. In this figure, *Component2* overrides the service "buffering" of the *Component1*. Since there are no references to the *Component1*, this process, which is based on the *Template Method* and *Strategy* design patterns [Gamma et al. 1995], can be performed at runtime. After deploying the *Component2*, the "readFile" service of *Component1* becomes based on the "buffering" service implemented by the *Component2*. It is important to note that *Component2* does not have to extend *Component1*, it is only necessary to know the provided and required services.

2.4. Recursive Composition: Applications as Components

One can think that a flat architecture could be always better. It could be more interesting in terms of performance and will not require the usage of several containers. Also, it will work similarly to service oriented architectures, like Jini [Waldo 1999]. However, the main motivation for a hierarchy of containers is that it allows to maintain cohesion of functionalities provided by their child components. It makes possible to reuse entire containers without needing to understand the internal components or other containers, at any level of the hierarchy. Also, it allows recursive composition of applications, since root containers can be viewed as components for other containers. Therefore, an application can be built by integrating containers of various CMS based applications (Figure 5).

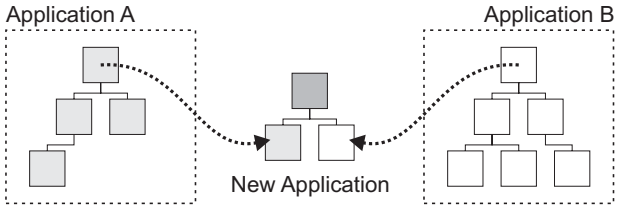


Figure 5. Composition of applications.

Based on the service and event models, black-box inheritance and recursive composition, the CMS supports all kinds of evolution scenarios: component change, addition, and removal; service and event changes; and architectural changes. In face of unanticipated evolution, an entire application could be dynamically changed. It is only necessary: i) to identify which components will change; ii) to identify which are their dependencies (services and events) and, if necessary, change them; iii) and finally change the identified components. The effort needed to perform this evolution depends on the functional cohesion and complexity of the application. In fact, it could be hard but using the CMS it will be possible.

3. Java Component Framework

The JAVA COMPONENT FRAMEWORK (JCF) is a Java implementation of the CMS. The JCF design is based on the *Composite* design pattern [Gamma et al. 1995], which can be applied to hierarchical architectures. Figure 6 shows a simplified version of the JCF class diagram, describing its main methods. `Container` and `FunctionalComponent` classes are instantiated for containers and functional components, respectively. The abstract class `AbstractComponent` assures the recursive composition [Gamma et al. 1995]. Thus, containers are not aware if their children are functional components or other containers. Additionally, it implements the accessor methods. The methods declared in the class `AbstractComponent` are differently implemented by `FunctionalComponent` and `Container` classes, for both the service and the event interaction models.

The service interaction model is implemented through iterative invocations of the `doIt` and `receiveRequest` methods. Such methods are invoked by the components and containers of the hierarchy until the service provider component is located. The function of the `doIt` method is to forward the service request, in a bottom-up way, until reaching the container that contains the reference to the provider. When this occurs, the `receiveRequest` method is invoked, in a top-down way, until reaching the functional

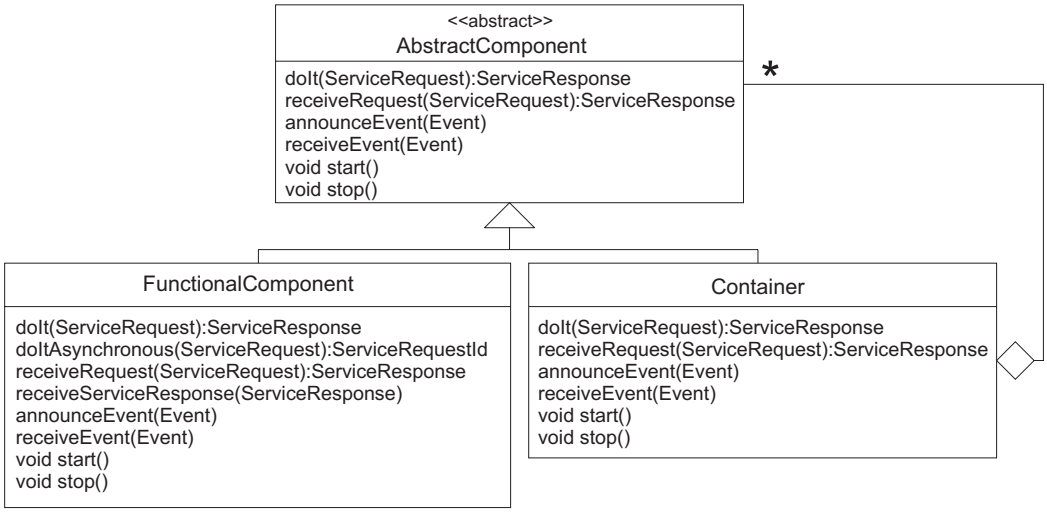


Figure 6. Simplified JCF class diagram.

component that implements the service (Figure 7). The syntax for the service invocation methods are `doIt(ServiceRequest)` and `receiveRequest(ServiceRequest)`, where `ServiceRequest` is an object that encapsulates a service name and the parameters needed to execute the service. The result of those methods is a `ServiceResponse`, which encapsulates the service execution result or the exception, if it occurs.

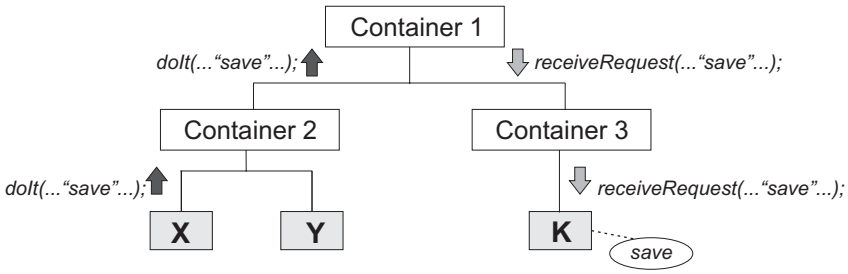


Figure 7. Execution of the `doIt` and `receiveRequest` methods.

The JCF also implements an asynchronous version of the service based model. The asynchronous interaction implementation is based on the *ActiveObject* design pattern [Vlissides et al. 1996]. A component asynchronously invokes a service through the method `doItAsynchronous` and receives a request identifier (`ServiceRequestId`). Then, a new thread is started to request the service through the method `doIt`. When the return from `doIt` occurs, it invokes the method `receive ServiceResponse` for the service requester component, forwarding the service request and the service identifier. Based on the service identifier, the requester component identifies to which invocation the reply refers to.

The implementation of the event based interaction model is based on the *Observer* [Gamma et al. 1995] and *ActiveObject* [Vlissides et al. 1996] design patterns. The functionality is implemented through the asynchronous invocation of the method `announceEvent` to announce events to the parent containers (bottom-up). On the other hand, the invocation of the method `receiveEvent` notifies the events to the interested components (top-down), as occurs with services.

Besides the interaction models specified by the CMS, the JCF implements initialization properties for components. Moreover, JCF provides a mechanism for starting and stopping the execution of the components. The initialization properties are stored in a table for each functional component and can be accessed through the `getInitialization`

`Parameter(String)` method, whose argument is the name of the required initialization parameter. The component initialization is implemented by the `start` and `stop` methods. For containers, these methods start/stop all of its components through the invocation of their respective `start` and `stop` methods. For functional components, these methods are template methods [Gamma et al. 1995] that invoke abstract methods implemented by the component developer. These methods initialize/interrupt the execution according to the component needs.

3.1. Security Support

As described in Section 2.1, an alias is used to uniquely identify services and events with the same name for different components. However, such a strategy introduces a security problem into the model. For example, it is possible to interpose a provider *X* between another provider *Y* and its clients in order to intercept the client requests towards *Y*. This may represent an intrusive way to make something undesirable in the system, since the interposed provider *X* may be seen as an intruder.

As this security issue is not tackled by the component model, the JCF must provides means for dealing with security policies for the interaction and deployment models. Such policies must then be satisfied when some service is requested or an event is announced as well as a component is inserted or removed from a container. This security infrastructure, shown in Figure 8, was developed using aspect oriented programming, with AspectJ [Kiczales et al. 2001]. Aspects have allowed to hide the complexity of the security mechanism from the developer as well as to simplify the development of systems without security requirements. The security mechanism illustrated in Figure 8 is explained as follows.

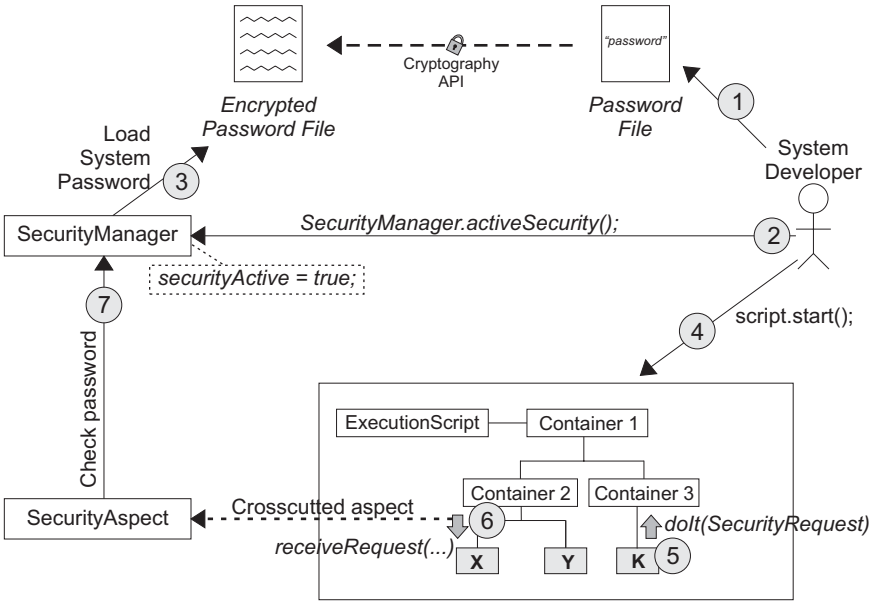


Figure 8. Aspect oriented security architecture.

1. The application developer creates a ".security" file containing the password for accessing the system as well as the service access policies. Then, uses the Java cryptography API to encrypt the file.

2. When developing the application, the security mechanism should be activated calling the `activeSecurity()` method of the `SecurityManager` singleton class. This operation defines that all service invocations, event announcements and component additions must be verified.

3. The `SecurityManager` retrieves the password and the policy information and stores them in memory.

4. After starting the root container, all of its components are also started and the application runs by means of a sequence of service invocations and event announcements.

5. A component invokes a service. With the security activated, the service requester component must forward a `SecurityServiceRequest` instance as parameter, containing the system password.

6. The component receives the request via the `receive Request` method, then the `SecurityAspect` aspect intercepts the method invocation and asks the `Security Manager` to verify the request password.

7. `SecurityManager` verifies the request password and allows the service execution. Otherwise, a `ComporSecurity Exception` is thrown.

4. Performance Analysis

There is a trade-off between flexibility and performance in the CMS. Although it allows dynamic composition of components, both its deployment and interaction models introduce an impact on the software performance when requesting services and announcing events. Considering performance a critical non functional requirement for some application domains, we propose a performance evaluation model that allows evaluating the performance of an application with specific architectures based on the CMS. Through this evaluation model, it is possible to identify and reduce potential overheads caused by the architectural design.

There are three main operations defined in the CMS to be evaluated: component deployment, service request, and event announcement. As mentioned before, the architecture of an application according to the CMS can be represented by a tree. Thus, the performance evaluation for these operations is based on a tree structure. The performance is measured based on the time spent to perform the two main operations for implementing the CMS: method invocations and accesses to data structures. Each tree edge represents a method invocation operation, followed by the invocation of a set of related methods, and accesses to data structures that store the event and service tables. To exemplify the application of the evaluation model, the mean time of method invocations and access to Java hash tables are considered.

4.1. Component Deployment Analysis

The component deployment operation is concerned with the insertion of functional components into containers, making their services and events available to other components. As mentioned before, after the insertion of a component, the tables of services and events for each container up to the root of the hierarchy is updated accordingly. Therefore, the deployment operation time can be evaluated as: $T_d = t_r + d \times (t_s + t_e)$, where: t_r is the mean time to register the new component on its parent container; d is the depth of the new component (the depth of a node is the length of the path from the root to that node); t_s is the mean time to register the provided/required services to the new component on all the containers between its parent and the root; and t_e has the same meaning as t_s, but announced/interested events are registered instead.

For instance, Figure 9 presents the deployment of the component *n*, which results in registering the component to its parent, and registering its provided/required services and announced/interested events to two containers. Thus, supposing $t_r = 20\mu s$ and $t_s = t_e = 22\mu s$, the time of the operation using the JCF is estimated as $T_d = 20\mu s + 2 \times (22\mu s + 22\mu s) = 108\mu s$.

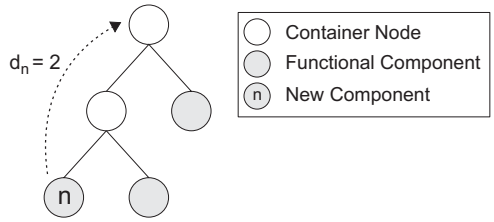


Figure 9. Component deployment performance evaluation.

4.2. Service Request Analysis

According to the CMS, when a service request is performed, the request is propagated through the hierarchy reaching the service provider, if it exists. Thus, the time for a service request operation can be given by $T_s = t_n + (d_r + d_p + 1) \times t_m$, where: t_n is the mean time to create a new service request; d_r is the depth of the node of the requester component; d_p is the depth of the node of the provider component; the constant $(+1)$ refers to an extra query, performed by the requester component to itself (for the scenarios where, possibly, the own component implements some desired service); and t_m is the mean time to access the data structures. Figure 10 illustrates the computation time of a service request in the JCF. In this case, the component *r* requests a service provided by the component *p*. Supposing $t_n = 25\mu s$ and $t_m = 30\mu s$, the time for this operation is given by $T_s = 25\mu s + (3 + 2 + 1) \times 30\mu s = 205\mu s$.

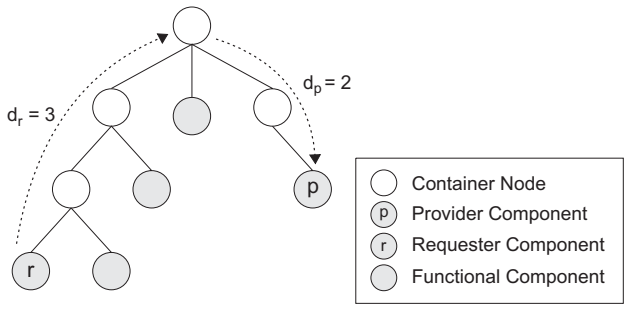


Figure 10. Service request performance evaluation.

4.3. Event Announcement Analysis

Unlike the service request operation, when an event is announced, the announcement is propagated through the hierarchy until reaching all the interested components. Since several components can be interested in the same event, various event targets may exist. If the formula for service request evaluation is used, an edge could be counted twice or more. In order to deal with this problem, besides containers and functional components, the concept of *common container node* is defined.

A *common container node* is a container node that: (i) contains two or more child nodes; (ii) it is composed of at least two "k nodes" (interested, announcer, or another common node), where each "k node" belongs to different subtrees. Furthermore, even if there are no interested components, the event is propagated to the root node. In this way, every root node is also defined as a *common container node*. The time for an event announcement operation is defined as: $T_e = t_a + t_t + \sum_{i=1}^{n} g_{k_i} \times t_c$, where: t_a is the mean time to create a new event announcement; t_t is the mean time to create and fire a new thread to deliver the event to all the interested components; g_{k_i} is the number of edges from each "k" to its *common container node*; and t_c is the mean time to query data structures about event interests.

For example, consider the tree representing an architecture of an application shown in Figure 11. In this figure, there is one component that announces an event and six components that are interested in it. According to this hierarchy and supposing $t_a = 20\mu s$,

$t_t = 80\mu s$ and $t_c = 30\mu s$, the time of the operation is $T_e = 20\mu s + 80\mu s + \sum_{i=1}^{11} g_{k_i} \times 30\mu s = 550\mu s$.

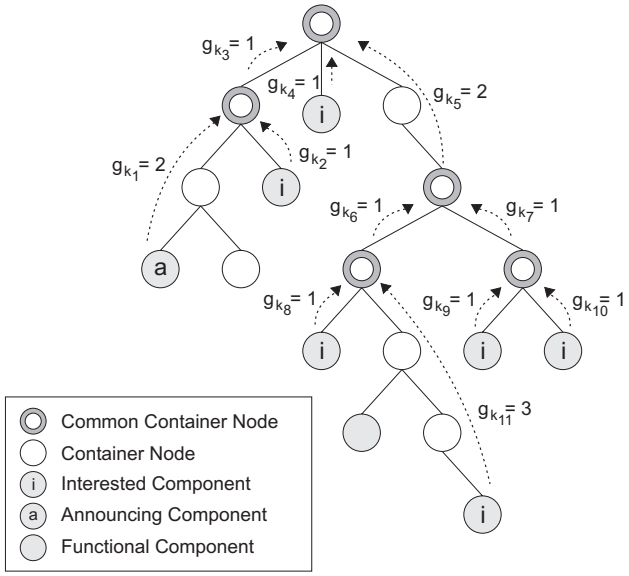


Figure 11. Event announcement performance evaluation.

4.4. JCF Code Profiling

In order to validate the evaluation model and obtain real performance evaluation measures we have performed a code profiling on the JCF, making possible to compare real data with the analytical evaluation results. The JProfiler (`http://www.ej-technologies.com`) profiler was used to verify the real cost for component deployment and execution of events and services. A dedicated Pentium IV 2.8 GHz machine, with 512MB of main memory, running Sun Java Virtual Machine version 1.4.2 and Windows XP Service Pack 2 were used to execute code profiling. Figure 12 presents the comparison between the results of the performance evaluation model and the profiling results. The x-axis should be read as: *depth of new nodes*, for the deployment of components; *total path length*, for the service requests; and *number of edges*, for the event announcement.

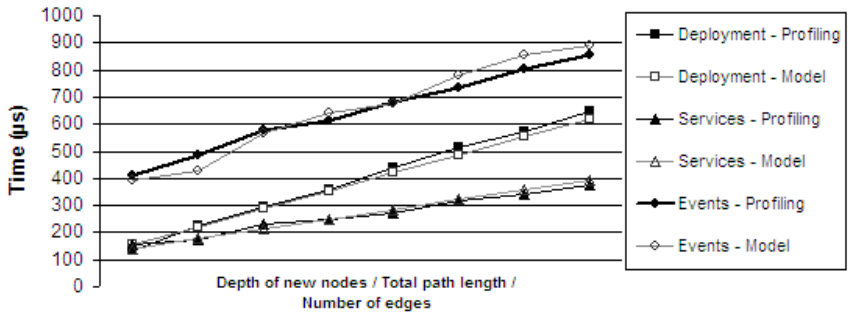


Figure 12. Model results vs. Profiling results.

For our experiments we have performed some changes in the component structures presented in the sections 4.1, 4.2, and 4.3. To evaluate the performance of component deployment, we have changed the tree depth. For the service requests, we have changed the tree depths for the requester and provider and explored different configuration of components for each combination of provider and requester depths. Finally, for event announcements, we have changed the number of interested components, exploring different configurations of components for each number of interested components.

4.5. Discussion of the Evaluation Results

The result of the evaluation indicates that the performance impact can be minimized by managing the depth of the container hierarchy. For example, consider an architecture where there is only one container (the root) and all its children are functional components, i.e, their depth is one. In this case: a component deployment operation is performed through only one service/event registering operation; a service request is performed via three accesses to the data structures; and an event announcement is performed through $(number\ of\ interested\ components + 1)$ accesses to data structures, with only one extra method execution compared to an usual *Observer* pattern implementation [Gamma et al. 1995].

Although improving the performance of the operations, a flat hierarchy reduces modularity, cohesion, and flexibility of the software architecture. It occurs because the use of containers allows the modularization of related components, making possible to change entire containers to other containers or components, as described in Section 2.4.

On the other hand, the deeper a hierarchy is, the higher is the number of method executions and accesses to data structures, and consequently the performance is reduced. An example of this kind of "deep hierarchy" is illustrated in Figure 11, where fifteen accesses to data structures occur for announcing an event to six interested components.

Based on the analysis of profiling results, we conclude that the measures obtained are very close to the analytical results, with mean errors around 5% for all operations. Probably, these errors are related to potential optimization operations performed by the Java Virtual Machine. Therefore, depending on the performance and flexibility requirements of an application, either a deeper or flatter hierarchy is more suitable. The evaluation model is useful for evaluating the performance of specific architectures still at design time.

5. Component Composition Tools

The `Component Composition Tools` (CCT) is a set of Eclipse plug-ins to develop software supporting dynamic unanticipated evolution based on the CMS. The main motivation for building the CCT was the significant effort required to compose a system using the JCF. For large scale applications, the programming effort to build several components, to compose various containers, and to define many services and events can be very high without automation tools.

The CCT main tools have been built over the Eclipse platform core: *component pallete/manager*, to deploy and make components available to be reused; *component tree/inspector*, to manage and configure the application components; *component test*, to perform integration tests; and a *component description wizard*, to describe new components. Moreover, a *component editor* is being constructed over the *AspectJ Development Tools* (`http://www.eclipse.org/ajdt`) to ease the component development activities. The support for aspects provided by the CMS is not covered in this paper.

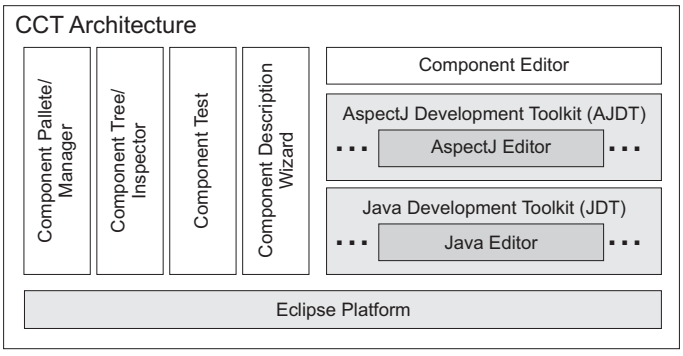


Figure 13. Architecture of the CCT.

The graphical interface of the CCT is implemented as a set of Eclipse views and wizards. A CCT perspective and project nature were created to provide a customized composition workspace for the developer. Figure 14 depicts the main screen of the CCT perspective.

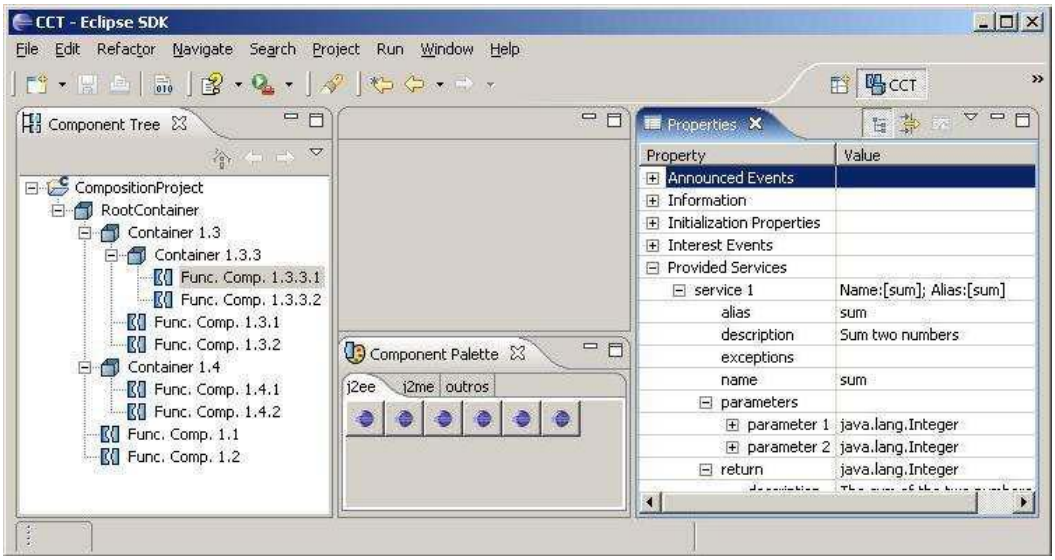


Figure 14. CCT perspective on the Eclipse platform.

With CCT, it is possible to compose CMS applications by assembling pre-existing components. However, it does not provide support for running and monitoring applications. For that, we are working on developing a Component Application Server (CAS) that runs on top of Oscar – an implementation of the *Open Services Gateway Initiative* (OSGi) (`http://www.osgi.org`). CAS provides an execution environment for the deployment and execution of CMS/JCF components and applications. Also, it manages component life cycle and controls component versioning. Oscar/OSGi is used for managing functionalities related to dynamic class loading. Using CCT and CAS, developers have full support for developing, composing and running applications based on CMS.

6. Case Study: Wings Pervasive Middleware

Wings is a middleware for pervasive computing that is guided by three issues: context-sensitivity; networking support flexibility; and interoperability both in terms of networking protocol stack and programming language [Loureiro et al. 2005]. The basis of the middleware lies on the concepts of *resource*, *context* and *peer*. We define a *resource* as an entity with a description, through which it can be shared, discovered and downloaded, such as an audio file. A *context* encapsulates information about a local peer and the environment in which it is immersed. Finally, *peers* are defined as network nodes having the following set of capabilities: search and sense other peers; share, discover, and download resources, as well as deliver context information.

Due to the sensing capability, Wings has been designed for "infrastructureless" environments. Therefore, the communication between peers must be performed in an ad-hoc way. This characteristic enhances the applicability of Wings in the world of pervasive computing, where the infrastructure is something we cannot always count on. This approach provides the necessary tools to develop applications (*Winglets*) for ad-hoc like pervasive environments such as mobile virtual communities and mobile file sharing. Using Wings, pervasive applications could take advantage of multiple configurations by performing host discovery over different network infrastructures, possibly at the same time. Based on this approach, an application could, for example, discover hosts

through *UPnP* (http://www.upnp.org), *JXTA* (http://www.jxta.org) and *Zeroconf* (http://www.zeroconf.org) protocols. This improves the acquisition of context information, since more hosts can be discovered by the applications.

However, it is very difficult to predict which of such protocols will supply the needs of different applications. Moreover, mobile devices are still very limited concerning memory and storage capacities. Therefore, it would not be reasonable to embed in such devices all the existing network infrastructure protocols for each of their wireless interfaces. It becomes necessary a mechanism for inserting and removing such implementations from a device, whenever needed. For that, we use the CMS to encapsulate the peer discovery algorithm and context information mechanisms in software components. Such components, namely Network Infrastructure Components (NICs), may be plugged in and out from the middleware even at runtime (Figure 15).

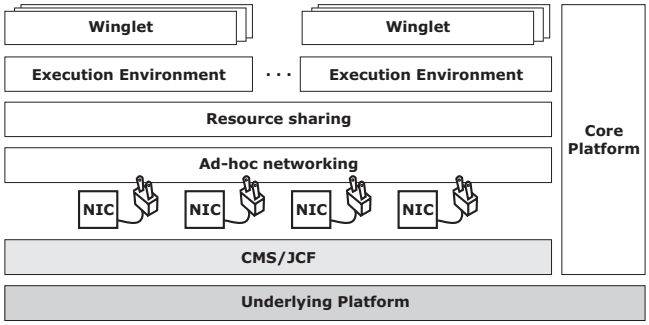


Figure 15. Wings Architecture.

The CMS successfully provides an effective way of changing *NICs* at runtime, without affecting the rest of the middleware. This is an important feature in a pervasive environment, where the networking protocols involved may change, but users do not want to stop their tasks for replacing one protocol for another. In other words, this process should be performed transparently.

7. Related Work

Different kinds of component models have been proposed. Some examples are *Sun JavaBeans* and *Enterprise Java Beans* (http://java.sun.com), and the *CORBA Component Model* (http://www.omg.org). Such models have been successfully applied for constructing corporative applications and their middleware implementations provide many interesting services for enterprise software development. However, these models were not conceived to support dynamic unanticipated software evolution. In some cases, their middlewares provide mechanisms and services to perform dynamic changes, but this is not defined in a component model level. The lack of this feature makes difficult the construction of systems supporting unanticipated evolution.

CMS has also some similarities with service oriented architectures: service publishing and provision, transparency of the service provider, flexibility for changes, among others. For instance, Jini [Waldo 1999] is a Java-based technology for the provisioning of services among network nodes. In Jini, services are advertised and discovered in central repositories, like a distributed CMS single (root) container. The work presented in [Handorean and Roman 2003] describes a Java middleware for providing services in ad-hoc networks. Such a middleware is based on a distributed service registry, where each node of the network is able to provide and use services. Other example is the OpenWings (http://www.openwings.org) framework, which aims at providing service provisioning features targeted to dynamic networks. In these works, dynamic evolution is not provided. In the work of Piccinelli et al [Piccinelli et al. 2003] the main focus is the dynamic composition of services, and recursive composition of services is also allowed. However, dynamic features are only related to service loading, unanticipated changes are not tackled.

In the context of dynamic composition models, we highlight HADAS [Ben-Shaul et al. 2001]. In HADAS, the focus is the interoperability between distributed components. The reflection concept is used to define the dynamic composition. The developer should define a set of *fixed* behaviors and another set of *extensible* behaviors for the application. Only *extensible* behaviors can be dynamically composed and changed. Considering that the changes cannot be predicted, the definition of fixed components imposes many difficulties and in some cases makes no sense, since any component can be eventually changed. Another work is Gravity [Cervantes and Hall 2004] which puts concepts from service and component orientation together for defining a model that supports the adding and removal of components at runtime. For these works, there are no mechanisms to provide recursive composition and the dynamic evolution is only allowed for explicitly defined non fixed parts.

8. Concluding Remarks

This paper presented a component based infrastructure to develop software supporting dynamic unanticipated evolution. We introduced a model, named CMS, which provides mechanisms for managing dynamic changes in the software on the fly, even if they have not been anticipated. A Java implementation of the CMS that allows developing Java applications supporting dynamic unanticipated software evolution was also presented. To make possible a large scale development, an Eclipse-based tool to support the composition activities, called CCT, was introduced.

Moreover, a performance evaluation model was described. Based on the model, the designer can identify CMS-based architecture that is more suitable for a specific application, taking into account performance and flexibility issues, still at design time. Also, we described the implementation of a complex middleware for pervasive computing, which uses the CMS as the key technology to provide flexibility.

The infrastructure presented in this paper represents a novel engineering support for constructing applications supporting to unanticipated evolution. Using CMS, JCF, and CCT, applications can be developed based on reuse, besides improving flexibility and reducing maintenance and evolution time and costs. In what follows, some current efforts and future perspectives are discussed.

Multi-Language Implementations

Multi-language implementations are very important to consolidate the CMS , and also to apply it to different contexts and platforms. For that, besides the Java implementation of the CMS introduced in this paper, a `Python Component Framework` (PCF) and a `C++ Component Framework` (CCF) are being developed. The simplicity of the CMS makes possible to implement dynamic composition without requiring object oriented languages complex mechanisms. For Python, which supports threads, dynamic loading, and dynamic binding as native features, the implementation is even more straightforward. In the case of C++, we had some problems to integrate these features from different sources. They are being applied to different contexts: the PCF is useful for rapid prototyping and the CCF is being applied to Linux-based embedded systems. We are developing the PCF and CCF as close as possible to the JCF design, to provide the same dynamic software composition infrastructure for the developer, regardless of the language.

Future Trends

The main feature trends related to the research reported in this paper are the support for non functional requirements and for formal dependency verification. The former is related to the description and implementation of non functional requirements of the components. Operation time, performance, and real-time restrictions, among other information, must be present in the component description available to the developers. For some domains, it is very important to define if two components could be interchanged, for example. The last is concerned with the formal description of the component interface - services and events.

This allows verifying if the dependencies (required services and events) of all components are being provided according to the formal specification. Also, it will make possible to determine if a remove or change operation could be performed while still maintaining the specification of the components and the system correct.

References

Ben-Shaul, I., Holder, O., and Lavva, B. (2001). Dynamic Adaptation and Deployment of Distributed Components In Hadas. *IEEE Trans. Softw. Eng.*, 27(9):769–787.

Cervantes, H. and Hall, R. S. (2004). Autonomous Adaptation to Dynamic Availability Using a Service-Oriented Component Model. In *Proceedings of the International Conference on Software Engineering (ICSE)*, pages 614–623. IEEE Computer Society.

Crnkovic, I. (2001). Component-based Software Engineering - New Challenges in Software Development. In *Software Focus*, volume 4, pages 127–133. Wiley.

Ebraert, P., Vandewoude, Y., D'Hondt, T., and Berbers, Y. (2005). Pitfalls in unanticipated dynamic software evolution. In *Proceedings of the Workshop on Reflection, AOP and Meta-Data for Software Evolution(RAM-SE'05)*.

Fayad, M., Johnson, R., and Schmidt, D. (2000). *Building Application Frameworks*. Wiley.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley.

Handorean, R. and Roman, G.-C. (2003). Secure Service Provision in Ad Hoc Networks. In *Proc. of the First International Conference on Service Oriented Computing*, volume 2910 of *Lecture Notes in Computer Science*, pages 367–383, Trento, Italy. Springer Verlag.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., and Griswold, W. (2001). An Overview of AspectJ. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK. Springer-Verlag.

Kniesel, G., Noppen, J., Mens, T., and Buckley, J. (2002). 1st Int. Workshop on Unanticipated Software Evolution. In *ECOOP Workshop Reader*, volume 2548 of *LNCS*. Springer Verlag.

Loureiro, E., Oliveira, L., Almeida, H., Ferreira, G., and Perkusich, A. (2005). Improving flexibility on host discovery for pervasive computing middlewares. In *3rd International Workshop on Middleware for Pervasive and Ad-hoc Computing*, Grenoble, France. ACM Press.

Mayer, J., Melzer, I., and Schweiggert, F. (2003). Lightweight Plug-In-Based Application Development. In *NODe '02: Revised Papers from the International Conference NetObjectDays on Objects, Components, Architectures, Services, and Applications for a Networked World*, pages 87–102. Springer-Verlag.

Papazoglou, M. P. (2003). Service-Oriented Computing: Concepts, Characteristics and Directions. In *Proc. of Fourth International Conference on Web Information Systems Engineering*, pages 3–12, Rome, Italy. IEEE.

Piccinelli, G., Zirpins, C., and Lamersdorf, W. (2003). The FRESCO Framework: An Overview. In *Proceedings of the 2003 Symposium on Applications and the Internet Workshops*, pages 120–123, Orlando, USA. IEEE Computer Society.

Vlissides, J., Coplien, J., and Kerth, N. (1996). *Pattern Languages of Program Design 2*. Addison-Wesley.

Waldo, J. (1999). The Jini Architecture for Network-centric Computing. *Communications of the ACM*, 42(7):76–82.

Injeção de Falhas na Fase de Teste de Aplicações Distribuídas

Juliano C. Vacaro, Taisy S. Weber

[1]Instituto de Informática – Universidade Federal do Rio Grande do Sul (UFRGS)
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brasil

{jcvacaro,taisy}@inf.ufrgs.br

Abstract. *Injecting communication faults, a test engineer can observe the behavior of network based applications in response to real faults and thus validate the fault tolerance of such systems. However, fault injection alone is not sufficient to validate other important functional aspects of an application. In order to increase the test coverage, fault injection must be applied together with well known software testing techniques integrated into conventional development and test environments. To reach these requirements, a fault injector to evaluate distributed applications based on RMI and its integration into JUnit and ANT frameworks are introduced.*

Resumo. *Injeção de falhas de comunicação visa emular falhas comuns a sistemas baseados em rede para assim observar o comportamento de aplicações em resposta a diferentes condições de falha. Apesar de eficiente, a técnica não é suficiente para validar outras características funcionais importantes de uma aplicação. Para aumentar a cobertura dos testes, injeção de falhas deve ser usada de forma complementar a técnicas convencionais de teste e operar integrada a plataformas de desenvolvimento e teste de sistemas. Visando atender estes requisitos, este trabalho apresenta um injetor de falhas para a avaliação de aplicações distribuídas baseadas em RMI, e sua integração aos frameworks JUnit e ANT.*

1. Introdução

Sistemas distribuídos vêm sendo desenvolvidos usando Java, sendo a invocação remota de métodos o mecanismo preferencial para a interação entre objetos. Para a validação destes sistemas, o engenheiro de testes se defronta com o desafio de, além de avaliar as funcionalidades de aplicações em resposta a entradas especificadas e situações esperadas, incluir testes em situações de erro. Estes últimos permitem verificar a robustez, disponibilidade e confiabilidade do sistema. Entretanto, o teste de aplicações distribuídas sob situações de erro não é uma tarefa trivial, pois o comportamento do ambiente de operação de tais sistemas, incluindo os diversos nodos de execução e os *links* de comunicação, não é facilmente emulado. Atuar fisicamente sobre o ambiente de execução, desligando nodos servidores e provocando desconexão de *links*, nem sempre é uma estratégia adequada principalmente quando o ambiente de teste é o próprio ambiente de desenvolvimento e monitoramento, quando não o próprio ambiente de produção da empresa.

Ferramentas de injeção de falhas por software mostram-se um método eficiente para a validação de sistemas. A abordagem visa a emulação de falhas de *hardware*, sem

comprometer fisicamente o ambiente, e a análise do comportamento do sistema na presença destas falhas. Porém, para que injeção de falhas possa ser usada de maneira complementar às técnicas já empregadas para o teste funcional ou estrutural das aplicações, é necessário que estas ferramentas sejam integradas a plataformas que tenham este objetivo. A adequada integração de injetores de falhas a ambientes usuais de teste permite ao engenheiro de teste criar cenários mais representativos das situações esperadas no ambiente de operação de tais aplicações.

Este artigo apresenta um injetor de falhas para aplicações Java distribuídas baseadas em RMI e sua integração a ambientes de teste. Vários injetores de falhas influenciaram o desenvolvimento deste trabalho, entre elas Jaca [Martins et al. 2002], uma ferramenta de injeção de falhas baseada em reflexão computacional para aplicações Java, e FIONA [Jacques-Silva et al. 2004], que insere falhas de comunicação no protocolo UDP. Entretanto, nem as ferramentas citadas nem as demais que foram analisadas [Dawson et al. 1996, Stott et al. 2000, Chandra et al. 2004, Drebes et al. 2005] se mostraram capazes de emular diretamente os cenários de falhas de RMI. Nenhuma das ferramentas previa a integração em ambientes de teste. Neste contexto, o objetivo deste trabalho foi desenvolver uma ferramenta capaz de emular os cenários de falhas de RMI de maneira eficiente e direta, e assim possibilitar a avaliação de aplicações construídas sobre este protocolo bem como os mecanismos de detecção e tratamento de erros de tais aplicações.

O injetor apresentado neste artigo, denominado FIRMI, complementa o ciclo de teste de aplicações através da integração em diferentes plataformas de desenvolvimento, agregando as vantagens da técnica de injeção de falhas voltada a sistemas distribuídos para estes ambientes. A Seção 2 apresenta o conceito de injeção de falhas. A Seção 3 descreve o modelo de falhas referente a arquitetura de RMI. Os detalhes do injetor são mostrados na Seção 4. A integração de FIRMI nas plataformas de teste e desenvolvimento de aplicações é detalhada na Seção 5. Na Seção 6 é demonstrado o uso da ferramenta através de um experimento de injeção de falhas. Trabalhos relacionados são descritos na Seção 7 e considerações finais são feitas na Seção 8.

2. Injeção de Falhas

Injeção de falhas corresponde à inserção artificial de falhas em um sistema computacional real para avaliar o seu comportamento na presença de falhas. Neste processo de validação deve ser especificada uma carga de trabalho (conjunto de ações que o sistema alvo irá executar) e a carga de falhas (falhas que serão injetadas no alvo durante a execução da carga de trabalho). Uma rodada de testes usando um injetor de falhas com uma determinada carga de trabalho e uma determinada carga de falhas é chamado experimento.

Este trabalho foca a injeção de falhas de comunicação por *software* [Hsueh et al. 1997]. Nesta abordagem são emuladas falhas em componentes de *hardware*, acelerando a manifestação de erros devidos a tais falhas e assim permitindo observar o comportamento do sistema sob teste. A injeção de falhas emuladas de *hardware* por *software* é diferente da injeção de falhas de *software*, onde é comum a geração de mutantes de código [Delamaro et al. 2001b, Delamaro et al. 2001a] para verificação da cobertura dos procedimentos de teste. A injeção de falhas por *software* opera principalmente alterando o fluxo de execução para que as rotinas do injetor responsáveis pela emulação das falhas de *hardware* possam inserir erros na aplicação sob teste durante sua execução. Uma das

maneiras mais eficientes de inserir erros é por instrumentação de código.

3. Comportamento de RMI na Presença de Falhas

A arquitetura de RMI [Sun Microsystems 1997] consiste basicamente de três módulos: cliente, servidor e registro de objetos. Para que objetos sejam referenciados por clientes, estes devem ser criados no nodo servidor e cadastrados no registro de objetos. A comunicação entre objetos remotos é baseada no conceito de *proxies*. Um cliente para ter acesso a objetos em um servidor deve primeiramente obter uma referência (*proxy*) para o objeto. Um *proxy* (Stub) de um objeto é um objeto que possui a mesma interface que o objeto original, mas que repassa as requisições de invocação de métodos via rede ao objeto "real" no servidor. Ao receber a requisição para a execução de um método, o servidor irá obter o objeto correto para processar a requisição. O resultado da computação é então retornado ao *proxy* no cliente e posteriormente para a aplicação.

Como RMI é uma abstração de alto nível de comunicação, falhas inerentes a mecanismos de mais baixo nível afetarão as aplicações baseadas neste protocolo. Assim, é importante identificar as situações de falha possíveis em sistemas distribuídos para posteriormente compreender como estas falhas podem afetar componentes de maior nível de abstração. Este trabalho utiliza um modelo de falhas de comunicação [Birman 1996], que define falhas de colapso (o componente pára sua execução e entra em colapso sem tomar ações incorretas), parada segura ou *fail-stop* (similar à falha de colapso, porém a falha é precisamente detectável pelos outros componentes do sistema), omissão de envio e recepção de mensagens (mensagens são perdidas no próprio nodo emissor ou receptor durante o processamento do sistema operacional), rede (mensagens são descartadas pela infraestrutura de comunicação), particionamento de rede (uma rede se divide em uma ou mais subredes desconectadas, onde os nodos de cada subrede não se comunicam), temporização (onde um requisito temporal do sistema é violado) e bizantinas (incluem uma grande variedade de comportamentos falhos, como o corrompimento de dados ou até um comportamento malicioso da aplicação).

Ambos nodos cliente e servidor estão sujeitos a falhas. É importante salientar que: 1) toda a comunicação RMI é sempre baseada no paradigma cliente/servidor, mesmo que um nodo possa assumir ambos os papéis, 2) todos os erros detectados no nodo servidor são sempre enviados ao cliente e 3) quando uma falha é detectada pelo sistema RMI, um erro será sinalizado para a aplicação através de exceções. Estas exceções podem ser de três tipos: as referentes ao protocolo de transporte de dados (TCP ou HTTP), as geradas pelo protocolo RMI e finalmente as geradas por objetos remotos (definidas pelo usuário).

Durante o processo de invocação, um cliente com acesso a uma referência remota solicita ao servidor a execução de um método. O colapso do servidor, antes que seja efetuada a requisição, é detectado por RMI e sinalizado à aplicação através de uma exceção. Entretanto, se o nodo entrar em colapso durante a execução da requisição, o nodo cliente não saberá se o servidor ainda está processando o pedido, fazendo com que o cliente fique esperando indefinidamente ou até que o servidor seja reiniciado. Particionamento de rede pode ocasionar problemas ao sistema de coletor de lixo distribuído de RMI. Quando um cliente obtém uma referência remota, é associada uma permuta (*Lease*), que define o tempo que um objeto é considerado pelo servidor como referenciado por um cliente. É responsabilidade do cliente atualizar o tempo de uso de cada objeto antes que o mesmo

expire. Uma falha de particionamento pode fazer com que clientes não sejam capazes de atualizar a permuta dos objetos que referenciam. Isto faz com que o servidor considere os objetos como não mais referenciados, portanto sendo coletados. Ao retornar da falha de particionamento, clientes não serão mais capazes de acessar seus objetos remotos.

Falhas de temporização são mais significativas no nodo servidor. Este tipo de falha pode ocorrer devido à sobrecarga do nodo em questão, degradando os serviços oferecidos pelo componente. Ainda, falhas bizantinas, como a alteração dos valores de parâmetros passados durante o processo de invocação podem não ser percebidas pelo sistema RMI, podendo ocasionar o colapso de aplicações. Na Seção 4.3 é mostrado como o injetor, através da descrição de cenários de falhas, emula as situações vistas nesta seção.

4. Injetor

FIRMI, *Fault Injector for RMI*, é capaz de emular os cenários de falhas de comunicação para aplicações baseadas em RMI. O objetivo da ferramenta é permitir avaliar o comportamento sob falhas de aplicações que usam o protocolo RMI bem como a cobertura de falhas dos mecanismos de tolerância a falhas por elas empregados. Para que o injetor seja usado de maneira plena na fase de teste de sistemas, FIRMI foi projetado para ter uma arquitetura simples e modular, possibilitando a integração em diferentes ambientes de desenvolvimento.

RMI é largamente usado em aplicações distribuídas, sendo alvo de otimizações e adaptações a diferentes necessidades. Cheng-Wei Chen [Chen et al. 2004] estende RMI para operar sobre ambientes como Bluetooth, GPRS e WLAN. Chang [Chang and Aharnad 2004] estende RMI para comunicação P2P, agregando escalabilidade e tolerância a falhas ao *framework*. Para que o injetor possa ser usado na avaliação de sistemas que usam diferentes implementações de RMI, como as citadas, o projeto de FIRMI segue a especificação de RMI [Sun Microsystems 1997], ficando transparente ao injetor os detalhes de uma determinada implementação.

4.1. Instrumentação do Protocolo

Além da compatibilidade com diferentes implementações de RMI, também é desejado que a descrição dos cenários de falhas seja similar à API do protocolo, portanto, é necessário que o local escolhido para a inserção dos ganchos de instrumentação seja o ponto mais abstrato em que não se perca a semântica do protocolo. A análise da especificação mostrou que a localização ideal para a instrumentação do protocolo são as estruturas pertinentes aos Stubs e Skeletons, já que estes concentram os fluxos de envio e recepção de requisições. A especificação define as interfaces `java.rmi.server.RemoteRef` e `java.rmi.server.ServerRef`, sendo a primeira usada no nodo cliente para o envio e a segunda no nodo servidor na recepção de requisições. Como são interfaces, deve-se alterar as classes que implementam tais definições, fato que permite ao injetor operar sobre qualquer implementação de RMI compatível com a especificação.

As classes a serem instrumentadas para o injetor são classes de sistema. Classes de sistema são carregadas pelo Carregador de Classes de Inicialização na ativação da máquina virtual. Uma vez carregada, uma classe não pode ser alterada. Consequentemente as técnicas convencionais de instrumentação não podem ser usadas. Desta forma, é necessário usar o suporte oferecido pela máquina virtual. FIRMI usa o pacote

`java.lang.instrument`, pois é específico para a substituição de classes e por permitir a definição de vários agentes simultaneamente, possibilitando a integração deste injetor com outros injetores como FIONA [Jacques-Silva et al. 2004] com modificações mínimas no seu sistema de instrumentação. O pacote `java.lang.instrument` tem acesso às classes carregadas pelo Carregador de Classes de Inicialização conforme estas são referenciadas na aplicação, podendo operar sobre o conteúdo das mesmas. Apesar de flexível, a interferência causada pelo agente de instrumentação durante a execução de uma aplicação pode não ser desejada. Então, FIRMI também foi projetado para suportar a substituição de classes durante a inicialização da máquina virtual. Aqui, é indicado o caminho para o Carregador de Classes de Inicialização das novas classes de sistema (opção `-Xbootclasspath`). Assim, o Carregador usará a primeira classe encontrada como a classe de sistema sendo procurada.

Note que tanto o uso do pacote `java.lang.instrument` quanto o uso da opção `-Xbootclasspath` apenas provêem o acesso a uma classe de sistema, possibilitando sua substituição. Porém FIRMI, além disso, necessita de uma API que permita manipular o *bytecode* das classes de RMI para inserir os ganchos nos fluxos de envio e recepção de requisições. Foi escolhido Javassist [Chiba 1998] por sua facilidade de uso e por já ter sido testada com sucesso no injetor Jaca [Martins et al. 2002].

4.2. Arquitetura

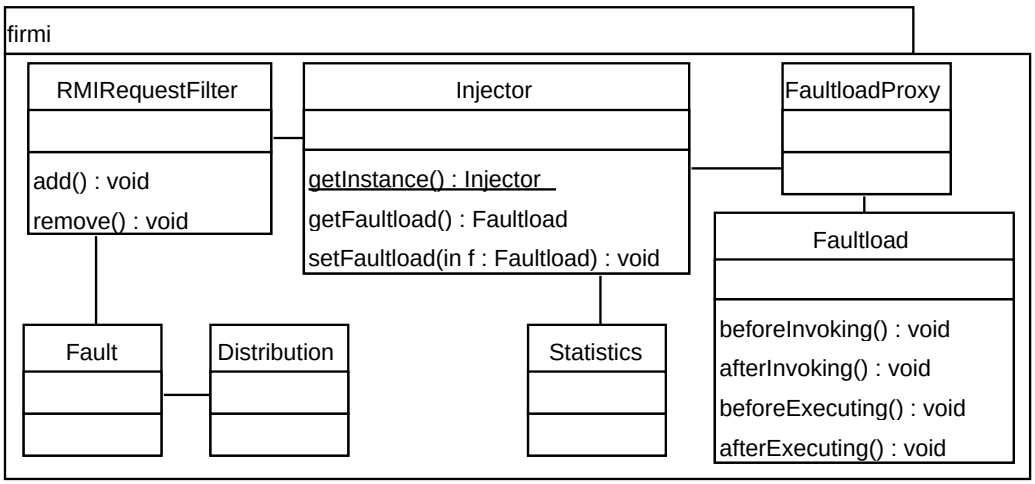


Figura 1. Diagrama de classes de FIRMI

A Figura 1 é um diagrama de classes de FIRMI. As principais classes do injetor são: `Injector`, `Faultload`, `Statistics` e `RMIRequestFilter`. A classe `Injector` controla as funcionalidades da ferramenta. Quando uma requisição RMI é interceptada pelos ganchos do mecanismo de instrumentação, o módulo irá acionar a classe de coleta de dados, registrar a requisição em log, invocar o `Faultload` para decidir a ação a ser tomada sobre a mensagem e finalmente injetar as falhas definidas pelo `Faultload` através da classe `RMIRequestFilter`. Para garantir que apenas uma instância de `Injector` seja criada, foi usado o padrão de projeto *Singleton* [Gamma et al. 1994]. Desta maneira, os demais componentes do sistema terão uma única visão do estado atual do injetor.

`Faultload` é a classe que representa um cenário de falhas. Toda a vez em que uma requisição RMI é originada de um cliente ou recebida pelo servidor, o injetor irá invocar o módulo de tratamento do cenário de falhas. Este comportamento é similar a noção de *listeners*, largamente utilizado na linguagem. A principal função desta classe é decidir, baseado nas informações do fluxo RMI, quando falhas devem ser ativadas ou interrompidas. FIRMI também define um conjunto de classes para permitir a emulação de erros causados por falhas de comunicação (Seção 3). `RMIRequestFilter` atua como um filtro de requisições RMI, injetando as falhas definidas por um `Faultload`. Para ativar ou interromper a ação de uma falha, deve-se adicionar/remover a mesma do filtro de requisições. Em FIRMI, cada falha é representada por uma classe (`CrashFault`, `LinkCrashFault`, `TimingFault` ou `NetworkPartitionFault`). Assim, quando uma requisição RMI for submetida ao filtro, todas as falhas ativas serão aplicadas sobre a mensagem.

Por fim, a classe `Statistics` contém as informações coletadas durante um experimento. A classe pode ser acessada para que as informações sejam armazenadas ou até usadas pelo cenário de falhas para tomada de decisão. Durante um experimento, as atividades realizadas por *faultloads* são monitoradas através da classe `FaultloadProxy`. A classe segue o padrão de projeto *Proxy* [Gamma et al. 1994]. Desta maneira, quando uma requisição RMI é percebida por FIRMI, o *proxy* coleta informações da execução do cenário de falhas ativo e atualiza as medidas contidas na classe `Statistics`.

4.3. Cenários de Falhas

A descrição de cenários de falhas é feita através da criação de classes Java. Como uma carga de falhas é dependente da aplicação sob teste e da carga de trabalho definida, todo módulo de cenário de falhas deve estender a classe `Faultload` para que o injetor possa interagir com o módulo de maneira padronizada. A classe `Faultload` é um *listener* de requisições RMI, possuindo métodos que indicam o fluxo de mensagens do protocolo. Os métodos `beforeInvoking` e `afterInvoking` são invocados respectivamente antes e após uma requisição no nodo cliente. `beforeExecuting` e `afterExecuting` são similares aos métodos anteriores, porém são invocados no nodo servidor na execução de uma requisição. Todos os métodos da classe `Faultload` possuem, praticamente, o mesmo conjunto de argumentos: a referência remota usada, o método sendo invocado, os valores dos parâmetros deste método e, após a execução de uma requisição, o valor de retorno do processamento.

Uma das vantagens da abordagem adotada para a descrição de cenários de falhas é a injeção de falhas baseada tanto no fluxo de mensagens quanto no conteúdo das mesmas, permitindo a validação de aplicações com precisão, já que falhas podem ser ativadas em pontos específicos no fluxo de execução de tais aplicações. É importante salientar que *faultloads* são responsáveis pela ativação de falhas e não pela sua injeção. FIRMI possui uma API para a injeção de falhas composta da classe `RMIRequestFilter` e por classes que modelam o comportamento exibido por RMI na presença de falhas de comunicação. *Faultloads*, após decidir pela ativação de uma falha, devem primeiramente criar um objeto do tipo desejado (`CrashFault`, `LinkCrashFault`, `TimingFault` ou `NetworkPartitionFault`) e então registrar o mesmo junto a classe `RMIRequestFilter`, a qual irá realizar o processo de injeção de falhas, gerenciando os aspectos de cada falha de acordo com o modelo adotado pela ferramenta.

Também é possível associar distribuições de probabilidade à ativação de falhas (classe `Distribution`), permitindo a especificação de cenários de falhas representativos, já que distribuições de probabilidade são usadas para modelar comportamentos observados na realidade.

Na Seção 3 foi definida a relação entre as falhas do sistema de comunicação e como estas influenciam os componentes de maior nível de abstração. FIRMI, através de sua API, deixa transparente tais detalhes do protocolo RMI, possibilitando a engenheiros de teste reproduzir situações inerentes a sistemas distribuídos sem a necessidade de dominar os detalhes específicos de RMI. Além disso, a arquitetura do injetor e as características inerentes a linguagem Java permitem que falhas sejam ativadas com base em informações estáticas, informações internas ao próprio módulo de cenário de falhas e informações extraídas diretamente da aplicação sob teste. Estas características possibilitam a criação de cenários de falhas representativos, pois o comportamento da injeção de falhas pode ser modificado dinamicamente durante a execução da aplicação sob teste.

5. Integração com outras plataformas de desenvolvimento

Paradigmas de desenvolvimento focadas unicamente na programação de sistemas (*Extreme Programming* [Beck and Andres 2004]) tem se tornado populares. O desenvolvimento baseado em testes (*Test-Driven Development*) é um dos pontos atacados por estas metodologias. A abordagem consiste no processo contínuo de desenvolvimento e teste, caracterizando uma metodologia bastante dinâmica. Consequentemente, são necessárias ferramentas de apoio ao desenvolvimento de aplicações capazes de agilizar este processo.

O modelo adotado em FIRMI para a descrição de cenários de falhas facilita o processo de integração do injetor em várias plataformas de desenvolvimento, permitindo estender *frameworks* já existentes com o conceito de injeção de falhas no auxílio à avaliação de sistemas. O uso de linguagens de *script* como BSF (Seção 5.1), de *frameworks* para teste de aplicações como JUnit (Seção 5.2) e o uso de ambientes de desenvolvimento como ANT (Seção 5.3), aceleram o processo de desenvolvimento de aplicações baseadas em Java. Como o objetivo de FIRMI também é a validação de sistemas, o processo de integração do injetor permite que as vantagens associadas à técnica de injeção de falhas sejam incorporadas de maneira natural às plataformas citadas.

5.1. Integração com Linguagens de *Script*

Scripts são frequentemente usados para estender funcionalidades de aplicações devido a sua facilidade na descrição de algoritmos. A integração de FIRMI com linguagens de *script* visa o mesmo objetivo, estender os mecanismos para a descrição dos módulos de cenários de falhas e assim agilizar o processo de avaliação de aplicações.

BSF [Apache Software Foundation 2002] é uma abstração para o uso de linguagens de *script*. A plataforma permite que *scripts* sejam invocados a partir de aplicações Java e que aplicações Java executem *scripts*. É importante salientar que BSF é um framework composto por uma série de classes abstratas que devem ser implementadas pelas reais linguagens. BeanShell [Niemeyer and Knudsen 2000], Jython [Pedroni and Rappin 2002] são linguagens bem conhecidas e compatíveis com este padrão. `javax.script` é um pacote nativo de Java que implementa funcionalidades similares ao *framework* BSF.

A especificação JSR-223 [Java Community Process 2005] define os requisitos para a padronização do suporte ao uso de *scripts* a partir da linguagem Java (o pacote será incluído na versão 1.6 da Linguagem).

Os *frameworks* mencionados nesta seção possibilitam a descrição de cenários de falhas em várias linguagens de *script* de maneira transparente, diminuindo a curva de aprendizado para o uso do injetor de falhas.

5.2. Integração com JUnit

A integração de FIRMI com o *framework* JUnit [Gamma and Beck 2001] introduz a noção de injeção de falhas na fase de teste de aplicações distribuídas, estendendo a definição de casos de teste. A integração da ferramenta permite reproduzir situações inerentes a sistemas distribuídos como colapso de nodos, particionamento de rede e falhas em canais de comunicação. Assim, agilizando o processo de avaliação de aplicações e de seus mecanismos de tolerância a falhas. Além disso, as medidas estatísticas coletadas pelo injetor juntamente com as medidas obtidas pelo próprio ambiente de teste auxiliam a caracterização das aplicações na presença de falhas.

Em JUnit, um caso de teste é modelado através da classe `TestCase`. Casos de teste podem ser agrupados através da classe `TestSuite`. Para conduzir testes em JUnit deve-se criar uma classe que estenda `TestCase`, criar um método para cada teste que se deseja executar e prefixar o nome destes métodos com a palavra `test`. Em cada método deve-se validar o resultado das operações para indicar ao *framework* se o teste foi bem sucedido ou não. Finalmente, os testes devem ser agrupados em um `TestSuite` e executados através de um *runner*. Assim, para a integração do injetor, foi criado o pacote `firmi.junit.framework` e estendidas as classes `TestCase` e `TestSuite`. A Figura 2 mostra a estrutura das novas classes desenvolvidas. A extensão possibilita definir cenários de falhas para um caso de teste específico, para um conjunto de casos de teste e também permite combinar as duas maneiras anteriores. Toda a interação com FIRMI é feita pelas classes estendidas, permitindo ao engenheiro de teste focar unicamente na especificação dos *faultloads* a serem utilizados.

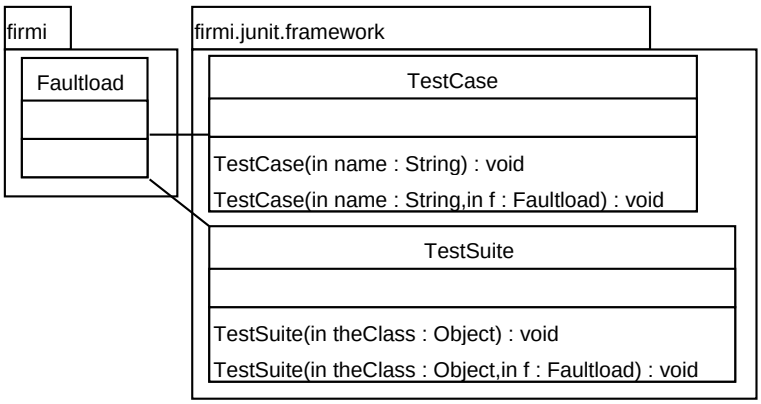


Figura 2. Extensão desenvolvida para a plataforma JUnit

Na classe `TestCase` foram alterados os construtores para aceitar um objeto `Faultload` como parâmetro. A modificação permite que seja definido um cenário de

falhas específico para este caso de teste, o que diminui a complexidade do módulo e permite a criação de cenários de falhas mais simples e claros. Também é possível definir um cenário de falhas para um conjunto de casos de teste. As modificações realizadas na classe `TestSuite` são similares as modificações da classe `TestCase`, sendo alterados apenas os construtores do módulo. Esta opção é útil quando deseja-se injetar um determinado perfil de falhas a um conjunto de testes relacionados. Ainda é possível combinar a definição de cenários de falhas para um único ou um conjunto de casos de teste. Deste modo, cenários de falhas podem ser associados a conjuntos de casos de teste, e para situações específicas, pode-se definir um cenário de falhas para um em especial.

```
1  public class MyTestCase extends firmi.junit.framework.TestCase {
2    public MyTestCase(String name)              {super(name);}
3    public MyTestCase(String name, Faultload f) {super(name, f);}
4
5    public void test1() throws Exception {...}
6    public void test2() throws Exception {...}
7    public void test3() throws Exception {...}
8
9    public static Test suite() {
10     TestSuite suite = new TestSuite();
11     suite.addTest(new MyTestCase("test1"));
12     suite.addTest(new MyTestCase("test2", new MyFaultload1()));
13     suite.addTest(new MyTestCase("test3", new MyFaultload2()));
14     return suite;
15   }
16 }
```

Figura 3. Caso de teste em JUnit com as classes estendidas por FIRMI

A Figura 3 mostra a estrutura para a definição de casos de teste usando as classes estendidas por FIRMI. A classe `MyTestCase` (linha 1) é a classe que representa um caso de teste. A classe estende `firmi.junit.framework.TestCase` (linha 1) para suportar a especificação de *faultloads* individuais para cada teste. `test1` (linha 5), `test2` (linha 6) e `test3` (linha 7) são métodos que representam testes a serem executados. `MyFaultload1` (linha 12) e `MyFaultload2` (linha 13) são módulos de cenários de falhas. O código mostra que a extensão de FIRMI possibilita a criação de casos de teste de forma convencional (linha 11) e também permite associar módulos de cenários de falhas a casos de teste específicos (linhas 12 e 13), oferecendo ao engenheiro de teste um maior controle em relação ao momento em que falhas deverão ser inseridas no sistema sob teste.

5.3. Integração com ANT

Para que FIRMI seja configurado a fim de injetar falhas em uma aplicação alvo é necessário fornecer informações adicionais ao ambiente Java. Entre estas informações estão a alteração do `bootclasspath` para conter os pacotes do injetor e as classes instrumentadas. Também é necessário indicar o módulo de cenário de falhas que deverá ser usado. ANT [Apache Software Foundation 2000] é um ambiente de execução similar a um Makefile em UNIX, porém destinado a aplicações Java. A integração da ferramenta a este ambiente de desenvolvimento facilita o processo de uso do injetor, ocultando detalhes de configuração para o engenheiro de teste.

O *script* ANT é um arquivo no formato XML organizado em projetos (*project*), alvos (*targets*) e tarefas (*tasks*). O projeto define as informações da aplicação sendo de-

senvolvida como nome, descrição e diretório base para execução de comandos. Alvos são a maneira com que um desenvolvedor organiza seu *script*. Pode-se criar um alvo para compilar uma aplicação, para executá-la ou verificar dependências externas. Cada alvo definido pelo usuário executa suas ações através de tarefas. Tarefas são estruturas pré-definidas por ANT. Existem tarefas para compilação (`javac`), execução (`java`) até tarefas para usar programas externos como FTP, SCP, CVS (*Control Version System*) e para o próprio ambiente JUnit.

Cada tarefa em ANT é associada a uma classe Java. Como é desejado integrar o injetor à execução de aplicações e ao ambiente JUnit, as classes associadas a estas tarefas foram estendidas para suportar as informações adicionais à execução de FIRMI. No caso, `java` é a tarefa responsável pela execução de uma aplicação e `junit` a responsável pela execução de casos de teste, sendo as duas classes os módulos estendidos pela ferramenta. A Figura 4 mostra o formato da tarefa `firmi-java`. `firmi-java` suporta os mesmos atributos da tarefa `java` como `classname`, `classpath` etc, com adição das informações para associar um módulo de cenário de falhas. O atributo `faultloadclasspath` indica a localização do cenário de falhas, pois este não precisa estar inserido na mesma estrutura da aplicação sob teste. O atributo `faultload` define o nome da classe Java que representa o cenário de falhas. A tarefa também possibilita controlar a instrumentação das classes de sistema de RMI responsáveis pelos fluxos de envio e recepção de requisições, o que é feito através dos atributos `remoterefimpl` e `serverrefimpl` (Seção 4.1). A integração com ANT elimina a necessidade de informar os pacotes do injetor para a JVM, pois são localizados e configurados automaticamente pela extensão desenvolvida.

```
1 <firmi-java
2   faultloadclasspath="classpath para o faultload"
3   faultload="classe Java do faultload"
4   remoterefimpl="implementação da interface RMI RemoteRef"
5   serverrefimpl="implementação da interface RMI ServerRef"
6   ...>
7     <!-- demais opções da tarefa java -->
8 </firmi-java>
```

Figura 4. Extensão da tarefa java

Para a extensão de JUnit foi criada a tarefa `firmi-junit` (Figura 5). Seguindo a idéia usada em `firmi-java`, são suportados todos os atributos da tarefa `junit` com a adição dos atributos para a integração do injetor. Entretanto, como descrito na Seção 5.2, a associação de cenários de falhas com casos de teste é feita no próprio módulo de teste, não sendo necessário informar o caminho e o nome da classe para o cenário de falhas. Ainda é possível controlar a instrumentação das classes de sistema de RMI através dos atributos `remoterefimpl` e `serverrefimpl`.

```
1 <firmi-junit
2   remoterefimpl="implementação da interface RMI RemoteRef"
3   serverrefimpl="implementação da interface RMI ServerRef"
4   ...>
5     <!-- demais opções da tarefa junit -->
6 </firmi-junit>
```

Figura 5. Extensão da tarefa junit

A integração de FIRMI no ambiente ANT faz com que o uso do injetor ocorra sem maiores modificações ao ambiente já existente, facilitando ainda mais o uso da ferramenta por desenvolvedores e engenheiros de teste.

6. Experimentos de Injeção de Falhas

Para demonstrar uma campanha de injeção de falhas com FIRMI, é usado o ambiente de computação em grade OurGrid [OurGrid 2005]. Não é objetivo deste trabalho validar todos os aspectos do ambiente, mas sim mostrar como o injetor proposto pode ser usado na validação de aplicações distribuídas baseadas em RMI. OurGrid é baseado no conceito de *Bag-of-Tasks*, aplicações paralelas onde as tarefas são independentes, ou seja, tarefas não se comunicam entre si para completar sua computação. OurGrid é formado por três componentes: `MyGrid`, `Peer` e Agente. `MyGrid` é o componente central do ambiente, sendo responsável pela coordenação e escalonamento de *jobs* [1]. *Peers* são responsáveis pela organização dos nodos de um dado domínio administrativo, atuando como um provedor de nodos disponíveis para `MyGrid`. Agentes são os componentes que executam as tarefas escalonadas pelo `MyGrid`.

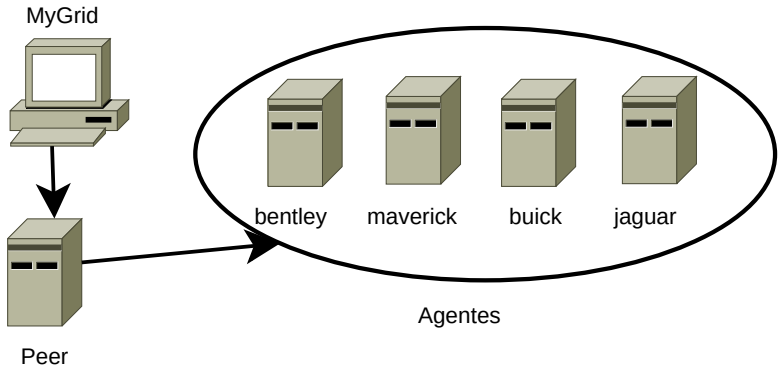


Figura 6. Organização do sistema OurGrid usado nos experimentos

A Figura 6 mostra a topologia criada para a condução dos experimentos. Como o objetivo é validar as funcionalidades do ambiente OurGrid, e não das aplicações executadas pelo mesmo, foi usada uma aplicação sintética da própria distribuição do sistema que realiza o fatorial de um número. Para a condução dos experimentos, FIRMI foi adicionado nos ambientes de execução dos agentes e também no `Peer` através da alteração dos *scripts* de inicialização de tais componentes para incluir as opções necessárias ao injetor.

6.1. Localização de Agents

Os nodos registrados no `Peer` são periodicamente monitorados a fim de manter uma lista atualizada de agentes disponíveis para a execução de tarefas. Uma análise dos logs gerados pelo injetor mostrou que este monitoramento é feito através da própria API de RMI, o que possibilita o uso de FIRMI para validar esta funcionalidade. Este caso de teste tem o objetivo de verificar se o `Peer` é capaz de detectar o colapso de agentes e consequentemente removê-los da lista de nodos disponíveis.

[1] Job é uma aplicação em OurGrid composta por tarefas independentes

O nodo `buick` foi escolhido para executar o injetor e emular o colapso desejado. Para especificar a carga de falhas é necessário decidir o momento em que a falha deve ser ativada. A análise dos logs mostrou que o método `UserAgentServer.ping` é usado pelo mecanismo de monitoramento de OurGrid. Assim, a carga de falhas para este experimento consiste na emulação do colapso do nodo `buick` após a terceira invocação de `ping`, o que permite o nodo ser primeiramente detectado pelo `Peer` para então sofrer o colapso. Para criar o módulo do cenário de falhas (Figura 7), foi criada a classe `BuickFaultload` (linha 1), que estende a classe `Faultload` da API de FIRMI, e definido o método `beforeExecuting` (linha 4), a fim de receber notificações da execução de métodos remotos. A falha de colapso (linha 2) é definida no momento da criação do *faultload* e ativada com base no fluxo de execução de RMI (linha 8).

```
1  public class BuickFaultload extends Faultload {
2    private CrashFault c = new CrashFault("c1");
3    ...
4    public void beforeExecuting(Remote obj,
5        Method method, Object[] params) {
6      times += 1;
7      if (times == 3 && method.getName().equals("ping"))
8        RMIRequestFilter.add(c);
9    }
10 }
```

Figura 7. Faultload para emular falha de colapso no nodo buick

O resultado do teste mostrou que o `Peer` foi capaz de detectar o colapso de `buick`. A validação foi feita visualmente através do comando `peer status` e também através dos próprios logs gerados por OurGrid (Figura 8). Este caso de teste ilustra a capacidade da ferramenta de injetar falhas em pontos específicos da execução de aplicações, dificilmente reproduzidas de forma manual sem a modificação direta do código fonte.

```
OurGrid Peer 3.2.1 is Up and Running
Local GuMs:
  bentley
  buick
  jaguar
  maverick
...
```

(a) antes do colapso de buick

```
OurGrid Peer 3.2.1 is Up and Running
Local GuMs:
  bentley
  jaguar
  maverick

...
```

(b) após o colapso de buick

Figura 8. Saída do comando peer status

6.2. Escalonamento de Tarefas

OurGrid implementa um mecanismo de tolerância a falhas que evita a perda de um *job* quando uma tarefa não é concluída por motivo de falha. Neste caso o escalonador irá executar a tarefa não concluída em outro nodo para então finalizar o *job* com sucesso. Este experimento irá exercitar esta funcionalidade com a definição de um *job* composto por 1000 tarefas, ou seja, todos os nodos deverão ser usados.

Neste experimento serão emuladas falhas de temporização (nodo `jaguar`) e colapso de *link* (nodo `maverick`) segundo uma distribuição de probabilidade uniforme

com taxa de erro de 5%, reproduzindo uma situação de congestionamento de rede onde pacotes são atrasados ou até perdidos, bem como falhas de componentes da infraestrutura de comunicação. A Figura 9(a) é o módulo de cenário de falhas usado para emular as falhas de temporização. Na linha 5 é definida a falha com atraso de 1 segundo por requisição, enquanto que na linha 7 é feita a ativação da mesma. O *faultload* para o nodo `maverick` (Figura 9(b)) é similar ao módulo anterior, mudando apenas o tipo de falha especificado. Note que para indicar o padrão de repetição baseado em uma distribuição de probabilidade, basta criar a classe correspondente e indicá-la na definição da falha (linha 6). Como o fluxo de requisições RMI não é relevante para a ativação das falhas, não é necessário definir nenhum método para a notificação de eventos.

```
1 public class JaguarFaultload
2     extends Faultload {
3   private TimingFault t;
4   public JaguarFaultload() {
5     t = new TimingFault(
6       "t1", 1000);
7     RMIRequestFilter.add(t);
8   }
9 }
```

(a) nodo jaguar

```
1 public class MaverickFaultload
2     extends Faultload {
3   private LinkCrashFault lc;
4   public MaverickFaultload() {
5     lc = new LinkCrashFault("lc1",
6       new UniformDistribution(0.05));
7     RMIRequestFilter.add(lc);
8   }
9 }
```

(b) nodo maverick

Figura 9. Faultloads usados neste experimento

nodos	tarefas		
	escalonadas	concluídas	falhas
bentley	212	212	0
buick	255	255	0
jaguar	255	255	0
maverick	278	278	0

(a) sem injeção de falhas

nodos	tarefas		
	escalonadas	concluídas	falhas
bentley	425	425	0
buick	522	522	0
jaguar	45	45	0
maverick	15	8	7

(b) com injeção de falhas

Figura 10. Resultado da execução do job com 1000 tarefas

Os resultados do experimento (Figura 10) mostram que os nodos `jaguar` e `maverick`, por apresentar falhas de temporização e colapso de *link*, causaram uma grande degradação no desempenho do sistema. O nodo `jaguar` foi capaz de executar apenas 45 tarefas, ou seja, 4.5% do total de tarefas da aplicação. No caso do nodo `maverick`, apenas 8 tarefas das 15 escalonadas foram processadas com sucesso (0.8%). Isso ocorre devido ao fato de que falhas de colapso de *link*, em comparação às falhas de temporização injetadas, causam maior indisponibilidade de nodos. Outro fator importante a ser considerado é que o tempo de execução de cada tarefa é muito baixo, tornando períodos de indisponibilidade mais críticos, já que muitas tarefas serão escalonadas para outros nodos. Apesar da queda de desempenho do sistema devido a falhas, nesta campanha de teste os mecanismos de tolerância a falhas do sistema alvo estão operando de acordo com a especificação. O experimento mostrou o comportamento sob falhas do sistema alvo e sua total cobertura das falhas injetadas.

7. Trabalhos Relacionados

Para diminuir a latência e permitir um maior controle sobre a manifestação da falha injetada, no lugar de introduzir falhas de memória e CPU várias ferramentas injetam diretamente falhas de comunicação. Mas nenhuma destas ferramentas apresenta a facilidade

de definir modelos de falhas adequados para validar especificamente aplicações baseadas em RMI.

DOCTOR [Han et al. 1995] injeta falhas de memória, CPU ou de comunicação em ambientes distribuídos de tempo real, que não são ambientes usuais para aplicações baseadas em RMI. ORCHESTRA [Dawson et al. 1996] foi usado para o teste da implementação de protocolos, especialmente TCP. A ferramenta é inserida na pilha de protocolos do sistema operacional, abaixo da camada onde reside o protocolo sob testes, e atua sobre cada mensagem que flui por esta pilha. Esta ferramenta inspirou ComFirm [Drebes et al. 2005], um módulo de kernel Linux que atua na camada de IP. ORCHESTRA e ComFirm poderiam ser usadas para avaliar aplicações RMI, porque tanto TCP como http, que suportam RMI, estão construídos sobre IP. Entretanto, por interceptar todas as mensagens de um nodo, a seleção de mensagens de mais alto nível específicas de um dado processo exige um esforço muito grande na construção da carga de falhas apropriada. Para evitar este custo na avaliação de aplicações baseadas em UDP, FIONA [Gerchman et al. 2005] foi desenvolvido com modelo de falhas apropriado a este protocolo. Os injetores de FIONA atuam instrumentando classes de comunicação de sistema. A menor intrusividade temporal foi alcançada com a versão usando JVMTI [Jacques-Silva et al. 2004]. Uma ferramenta semelhante foi desenvolvida em paralelo na IBM [Farchi et al. 2004], também baseada na instrumentação da comunicação UDP. Como UDP não é base de RMI, nenhum dos injetores para UDP é apropriado para validar aplicações RMI.

Além de injetores, são encontrados ambientes de injeção de falhas para aplicações distribuídas. NFTAPE [Stott et al. 2000] usa injetores leves, um componente de software compacto e substituível. Entre esses injetores estão previstos componentes para falhas de comunicação, mas relatos de sua utilização não foram localizados. Loki [Chandra et al. 2004] apresenta uma arquitetura distribuída para testar aplicações em cenários com falhas em múltiplos nodos. Loki mantém um estado global entre os nodos do experimento. Gatilhos definidos neste estado global permitem ativar falhas nos processos, mas os injetores em si, assim como a carga de falhas, são deixados a cargo do engenheiro de teste. FIONA também apresenta componentes de controle e monitoramento distribuídos para permitir ativação de falhas em múltiplos nodos mas, ao contrário de Loki, não se baseia em um estado global mas apenas na troca de mensagens entre nodos.

A dificuldade de lidar com modelos de falhas de protocolos de baixo nível de abstração, a inadequação de ferramentas que instrumentam UDP para validar aplicações RMI, e a não disponibilidade de injetores para RMI foram motivações para a construção de FIRMI. Os ambientes providos por NFTAPE, Loki e FIONA não comportam integração com ambientes de teste, outra motivação do projeto de FIRMI, e portanto não estão sendo considerados no momento.

8. Conclusão

Aplicações com fortes requisitos de confiabilidade e disponibilidade devem ser submetidas a procedimentos de teste para validar suas funcionalidades tanto em condições normais de operação quanto em situações de falha. Injeção de falhas, em particular a injeção de falhas de comunicação por *software*, é uma técnica eficiente na validação de tais aplicações. No entanto, esta não é suficiente para exercitar todas as características de um dado sistema, o que não elimina a necessidade de outras técnicas já existentes de teste

de aplicações. Este trabalho apresentou a ferramenta FIRMI, um injetor de falhas para a avaliação de aplicações distribuídas baseadas em RMI que possui como uma de suas principais características a integração em diferentes ambientes de teste e desenvolvimento de aplicações, agregando as vantagens da técnica de injeção de falhas voltada a sistemas distribuídos para estes ambientes.

A descrição de cenários de falhas baseada em linguagens de *script* e a integração de FIRMI em JUnit e ANT permitem a engenheiros de teste usar a ferramenta de modo complementar nas fases de teste de aplicações e assim aumentar a cobertura dos testes empregados na validação de sistemas. Os experimentos conduzidos ilustram a capacidade da ferramenta de injetar falhas em pontos específicos da execução de uma aplicação, e também permite a definição de cenários de falhas mais genéricos que não dependam das características de uma aplicação em particular. Estes fatores fazem de FIRMI uma alternativa viável na validação de sistemas baseados em RMI.

Referências

Apache Software Foundation (2000). ANT. `http://ant.apache.org/`.

Apache Software Foundation (2002). BSF. `http://jakarta.apache.org/bsf`.

Beck, K. and Andres, C. (2004). *Extreme Programming Explained : Embrace Change*. Addison-Wesley Professional, Workingham, 2 edition.

Birman, K. P. (1996). *Building Secure and Reliable Network Applications*. Prentice Hall, 1 edition.

Chandra, R., Lefever, R., Joshi, K., Cukier, M., and Sanders, W. (2004). A global-state-triggered fault injector for distributed system evaluation. *IEEE Transactions on Parallel and Distributed Systems*, 15(7):593–605.

Chang, T. and Aharnad, M. (2004). GT-P2PRMI: improving middleware performance using peer-to-peer service replication. In *Proceedings of the 10th IEEE International Workshop on Future Trends of Distributed Computing Systems (FTDCS'04)*, pages 172–177, Suzhou, China. IEEE Computer Society Press.

Chen, C.-W., Chen, C.-K., Chen, J.-C., Ko, C.-T., Lee, J.-K., Lin, H.-W., and Wu, W.-J. (2004). Efficient support of java rmi over heterogeneous wireless networks. *IEEE International Conference on Communications*, 3(1):1391–1395.

Chiba, S. (1998). Javassist - a reflection-based programming wizard for java. In *Proceedings of OOPSLA'98 Workshop on Reflective Programming in C++ and Java*, Vancouver, Canada.

Dawson, S., Jahanian, F., Mitton, T., and Tung, T.-L. (1996). Testing of fault-tolerant and real-time distributed systems via protocol fault injection. In *Proceedings of the 26th IEEE International Symposium on Fault-Tolerant Computing (FTCS '96)*, pages 404–414, Sendai, Japan. IEEE Computer Society Press.

Delamaro, M. E., Maldonado, J. C., and Mathur, A. P. (2001a). Interface mutation: An approach for integration testing. *IEEE Trans. Software Engineering*, 27(3):228–247.

Delamaro, M. E., Pezzè, M., Vincenzi, A. M. R., and Maldonado, J. C. (2001b). Mutant operator for testing concurrent java programs. In *Anais do 15th Simpósio Brasileiro de Engenharia de Software (SBES'2001)*, pages 386–391, Rio de Janeiro/RJ. SBC.

Drebes, R. J., Leite, F. O., Jacques-Silva, G., Mobus, F., and Weber, T. S. (2005). ComFIRM: a communication fault injector for protocol testing and validation. In *IEEE Latin American Test Workshop, 6th (LATW'05)*, pages 115–120, Salvador, Brazil.

Farchi, E., Krasny, Y., and Nir, Y. (2004). Automatic simulation of network problems in UDP-based Java programs. In *Proceedings of the International Parallel and Distributed Processing Symposium, 18th (IPDPS'04)*, page 267, Santa Fe, New Mexico, USA. IEEE Computer Society.

Gamma, E. and Beck, K. (2001). Junit. `http://www.junit.org`.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional Computing Series. Addison-Wesley, 1st edition.

Gerchman, J., Jacques-Silva, G., Drebes, R., and Weber, T. S. (2005). Ambiente distribuído de injeção de falhas de comunicação para teste de aplicações java de rede. In *Anais do 19 Simpósio Brasileiro de Engenharia de Software (SBES 2005)*, pages 232–246, Uberlândia, MG. SBC.

Han, S., Shin, K. G., and Rosenberg, H. (1995). DOCTOR: An integrateD sOftware fault injeCTiOn enviRonment for distributed real-time systems. In *Proceedings of the International Computer Performance and Dependability Symposium*, pages 204–213, Erlangen, Germany. IEEE Computer Society Press.

Hsueh, M.-C., Tsai, T. K., and Iyer, R. K. (1997). Fault injection techniques and tools. *IEEE Computer*, 30(4):75–82.

Jacques-Silva, G., Drebes, R. J., Gerchman, J., and Weber, T. S. (2004). FIONA: A fault injector for dependability evaluation of Java-based network applications. In *Proceedings of the 3rd IEEE International Symposium on Network Computing and Applications (NCA'04)*, pages 303–308, Washington, DC, USA. IEEE Computer Society.

Java Community Process (2005). JSR-000223 scripting for the javatm platform. `http://jcp.org/aboutJava/communityprocess/pr/jsr223/`.

Martins, E., Rubira, C. M. F., and Leme, N. G. M. (2002). Jaca: A reflective fault injection tool based on patterns. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'02)*, pages 483–487, Washington, DC. IEEE Computer Society Press.

Niemeyer, P. and Knudsen, J. (2000). *Learning Java*. O'Reilly, 1 edition.

OurGrid (2005). Ourgrid online manual. `http://www.ourgrid.org/`.

Pedroni, S. and Rappin, N. (2002). *Jython Essentials*. O'Reilly, 1 edition.

Stott, D. T., Floering, B., Kalbarczyk, Z., and Iyer, R. K. (2000). NFTAPE: A framework for assessing dependability in distributed systems with lightweight fault injectors. In *Proceedings of the IEEE International Computer Performance and Dependability Symposium (IPDS 2000)*, pages 91–100, Chicago, Illinois, USA. IEEE Computer Society Press.

Sun Microsystems (1997). Java RMI specification. `http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmiTOC.html`.

Implementing Framework Crosscutting Extensions with EJPs and AspectJ

Uirá Kulesza[1], Roberta Coelho[1], Vander Alves[2], Alberto Costa Neto[2], Alessandro Garcia[3], Carlos Lucena[1], Arndt von Staa[1], Paulo Borba[2]

[1]Departamento de Informática – Pontifícia Universidade Católica do Rio de Janeiro (PUC-Rio)
{uira, roberta,lucena,arndt}@inf.puc-rio.br

[2] Centro de Informática – Universidade Federal de Pernambuco
{vra, acn, phmb}@cin.ufpe.br

[3]Lancaster University, Computing Department, Lancaster - United Kingdom
garciaa@comp.lancs.ac.uk

Abstract. *In a previous work, we proposed a framework extension approach based on the use of a new concept, called Extension Join Points (EJPs). EJPs enable the framework systematic extension by means of variability and integration aspects. In this paper, we show how EJPs can be implemented using the mechanisms of the AspectJ language. To evaluate the usefulness of the EJPs in the framework extension process, we have used them in the development of three OO frameworks from different domains. As a result of our case studies, we present: (i) an initial categorization of different kinds of contracts between frameworks, EJPs and aspects which can be implemented in AspectJ; and (ii) a set of lessons learned when specifying the EJPs.*

Resumo. *Uma abordagem para extensão de frameworks baseada em um novo conceito, denominado Extension Join Points (EJPs), tem sido proposta anteriormente. EJPs possibilitam a extensão sistemática de frameworks, através do uso de aspectos de variabilidade e integração. Neste artigo, nós mostramos como os EJPs podem ser implementados usando os mecanismos da linguagem AspectJ. Para avaliar a utilidade dos EJPs no processo de extensão de frameworks, nós os utilizamos no desenvolvimento de 3 frameworks OO de diferentes domínios. Como um resultado de nossos estudos de caso, nós apresentamos: (i) uma categorização inicial de diferentes tipos de contratos entre frameworks, EJPs e aspectos, os quais podem ser implementados em AspectJ; e (ii) um conjunto de lições aprendidas quando especificando os EJPs.*

1. Introduction

Object-oriented (OO) frameworks [11] represent nowadays a common and important technology to implement program families. They enable modular, large-scale reuse by encapsulating one or more recurring concerns of a given domain, and by offering different variability and configuration options to the target applications. In the framework based development, applications are implemented by reusing the architecture defined by the frameworks and by extending their respective variation points or hot-spots [11]. Hence, the adoption of the framework technology brings in general significant productivity and quality in the development of applications. Besides their advantages, some researchers [5, 8, 23, 24, 28] have recently described the inadequacy of OO mechanisms to address the modularization and composition of many framework features, such as, optional [5], alternative and crosscutting composition features [23, 24]. As a consequence, the limited modularity provided by the OO

mechanisms brings difficulties to configure many framework features for specific needs, thus impeding the framework adaptation and reuse [5, 8, 23, 24, 28].

Aspect-oriented software development (AOSD) [12, 17] has been proposed as a technology which aims to offer enhanced mechanisms to modularize crosscutting concerns. Crosscutting concerns are concerns that often crosscut several modules in a software system. AOSD has been proposed as a technique for improving the separation of concerns in the construction of OO software, supporting improved reusability and ease of evolution. Recent work [2, 18, 19, 20, 21, 25, 27, 31] has explored the use of aspect-oriented (AO) techniques to enable the implementation of flexible and customizable software family architectures. In these research works, aspects are used to modularize crosscutting variable (optional or alternative) and integration features. In a previous work [19], we have proposed an approach which aims to improve the extensibility of object-oriented frameworks using aspect-oriented programming. Our approach proposes the definition of extension join points in the framework code, which can be extended by means of variability and integration aspects. These aspects are responsible to implement optional, alternative and integration features in the framework. Since the aspects can be automatically unplugged from the framework code, our approach makes easier to customize the framework to specific needs.

This paper shows and evaluates how the framework extension join points (EJPs) from our approach can be implemented in the AspectJ language. The EJPs codification in AspectJ gives us the advantages of explicitly exposing some framework join points and writing contracts that must be satisfied when extending those join points. Hence, it gives more systematization and robustness for our approach in the process of framework extension. To evaluate the usefulness of the EJPs in the framework extension process, we have used them in the development of three OO frameworks from different domains. As a result of our case studies, we present: (i) an initial categorization of different kinds of contracts between frameworks, EJPs and aspects which can be implemented in AspectJ; and (ii) a set of lessons learned when specifying the EJPs.

The remainder of this paper is organized as follows. Section 2 presents background by detailing framework modularization problems addressed by our approach and by introducing AOSD basic concepts. Next, Section 3 gives an overview of our approach for framework development with aspect-oriented programming based on the specification of EJPs. Section 4 then details how our EJPs can be implemented using AspectJ, including the specification of their contracts. This section also presents a categorization of contracts that need be defined when adopting our approach. Subsequently, Section 5 illustrates the implementation of EJPs using AspectJ for two different case studies. Section 6 presents the lessons learned from our case studies. Related work is discussed in Section 7. Finally, Section 8 summarizes our contributions and provides directions for future work.

2. Background

This section briefly revisits research work that describes the inadequacy of object-oriented mechanisms to modularize specific framework features. We also present the basic concepts of AOSD and discuss emerging aspect-oriented design approaches.

2.1 Issues in Modularizing Framework Features

Despite the well-known benefits of OO frameworks in implementing program families, recent research has exposed the inadequacy of framework technology in modularizing features with particular properties, such as optional [5] and crosscutting composition [23, 24] features. These issues hinder the framework instantiation process to meet specific user needs. As a

result, framework reuse can become unmanageable or even impracticable. Next, we describe these two problems of framework feature modularization.

Modularizing Optional Framework Features. Batory et al [2] address the issues of the framework technique in modularizing optional features. An optional feature is a framework functionality that is not used in every framework instance. According to such research, developers typically deal with this problem either by implementing the optional feature in the code of concrete classes during the framework instantiation process, or by creating two different frameworks, one addressing the optional feature and the other one without it. As a result, many framework modules are replicated just for the sake of exposing optional features, thus leading to "overfeatured" frameworks [8], in which several instance-specific functionalities can be present.

By analyzing a number of available frameworks (such as JUnit and JHotDraw), we note that the most widespread practice in implementing framework optional features is the use of inheritance mechanisms to define additional behavior in the framework classes. In the JUnit framework, for example, inheritance relationships are used to define a specific kind of test case as well as additional and optional extensions to test cases and suites.

Crosscutting Feature Compositions in Frameworks Integration. Mattsson et al [23, 24] have analyzed the issues in integrating OO frameworks and proposed several OO solutions. Their research relates the composition of two frameworks to the composition of a new set of features (represented as a framework) in the structure of another framework. For example, suppose we need to extend the JUnit framework to send specific failures that occur to software developers. A specific test failure report could be send by e-mail to different software developers, every time a specific and critical failure happens. Imagine we have available an e-mail framework to support our implementation. The problem here is how we could implement this functionality in the JUnit framework. It involves the integration of the JUnit and the e-mail framework. This composition could be characterized as crosscutting since we are interested to send a failure report by e-mail during the execution of the tests.

Based on a case study [20] with feature compositions involving four OO frameworks of varying complexity and addressing concerns from distinct horizontal and vertical domains [10], we have concluded that the framework integration solutions presented by Mattson et al [23, 24] are invasive and bring several difficulties to the implementation, understanding, and maintenance of the framework composition code. Our analysis has shown that 6 out of 9 solutions described by those authors have poor modularity and a crosscutting nature, requiring invasive internal changes in the framework code.

2.2 Aspect-Oriented Software Development

Aspect-oriented software development (AOSD) [12, 17] is an evolving approach aiming at modularizing concerns, which existing paradigms are not able to capture explicitly. It encourages modular descriptions of complex software by providing support for cleanly separating the basic system functionality from its crosscutting concerns. Crosscutting concerns are concerns that often crosscut several modules in a software system. AOSD supports the modularization of crosscutting concerns by providing abstractions to extract these concerns and later compose them back when producing the overall system. AOSD proposes the notion of aspect as a new abstraction and provides new mechanisms for composing aspects and components (classes, methods, etc.) together at specific join points.

AspectJ [3] is an aspect-oriented extension to the Java programming language. The aspect abstraction in AspectJ is composed of inter-type declarations, pointcuts and advices. Pointcuts have a name and are collections of join points. Join points are well-defined points in

the dynamic execution of system components. Examples of join points are method calls and method executions. Advice is a special method-like construct attached to pointcuts. Advices are dynamic crosscutting features since they affect the dynamic behavior of components. Inter-type declarations specify new attributes or methods to be introduced in specific classes. In this work we will focus on the use of aspect-oriented abstractions to modularize framework extensions which implement optional, alternative or crosscutting composition features.

2.2.1 Obliviousness and Crosscutting Interfaces (XPIs)

Filman and Friedman [13] have identified two properties, *quantification* and *obliviousness*, which they believe are fundamental for aspect-oriented programming. The *Quantification* property refers to the desire of programmers to write programming statements with the following form: "*In programs* P, *whenever condition* C *arises*, *perform action* A". The AspectJ programming language, for example, supports this property by means of the pointcut, join point and advice mechanisms described above. Obliviousness establishes that programmers of the base code – the classes which will be affected by the aspects – do not need to be aware of the aspects which will affect it. It means that programmers do not need to prepare the base code to be affected by the aspects. The following sentence from the authors synthesizes both properties [13]: "*AOP can be understood as the desire to make quantified statements about the behavior of programs, and to have these quantifications hold over programs written by oblivious programmers.*"

In a recent study, Sullivan et al [30] have compared the obliviousness methodology with a new approach to AO development based on *design rules* [4]. In their approach, the authors propose the specification of interfaces between the base code and the aspects, which determine the anticipated definition of join points from the base code before its implementation. These join points are used subsequently in the implementation of the system aspects. The design rule based approach [30] addresses the decoupling of the base and aspect code by offering a clear specification of the interaction and contracts between them and by allowing their parallel development. In the study, the authors have also observed how their approach helps to reduce or eliminate several disadvantages of the obliviousness approach, such as, the codification of complex and fragile pointcuts expressions and the tight coupling of the aspects to changeable and complex details from the base code.

Griswold et al [16] have recently shown how the interfaces between the base code and the aspects, called crosscutting interfaces (XPIs) and previously proposed by the design rules based approach, can be partially implemented in AspectJ. The XPIs are used to abstract a crosscutting behavior existing in the base code. The implementation of XPIs in AspectJ is composed of: (i) *a syntactic part* – which allows to expose specific join points by specifying pointcuts in aspects; and (ii) *a semantic part* – which details the meaning of the exposed join points and it can also define constraints (such as, pre- and post-conditions) that must be satisfied when extending those join points. This semantic part can be partially implemented with enforcement aspects (implemented with `declare error` and `declare warning` AspectJ constructs) [9] or by defining contract aspects which guarantee specific constraints are satisfied before and after the advices execution.

The definition of XPIs has inspired the central idea of our approach to extend object-oriented frameworks by exposing a set of extension join points (EJPs) present in their implementation. Next section gives an overview of the approach. Section 4 details our study of implementation of framework EJPs in AspectJ.

3. An Approach to Extending OO Frameworks with Aspects

This section gives an overview of our framework development approach [19]. Section 4 details our study of realization of the approach using AspectJ.

3.1 Extension Join Points (EJPs)

In our approach, an OO framework specifies and implements not only its common and variable behavior using OO classes, but it also exposes a set of extension join points (EJPs) which can be used to also extend its functionality. Similar to XPIs [16, 30], EJPs establish a contract between the framework classes and a set of aspects extending the framework functionality. Unlike XPIs, however, EJPs aims at increasing the framework variability and integrability. Accordingly, we propose to use the XPI concept in the framework development context, in which EJPs serve three different purposes:

(i) to expose a set of framework events that can be used to notify or to facilitate a crosscutting integration with other software elements (such as, frameworks or components);

(ii) to offer predefined execution points spread and tangled in the framework into which the implementation of optional features can be included;

(iii) to expose a set of join points in the framework classes that can have alternative implementations of a crosscutting variable functionality.

In this context, EJPs document crosscutting extension points for software developers that are going to instantiate and evolve the framework. They can also be viewed as a set of constraints imposed on the whole space of available join points in the framework design, thereby promoting safe extension and reuse. A key characteristic of EJPs is that framework developers and users do not need to learn totally new abstractions to use them, as they can mostly be implemented using the mechanisms of AOP languages (Section 4).

3.2 Framework Core and Extension Aspects

Our approach promotes framework development as a composition of a core structure and a set of extensions. A framework extension can define one of the following: (i) the implementation of optional or alternative framework features; or (ii) the integration with an additional component or framework. The composition between the framework core and the framework extensions is accomplished by different types of *extension aspects*, each one defining a crosscutting composition with the framework by means of its exposed EJPs. We next describe the main concepts of our approach:

(i) *framework core* implements the mandatory functionality of a software family. Similar to a traditional OO framework, this core structure contains the frozen-spots that represent the common features of the software family and hot-spot classes that represent non-crosscutting variabilities from the domain addressed;

(ii) *variability aspects* implement optional or alternative features existing in the framework core. These elements extend the framework EJPs with any additional crosscutting behavior;

(iii) *integration aspects* define crosscutting compositions between the framework core and other existing extensions, such as an API or an OO framework. These elements also rely on the EJPs specification to define their implementation.

The design of an OO framework with aspects following our approach is shown in Figure 1. According to this figure, both variability and integration aspects intercept only join points matched by pointcuts in the EJPs provided by the framework; further, such aspects must comply with all the constraints defined by the EJPs. This brings systematization to the

framework extension and composition with other artifacts, providing a number of benefits [19], such as enhanced understandability and evolution of the framework core, safe framework reuse, and pluggable/unpluggable crosscutting framework extensions.

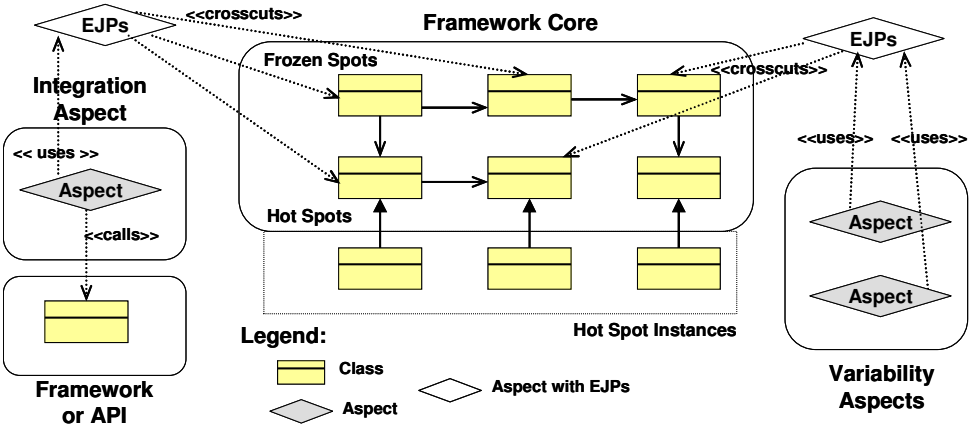


Figure 1. Elements of our Framework Development Approach

4. Implementing Extension Join Points with AspectJ

In this section, we explore the use of AspectJ language to specify the framework extension join points. The EJP codification in AspectJ language brings the following advantages to the framework extension process: (i) it enables the developer to expose a set of join points that are spread in the framework in a single aspect, that can be used to extend the framework functionality with integration and variability aspects; and (ii) it allows the representation of many constraints – that must be satisfied when extending those join points – in a way that they will not just be stated but they will be enforced during compilation and runtime. Next sections detail how we have implemented our EJPs in AspectJ.

4.1 EJPs Structure

The way we codified the EJP in AspectJ-style was inspired in the way Griswold et al [16] codified the XPIs. Each EJP is represented by an aspect comprising a set of pointcut descriptors that represents the set of extension join points of a framework. The EJP constraints which regulate the relationships between the framework, EJPs and extension aspects (mentioned in Section 3.1) are represented, in our approach, by separate aspects. However, we have defined a different methodology from the proposed by Griswold et al [16] to specify these constraints. We have classified them in the following categories: (i) *framework internal contracts* - contracts between the framework and its EJPs – and (ii) *framework extension contracts* - contracts between the EJPs and its extension aspects. The next section describes in detail the kinds of contracts defined in our categorization. Table 1 presents the main elements which comprises an EJP in AspectJ.

4.2 EJPs Contracts

During the definition of the EJPs` contracts, we first categorized the kinds of contracts that should exist between the elements of our approach (Figure 1); we next evaluated different ways to specify them in AspectJ. In the following, we detail our categorization of contracts and the guidelines on their implementation.

The *framework internal contracts* define constraints whose purpose is to assure that framework refactorings and evolution do not affect the functionality of its extension aspects. They are classified in the following categories: (i) *structural* – which aims to guarantee the

framework implements specific interfaces defined by the EJPs; and (ii) *behavioral* – which assures the framework EJPs comprises all and only the framework events (or states) that the EJP is intended to expose.

The *framework extension contracts* are used to assure that each extension aspect respects constraints and invariants of the framework. The following categories were defined: (i) *structural* – these contracts assure that aspects only extend the framework join points exposed by the EJPs; (ii) *behavioral* – specify the framework classes' methods that can be invoked by the extension aspects; and (iii) *invariants* – define specific pre- and pos-conditions that must be preserved before and after the execution of extension aspect advices.

Tables 2 and 3 present the EJP contracts categorization. They also show the different mechanisms of AspectJ that we have used to implement them. AspectJ offers several mechanisms that can be used to specify our different contracts. When choosing mechanisms for each contract type, we prefer static mechanisms to dynamic ones, since only the former can be verified in compilation time, which is the case of the `declare parents`, `declare error` and `declare warning` statements. Some kinds of contracts, however, depend on dynamic information to be implemented. For these specific cases (such as verification of framework invariants), we have used the `adviceexecution` pointcut designator of AspectJ, which allows to intercept the execution of advices. Next section details the specification of EJPs for our case studies, including the implementation of their respective contracts.

Element Name	Purpose
Name	Specifies the name of the EJP, and is represented by the aspect's name in AspectJ.
Scope	Defines all the framework elements that are "encapsulated" by the EJP. It is represented by an AspectJ pointcut descriptor using the within designator including all the packages that comprises the framework (a scope example can be seen in Figure 2).
Crosscutting Extension Points	Quantifies the framework join points that represent relevant events or transition states occurring during the execution of the framework functionalities.
Accessors	Defines a set of pointcuts whose goal is to act for an aspect like accessor methods acts for a class. They expose EJP-specific information, which is useful for the definition of EJP contracts, such as: • EJP main purpose: each EJP should have a main purpose which can be, for example, to expose a specific event or an abstract state of the system. • All exposed join points They are defined as protected because they should be used only by EJP contracts.
Framework Internal Contracts	These contracts constrain the framework developer to expose in the EJP all the events that are expected to be exposed and to implement (in the framework) any interface, which is necessary for the exposure of such events.
Framework Extension Contracts	These contracts regulate the interaction between extension aspects and EJPs. The internal and extension contracts are defined in a separate aspect, in AspectJ.

Table 1. EJP Main Elements

Contract Type	AspectJ Implementation
Structural	Specification of interfaces that must be implemented by framework classes. The obligation to implement these interfaces is assigned by the EJPs using the `declare parents` inter-type construction of AspectJ. The interfaces are also declared inside the aspects that represent the EJPs.
Behavioral	Implementation of enforcement policies guaranteeing that the extension join points are called only and in all appropriate places inside the framework. This contract can be specified using `declare warning` and `declare error` AspectJ statements.

Table 2. Framework Internal Contracts

Contract Type	AspectJ Implementation
Structural	This contract can not be implemented in AspectJ, due to a current limitation of the language which does not allow the developer to restrict specific join points to be affected. Hence, to assure that extension aspects can only extend the EJPs, the developers must follow the programming practice of using only pointcuts specified in the EJP aspects.
Behavioral	This kind of contract restricts the framework classes' methods that can be accessed inside the extension aspects. There are two different ways to specify it: (i) using `declare warning` and `declare error` AspectJ statements, which allow the static verification of policies; and (ii) by defining advices which intercept every advice execution that realizes calls to the framework classes' methods. The `adviceexecution()` pointcut designator is used to intercept the advices execution.
Invariants	This contract defines pre- and pos- conditions that must be assured before and after the advice execution. These contracts are also defined using `adviceexecution()` pointcut designator to intercept the advices execution.

Table 3. Framework Extension Contracts

5. Case Studies

We have conducted three different case studies in which we analyze the use and suitability of AspectJ language to codify our framework EJPs. We selected frameworks from different domains and codified their EJPs and extension aspects using AspectJ language. Due to space limitation, the following sections briefly describe the implementation of EJPs for two case studies. For a complete description of the implementation of EJPs and extensions aspects for these case studies, please refer to [18]. Section 6 discusses lessons learned and guidelines derived from our case studies.

5.1 JUnit

The main purpose of the JUnit framework is to allow the design, implementation and execution of the unit tests in Java applications. According to the JUnit framework, each unit test is responsible for exercising one class method in order to assure that it performs as expected. The JUnit main functionalities are: the definition of test cases or suites to be executed; the execution of a selected test case or suite; and the collection and presentation of the test results. However, different extensions can be implemented to add new functionalities into the JUnit framework core. Some examples of simple extensions are the following:

(i) enable JUnit to execute each test suite in a separate thread, and wait until all tests finish. In order to implement this extension we need to observe the event when the test suite starts running, the event when each test method runs, and the event when the test suite stops running.

(ii) enable JUnit to run each test repeatedly. In order to implement this extension we need to observe the event when each test method runs.

These extensions need to observe JUnit internal events, which are spread over JUnit classes. In other words, such extensions are not well modularized in the object-oriented design. In our approach, an EJP was used to expose such *key events* that are not adequately captured by the OO design and that are useful for crosscutting compositions scenarios. Figure 2 presents an EJP, called `TestExecutionEvents`, which exposes a set of join points in the JUnit framework. Some of these join points were discovered by checking them against these anticipated crosscutting extension scenarios. Based on this first set of discovered join points, we could foresee other relevant events that may be of interest when extending JUnit.

The `TestExecutionEvents` EJP facilitates the definition of JUnit framework crosscutting extensions, since we can implement the extension aspects by reusing join points exposed by it. If necessary, extension aspects can also define more specific EJP-based pointcuts. Therefore, it is possible to codify aspects that affect only specific test cases or suites defined to test an application. In order to do it, it is only necessary to append a sub-expression to the

EJP pointcuts when defining an advice (e.g. `<EJP_pointcut> && within(<AppTestCase>))`. Besides the public pointcut descriptors, the EJP also contains a set of protected pointcuts which represents the EJP scope and the EJP accessors detailed in Section 4.1.

```
public aspect TestExecutionEvents {
       //Needed by: RepeatAllTests extension
       public pointcut testExecution(Test test):
              target(test) && call (void Test.run(TestResult));
       //Needed by: ActiveTestSuite extension
       public pointcut testSuiteExecution (TestSuite ts,TestResult rs):
              target(ts) && call (void TestSuite.run(TestResult)) && args(rs);
       //Needed by: ActiveTestSuite extension
       public pointcut testExecutionFromSuite(TestSuite ts,Test t,TestResult rs):
              target(ts) && call (void TestSuite.runTest(Test, TestResult)) &&
             args(test, result);
        //It is not already used any anticipated extension
       public pointcut testCaseExecution (TestCase tc, TestResult rs):
              target(tc) && call (void TestCase.run(TestResult)) && args(rs);
       //AUXILIARY METHODS:
        protected pointcut EJPMethodsScope():
                     withincode (void TestSuite.runTest(Test, TestResult)) ||
                     withincode (void TestCase.runTest()) ||
                     withincode (void TestSuite.run(TestResult));
                     withincode (void Test.run(TestResult));
       // Framework Scope
       protected pointcut FWScope(): within(junit..*);
        //The main propose of this EJP is to expose all the points in the
        // framework that result in a test execution.
       protected pointcut MainPurpose(): call (void TestResult.run(Test));
}
```

Figure 2. The AspectJ code of one EJP for JUnit framework

```
1. public aspect TestExecutionEventsContracts {
2.     //Behavioral Internal Contract
3.     declare error:
4.             (!TestExecutionEvents.EJPMethodsScope() &&
5.                TestExecutionEvents.MainPurpose() ):
6.               "Contract violation: Test execution should occur "+
7.                "through one of the methods: Test.run(), TestSuite.run(),"+
8.                "TestSuite.runTest(),TestCase.run(), TestCase.runTest()";
9.     //Behavioral Extension Contract
10.    public pointcut variabilityaspects(): within(variabilityaspects..*);
11.    before() : cflow ( adviceexecution() && !variabilityaspects() ) &&
12.            ( call(* *(..)) && TestExecutionEvents.FWScope() ){
13.           throw new RuntimeException("Contract Violation: no aspects, except" +
14.           " variability aspects, can access the elements of JUnit framework.");
15.    }
16.    ...
17.}
```

Figure 3. Corresponding contract of TestExecutionEvents EJP.

As discussed in the previous sections, each EJP contains a set of contracts regulating the internal and extension constraints. Figure 3 illustrates the `TestExecutionEventsContracts` aspect. This aspect contains one internal contract constraining the designer to assure that the pointcut descriptors (PCD) defined in the EJP comprises all and only the join points that results in test method executions. In other words, if any method not specified in an EJP pointcut (`!TestExecutionEvents.EJPMethodsScope()`) tries to call a unit test (`TestExecutionEvents.MainPurpose()`) a contract violation will be signed at compilation time.

The extension contract illustrated in Figure 3, assures that: no aspect, except the variability ones, can directly or indirectly, call a method, create an instance, or access an attribute of an element defined inside JUnit framework. The `adviceexecution()` matches the join points representing the execution of any advice. The expression `adviceexecution() && !variabilityaspects()`, defined in line 11, matches join points that occur during the execution of an advice and that are not defined inside a variability aspect – we defined, in line 10, that every variability aspect will be stored on packages matching the pattern `variabilityaspects..*`. This expression surrounded by `cflow` designator, matches the advice execution of non-variability aspects, or any method in the control flow of the advices defined in such aspects. Finally, the expression `call(* *(..)) && TestExecutionEvents.FWScope()` matches any method call, instance creation, or access an attribute of an element defined inside JUnit framework. Figure 4 shows the `TestExecutionEvents` EJP, which crosscuts JUnit elements and is used by a set of extension aspects.

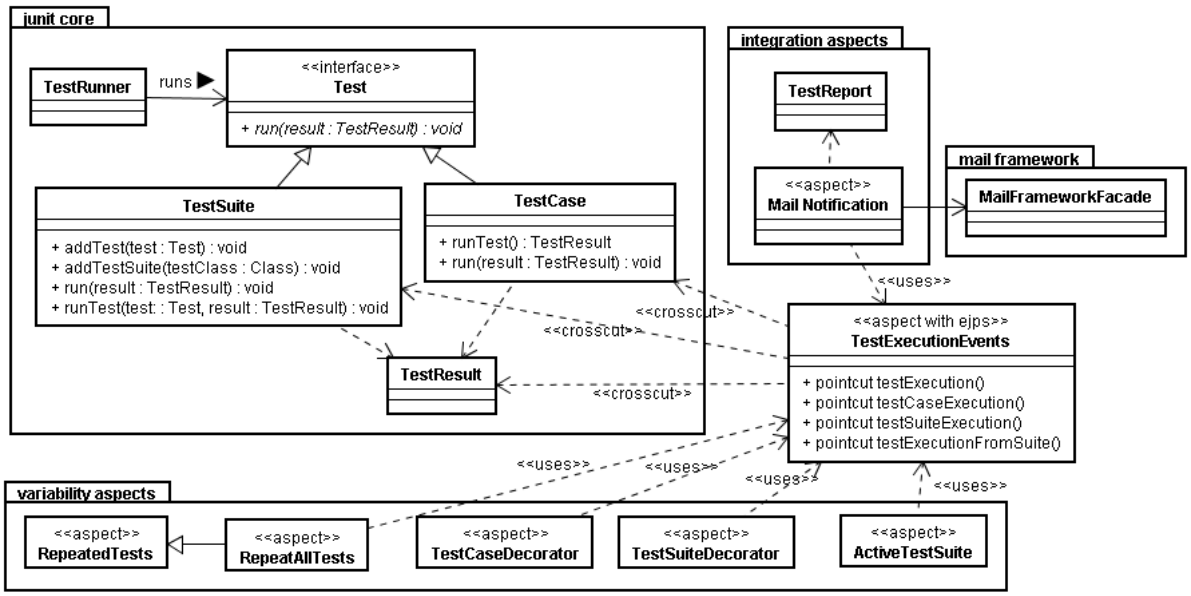


Figure 4. Overview of JUnit Framework and some crosscutting extensions.

5.2 J2ME Game Software Product Line

In this case study, we implemented variant features of an industrial J2ME game Software Product Line[1] based on EJPs. J2ME games are mainstream mobile applications of considerable complexity [2]. Their overall structure and behavior are defined by a framework known in this domain as the *game engine*. Essentially, this is a state machine whose state change is driven by elapsed time and user input through the device keypad. State changes affect the state of various drawing objects (*game actors*) and how they interact. Then, these objects are drawn again after such state changes. Typical hot-spots of this framework include some abstract classes defining basic drawing capability for game actors.

The case study implementation exposed game engine EJPs in order to allow the composition of crosscutting extensions in its basic functionality. Some interesting EJPs are the following: (i) how images are initialized and used; (ii) drawing of specific images; and (iii) game startup and changing screens. We have chosen these EJPs because they represent relevant events that can be of interest when extending the game engine core workflow. The resulting SPL architecture is shown in Figure 5. Package `rain.core` denotes the SPL core,

[1] Access to the game SPL instances was provided by Meantime Mobile Creations/CESAR.

i.e. the game engine. Package `extension join points` encapsulate all the EJPs, which are used by variability aspects and integration aspects in corresponding packages to implement crosscutting extensions.

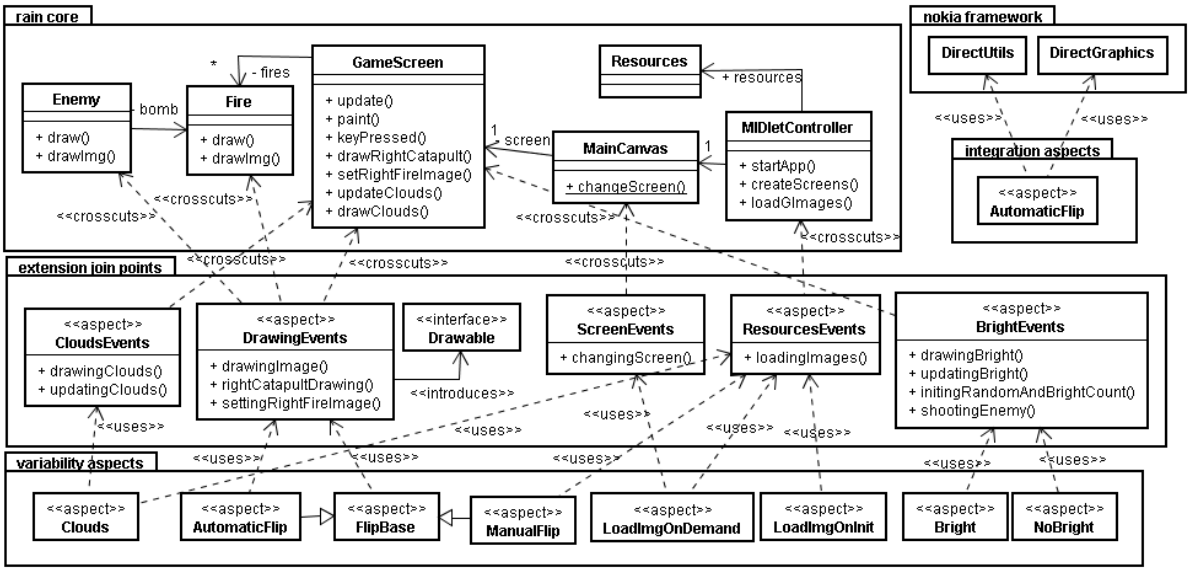


Figure 5. Architecture of the J2ME Game Product Line.

For example, the `DrawingEvents` and the `ResourceEvents` EJPs were composed with variability aspects to implement the alternative features for drawing some images. Specific images may be drawn at various locations and, under certain circumstances, may be transformed (rotated, flipped), which may be accomplished either manually (`ManualFlip` variability aspect) by using fresh new images or automatically (`AutomaticFlip` variability aspect) by transforming the original ones by calling device proprietary drawing API. In fact, this latter aspect also behaves as an integration aspect, due to the interaction with the proprietary API (another framework). Therefore, the fact that such aspect is in two packages is merely a logical, but not a physical view. By exposing these EJPs and composing them with variability and integration aspects, we provide modular implementation for the variant features the aspects represent.

In particular, the `FlipBase` aspect depends on the `DrawingEvents` EJP, which specifies all relevant events needed by such aspect, namely the drawing of images of game objects (Figure 5). We sketch this EJP in Figure 6:

```
public abstract aspect DrawingEvents {
 /* The purpose of the drawingImage PCD is to expose all and only
 drawing requests of images associated to game items that move around the
 game screen. All such requests must be implemented by call to a method
 matching the PCD. We require aspects advising this PCD to access only
 some framework objects through the Drawable or Graphics types. */
       public interface Drawable {
              public void drawImg(Graphics g, int ofsX);
              ...
       }
       declare parents: Enemy implements Drawable;
       declare parents: Fire implements Drawable;
       public pointcut drawingImage(Drawable d, int offSetX, Graphics g) :
              execution(public void Drawable.drawImg(Graphics, int))
                && this (d) && args(g, offSetX);
       ...
}
```

Figure 6. Structure of the DrawingEvents EJP.

The comment in the EJP is a semantic specification of the framework internal contract and the framework extension contract. In the former, the core must signal its intent of drawing image of game items by calling specific methods of the `Drawable` interface, which it must implement (`declare parents` constructs), thus forcing the contract; in the latter, the variability aspect should access framework context only through the EJP; further, such aspect cannot access internal framework details. This constraint can be checked with the `declare warning` construct in the aspect in Figure 7.

```
public aspect DrawingExternalContractChecker {
   // Framework Scope - Calls Not Allowed
   public pointcut FWScopeNotAllowed():
       call ( * !(Drawable+||Graphics).*(..) ) && call (* raincore.*.*(..));

   public pointcut aspectsPackages(): within(variabilityaspects..*);

   declare warning: FWScopeNotAllowed() && aspectsPackages():
              "Extension aspects are accessing internal framework details";
}
```

Figure 7. Contract checking for EJP.

The `FWScopeNotAllowed()` pointcut denotes calls to framework internal types, where we assume that the `Drawable` interface and the `Graphics` class should be visible to the variability aspects. The `aspectsPackages()` denotes calls within such aspects.

Figure 5 also shows other EJPs in the SPL, representing additional crosscutting extensions; it further illustrates that one EJP may be used by more than on variability or integrability aspects, and, conversely, that each such aspect may extend more than one EJP.

6. Discussion and Lessons Learned

This section provides further discussion of issues and lessons we have learned in the evaluation of our approach and the use of AspectJ to implement our EJPs.

6.1 EJP-based Approach Analysis

EJPs stability. EJPs specify not only a set of extension join points in which frameworks can be extended, but they also represent the interfaces between the framework classes and extension aspects. In this sense, they have the same purpose of the XPIs, proposed by Griswold et al [16]. Hence, the implementation of EJPs gives us the benefit to evolve the framework classes without break the aspects that extend its functionality. However, to achieve this benefit is important that joins points exposed by EJPs and their respective contracts remain working when refactoring the framework classes. EJPs can also evolve to accommodate new requirements required by the extension aspects, such as, the exposition of new framework join points or the exposition of additional arguments in the existing join points exposed. All the contracts defined by the EJPs must be revalidated and if necessary rewritten during the refactoring and evolution of EJPs due to change in the framework classes or new demands in the extension aspects.

EJPs modeling. EJPs can also be considered framework hot-spots [11]. They represent flexible points in the execution of specific framework scenarios that can have a crosscutting extension inserted. We have encountered in our case studies that although the modeling of EJPs is dependent on the framework domain, they in general represent relevant events or transition states occurring during the execution of the framework functionalities. Since the EJPs are modeled to accommodate the insertion of optional, alternative and integration

features in the framework, the early identification of these elements in the domain analysis [10] also helps the EJPs modeling.

EJPs as Architectural Enforcement. In our approach, framework classes and extension aspects are constrained to interact in a manner that respects EJPs' contracts. Since EJPs provide a way for the definition of inter-module interactions, it can also be useful for the enforcement of architectural properties in general. Architectural properties, like the extension join points, are not localized in a single system module, they must be observed in many of them. Using EJPs contracts to enforce architectural policies can bring more robustness to aspect oriented systems.

EJPs Specialization. EJPs expose framework join points in which can be inserted new crosscutting behaviors by means of the extension aspects. Since many of these join points can be some of the hook methods of framework hot-spots, EJPs can also be specialized to affect only specific hot-spots instances. In JUnit case study (Section 5.1), for example, we have presented an example of EJP pointcut which can be customized to affect only specific instances of test cases and suites. In that case, the pointcut defined in the EJP need be redefined by the developer implementing the extension aspect that will use it.

6.2 EJPs Implementation in AspectJ

Contracts Implementation in AspectJ. The AspectJ available mechanisms allowed the implementation of four kinds of EJP contracts defined in our category (Section 4.2). Three different mechanisms were used to implement them: (i) the `declare error` and `declare warning` statements to enforce policies between the framework, EJP and aspects; (ii) the `declare parents` statement that guarantees framework classes implement interfaces defined by the EJPs; and (iii) `adviceexecution` pointcut designator which allows to intercept advices and define specific contracts to be validated before and after their executions. There are specific constraints that cannot be checked with aspects; for example, fields introduced into core classes by means of inter-type declaration should not be accessed by the core (thus resulting in a dependency of the core into the aspects). This cannot be checked statically by aspects, thus requiring an enhanced analysis tool.

Join Point Encapsulation. The only kind of contract not implemented in AspectJ language was the extension contract which determines that aspects can only extend the framework join points exposed by the EJPs. Currently, there is no existing mechanism in AspectJ to restrict advices to extend specific join points. The programming practice to allow developers to only reuse the join points exposed in the EJPs was used in our case studies to guarantee this kind of contract. Larochele et al [22] have proposed a mechanism, called join point encapsulation, which aims to prevent selected join points from being modified by aspects. They extend the AspectJ language to support their `restrict` statement whose implementation allows to prevent the access to specific join points. Since this mechanism was implemented only to previous versions of AspectJ, we did not have the chance to experiment it in our case studies.

Annotation-based Pointcuts. AspectJ [3] has recently incorporated mechanisms to specify join points using Java annotations. EJPs can benefit from this mechanism by allowing inserting the annotations directly in the framework classes' join points being affected. This implementation decision can give more stability (Section 6.1) to the EJPs, since signature-based pointcuts are subject to changes when the framework needs to be constantly refactored. In other words, annotation-based pointcuts now available in AspectJ can become the EJPs more robust in scenarios where the use of the signature-based model generates pointcuts complex and difficult to maintain.

Interface-based Contracts. EJP can define interfaces and specify that types in the framework implement these interfaces by means of the `declare parents` static crosscutting construct available in AspectJ. This structural internal contract is important for promoting higher abstraction for the extension aspects, since these will intercept events on generic rather than specific types, thus leading to reduced coupling with the framework and to higher reuse of extension aspects.

7. Related Work

Our concept of EJPs is inspired by Sullivan et al's work [30] on specification of crosscutting interfaces (XPIs). XPIs abstract crosscutting behavior, isolating aspect design from base code design and vice-versa. Continuing this work, Griswold et al show how to represent XPIs as syntactic constructs [16]. EJPs play a similar role to XPIs, but specifically in the context of framework development, by exposing a set of framework events for notification and crosscutting composition, and by offering predefined execution points for the implementation of optional and alternative features. In the specification of the semantic part of EJPs, however, we have defined a different methodology to specify the constraints which regulate the relationships between the framework, EJPs and extension aspects.

Open Modules [1] introduces a strong form of encapsulating join points occurring inside a module. It permits defining an interface composed by set of pointcuts that can be advised by clients. Any other join point that occurs inside the module is protected from external advising. It permits evolution of a module implementation without considering the aspects advising exported pointcuts, since no changes are made on the interface. It's possible because the aspects are coupled with the module exclusively by the module's interface. However, Open Models has a limitation on the pointcuts that can be written on the interfaces. These poincuts can only intercept join points occurring inside the module, making impossible writing an interface that crosscuts more than one module. Our approach doesn't have this limitation, since an EJP can declare pointcuts (extension points) involving join points occurring in any number of classes. We use contracts based on AspectJ's inter-type constructions (declare error and warning) to control coupling between framework core, EJPs and extension aspects.

Feature oriented approaches (FOAs) have been proposed [29] to deal with the encapsulation of program features that can be used to extend the functionality of existing base program. Batory et al [5] argue the advantages that feature-oriented approaches have over OO frameworks to design and implement product-lines. Mezini and Ostermann [25] have identified that FOAs are only capable of modularizing hierarchical features, providing no support for the specification of crosscutting features. These researchers propose CaesarJ [26], an AO language that combines ideas from both AspectJ and FOAs, to provide a better support to manage variability in product-lines. Our approach is directly related those authors work, since we believe that the design of product-line architectures may benefit from the composition and extension of different frameworks using integration and variability aspects. Additionally, we propose the definition of EJPs as a form of reducing and exposing coupling between the framework core and its extensions, witch are implemented using aspects.

Zhang and Jacobsen [30] propose the Horizontal Decomposition method (HD), a set of principles guiding the definition of functionally coherent core architecture and customizations of it. They suggest dividing the middleware in core and aspects that customize the core with orthogonal functionality. HD adopts obliviousness as a principle, suggesting that framework core should be unaware of the aspects. Our approach suggests that is necessary to use some mechanism to control and expose coupling between framework core and its extension, witch we called EJPs.

Mortensen and Ghosh [27] investigate how AOP helps using and extending object-oriented frameworks in VLSI CAD applications. They suggests that in general the code necessary for integrating the framework into the application is crosscutting, and shows that using AOP it was possible to better modularize that code, reducing the number of lines of code and also improving the application structure. They propose using AOP for constructing a reusable library of framework-based aspects useful in a family of framework-based applications. Our approach suggests using aspects not only for integrating frameworks into applications, but also for composing independent frameworks. Another difference is that we advocate using aspects inside the frameworks to capture crosscutting concerns and expose these extension points.

8. Conclusions and Future Work

In a previous work, we proposed a framework extension approach based on the use of Extension Join Points (EJPs). EJPs enable the framework systematic extension by means of variability and integration aspects. In this paper, we have shown how EJPs can be implemented using the mechanisms of the AspectJ language. Our EJPs were implemented by exposing specific framework join points using AspectJ pointcuts and by defining a set of contracts specified using different static and dynamic AspectJ mechanisms. These contracts play a fundamental role in our approach because they help to govern the relationships between the framework and extension aspects by ensuring that important constraints respected.

As future work, we intend to continue the evaluation of the approach in the development and refactoring of object-oriented frameworks. We also plan to realize quantitative studies [6, 15] to compare the approach with the use of OO techniques with respect to traditional software metrics. In order to enable the adoption of our approach, we intend to derive a more systematic implementation method which offers more detailed steps and guidelines to the design and implementation of extensible OO frameworks with aspects. Finally, we plan to explore the extension of current domain analysis and design methods [10] to support the early modeling of extension join points and framework extension aspects. This also involves to investigate the suitability of UML-based notations to represent the EJPs, such as the aSideML crosscutting interfaces [7].

Acknowledgments. We would like to thank the members of Software Productivity Group at Federal University of Pernambuco for valuable suggestions for improving this paper. This research was partially sponsored by FAPERJ (grant No. E-26/151.493/2005 and No. E-26/100.061/06), CNPq (grants No. 552068/2002-0, 481575/2004-9 and 141247/2003-7), MCT/FINEP/CT-INFO (grant No. 01/2005 0105089400), and European Commission Grant IST-2-004349: European Network of Excellence on AOSD (AOSD-Europe).

References

[1] J. Aldrich, “Open Modules: Modular Reasoning about Advice,” Proceedings of ECOOP’05, LNCS 3586, Springer 2005, pp. 144–168.

[2] V. Alves, P. Matos, L. Cole, P. Borba, G. Ramalho. “Extracting and Evolving Mobile Games Product Lines”. Proceedings of SPLC'05, LNCS 3714, pp. 70-81, September 2005.

[3] AspectJ Team. The AspectJ Programming Guide. http://eclipse.org/aspectj/.

[4] C. Baldwin, K. Clark. Design Rules: The Power of Modularity. MIT Press, Cambridge, MA, 2000.

[5] D. Batory, R. Cardone, and Y. Smaragdakis, Object-Oriented Frameworks and Product-Lines. 1st Software Product-Line Conference (SPLC), pp. 227-248, Denver, August 1999.

[6] N. Cacho, et al. Composing Design Patterns: A Scalability Study of Aspect-Oriented Programming. Proceedings of AOSD'06, Bonn, Germany,2006.

[7] C. Chavez, A. Garcia, U. Kulesza, C. Sant'Anna, C. Lucena. Taming Heterogeneous Aspects with Crosscutting Interfaces. Journal of the Brazilian Computer Society, 2006 (to appear).

[8] W. Codenie, et al. "From Custom Applications to Domain-Specific Frameworks", Communications of the ACM, 40(10),October1997.

[9] A. Colyer, et al. Eclipse AspectJ: Aspect-Oriented Programming with AspectJ and the Eclipse AspectJ Development Tools, Addison-Wesley, 2004.

[10] K. Czarnecki, U. Eisenecker. Generative Programming: Methods, Tools, and Applications, Addison-Wesley,2000.

[11] M. Fayad, D. Schmidt, R. Johnson. Building Application Frameworks: Object-Oriented Foundations of Framework Design. John Wiley & Sons, September 1999.

[12] R. Filman, T. Elrad, S. Clarke, M. Aksit. Aspect-Oriented Software Development. Addison-Wesley, 2005.

[13] R. Filman, D. Friedman. Aspect-oriented programming is Quantification and Obliviousness. In [12], pp. 21–35. Addison-Wesley, 2005.

[14] E. Gamma, et al. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[15] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, A. Staa. Modularizing Design Patterns with Aspects: A Quantitative Study. Proc. 4th Intl. Conference on Aspect-Oriented Software Development, Chicago USA, March 2005.

[16] W. Griswold, et al, "Modular Software Design with Crosscutting Interfaces", IEEE Software, Special Issue on Aspect-Oriented Programming, January 2006.

[17] G. Kiczales, et al. Aspect-Oriented Programming. Proc. of ECOOP'97, Finland, 1997.

[18] U. Kulesza, et al. "Implementing Framework Crosscutting Extensions with EJPs and AspectJ", Technical Report, PUC-Rio, Brazil, August 2006.

[19] U. Kulesza, V. Alves, A. Garcia, C. Lucena, P. Borba. Improving Extensibility of Object-Oriented Frameworks with Aspect-Oriented Programming, Proceedings of ICSR'2006, Springer Verlag, LNCS 4038, pp. 231-245, Torino, Italy, June 2006.

[20] U. Kulesza, A. Garcia, C. Lucena. "Composing Object-Oriented Frameworks with Aspect-Oriented Programming", Technical Report, PUC-Rio, Brazil, April 2006.

[21] U. Kulesza, A. Garcia, C. Lucena, A. von Staa. "Integrating Generative and Aspect-Oriented Technologies", Proceedings of SBES'2004, pp. 130-146, Brasilia, Brazil, October 2006.

[22] D. Larochelle, et al., "Join Point Encapsulation," Proc. Workshop Software Eng. Properties of Languages for Aspect Technologies (SPLAT), 2003.

[23] M. Mattson, J. Bosch, M. Fayad. Framework Integration: Problems, Causes, Solutions. Communications of the ACM, 42(10):80–87, October 1999.

[24] M. Mattsson, J. Bosch. Framework Composition: Problems, Causes, and Solutions. In [7], 1999, pp. 467-487.

[25] M. Mezini, K. Ostermann: "Variability Management with Feature-Oriented Programming and Aspects". Proceedings of FSE'2004, pp.127-136, 2004.

[26] M. Mezini, K. Ostermann. "Conquering Aspects with Caesar". Proc. of AOSD'2003, pp. 90-99, March 17-21, 2003, Boston, Massachusetts, USA.

[27] M. Mortensen, S. Ghosh. Using Aspects with Object-Oriented Frameworks, Proceedings of AOSD'2006, Industry Track, Bonn, Germany, March 20-24, 2006.

[28] D. Riehle, T. Gross. "Role Model Based Framework Design and Integration". Proceedings of OOPSLA'1998, pp. 117-133, Vancouver, BC, Canada, October 18-22, 1998.

[29] Y. Smaragdakis, D. Batory. Mixin Layers: An Object-Oriented Implementation Technique for Refinements and Collaboration-Based Designs, ACM TOSEM, 11(2): 215-255 (2002).

[30] K. Sullivan, et al. Information Hiding Interfaces for Aspect-Oriented Design, Proceedings of ESEC/FSE´2005, pp.166-175, Lisbon, Portugal, September 5-9, 2005.

[31] C. Zhang, H. Jacobsen. "Resolving Feature Convolution in Middleware Systems". Proceedings of OOPSLA'2004, pp.188-205, October 24-28, 2004, Vancouver, BC, Canada.

Um Estudo Comparativo do Tempo de Composição de um Framework Orientado a Aspectos de Persistência e de um Framework Orientado a Objetos de Persistência

Valter Vieira de Camargo[1,2], Erika Nina Höhn[1], José Carlos Maldonado[1]

[1]Instituto de Ciências Matemáticas e de Computação (ICMC)

Universidade de São Paulo (USP) – Avenida do Trabalhador São Carlense, 400

Cep 13.560-970 – São Carlos – SP

[2]Fundação de Ensino Eurípides Soares da Rocha – Univem

Avenida Hygino Muzzi Filho, 529 - Cep 17525-901 - Cx.Postal 2041 – Marília - SP

{valter, hohn, jcmaldon}@icmc.usp.br

Abstract. *This paper presents a quantitative study that investigated whether the use of aspect-oriented programming in implementation of persistence frameworks affects the composition time with a base-code. To conduct the study, two versions of the same persistence framework were used – one of them was implemented using aspect orientation, while the other was implemented using object orientation. Some students of a course participated in the study collecting data from the composition of the two frameworks with two applications. The results have shown that the composition time is smaller when using aspect-oriented programming.*

Resumo. *Este artigo apresenta um estudo quantitativo que investigou se o uso da orientação a aspectos na implementação de frameworks de persistência afeta o tempo de composição com um código-base. Para conduzir esse estudo, foram utilizadas duas versões do mesmo framework de persistência – uma implementada com orientação a aspectos, e outra implementada com orientação a objetos. Alunos de uma disciplina de pós-graduação participaram do estudo coletando os dados da composição dos dois frameworks com duas aplicações. Os resultados mostraram que o tempo de composição é menor quando se utiliza programação orientada a aspectos.*

1 – Introdução

Diferentemente dos frameworks orientados a objetos (FOO) de aplicação, cujo processo de reúso possui apenas a etapa de instanciação, os frameworks que tratam de apenas um interesse são reusados realizando-se duas etapas: instanciação e composição. A etapa de instanciação é o mesmo processo utilizado pelos FOO de aplicação, em que implementam-se ganchos do framework, escolhe-se alguma funcionalidade alternativa ou implementa-se uma nova. A atividade de composição, por sua vez, tem o objetivo de compor as variabilidades escolhidas na primeira na etapa de instanciação com o código-base, o que geralmente é feito com os mesmos mecanismos orientados a objetos utilizados na instanciação, causando entrelaçamento e espalhamento de código de diferentes interesses.

[1] Trabalho realizado com o apoio financeiro da CAPES.

Exemplos de frameworks orientados a objetos (FOOs) de persistência que se enquadram nessa categoria são: *Hibernate* (2006), *Cayenne* (2006) e *OJB* (2006).

Com o surgimento da programação orientada a aspectos (POA), e a possibilidade de encapsular interesses transversais (*crosscutting concerns*) de forma mais adequada [Kiczales et al., 1997], vários pesquisadores voltaram sua atenção para frameworks que tratam de um único interesse [Constantinides et al., 2000], [Pinto et al, 2002], [Rashid and Chitchyan, 2003], [Soares et al., 2002], [Vanhaute et al., 2001], [Couto et al., 2005], [Hanenberg et al., 2004]. Com os novos mecanismos disponíveis na POA, a composição desses frameworks com o código-base é facilitada e não apresenta os problemas de entrelaçamento e espalhamento existentes quando apenas orientação a objetos é usada. Os frameworks que tratam de um único interesse e que utilizam os novos mecanismos de composição da POA são denominados neste trabalho de Frameworks Transversais (*Crosscutting Frameworks*) (FTs) [Camargo e Masiero, 2005].

Vários trabalhos relatam que o uso da POA na implementação de frameworks de um único interesse resulta em benefícios de manutenção, reúso e legibilidade de código. Em particular, o interesse de persistência tem sido bastante pesquisado nesse sentido [Soares et al., 2002], [Couto et al., 2005], [Rashid and Chitchyan, 2003]. Contudo, não foram encontrados trabalhos cujo foco fosse averiguar se a POA também afeta o tempo de composição desses frameworks com o código-base. Esse é um fator importante porque a adoção de uma nova tecnologia, por exemplo, um framework, pode depender, além de outros fatores, da produtividade e eficiência que essa nova técnica traz para a equipe de desenvolvimento.

Dessa forma, este artigo apresenta um estudo quantitativo que comparou o tempo de composição de um framework OO e de um FT com um código-base. Para a realização do estudo foram utilizadas duas versões de um mesmo framework de persistência: uma orientada a objetos e uma orientada a aspectos. Esses frameworks foram compostos com duas aplicações similares e dados foram coletados.

Este artigo está organizado da seguinte forma: na seção 2, são apresentados os frameworks utilizados no estudo, bem como os passos necessários para sua composição com um código-base. Na Seção 3 o estudo de caso é descrito no formato proposto por Wohlin (2000). Na Seção 4 são mostrados os dados coletados e na Seção 5 a análise dos resultados. Na Seção 6 discutem-se possíveis ameaças aos resultados obtidos neste estudo e na Seção 7 encontram-se os trabalhos relacionados. Na Seção 8 estão as conclusões e os trabalhos futuros.

2. Frameworks Utilizados no Estudo

O estudo utilizou dois frameworks de persistência, que são, na realidade, duas versões – uma orientada a objetos e uma orientada a aspectos – de um mesmo framework. A versão orientada a objetos foi desenvolvida a partir da versão orientada a aspectos, e manteve a arquitetura, funcionalidades e características dessa versão, sendo modificada apenas a técnica utilizada para a composição com o código-base.

Como o FT de persistência havia sido desenvolvido com a preocupação de utilizar aspectos apenas onde fosse necessário, a criação de uma versão OO foi facilitada. Por exemplo, no FT de persistência, uma das tarefas da etapa de composição é fazer com que as classes persistentes da aplicação estendam uma interface do FT chamada `PersistentRoot`, o que é feito com declarações inter-tipo da linguagem AspectJ. No caso do framework de persistência OO, as classes persistentes da aplicação também devem estender a mesma interface, porém isso é feito introduzindo-se declarações

"`extends PersistentRoot`" em cada uma das classes. Outro exemplo é a composição da conexão, que no FT é feita concretizando-se dois conjuntos de junção (*pointcuts*) por meio da criação de um aspecto concreto. Já no framework OO, essa composição é feita por meio da invocação de métodos para abrir e fechar a conexão em vários.

Como comentado anteriormente, o processo de reúso de um framework que trata de um único interesse é feito em duas etapas: instanciação e composição. A etapa de instanciação para as duas versões do frameworks de persistência é idêntica, e consiste em escolher o tipo de conexão com o banco de dados. Como o objetivo do estudo foi comparar apenas a etapa de composição, os participantes do estudo receberam os frameworks com a instanciação já realizada, bastando apenas realizar a etapa de composição do framework com o código-base.

O framework de persistência utilizado no estudo, independentemente da versão (OO ou OA), possui duas partes distintas. A primeira parte corresponde ao que é chamado de "comportamento da persistência", que é um conjunto de operações básicas de persistência que devem ser adicionadas nas classes persistentes da aplicação. A segunda parte é o tratamento da conexão com o banco de dados, o qual consiste em determinar pontos da aplicação em que a conexão deve ser aberta e fechada. A etapa de composição desse framework possui três passos, sendo que os dois primeiros correspondem ao acoplamento dessas duas partes com o código base, e o último aos testes de acoplamento. Embora essas duas partes formem a persistência como um todo, estão sendo tratadas de forma separada porque a composição com o código-base utiliza técnicas diferentes de programação, tanto na versão OO quanto na OA. Coletando separadamente os dados foi possível averiguar qual das partes tem mais impacto no tempo de composição total.

2.1 – Passos da Etapa de Composição dos Frameworks

Os passos da etapa de composição são: 1) Acoplar Comportamento de Persistência; 2) Acoplar Conexão e 3) Testar Acoplamento. Esses são os passos que os participantes do estudo deveriam realizar para coletar dados. Para os dois primeiros passos, os participantes deveriam registrar o tempo de realização do passo, a quantidade de linhas de código escritas, a quantidade de locais modificados e a quantidade de unidades criadas, isto é, classes, pacotes e aspectos. Para o último passo apenas o tempo deveria ser registrado. Na Tabela 1 são mostrados os dados que deveriam ser coletados e a respectiva sigla de cada um.

Tabela 1 – Significado das Siglas dos Dados Coletados

Legenda dos Dados Coletados	
Dados para Coletar	**Siglas**
Tempo de Acoplamento do Comportamento de Persistência **(Passo 1)**	TACP
Tempo de Acoplamento da Conexão **(Passo 2)**	TAC
Tempo de Teste do Acoplamento **(Passo 3)**	TTeste
Quantidade de Linhas de Código Escritas	LC
Quantidade de Unidades (classes, aspectos, pacotes) Criadas	UC
Número de Locais modificados	LM

O primeiro passo da etapa de composição do FT consiste em criar um aspecto que estende o aspecto abstrato `OORelationalMapping` do framework e que declare, por meio de declarações inter-tipo, todas as classes persistentes da aplicação que devem implementar a interface `PersistentRoot`, como é mostrado na Figura 1. Para cada classe persistente de aplicação, por exemplo: `Endereco` e `Funcionário`, uma linha deve ser escrita informando que essa classe implementa a interface `PersistentRoot`.

Já no caso do framework OO de persistência, a realização do primeiro passo consiste em inserir a declaração "`extends PersistentRoot`" e a declaração de importação "`import persistence.PersistentRoot`", em cada classe persistente de aplicação, assim como está sendo mostrado na Figura 2.

```
public aspect MyOORelationalMapping extends OORelationalMapping {
       declare parents : Departamento    implements PersistentRoot;
       declare parents : Endereco        implements PersistentRoot;
       declare parents:  Funcionario     implements PersistentRoot; }
```

Figura 1 – Acoplamento do Comportamento de Persistência – AO

```
import persistence.PersistentRoot;
       public class [ClassName] extends PersistentRoot { … }
```

Figura 2 – Acoplamento do Comportamento de Persistência – OO

O segundo passo da etapa de composição consiste em estabelecer os locais do código onde a conexão deve ser aberta e fechada. Geralmente isso deve ser feito em programas (código-cliente) que usam objetos das classes persistentes definidas no passo anterior. No caso do FT, um aspecto concreto deve ser criado para estender o aspecto abstrato `ConnectionComposition` do FT, como está sendo mostrado na Figura 3. O aspecto abstrato `ConnectionComposition` possui dois conjuntos de junção abstratos: `openConnection()` e `closeConnection()` que devem ser concretizados. O conjunto de junção `openConnection()` deve ser concretizado de forma a entrecortar pontos da aplicação em que a conexão deve ser aberta, e o `closeConnection()` deve entrecortar pontos em que a conexão deve ser fechada. Além disso, também deve-se implementar o método gancho `getNameOfConnectionVariabilitiesClass()`. Esse método deve ser implementado de forma a retornar o nome da classe que foi criada na etapa de instanciação. Como a instanciação já havia sido realizada, o nome dessa classe também foi informado aos participantes.

```
public aspect myConnectionCompositionRules extends ConnectionComposition {

 pointcut openConnection(): execution (public static void SomeClass.main(..));
 pointcut closeConnection(): execution(public static void SomeClass.main(..));

  public String getNameOfConnectionVariabilitiesClass(){
             return "instantiation.myConnectionVariabilities";
   }}
```

Figura 3 – Acoplamento da Conexão – AO

Para o framework OO, a realização do segundo passo consiste em instanciar um objeto da classe `MyConnectionVariabilities` e chamar o método `ConnectDB()` em todos os pontos do código-base onde a conexão deve ser aberta e fechada. O estabelecimento da conexão deve ser feito logo no início dos programas e o encerramento deve ser feito no final. Na Figura 4 são mostradas em negrito as linhas de código que devem ser inseridas para que a composição possa ser realizada com um programa.

A etapa de teste foi idêntica para ambos os frameworks. Consistia em executar determinados programas e verificar se a saída correspondia à saída esperada. Por exemplo, uma das aplicações usadas no estudo possui um programa chamado `RegisterOfEvents`, cujo objetivo é armazenar em várias tabelas do banco de dados eventos que os

funcionários podem gerar durante o mês, por exemplo faltas, horas extras e atrasos. Se a execução desse programa não armazenar todos os dados esperados, a etapa de composição não pode ser considerada concluída e o erro deve ser rastreado e corrigido. Optou-se por não fornecer interfaces para as aplicações para que o tempo dos testes não fosse influenciado. O fornecimento dos programas fez com que todos realizassem os mesmos testes em tempos bastante próximos.

```
public class admissaoDeAssalariado {
  public static void main(String[] args) {
   MyConnectionVariabilities con = new MyConnectionVariabilities();
   con.ConnectDB();
      // Todo o comportamento do programa
   con.DisconnectDB();
   }}
```

Figura 4 – Acoplamento da Conexão - OO

3 – Descrição do Estudo Comparativo

Esta seção apresenta a definição, o planejamento e a execução do estudo de caso no formato proposto por Wohlin (2000). A análise dos resultados é mostrada na Seção 6.

3.1 - Definição do estudo

O objetivo foi **Analisar** o tempo de composição de um FT de persistência com uma aplicação e de um FOO de persistência com uma aplicação **para o propósito de** avaliação **com respeito à** eficiência **do ponto de vista de** desenvolvedores de software n**o contexto de** estudantes de pós-graduação em Ciências da Computação.

3.2 - Planejamento do estudo

a) Seleção do contexto. O estudo foi realizado com estudantes de pós-graduação em Ciência da Computação, no contexto da disciplina de Tópicos Avançados em Engenharia de Software, utilizando aplicações de domínio genérico (Loja de CDs e Oficina Eletrônica).

b) Formulação de hipóteses. Foram elaborados três tipos de hipóteses para o estudo: as hipóteses para o efeito do uso do Framework de Persistência no resultado podem ser vistas na Tabela 2; as hipóteses para o efeito do uso das Aplicações no Resultado encontram-se na Tabela 3, e as hipóteses para o efeito do grupo ou efeito da Interação Aplicação X Framework estão na Tabela 4.

c) Seleção de variáveis. As variáveis independentes são todas aquelas que são manipuladas e controladas durante o estudo. No estudo realizado essas variáveis são os Frameworks de persistência (OO e OA) e as aplicações (Loja de CDs e de Oficina Eletrônica).

As variáveis dependentes são aquelas que estão sob análise. Devem-se observar suas variações com base nas mudanças feitas nas variáveis independentes. No caso do estudo realizado a variável dependente é o tempo de composição.

d) Seleção dos participantes. Os participantes do estudo foram selecionados por meio de amostragem não-probabilística por conveniência. Eram alunos de pós-graduação da disciplina de Tópicos Avançados em Engenharia de Software, ministrada no ICMC/USP.

Tabela 2 – Hipóteses para o Efeito do Framework de Persistência

H_0:	Não há diferença entre desenvolvedores utilizando Framework de Persistência OA e desenvolvedores utilizando Framework de Persistência OO com respeito à eficiência individual.
Ha:	Há diferença entre desenvolvedores utilizando Framework de Persistência OA e desenvolvedores utilizando Framework de Persistência OO com respeito à eficiência individual.

Tabela 3 – Hipóteses para o Efeito do Uso das Aplicações no Resultado

H_0:	Não há diferença entre desenvolvedores fazendo o acoplamento na Aplicação Loja de CDs e desenvolvedores fazendo o acoplamento na Aplicação Oficina Eletrônica com respeito à eficiência individual.
Ha:	Há diferença entre desenvolvedores fazendo o acoplamento na Aplicação Loja de CDs e desenvolvedores fazendo o acoplamento na Aplicação Oficina Eletrônica com respeito à eficiência individual.

Tabela 4 – Hipóteses para o Efeito do Grupo ou Efeito da Interação

H_0:	Não há diferença entre o grupo que fez o acoplamento de um FOA com uma aplicação e o grupo que fez o acoplamento de um FOO em relação a eficiência.
Ha:	Há diferença entre o grupo que fez o acoplamento de um FOA com uma aplicação e o grupo que fez o acoplamento de um FOO em relação a eficiência.

e) Projeto do estudo realizado. O estudo foi planejado em bloco e balanceado para assegurar que os tratamentos tivessem igual número de participantes e para possibilitar a comparação dos efeitos das variáveis independentes, como pode ser visto na Figura 5. Na segunda fase do estudo, os 4 participantes mais experientes em OO e OA e nas linguagens Java e AspectJ executaram novamente as etapas do projeto experimental anterior, sendo invertida as combinações dos frameworks com as aplicações (Figura 5).

			Aplicações	
			Loja de CDs	**Oficina Eletrônica**
Framework de Persistência	1ª. Fase	**Treinamento**: exemplo de Composição do Framework de Persistência OO e OA em uma terceira aplicação.		
		Framework OO	Grupo 1	Grupo 2
		Framework OA	Grupo 2	Grupo 1
	2ª Fase	**Framework OO**	P4, P12	P1, P11
		Framework OA	P1, P11	P4, P12

Figura 5 – Projeto Experimental

f) Tipo de projeto: dois fatores com dois tratamentos (2*2 fatorial)

A distribuição dos participantes nos grupos foi realizada visando a colocar a mesma quantidade de participantes experientes nos dois grupos. A experiência dos participantes foi avaliada pelo questionário de caracterização dos participantes – documento utilizado para caracterizar experiência profissional e experiência nos assuntos relacionados com o estudo (conhecimento em OO e OA e das linguagens JAVA e AspectJ). O gráfico mostrado na Figura 6 (a) mostra a experiência dos participantes em Java, AspectJ e POA, e o gráfico (b) mostra a experiência com programação em Java e AspectJ. Os participantes P1, P2, P7 e P8 foram considerados experientes em POA.

g) Instrumentação. Os participantes deveriam contabilizar o tempo gasto para realizar a composição utilizando cada um dos frameworks.

Os documentos utilizados no estudo foram: formulário de caracterização dos participantes, para obtenção da experiência profissional e nos assuntos relacionados diretamente ao estudo; Formulário de registro, para preenchimento de todas as

informações durante a execução do estudo; Roteiro de execução, roteiro com todos os passos a serem seguidos para execução do estudo.

Os frameworks utilizados foram um de persistência orientado a objetos e um de persistência orientado a aspectos. E as duas aplicações foram uma para loja de CDs e uma para Oficina Eletrônica. A aplicação de loja de CDs possui 6 classes e 13 programas. O sistema de oficina eletrônica, possui 7 classes e 12 programas. As aplicações possuem complexidade similar e aproximadamente o mesmo número de linhas de código. Assim como os frameworks, essas aplicações foram desenvolvidas no meio acadêmico.

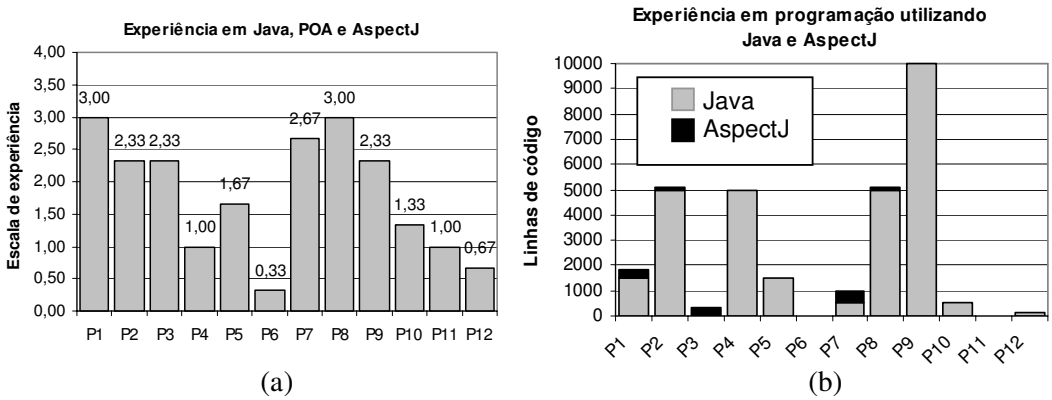


Figura 6 – (a) Experiência dos participantes em POA, Java e AspectJ; (b) Experiência dos participantes nas linguagens JAVA e AspectJ (linhas de código)

Quanto ao treinamento, foi realizado um pequeno treinamento visando a uniformizar o conhecimento sobre programação orientada a aspectos e de Frameworks (OO e OA). Também fez parte do treinamento a utilização de uma terceira aplicação, com a qual deveriam ser feitos os acoplamentos utilizando os frameworks OO e OA e todos os formulários que seriam utilizados no estudo posteriormente, para que o aprendizado sobre como fazer uma composição utilizando os frameworks e como preencher os formulários, não interferisse no tempo de acoplamento durante o estudo.

O objetivo da primeira fase foi realizar uma avaliação pura do tempo de composição e por isso, os pontos de acoplamento do código-base foram informados. Já na segunda fase, o objetivo foi avaliar, além do tempo que a técnica de programação causa, se a existência de uma nova dificuldade também interfere nos resultados.

3.3 - Execução. Os participantes executaram as atividades planejadas seguindo o projeto experimental. Primeiro treinaram a composição de um framework de persistência OO e depois de um FT em uma aplicação desenvolvida para exemplo.

Na primeira etapa da execução, o Grupo 1 utilizou o Framework de Persistência OO com a aplicação Loja de CDs, enquanto que o Grupo 2 utilizou o mesmo Framework com a aplicação Oficina Eletrônica. Na segunda etapa, os grupos utilizaram o Framework de Persistência OA, sendo que trocaram as aplicações, o grupo 1 utilizou a aplicação Oficina Eletrônica e o Grupo 2 utilizou a aplicação Loja de CDs.

4 – Dados Coletados

Esta seção mostra os dados coletados das duas fases do estudo. Nas Tabelas 5 e 6 são mostrados os dados da primeira fase do estudo, em que havia tanto participantes

experientes quanto novatos, e na Tabela 7, são mostrados os dados da segunda fase, em que havia apenas participantes experientes. Os significados das siglas dos dados coletados podem ser visto na Tabela 1.

Tabela 5 – Primeira Etapa da Primeira Fase – Composição com o Framework Orientado a Objetos

Primeira Etapa: Composição com Framework Orientado a Objetos de Persistência														
	Aplicação Loja de CDs – Grupo 1							Aplicação Abaco – Grupo 2						
	P1	P2	P3	P4	P5	P6	**média**	P7	P8	P9	P10	P11	P12	**média**
Dados														
1- TACP	01:55	10:00	02:11	02:07	01:58	04:00	**03:42**	05:00	01:58	01:40	03:05	04:00	06:00	**03:37**
1.1 - LC	10	10	10	10	10	10		16	16	16	16	16	16	
1.2 - UC	0	0	0	0	0	0		0	0	0	0	0	0	
1.3 - LM	5	5	5	5	5	5		8	8	8	8	8	8	
2- TAC	07:45	05:00	05:14	07:50	05:17	15:00	**07:41**	06:00	04:00	05:51	08:10	05:30	08:00	**06:15**
2.1 - LC	48	48	48	48	48	48		44	44	44	44	44	44	
2.2 - UC	0	0	0	0	0	0		0	0	0	0	0	0	
2.3 - LM	12	12	12	12	12	12		11	11	11	11	11	11	
3-TTeste	09:20	06:00	03:26	04:04	03:50	05:00	**05:17**	08:00	00:56	07:09	06:36	06:35	12:00	**06:53**
Soma dos Tempos	19:00	21:00	10:51	14:01	15:57	24:00	**16:40**	19:00	06:54	14:40	17:51	16:05	26:00	**16:45**

Tabela 6 – Segunda Etapa da Primeira Fase – Composição com o FT de Persistência

Segunda Etapa: Composição com FT de Persistência														
	Aplicação Abaco – Grupo 1							Aplicação Loja de CDs – Grupo 2						
	P1	P2	P3	P4	P5	P6	**média**	P7	P8	P9	P10	P11	P12	**média**
Dados														
1- TACP	05:45	04:00	03:10	03:00	03:10	02:02	**03:31**	04:00	02:22	03:33	02:58	03:12	10:00	**04:21**
1.1-LC	13	13	10	13	12	12		10	9	8	9	9	8	
1.2-UC	2	2	2	2	2	2		2	2	2	2	2	2	
1.3- LM	0	0	1	0	0	0		0	0	0	0	0	0	
2- TAC	02:30	01:00	03:35	01:54	02:30	04:00	**02:35**	02:00	00:42	03:40	02:57	02:32	03:00	**02:29**
2.1 - LC	10	12	8	12	10	7		10	8	9	12	10	7	
2.2 – UC	1	1	1	1	1	1		1	1	1	1	1	1	
2.3 – LM	0	0	1	0	0	0		0	0	0	0	0	0	
3-TTeste	08:50	07:00	04:30	07:41	03:56	07:00	**06:29**	06:00	00:40	05:55	05:30	03:55	03:00	**04:10**
Soma dos tempos	17:05	12:00	11:15	12:35	09:36	13:02	**12:35**	12:00	03:44	13:08	11:25	09:39	16:00	**10:59**

Os dados foram coletados separadamente para cada passo da etapa de composição do framework, isto é, foram coletados tempo, número de linhas de código escritas, número de locais modificados e número de unidades criadas para cada um dos três passos do processo de composição, a menos do passo de teste, em que foi contabilizado apenas o tempo. Os dados mostrados nas Tabelas 5, 6 e 7 estão agrupados pelos três passos da composição. No primeiro passo, representado pelo dado "1 - TACP" (veja Tabela 4), os dados "1.1 – LC", "1.2 – UC" e "1.3 – LM", que se encontram logo abaixo desse dado e que estão numerados como subseções, representam os dados coletados durante a realização desse primeiro passo. O mesmo ocorre com o segundo passo "Acoplar Conexão", representado pelo dado "2 – TAC". A letra "T" significa "tempo".

A coluna "média" mostra a média de tempo que os participantes gastaram na realização de cada um dos três passos e a última linha mostra a soma dos tempos de cada

participante e também a soma dessas médias, o que representa o tempo médio de composição do framework com a aplicação.

Note-se na Tabela 5 que a média de tempo de composição do FOO com a aplicação de loja de CDs (16:40) foi muito similar à média de tempo da composição desse mesmo framework com a aplicação Abaco (16:45), o que evidencia que as aplicações são similares em tamanho e funcionalidades. Mesmo no caso do FT, essa mesma similaridade é mantida, como pode ser visto na Tabela 6, a composição com a aplicação de loja de CDs foi realizada em 10:59 e com a aplicação Abaco em 12:35. Não se pode afirmar que a hipótese Ha mostrada na Tabela 3 pode ser refutada, porém há mais indícios que sim do que não.

Tabela 7 – Primeira e Segunda Etapas da Segunda Fase do Estudo

Primeira Etapa – Framework Orientado a Objetos								**Segunda Etapa – Framework Transversal**					
	Abaco – Grupo1			**Loja de CDs – Grupo 2**				**Loja de CDs – Grupo 1**			**Abaco – Grupo 2**		
	P1	**P2**	**média**	**P7**	**P8**	**Média**		**P1**	**P2**	**Media**	**P7**	**P8**	**Média**
Dado													
1- TACP	07:42	07:00	07:21	06:35	03:10	04:53		03:20	03:00	03:10	04:00	04:25	**04:13**
1.2-LC	16	16		10	10			10	11		9	14	
1.3-UC	0	0		2	0			1	1		1	1	
1.4- LM	8	8		5	5			0	0		0	0	
2-TAC	06:30	07:00	06:45	06:15	06:24	06:19		03:15	02:00	02:38	05:00	00:45	**02:53**
2.1 - LC	44	44		36	48			12	12		9	9	
2.2 – UC	0	0		0	0			1	1		1	1	
2.3 – LM	11	11		12	12			0	0		0	0	
Teste	06:05	07:00	06:32	05:00	04:25	04:43		04:18	07:00	05:39	04:00	04:10	**04:05**
Soma dos Tempos	20:17	21:00	**20:39**	17:50	13:59	**15:54**		10:53	12:00	**11:27**	13:00	09:20	**11:10**

Sempre que problemas ocorriam durante a coleta dos dados, os participantes eram instruídos a relatar o problema. Por exemplo, durante a realização da primeira etapa da primeira fase, o participante P2 realizou o acoplamento do comportamento de persistência (TACP) em 10 minutos, um valor bem acima da média dos outros participantes. Contudo, ele relatou que teve problemas com o ambiente Eclipse e que demorou cerca de 7,5 minutos para corrigir o problema. Note-se que se esse valor fosse descontado do tempo relatado pelo participante, ele estaria na média. Isso não foi feito porque problemas desse tipo são comuns durante qualquer desenvolvimento.

5 - Análise dos Dados

5.1 – Primeira Fase do Estudo

Os dados coletados na primeira fase do estudo mostraram que a composição utilizando o FT foi mais eficiente do que com o framework OO para 11 dos 12 participantes, como pode ser observado na Figura 7. Nessa figura, cada par de colunas representa os tempos de composição de um determinado participante. As colunas brancas representam o tempo gasto na composição com o framework OO e as colunas cinza com o FT. As etiquetas sobrepostas sobre cada par de colunas representam a diferença percentual entre os tempos. Por exemplo, o participante P2 realizou a composição com o FT 42,8% mais rápido do que com o framework OO. Note-se que apenas o participante P3 gastou mais tempo com o FT do que com o framework OO, porém foi uma diferença muito pequena: 21 segundos.

Analisando-se os formulários de coleta de dados desse participante não foi constatado nenhum problema.

Um fato interessante foi que os participantes com menos experiência em linguagens de programação obtiveram um nível de eficiência elevado com o FT. Por exemplo, o participante P6 fez a composição com o FT 31,4% mais rápido do que com o FOO, e o participante P12 obteve um índice ainda melhor, 38,4%.

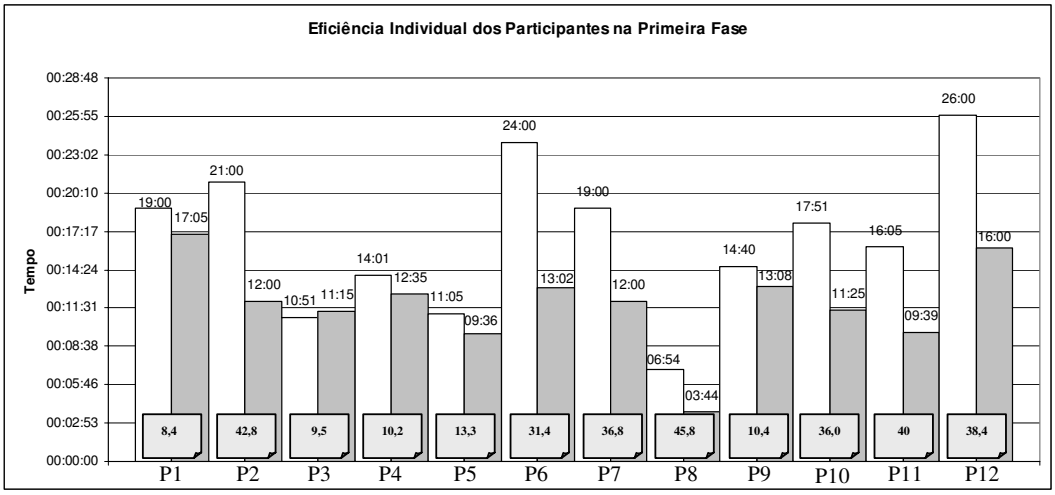


Figura 7 – Diferenças Percentuais entre os Tempos de Composição (Primeira Fase)

Desconsiderando os participantes experientes (P1, P2, P7 e P8), a média de tempo de composição dos novatos com o FT foi 23,65% menor do que usando o framework OO. Mesmo se os dados dos participantes P6 e P12, que são inexperientes em programação, forem desconsiderados, ainda assim os demais novatos foram 21,27% mais eficientes com o FT do que com o framework OO.

Os participantes experientes também foram mais eficientes com o FT do que com o framework OO. A média do tempo de composição foi 8,33% mais rápida com o FT do que os novatos – enquanto os experientes realizaram a composição com o FT 31,98% em média mais rápido do que com o framework OO, a média dos novatos foi de 23,65%.

Apenas o P1 obteve uma diferença percentual pequena entre os tempos, como pode ser observado na Figura 7. Porém, analisando-se os formulários desse participante, pôde-se constatar que ele obteve problemas com o ambiente durante a composição com o FT, o que elevou o tempo de composição com esse framework. Isso também pode ser constatado observando que o tempo de composição usando o FT desse participante foi superior ao de todos os outros participantes (17:05).

Sem considerar diferença entre experientes e novatos, o tempo médio de composição do FT com a aplicação de loja de CDs foi de 10:59 (Tabela 6), um valor 34,1% menor do que o tempo médio de composição dessa mesma aplicação com o framework OO, que foi de 16:40 (Tabela 5). Da mesma forma, o tempo médio de composição do FT com a aplicação Abaco foi de 12:35 (Tabela 6), um valor 24,87% menor do que com o framework OO, que foi de 16:45 (Tabela 5).

Assim, a média de tempo utilizada pelos dois grupos para a composição do FT foi de 11:47, enquanto que utilizando o framework OO foi de 16:42. O FT foi 29,3% mais rápido em média do que com o framework OO nessa primeira fase do estudo.

5.2 – Segunda Fase do Estudo

Os dados obtidos na segunda fase do estudo podem ser vistos no gráfico da Figura 8. Assim como na Figura 7, as notas sobrepostas sobre cada par de colunas representam a diferença percentual do tempo de composição usando cada um dos frameworks. Nesta fase, que só envolveu participantes com experiência em POA, o tempo de composição com o FT foi menor do que com o framework OO para todos os participantes. A eficiência média também foi melhorada para os participantes experientes, enquanto que na primeira fase a composição com o FT foi 31,98% mais rápida, nessa segunda fase foi de 37,37%. Uma diferença provavelmente ocasionada pelo conhecimento do framework.

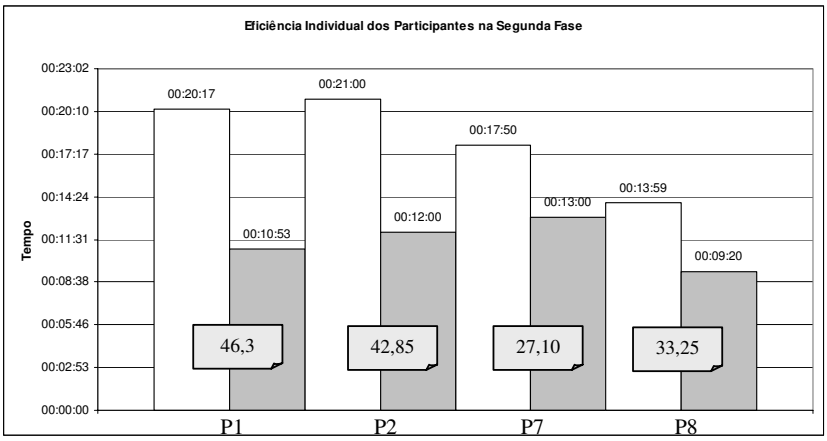


Figura 8 - Diferenças Percentuais entre os Tempos de Composição (Segunda Fase)

Considerando-se as médias de tempo, a composição da aplicação Abaco com o framework OO foi realizada em 20:39, enquanto que com o FT o tempo foi de 11:10, uma diferença de 54,0% a favor da orientação a aspectos. Já, com a aplicação de Loja de CDs a diferença foi bem menor, mas ainda assim favoreceu o FT, a composição com o framework OO foi realizada em 15:54, enquanto que com o FT foi realizada em 11:27, uma diferença de 27,98%.

A média de tempo utilizada pelos dois grupos de experientes para a composição do FT foi de 11:18 nesta segunda fase, enquanto que na primeira foi de 11:12, uma diferença insignificante. Embora também pequena, a diferença com o framework OO foi maior entre as duas fases. Enquanto que na primeira fase a composição com o framework OO foi realizada pelos experientes em 16:28 em média, na segunda foi realizada em 18:17.

5.3 – Resultados Parciais

Como pode ser observado, tanto na primeira fase quanto na segunda, a média de tempo da composição utilizando o FT foi menor do que utilizando o framework OO, mostrando evidências de que a hipótese H_0 mostrada na Tabela 2 talvez pudesse ser refutada se experimentos formais fossem conduzidos.

Se os tempos forem analisados individualmente para cada um dos três passos da composição, tanto na primeira fase quanto na segunda, observa-se que a diferença de tempo relevante ocorre na verdade durante o acoplamento da conexão, e não durante o acoplamento do comportamento da persistência ou dos testes.

Na Tabela 8 é mostrado o tempo médio de cada um dos passos da composição para as duas fases do estudo. Essas médias foram obtidas calculando-se a média das médias já obtidas para cada passo do estudo comparativo. Por exemplo, o tempo médio de

acoplamento do comportamento da persistência da primeira fase é 03:40 com o framework OO (considerando as duas aplicações), como pode ser visto na Tabela 8. Esse valor foi obtido calculando-se a média das médias das duas aplicações com o mesmo framework, não fazendo distinção entre experientes e novatos – 3:42 e 3:37, como pode ser visto na terceira linha na Tabela 5. O mesmo foi feito para os demais valores.

Tabela 8 – Tempo Médio dos Passos de Composição da Primeira e Segunda Fases

Primeira Fase	
Composição com Framework OO	
Média do Tempo de Acoplamento da Persistência com FOO	03:40
Média do Tempo de Acoplamento da Conexão com FOO	**|--- 06:58 ---|**
Média do Tempo de Realização dos Testes com FOO	06:05
Composição com Framework Transversal	
Média do Tempo de Acoplamento da Persistência com FT	03:56
Média do Tempo de Acoplamento da Conexão com FT	**|--- 02:32 ---|**
Média do Tempo de Realização dos Testes com FT	05:20
Segunda Fase	
Composição com Framework OO	
Média do Tempo de Acoplamento da Persistência com FOO	06:07
Média do Tempo de Acoplamento da Conexão com FOO	**|--- 06:32---|**
Média do Tempo de Realização dos Testes com FOO	05:37
Composição com Framework Transversal	
Média do Tempo de Acoplamento da Persistência com FT	03:41
Média do Tempo de Acoplamento da Conexão com FT	**|--- 02:45 ---|**
Média do Tempo de Realização dos Testes com FT	04:52

O passo de acoplamento da persistência possui diferenças muito sutis de tempo entre as versões OO e OA. A justificativa para isso é que a quantidade de linhas de código escritas para realizar esse passo é bastante similar entre as duas versões, como pode ser visto comparando a Figura 1 com a Figura 2. A diferença entre as versões OO e OA é que nessa última, todas as linhas de código estarão em um único aspecto e não espalhadas pelo sistema. Dessa forma, pode-se deduzir que embora determinados interesses transversais tenham um impacto significativo na manutenção e legibilidade do código, influenciam minimamente o tempo da composição.

Com base nesses dados, observa-se que a diferença de tempo significativa somente ocorre no passo de acoplamento da conexão. Na primeira fase do estudo, o tempo médio de realização desse passo usando o FOO foi de 6:58, enquanto que usando o FT foi de 2:32, uma diferença de 63,63%. Na segunda fase a diferença foi de 57,90%.

Observando-se os dados coletados pôde-se concluir que essa diferença de tempo está diretamente relacionada com a quantidade de locais do código que devem ser manipulados, mas principalmente, com a quantidade de linhas de código que são escritas para a realização do acoplamento. Tanto na primeira fase quanto na segunda, houve uma diferença significativa na quantidade de linhas de código escritas quando a conexão é acoplada utilizando-se o FT, fazendo com que o tempo de acoplamento diminua. Comparando-se a Tabela 5 e a 6 pode-se observar que a quantidade de linhas de código escritas (LC) para o acoplamento da aplicação de loja de CDs com o framework OO foi de 48 linhas, enquanto que o acoplamento do FT com essa mesma aplicação utilizou entre 7 e 12 linhas. O mesmo ocorreu com a aplicação Abaco, que utilizou 44 linhas de código para o acoplamento com o framework OO e entre 7 e 12 para o acoplamento com o FT.

Mesmo considerando a escrita de uma linha com orientação a aspectos mais difícil e demorada do que a escrita de uma linha com orientação a objetos, a grande diferença na quantidade de linhas escritas favorece a orientação a aspectos. Contudo, para interesses cuja quantidade de linhas de código escritas para o acoplamento é similar entre as duas versões, a orientação a objetos pode ser beneficiada. Note-se, por exemplo, o passo de

acoplamento da persistência na primeira fase do estudo, que envolveu tanto participantes experientes quanto novatos. A Tabela 8 mostra que a média de tempo usando orientação a objetos (3:40) foi 6,7% mais rápida do que usando orientação a aspectos (3:56). Embora seja uma diferença pequena, está diretamente relacionada com o tamanho das aplicações usadas no estudo. Para aplicações maiores essa diferença tende a aumentar. Note-se também que na segunda fase do estudo, que só envolveu participantes experientes, o tempo de realização do primeiro passo favoreceu a orientação a aspectos. Isso ocorre porque, para esse caso, a dificuldade de se escrever uma linha de código com orientação a aspectos é similar à dificuldade com orientação a objetos.

Os gráficos mostrados na Figura 9 apresentam uma comparação entre o número médio de linhas de código escrito durante a composição com o framework OO e com o FT. Cada coluna representa um passo da etapa de composição: a primeira é o acoplamento da persistência e a segunda o acoplamento da conexão. As colunas brancas representam a quantidade média de linhas de código escritas durante a composição com o framework OO e as cinza com o FT. Sobre cada coluna também é encontrado o tempo médio que os participantes gastaram na realização de cada um dos passos, sem fazer distinção entre experientes ou novatos. Os dois gráficos da parte superior da figura correspondem à primeira fase e os dois da parte inferior correspondem à segunda fase. Note-se que em todos os casos, a quantidade de linhas de código necessárias para se realizar o acoplamento da conexão é maior quando se utiliza o framework OO.

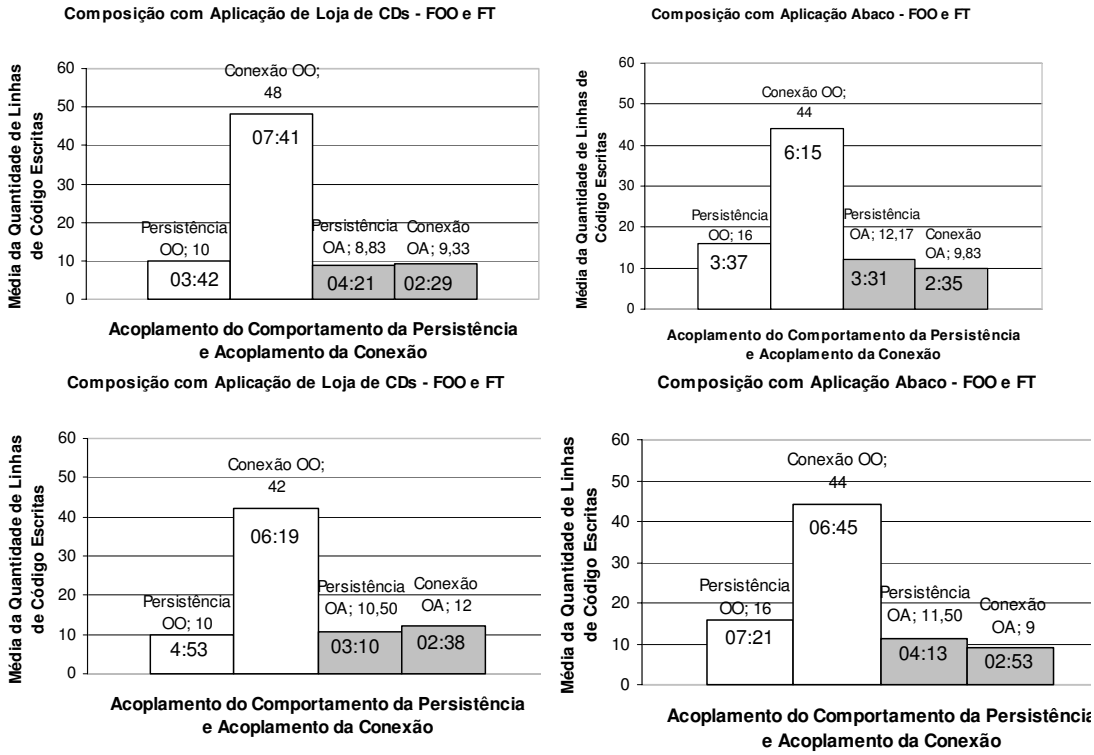


Figura 9 – Comparação do Dado LC

Os resultados da primeira e segunda fase do estudo mostraram uma pequena diferença no tempo de composição a favor do FT, cerca de 4 minutos em média em ambas

as fases, com pode ser observado na Tabela 8. Embora seja uma diferença pequena, o tamanho das aplicações tem influência direta sobre esse número. A composição do framework OO foi em média 30% mais demorada do que com o FT tanto na primeira fase quanto na segunda.

Para muitas organizações essa diferença pode ser insignificante, principalmente para aquelas que trabalham com aplicações de pequeno porte. Além disso, para essas organizações, os recursos investidos no treinamento em uma nova tecnologia pode inviabilizar a adoção de uma nova técnica de programação. Contudo, observa-se que, como essa diferença de tempo está ligada com a quantidade de locais modificados do código, aplicações de grande porte podem se beneficiar com a adoção de um FT, embora novos experimentos devam ser feitos para apresentar dados quantitativos mais sólidos.

Considerando apenas o passo de acoplamento da conexão a diferença percentual é bem maior. Por exemplo, na primeira fase do estudo, a média do tempo de acoplamento da conexão usando o framework OO foi de 6:58, enquanto que com o FT foi de 2:32, uma diferença de 63,64%, um número bastante significativo em se tratando de tempo. Um número relativamente similar é encontrado na segunda fase: 57,91%. Com essa última análise pode-se inferir que determinados interesses transversais que afetam um grande número de locais do código, como ocorre com a conexão, possuem uma diferença significativa no tempo de composição.

6 – Ameaças à Validade do Estudo

Um ponto que pode influenciar os resultados é a forma de implementação dos frameworks. Apesar dos frameworks utilizados no estudo seguirem basicamente a mesma estrutura da maioria dos outros frameworks pesquisados na literatura, pode ser que haja alguma implementação orientada a objetos que diminua o trabalho de acoplamento, e conseqüentemente o tempo total de acoplamento.

O fato de apenas a etapa de composição ter sido submetida à análise não influencia os resultados, pois geralmente a etapa de instanciação de um framework, seja OO ou OA, é feita com mecanismos orientados a objetos. Com base nos resultados obtidos, infere-se que mesmo se a etapa de instanciação de algum FT utilizasse mecanismos orientados a aspectos, o tempo diminuiria, o que não inviabilizaria os resultados. Entretanto, experimentos formais precisam ser conduzidos também nesse sentido.

Outro ponto que pode ter influenciado é a utilização de alunos de pós-graduação como participantes do estudo. Contudo, procurou-se não influenciar os alunos demonstrando expectativas a favor ou contra algum dos frameworks.

7 - Trabalhos Relacionados

Uma série de pesquisas já foi realizada no sentido de mostrar os benefícios de manutenção, legibilidade e reúso que o uso da POA pode trazer para o interesse de persistência [Soares et al., 2002], [Couto et al., 2005], [Rashid and Chitchyan, 2003]. Entretanto, esses trabalhos não apresentam dados quantitativos e experimentos que evidenciem esses benefícios. Além disso, também não comentam se o tempo de composição com o código-base é afetado pelo uso da POA.

Foram encontrados poucos estudos quantitativos envolvendo programação orientada a aspectos em geral. Garcia e outros (2006) fizeram um estudo que procurou averiguar os prós e contras do uso da POA na implementação de padrões de projeto. O trabalho desses autores complementa o trabalho qualitativo anterior de Hanemman e Kiczales (2002) e é baseado nas métricas de coesão, acoplamento e separação de

interesses. Outros estudos experimentais também foram realizados anteriormente, como, por exemplo, o estudo realizado por Kersten e Murphy (1999), que apresenta regras e políticas utilizadas para alcançarem seus objetivos de manutenibilidade, e o trabalho de Zhao e Xu (2004), que apresenta certas métricas que podem ser utilizadas para medir a coesão de um programa orientado a aspectos.

8. Conclusão e Trabalhos Futuros

O contínuo interesse de pesquisadores em frameworks de persistência [Soares et al., 2002], [Couto et al., 2005], [Rashid and Chitchyan, 2003] e a quantidade de frameworks distribuídos gratuitamente e amplamente utilizados [Hibernate, 2006], [OJB, 2006], [Cayenne, 2006] mostram a importância desse tema para a área de engenharia de software. O estudo apresentado por este artigo complementa pesquisas realizadas anteriormente que mostraram que o uso da POA traz benefícios de reúso e manutenção na implementação do interesse de persistência.

Embora o estudo tenha sido realizado com frameworks, os resultados mostram indícios de que o tempo de composição de outros interesses implementados com aspectos, desde que sejam transversais, é menor do que de um interesse implementado com orientação a objetos. Isso se dá pela quantidade de linhas de código que devem ser escritas quando a implementação do interesse se espalha pelos módulos do sistema.

Após a realização do estudo concluiu-se que há diferença no tempo de composição quando se utiliza orientação a aspectos. Em conseqüência da quantidade de linhas de código que deve ser escrita para realizar a composição, a orientação a aspectos é beneficiada pelo mecanismo de quantificação, em que vários pontos de acoplamento são informados em poucas linhas.

Note-se que os dados apresentados por este artigo só são válidos para frameworks caixa branca, em que a composição é realizada com acesso ao código-fonte do framework. Quando há ferramentas que auxiliam no processo de reúso, o tempo de composição passa a ser algo irrelevante, pois será realizado rapidamente independentemente se a implementação é feita com orientação a aspectos ou não [Couto et al., 2005]. Contudo, a maior parte dos frameworks que tratam de um interesse encontrado na literatura não possui apoio automatizado.

A decisão por utilizar um FT desenvolvido pelo próprio grupo de pesquisa se deu pelo acesso livre ao código-fonte, e pelo conhecimento total das funcionalidades disponíveis no framework. Além disso, o acesso ao código-fonte desse FT facilitou o desenvolvimento do framework OO de persistência, já que grande parte do código foi reaproveitada. O desenvolvimento do framework OO com funcionalidades idênticas faz com que o nível de confiabilidade do estudo de caso aumente. Não faria sentido comparar o FT que já estava disponível com o framework OO de persistência *Hibernate*, por exemplo, pois ambos possuem arquiteturas diferentes, com funcionalidades e técnicas de instanciação diferentes. Isso poderia acarretar diferenças favorecendo ou prejudicando o tempo de composição de um dos frameworks.

Como trabalhos futuros, pretende-se replicar o experimento para aplicações maiores, com o objetivo de averiguar se o tempo de composição continua aumentando conforme o tamanho da aplicação. Além disso, pretende-se também realizar o mesmo tipo de experimento com outros frameworks, por exemplo, controle de acesso, autenticação e frameworks que tratam de requisitos funcionais.

Espera-se que o pacote de experimentação, que está disponível em www.icmc.usp.br/~valter, seja utilizado por outros pesquisadores e/ou profissionais em novos experimentos que utilizam aplicações de grande e médio porte.

Referências Bibliográficas

Camargo, V.V., Masiero, P.C. (2005) "Frameworks Orientados a Aspectos". In: anais do 19º Simpósio Brasileiro de Engenharia de Software (SBES'2005), Uberlândia-MG, Brasil, outubro.

Cayenne. (2006). http://www.objectstyle.org/cayenne/ (último acesso: 04 de abril de 2006)

Constantinides, C.A., Bader, A, Fayad, M.F. (2000) "Designing an Aspect-Oriented Framework in an Object-Oriented Environment". ACM Computing Surveys, v.32.

Couto, C.F.M., Valente, M.T.O., Bigonha, R.S. (2005) "Um Arcabouço Orientado por Aspectos para Implementação Automatizada de Persistência". In: anais do 2º. Workshop Brasileiro de Desenvolvimento de Software Orientado a Aspectos (WASP'05), evento satélite do XIX SBES, Uberlândia, MG, Brasil, outubro.

Garcia, A., Santanna, C., Figueiredo E., Kulesza, U., Lucena, C., Staa, A. (2006) Modularizing Design Patterns with Aspects: A Quantitative Study. Transactions On Aspect-Oriented Software Development I, Series: Lecture Notes in Computer Science, Vol. 3880, p. 36-74.

Hanenberg, S., Hirschfeld, R., Unland, R., Kawamura, K. (2004) "Applying Aspect-Oriented Composition to Framework Development – A Case Study". In: proc of 1st International Workshop on Foundations of Unanticipated Software Evolution, Barcelona, Spain, march 28.

Hibernate. (2006) www.hibernate.org (último acesso em 4 de abril de 2006).

Hannemann, J., Kiczales, G. (2002) "Design Pattern Implementation in Java and AspectJ". In: proc. of OOPSLA'02 (November), 161-173.

Kersten, A., Murphy, G. (1999) Atlas: "A Case Study in Building a Web-based learning environment using aspect-oriented programming". In: proc. of OOPSLA'99, November.

Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, C.; Lopes, C.; Loingtier, J.; Irving, J. (1997) "Aspect Oriented Programming". In: proceedings of ECOOP. pp. 220-242.

OJB. (2006) http://db.apache.org/ojb/. (último acesso em 4 de abril de 2006).

Pinto, M., Fuentes, L., Fayad, M.E, Troya, J.M. (2002) "Separation of Coordination in a Dynamic Aspect Oriented Framework". In: proc. of the 1st International Conference on Aspect-Oriented Software Development, April.

Rashid, A., Chitchyan, R. (2003) "Persistence as an Aspect". In: proc. of 2nd International C. on Aspect Oriented Software Development(AOSD) Boston–USA, March.

Soares, S., Laureano, E., Borba, P. (2002) "Implementing Distribution and Persistence Aspects with AspectJ". In: proc. of ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications.

Vanhaute, B., Win, B., Decker, B. (2001) "Building Frameworks in AspectJ". In: European Conference on Object-Oriented Programming (ECOOP), Separation of Concerns Workshop. pp. 1-6, June.

Wohlin, C.; Runeson, P.; Höst, M.; Regnell, B.; Wesslén, A. Experimentation in Software Engineering: an Introduction. Kluwer Academic Publishers, 2000.

Zhao, J., Xu, B. (2004) "Measuring Aspect Cohesion". In: Proc of the Conference on Fundamental Approaches to Software Engineering (FASE'2004), LNCS 2984, Springer, Barcelona, Spain, March 29-31, 54-68.

Aspect-oriented Code Generation

**Marcelo Victora Hecht, Eduardo Kessler Piveta,
Marcelo Soares Pimenta, R. Tom Price**

Instituto de Informática
Universidade Federal do Rio Grande do Sul
Av. Bento Gonçalves, 9500 - Campus do Vale - Bloco IV
Bairro Agronomia - Porto Alegre - RS -Brasil
CEP 91501-970 Caixa Postal: 15064

{mvhecht, epiveta, mpimenta, tomprice}@inf.ufrgs.br

Abstract. *The maturing of aspect-oriented software modeling approaches provides support for the automatic generation of aspect-oriented code. In this paper we describe several means for automatic code generation from Theme/UML models, and discuss some difficulties involved in this process.*

Resumo. *O amadurecimento das abordagens de modelagem de software orientado a aspectos fornece subsídios para a geração automática de código a partir de modelos UML. Neste artigo são descritos diversos mecanismos para a geração de código a partir de modelos Theme/UML, e são discutidas algumas dificuldades envolvidas neste processo.*

1. Introduction

High-level programming languages and Code generation are among the oldest and most widespread techniques to accelerate the transformation process between the design of software and its implementation in executable code. Programming languages facilitate the expression of the solution for a given problem, reducing the amount of code needed and increasing the proximity between the software and the artifacts whose behavior it tries to represent. Code generation, by its turn, reduces the amount of code that has to be written, eliminating the necessity of manual labor, in addition to reduce the chance of accidental programming errors and simplifying the propagation of changes in design.

In the context of programming languages, one of the more recent advances is *aspect-oriented programming* [Kiczales, *et al.* 1997]. It allows an explicit separation of concerns that affect several parts of a software system, like security, persistence and tracing. In traditional analysis and design methods, those concerns end up scattered throughout a system and tangled with its functional requirements – this is called *crosscutting*. Aspect-oriented software development allows those concerns to be modeled and implemented independently from each other and from the main concerns of the system, improving several software quality factors such as extensibility, maintainability and reusability.

In automatic code generation, extensive use has been made of standard system modeling languages as UML [OMG UML 2004], allowing software specifications that are detailed, understandable and with less ambiguities. Currently, a major topic of research within code generation is MDA (Model-Driven Architecture) [OMG MDA 2001], a normalization of the code generation concept using UML models, as well as other elements associated to them, such as MOF [OMG MOF 2006], the metamodel on which UML itself is defined, and XMI [OMG XMI 2005], the standard for interchange of UML data using XML documents. The goal of MDA is to reduce as much as possible the cognitive distance between a system's design and its final implementation.

The use of aspect-oriented programming generated a demand for software design techniques that support its characteristics. One of the proposals for this is the Theme/UML approach [Clarke 2001, Clarke and Baniassad 2004] using *Composition Patterns.* It has been continuously developed since 1998 [Rashid, *et al.* 2005], aiming at managing the separation of concerns earlier in the development cycle. Initially focused on *subject-oriented programming* [Clarke, *et al.* 1999], it evolved to become the most general proposal currently available, and at the same time compatible with several aspect-oriented programming languages [Clarke and Walker 2001, 2002].

The existence of this type of model, and of its computer-understandable versions (through XML-based representations), opens up the path to several possibilities, such as quality metrics analysis, *bad smell* detection, refactoring, aspect extraction from traditional object-oriented programs, and automatic aspect-oriented code generation.

The goal of this paper is to describe a way to generate code in the AspectJ language [Kiczales, *et al.* 2001], the most extensively used aspect-oriented programming language today, from UML models extended according to Theme/UML [Clarke 2001]. To this purpose, we developed a XML representation of Theme/UML models, to serve a the input for a code generator programmed using XSLT (Extensible Stylesheet Language Transformations) [XSLT 2.0 2005].

The paper has been divided as follows: in Section 2, we revise the requirements to build a code generator, as well as the state of the art in this area, including MDA; in Section 3, we briefly investigate possible intersections between code generation and aspect-oriented programming; in section 4, we present a case study of the implementation of a aspect-oriented code generator based on Theme/UML; Section 5 cites related work in the area of code generation and aspect-oriented programming; and Section 6 contains our conclusions.

2. Code Generation

Code generation is the process of transforming high level artifacts – closer to the problem domain – in lower-level ones – nearer to the solution's architecture [Krueger 1992]. Although the term is used in various contexts, in this paper we will assume that it refers to the automatic conversion of a specification, in the form of high level models (like UML), into code (high level or binary), circumventing costly and error-prone cycles of manual programming.

2.1. Requirements for code generation

A code generator has some prerequisites. A model of the desired system has to be provided and

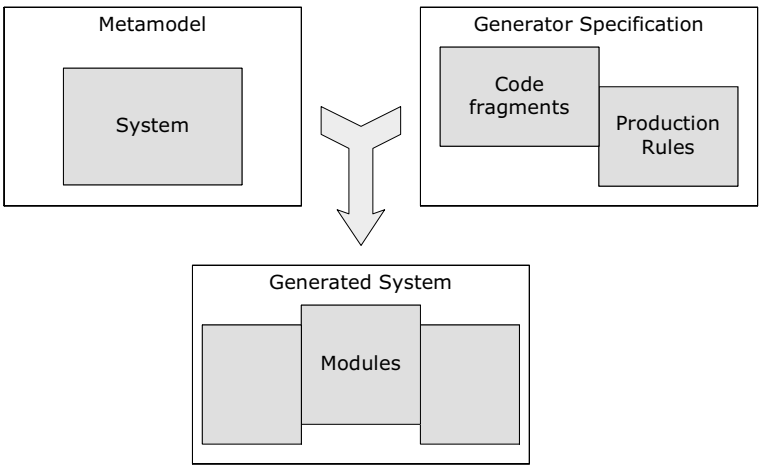


Figure 1. Typical structure of a code generator.

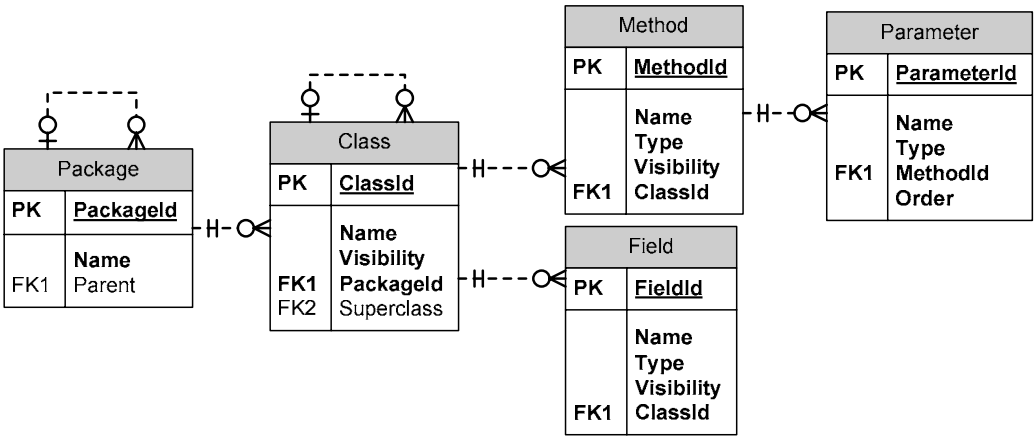


Figure 2. Metamodel for a simplified Java class model.

described within a metamodel that is sufficiently detailed and free of ambiguities to provide the structure to be generated. Also necessary is a specification of the elements of the target domain that will be composed to shape the final product, and rules, based on the contents of the metamodel, specifying how this composition is accomplished (see Figure 1).

The metamodel is a data model able to store other system models in a way that a computer can understand. It has to be specific to the generator's purpose: a metamodel used by an interface generator stores information about forms, fields, and user messages, while a metamodel for object-oriented software contains specifications of packages, classes, properties and methods. In Figure 2 we describe a metamodel regarding a simplified Java class model.

A code generator specification can be divided in two main pieces: the *code fragments* and the *production rules*. Code fragments are usually pieces of programs, in the generator's target language, that will be assembled to generate the source code that will be the output of the generator. They may have sections that are replaced with information contained in the metamodel, such as element names. Production rules define how the structure described in the metamodel will be converted to the structure of the composed code fragments in the output files. It usually contains alternative, iterative and recursive elements.

Simpler code generators, usually developed for use by a small team to solve a specific problem, such as creating object-relational mapping classes, generally don't make a clear distinction between the two parts. Code fragments are strings embedded in statements of an all-purpose programming language that shape the production rules. More mature generators divide the two parts more clearly, and frequently provide a form o programming devised specifically for the definition of production rules.

Many modeling tools commercially available have integrated code generators ([Rational Software 2005, ArgoUML 2005, Poseidon 2005]). They usually generate code according to a fixed structure, or allow limited modification. There are also several tools available that are designed specifically to build code generators ([codegen 2005, Velocity 2005, autogen 2005, CodeSmith 2005]). Practically all of them combine a script language for the programming of the production rules, and code fragments either embedded in the script or in separate files. One alternative, with the increasing popularity of the XML metalanguage for data interchange, is its use both to declare code fragments and to store the metamodel, and the use of XSLT [XSLT 2.0 2005], a programming language specialized in the transformation of XML documents, to define production rules [Dodds 2005].

2.2. Difficulties in Code Generation

The adoption of code generators can be hard, despite the gains this technology can convey. The first difficulty is economical: for something to be reused, it first must be developed, and sometimes this expense can't be justified [Biddle, *et al.* 2003]. Other common demands for to any kind of software, like the necessity of a detailed documentation, and concern for changes that can cause collateral effects with interacting systems, become especially important with code generators [Krueger 1992]. To maximize the gain offered by the generator, it is important to educate as many users as possible about its capabilities. And a badly planned modification in a code generator might impact the entire code base that depends on it.

Another problem comes up when the standard for the generated artifacts need to be modified, because changes made to those after the generation may be lost. Code generators can be divided as *passive* – creating each artifact only once – or *active* – designed to generate the same artifact many times according to changes in the model or in the generator [CGN 2005]. Passive generators create the initial code for the software, but can't be used for maintenance of already modified artifacts, and thus don't have the benefit of propagating changes in the model or in the generated standard to an existing system. Active generators allow the same code module to be generated any number of times, by the use of features like protected code sections that are copied from one version to the next.

Finally, often users of a generator face the need to use newer versions of a generator, in order to use new features, but to simultaneously keep in place older versions, to keep the compatibility with existing artifacts. Therefore, code generators are yet another dimension where maintainability problems can take place.

In section 3 we suggest ways by which AOP can help with some of those issues.

2.3. Model Driven Architecture

The advent of UML as a *de facto* standard for object-oriented software specification and modeling made it also the metamodel of choice to feed code generators. Because of that, the OMG developed the MDA (*Model Driven Architecture*) [OMG MDA 2005, Miller and Mukerji 2003] standard, with the intention of offering an vendor- and technology-independent platform for application development. According to OMG, development using MDA focuses in the functionality and behavior of a system, undistorted by the idiosyncrasies of the technological platform where it will be used. MDA attempts to separate implementation details from the business requisites, and consequently it is not necessary to repeat the whole process of modeling application requisites when there is a technology change.

MDA is based on several OMG specifications: the UML language is used to specify the system; the MOF (MetaObject Facility [OMG MDA 2005]) is used to store the UML model; and XMI (XML Metadata Interchange [OMG XMI 2005]) is used for the communication between the metamodel and the code generator tools.

3. Applying aspect-oriented concepts to code generation

Application generators are one of the most powerful reuse techniques available, able to output thousands of lines of code from a relatively small input set. On the other hand, they are also one of the least adaptable techniques, each one being designed to a specific type of output. High-level programming languages are also a form of reuse, in the sense that they allow more functions to be included in a system with less lines of code [Biggerstaff 1999].

It is appealing then to investigate how aspect-oriented programming can be combined to automatic code generators in order to combine the benefits of both technologies. This intersection may happen in two ways: aspect-oriented programming can be used in the

development of code generators, and a code generator can have aspect-oriented code as its target. While our focus in this paper is on the latter approach, in this section we will briefly investigate the former.

Initially, the advantages that AOP brings to the development of code generators are the same that it adds to any system: simplified development, ease in maintenance, increased understandability, etc.

Some ideas from aspect-orientation may be especially useful for the creation of code generators. It can, for example, be used to parameterize a generator, so that it can generate different outputs, according to the aspects selected to be weaved with it. This device can be used even to maintain several versions of the generator active.

Aspects may also be used to simplify the problem of evolving code generators, if the modification of the generated artifacts is implemented through advices and intertype declarations. That would allow the base artifacts to be generated again at will without breaking the modifications (unless join points in the base artifact were modified). For that to be possible, the generator itself could be unaware of AOP, as long as the programming language of the generated artifacts contemplates concern separation, or has some extension for that purpose. As an example, classes created by a Java code generator can be modified by aspects using AspectJ.

Biggerstaff [Biggerstaff 1998] draws a parallel between code generation and aspect-oriented programming, stating that code generation is a form of weaving code fragments. He also suggests that aspect-oriented techniques may increase the power of component libraries, granting the ability to change its components and multiplying their capabilities.

4. Implementing an automatic generator for Aspect-oriented code

In this section, we describe our efforts to implement an automatic code generator capable of transforming an aspect-oriented model into aspect-oriented code. We do not propose, at this time, to contemplate the contents of methods and advices, although, with sufficiently detailed interaction diagrams, it might be possible.

4.1. Challenges in generating aspect-oriented code

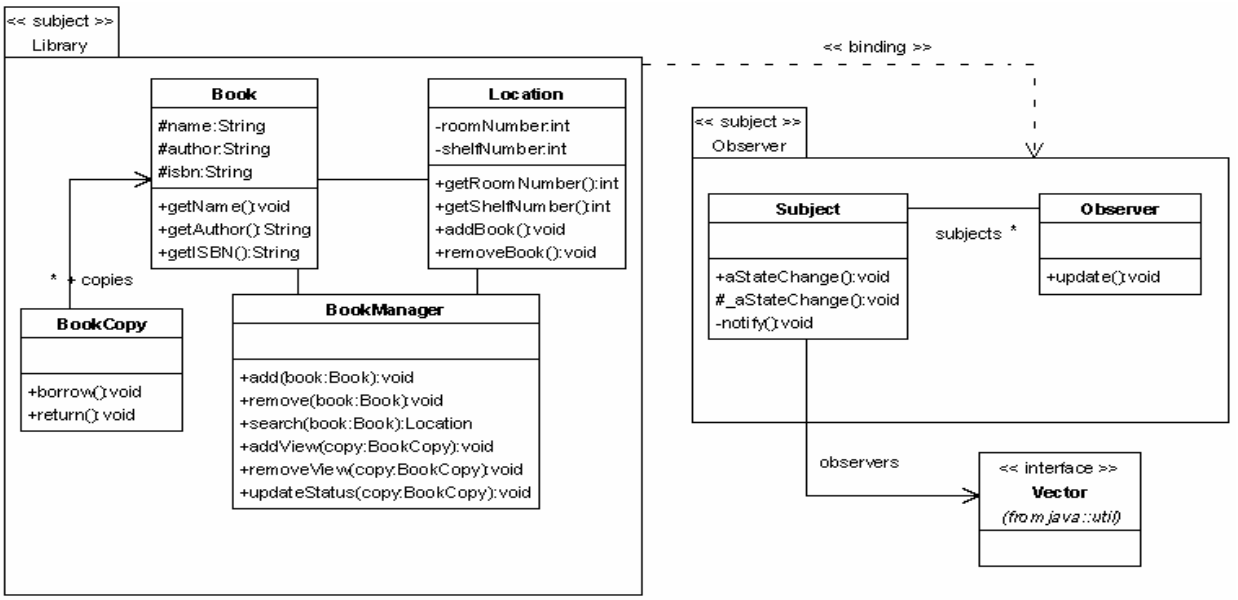


Figure 3. UML model of the example system (without Theme/UML extensions)

```
package library;

import java.util.Vector;

public abstract aspect Observer {
  interface ISubject {}

  interface IObserver {
    public void update();
  }

  Vector ISubject.observers;

  private void
    ISubject.notify_() {}

  abstract pointcut
    aStateChange(ISubject s);
  after(ISubject s):
    aStateChange(s) {
    s.notify_();
  }
}
```

```
package library;

public aspect LibraryObserver
    extends Observer {
  declare parents: BookCopy
    implements ISubject;
  declare parents: BookManager
    implements IObserver;

  public void BookManager.update() {
    this.updateStatus();
  }

  pointcut aStateChange(ISubject copy):
    target(copy)
    && args(..)
    && (execution
       (* BookCopy.borrow(..))
     || execution
       (* BookCopy.returnIt(..)));
}
```

Figure 4. Listing of AspectJ code for the *Observer* composition pattern and *LibraryObserver* binding (manually indented for better understanding).

A problem in generating aspect-oriented code is the lack of a standard modeling form that explicitly contemplates separation of concerns. Most of the proposed methods ([Groher and Baumgarth 2004, Chavez and Lucena 2002, Stein, *et al.* 2002, Basch and Sanchez 2003], [Pawlak, *et al.* 2002]) provide, as evidence of its validity, the correspondence between the models they define and an implementation in AspectJ. However, none has the required level of detail for automatic code generation, relying on human interpretation of the models to create code.

The paper that presents the most detailed implementation is [Clarke and Walker 2002], where the authors try to demonstrate that the Theme/UML approach [Clarke and Baniassad 2004], based on the *composition patterns* approach, is compatible with several implementations of aspect-oriented programming, including AspectJ. Even so, their approach is very complex, and the mapping between it and AspectJ is not trivial.

We developed a library maintenance system similar to the one used as an example in [Clarke and Walker 2001, 2002]. The base system controls the location of books, as well as loans and returns. Over that, the Observer design pattern [Gamma, *et al.* 1994] is applied as a crosscutting concern, for the book manager class to be aware when a copy is lent or returned (Figure 3). On aspect weaving, the *aStateChange* advice of *Subject* is to be associated with the join points corresponding to the *borrow()* and *return()* methods from class *BookCopy*, while the *update()* method from *Observer* is to be introduced to class *BookManager*. The use of aspects to implement design patterns is explored in more depth in [Garcia, *et al.* 2005] and [Hannemann and Kiczales 2002].

The target code for the system is presented in Figure 4. For reasons of reusability and evolvability described in [Clarke and Walker 2002], each CP in the model becomes an abstract aspect with abstract pointcuts, and the binding between the CP and the base packages turns into an aspect that makes those pointcuts concrete. This process is detailed in section 4.4. The code depicted is the output of the code generator; the only hand-made modification made was in the formatting and indentation for presentation purposes.

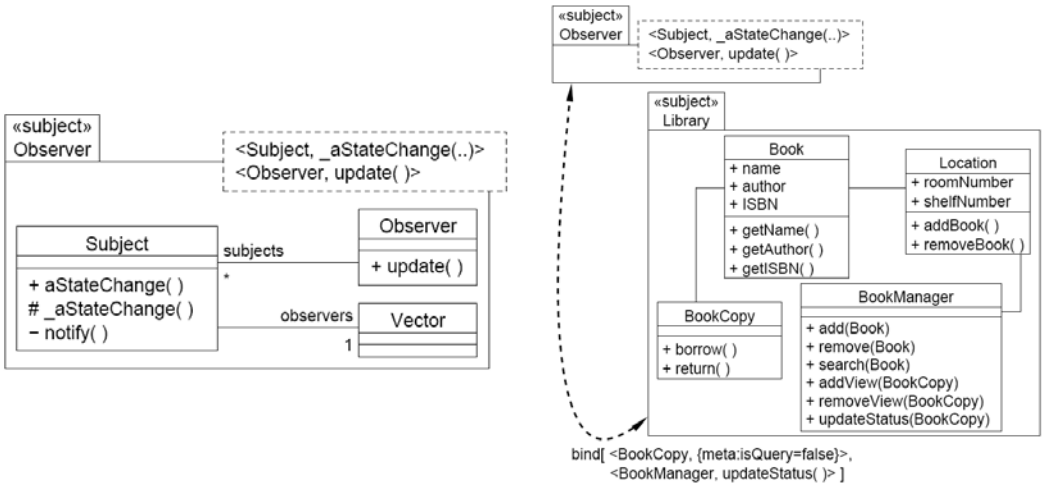


Figure 5.Class diagram using *Composition Patterns*

4.2. The Theme/UML approach [Clarke and Baniassad 2004]

This approach is divided in two parts: Theme/Doc, for requisite engineering, and Theme/UML, for system design, both considering separation of concerns in every level. Our interest here is in the second one.

The construction added by Theme to UML is the *composition pattern* (in short, CP), a definition of how to integrate artifacts (classes or operations) from two different packages. They work in a manner similar to UML *templates*, that allow elements in a model not to be fully defined, but to have *parameters* so that specific parts are replaced later by concrete elements of the model.

As used in Theme/UML, template parameter substitution can be used both to indicate dynamic crosscutting, where advices are connected to join points in the system through pointcuts, and to produce static crosscutting, where the very structure of the base system is modified [Clarke and Walker 2002], as can be seen in Figure 5. Advices are always represented by a pair of operations. The one with the name prefixed by an underscore represents the execution of the base method. Representing those operations on an interaction diagram, it is possible to define *before*, *after* and *around* advices. In the example displayed in Figure 6, the *aStateChange()* operation (the advice) executes the *_aStateChange()* operation firstly, and as a result represents an advice of type *after*.

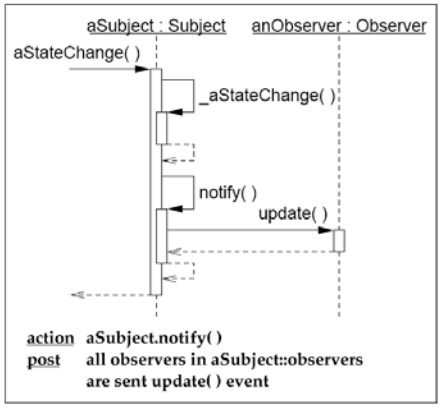


Figure 6. Sequence diagram depicting the interaction between a *composition pattern* and a modified class

```
<UML:Package xmi.id="c28" name="Observer" visibility="public"
isSpecification="false" isRoot="false" isLeaf="false" isAbstract="false">
  <UML:ModelElement.stereotype>
    <UML:Stereotype xmi.idref="d6a"/>
  </UML:ModelElement.stereotype>
  <UML:Namespace.ownedElement>
    <Theme:CompositionPatternParameters xmi.id="03a9">
      <UML:Operation xmi.id="8e09" xmi.idref="d4a"/>
      <UML:Operation xmi.id="8e1c" xmi.idref="d2a"/>
    </Theme:CompositionPatternParameters>
```

Figure 7. Section of a XMI document defining parameters for a package representing an Composition Pattern

4.3. Metamodel

It was necessary to define a computer-understandable format to store the class model so that it could be analyzed by the code generator system. Taking in consideration the predominance of the UML and XML metalanguages, respectively on the areas of system modeling and information exchange, it was natural to choose XMI, which combines both technologies, as the input of the generator. But there is no definition of the precise format a UML model should be stored, and no common schema for XMI documents. Instead, OMG defines a set of rules that vendors have to follow to define their own schemas. In this work, we started from the XMI generated by the Poseidon tool [Poseidon 2005], and added the necessary information to specify composition patterns – template parameters associated to packages, and parameter binding to relationships between packages.

Basically, two structures had to be added to the original XMI file. An element, labeled `CompositionPatternParameters`, is used inside a package (Figure 7) to indicate elements that can be bound to actual classes during composition to a base design – in this case, the operations *Subject._aStateChange()*, that represents an aspect, and *Observer.update()*, that is going to be

```
<UML:Dependency xmi.id="c26" isSpecification="false">
  <UML:ModelElement.stereotype>
    <UML:Stereotype xmi.idref="c24"/>
  </UML:ModelElement.stereotype>
  <UML:Dependency.client>
    <UML:Package xmi.idref="d6c"/>
  </UML:Dependency.client>
  <UML:Dependency.supplier>
    <UML:Package xmi.idref="c28"/>
  </UML:Dependency.supplier>
  <Theme:Bindings>
    <Theme:Bind>
      <UML:Class xmi.idref="e10" />
      <Theme:BoundParameter xmi.idref="8e09" />
      <UML:Constraint xmi.id="d88" name="" isSpecification="false">
        <UML:Constraint.body>
          <UML:BooleanExpression xmi.id="d8a" body="meta:isQuery=false"/>
        </UML:Constraint.body>
      </UML:Constraint>
    </Theme:Bind>
    <Theme:Bind>
      <Theme:BoundElement xmi.type="uml:Operation" xmi.idref="d74" />
      <Theme:BoundParameter xmi.idref="8e1c" />
    </Theme:Bind>
  </Theme:Bindings>
</UML:Dependency>
```

Figure 8. Section of a XMI document extending a dependency between packages with parameter binding.

Input: A composition pattern *CP*.
Output: An abstract aspect *A* corresponding to *CP*.

- Declare abstract aspect *A*.
- For each pattern class $PClass_i$ in *CP*:
  - Declare interface I_i in *A*.
  - For each template operation with no supplementary behaviour, $\langle op \rangle_{i,j}$, on $PClass_i$, declare corresponding (abstract) method $m_{i,j}$ on I_i.
  - For each template operation with supplementary behaviour[2], $\langle \_op \rangle_{i,k}$, on $PClass_i$:
    - Declare corresponding abstract pointcut $pc_{i,k}$ with formal parameters consisting of one to capture target object (of type I_i) on which $\langle \_op \rangle_{i,k}$ is called plus one corresponding to each of the specified formal parameters on $\langle \_op \rangle_{i,k}$.
    - Declare advice on $pc_{i,k}$ according to the supplementary behaviour.
  - For each non-template operation $op_{i,l}$ on $PClass_i$, introduce method $m_{i,l}$ (which implements any behavioural specifications for $op_{i,l}$) on I_i.
  - For each non-template association from $PClass_i$ to some non-pattern class, introduce field on I_i.
- For each non-pattern class in *CP*, implement it directly if not already available.

Figure 9. Algorithm for mapping certain constructs in an uncomposed CP to AspectJ [Clarke and Walker 2002].

introduced on a base class. Another element, called `Bindings` (Figure 8), is associated to a dependency relationship between the base package and the Composition Pattern, and indicates how the composition is to proceed.

4.4. Generator and Production Rules

The generator itself was implemented using XSLT (Extensible Stylesheet Language Transformations), version 2.0 [XSLT 2.0 2005]. This language was selected because it is a standard language, with extensive tool support, and its focus on transforming XML documents

```
<xsl:stylesheet version="2.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns:xs="http://www.w3.org/2001/XMLSchema"
   xmlns:fn="http://www.w3.org/2005/02/xpath-functions"
   xmlns:xdt="http://www.w3.org/2005/02/xpath-datatypes"
   xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:UML="org.omg.xmi.namespace.UML"
   xmlns:Theme="org.my.Theme">
 <xsl:import href="Theme2AspectJAspect.xslt"/>
 <xsl:import href="Theme2AspectJBinding.xslt"/>
 <xsl:import href="Theme2AspectJClass.xslt"/>

 <xsl:output method="text"/>
 <xsl:output method="text" name="textFormat"/>

 <xsl:template match="/">
   <xsl:apply-templates select="/XMI/XMI.content/UML:Model
/UML:Namespace.ownedElement//UML:Package/UML:Namespace.ownedElement
/Theme:CompositionPatternParameters/../.." mode="T2AJAspect"/>
   <xsl:apply-templates
select="/XMI/XMI.content/UML:Model/UML:Namespace.ownedElement
//UML:Dependency/Theme:Bindings/Theme:Bind" mode="T2AJBinding"/>
   <xsl:apply-templates
select="/XMI/XMI.content/UML:Model/UML:Namespace.ownedElement//UML:Package
[count(UML:Namespace.ownedElement/Theme:CompositionPatternParameters)=0]"
mode="T2AJClass"/>
 </xsl:template>
</xsl:stylesheet>
```

Figure 10. Section of XSLT transformation selecting the types of elements to be generated.

makes it very suitable for code generation based on XMI models [Marchal 2004].

It is important to emphasize, however, that the use of general-purpose XSLT processors for code generation is best suited for the creation of *passive* generators (section 2.2), because those processors have no support for integrating generated code with existing artifacts. If it is required that changes made to generated code are kept on within new generation cycles, other alternatives should be considered. Besides, the paradigms used on XSLT programming, based on tree traversals, are somewhat different from what programmers of imperative languages are used to. With few exceptions, XSLT processors also lack features like step-by-step execution and variable state evaluation, which would facilitate debugging in case the generator output does not correspond to the desired one.

The algorithm used to generate the files is similar to the one presented in [Clarke and Walker 2002] and reproduced in Figure 9.

As seen in the algorithm, each Composition Pattern becomes an *abstract aspect*, and the binding of the aspect to a class becomes a concrete aspect that realizes the abstract pointcuts defined in its parent. This promotes the reusability of the aspects.

Due to space constraints, it is impossible to reproduce the full listing for the XSLT files used in generation. We chose to represent here the main generator file, which distributes the appropriate elements among the three other files, which in turn are used to generate Aspects, Classes, and Bindings (Figure 10). Since in Theme Composition Patterns are just like any other UML Class Diagram package, except for the fact that they have parameters that can be bound to elements in other packages, the deciding factor for generating classes or aspects is the presence

```
<xsl:template match="UML:Package" mode="T2AJAspect">
   <xsl:variable name="uri" select="@name"/>
   <xsl:result-document href="{$uri}.aj" format="textFormat"><xsl:call-
template name="lowercase">
       <xsl:with-param name="name" select="@name"/>
     </xsl:call-template>;
<xsl:value-of select="@visibility"/> aspect <xsl:call-template
name="capitalize">
       <xsl:with-param name="name" select="@name"/>
     </xsl:call-template> {
     <xsl:apply-templates select="UML:Namespace.ownedElement/UML:Class"
mode="Interface"/>
}<!---->
   </xsl:result-document>
 </xsl:template>

 <xsl:template match="UML:Class" mode="T2AJAspect Interface">
 interface I<xsl:value-of select="@name"/> {
   <xsl:apply-templates select="UML:Classifier.feature/UML:Operation"/>
 }
 </xsl:template>

 <xsl:template match="UML:Operation" mode="T2AJAspect Interface">
   <xsl:if test="substring(@name, 1, 1) != '_' and
not(exists(../UML:Operation[@name = concat('_',
current()/@name)]))"><xsl:text>
   </xsl:text><xsl:call-template name="methodSig"/>;<!---->
   </xsl:if>

 </xsl:template>
```

Figure 11. Section of XSLT transformation converting packages from a XMI document into AspectJ aspects.

of an element called `CompositionPatternParameters` within the package.

We also reproduce a section of XSLT that generates abstract aspects from the definition of Composition Pattern packages (Figure 11). In this section, the packages received from the main file are processed according to the rules specified by the algorithm (Figure 9), generating an abstract aspect with interfaces for each class contained in the Composition Pattern and introducing methods for every operation that does not designate an advice.

5. Related works

In [Amaya, *et al.* 2005], an approach to deal with aspect oriented modeling in MDA is presented. In this approach, typical crosscutting concerns are considered as different perspectives of the system modeled using UML. This separation will keep from CIM to PSM. They also propose to model requirements using use cases and design them using composition patterns. In our approach, we do not deal with requirements, but the design is done using Theme/UML artifacts instead of subject-oriented design in UML. Theme/UML was designed with experiences in modeling subject-oriented (SO) systems, improving the former subject-oriented modeling techniques.

The Libra approach [Chaves 2004] describes potential benefits of mixing Model Driven Software Development and Aspect-Oriented Software Development. The authors propose an approach for combining them based on a new dynamic join point model for UML action semantics. Our approach could be adapted to work with the action semantics described by the authors. The main benefit of the Libra approach is the possibility to use of behavioral specification to describe aspect oriented software.

[Kulesza, *et al.* 2005] presents a generative approach for the development of multi-agent systems (MAS). The approach explores the MAS domain to enable the code generation of heterogeneous agent architectures. Aspect-oriented techniques are used to allow the modeling of crosscutting agent features. Although they provide a domain specific language (DSL) and an aspect-oriented architecture for agent systems, the relation to our work is associated to the code generation techniques. It maps abstractions of Agent-DSL to the depicted architecture. This is useful for agent-oriented systems, but not abstract enough to deal with other types of systems without adaptation. Our approach uses XMI to generate AO code, regardless of the system domain and reference architecture.

[Kulesza and Lucena 2004] provide a preliminary version of a method to develop aspect-oriented generative approaches, including a description of required adaptations of the domain engineering method to accommodate the use of aspect-oriented technologies. They describe adaptation to domain analysis, domain design and domain implementation. Our work focuses on the MDA view of code generation. Instead of using domain modeling, we focus on software modeling using UML models.

[González, *et al.* 2005] discuss principles to make an aspect-oriented analysis in software development through the identification and analysis of the dependencies model between system properties (in the context of MDA), but it does not include any attempt to implement those principles.

Finally, CAM/DAOP [Pinto, *et al.* 2005] seeks to combine component-based software engineering and aspect-oriented concepts with the use of an XML-based Architecture Description Language (ADL), called DAOP-ADL, which contemplates aspect composition. There is some similarity between the information conveyed by this ADL and by our XMI-based metamodel: they both attempt to represent aspect composition in a computer-understandable format. However, their approach is based in a proprietary platform for dynamic aspect weaving, while we base our efforts in existing modeling and programming languages.

6. Conclusion and Future Work

Aspect-oriented design and development and code generation are two technologies with a great potential to increase development productivity and software quality.

However, as we have shown, a few obstacles exist in the merging of the two technologies. The degree of productivity and quality gain from a code generator is directly dependent of the degree of quality of the metamodel it is based on, and of the preciseness of its production rules. This means that to create a generator for aspect-oriented code, we need to have a method of expressing separation of concerns in a software model in a way that is compatible with an aspect-oriented programming language.

We chose Theme/UML as the basis for our metamodel because it is a flexible and general modeling language. This is important because our interest is not only in the generation of aspect-oriented code, but also in use this model for code refactoring [Fowler 1999], detection of *bad smells* [Piveta, *et al.* 2005], and extraction of aspects from existing object-oriented systems [Monteiro and Fernandes 2005] (Figure 12). The basic structures of AspectJ – aspects, advices, pointcuts, join points, and introductions – can all be represented in Theme/UML. One of the tasks that our group is currently working on is to determine how each feature of AspectJ can be modeled in Theme/UML, and vice-versa.

We also chose to design the production rules so as to convert from Theme/UML directly to AspectJ code. The alternative approach would be to use a software model specific to the structure of AspectJ code, and to define a transformation in two stages: from Theme/UML to the AspectJ/UML model, and from this to AspectJ code. This would have the advantage of simplifying greatly the production rules at each step, making the code clearer and more maintainable. But the extra steps involved might be in itself a maintainability problem, especially if changes in AspectJ demand a modification of the intermediate model.

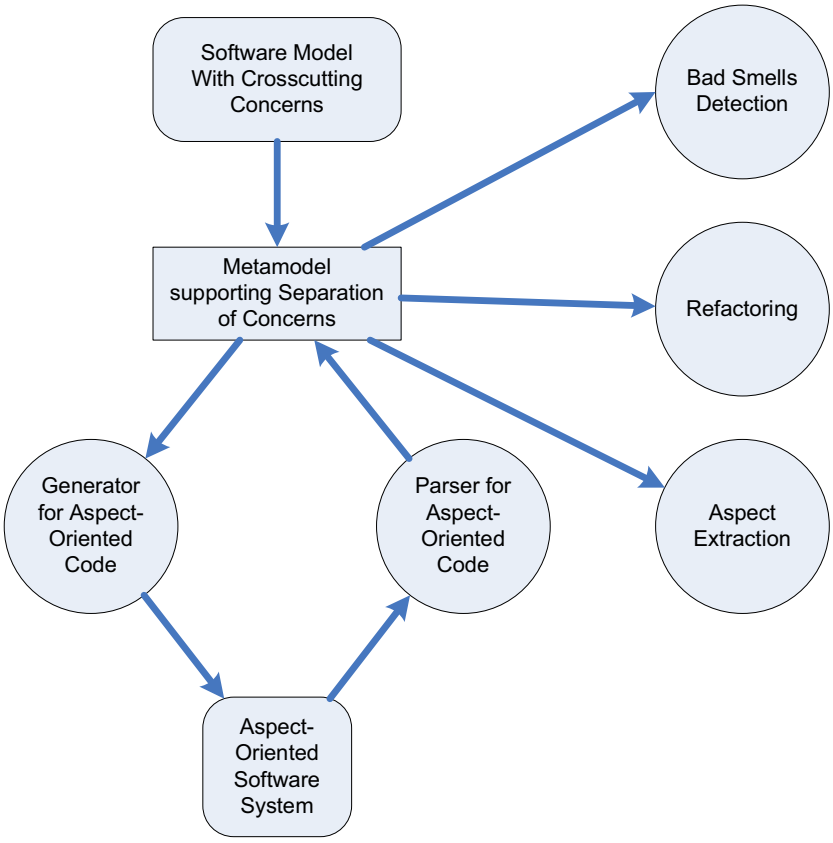


Figure 12. Activities based on a metamodel supporting separation of concerns

References

Amaya, P., Gonzalez, C. and Murillo, J. M. (2005), "Towards a Subject-Oriented Model-Driven Framework," in *Aspect-Based and Model-Based Separation of Concerns in Software Systems*.

ArgoUML (2005), http://argouml.tigris.org/, accessed on 2005-10.

Autogen (2005), http://www.gnu.org/software/autogen/, accessed on 2005-09.

Basch, M. and Sanchez, A. (2003), "Incorporating Aspects into the UML," in *Third International Workshop on Aspect Oriented Modeling*.

Biddle, R., Martin, A. and Noble, J. (2003), "No Name: Just notes on software reuse," in *ACM SIGPLAN Notices*, vol. 38, pp. 76-96.

Biggerstaff, T. J. (1998), "A perspective of generative reuse," in *Annals of Software Engineering*, vol. 5, pp. 169-226.

Biggerstaff, T. J. (1999), "Reuse technologies and their niches," in *Proceedings of the 21st international conference on Software engineering*. Los Angeles, California, United States: IEEE Computer Society Press.

Chaves, R. (2004), "Aspects and MDA: Creating aspect-based executable models," Master Thesis. Florianópolis.

Chavez, C. and Lucena, C. (2002), "A Metamodel for Aspect-Oriented Modeling," in *Workshop on Aspect-Oriented Modeling with UML (AOSD-2002)*.

Clarke, S. (2001), "Composition of Object-Oriented Software Design Models," PhD thesis, Dublin City University, Dublin, Ireland.

Clarke, S. and Baniassad, E. (2004), "Theme: An Approach for Aspect-Oriented Analysis and Design," in *Proceedings of the International Conference on Software Engineering 2004*.

Clarke, S., Harrison, W., Ossher, H. and Tarr, P. (1999), "Subject-oriented design: towards improved alignment of requirements, design, and code," in *ACM SIGPLAN Notices*, vol. 34, pp. 325-339.

Clarke, S. and Walker, R. J. (2001), "Separating Crosscutting Concerns across the Lifecycle: From Composition Patterns to AspectJ and Hyper/J," in *Technical Report TCD-CS-2001-15 and UBC-CS-TR-2001-05*, Trinity College, Dublin and University of British Columbia.

Clarke, S. and Walker, R. J. (2002), "Towards a standard design language for AOSD," in *Proceedings of the 1st international conference on Aspect-oriented software development*. Enschede, The Netherlands: ACM Press.

CGN (2005), Code Generation Network, http://www.codegeneration.net/, accessed on 2005-09.

Dodds, L. (2005), "Code generation using XSLT", https://www6.software.ibm.com/developerworks/education/x-codexslt/x-codexslt-3-1.html, accessed on 2005-12.

Codegen (2005), http://forge.novell.com/modules/xfmod/project/?codegen, accessed on 2005-09.

CodeSmith (2005), http://www.ericjsmith.net/codesmith/, accessed on 2005-09.

Fowler, M. (1999), "Refactoring: improving the design of existing code," Addison-Wesley Longman.

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1994), "Design Patterns," Addison-Wesley.

Garcia, A., Sant'Anna, C., Figueiredo, E., Kulesza, U., Lucena, C. and Staa, A. v. (2005), "Modularizing Design Patterns with Aspects: A Quantitative Study," in *Proc. 4rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2005)*.

González, C., Murillo, J. M. and Amaya, P. A. (2005), "Aspect-Oriented Analysis: A MDA Based Approach," in *Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design*.

Groher, I. and Baumgarth, T. (2004), "Aspect-Orientation from Design to Code," in *Early Aspects: Aspect-Oriented Requirements Engineering and Architecture Design*.

Hannemann, J. and Kiczales, G. (2002), "Design pattern implementation in Java and AspectJ," in *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications*.

Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J. and Griswold, W. G. (2001), "An overview of AspectJ," in *Proceedings of the 15th Europeen Conference on Object-Oriented Programming (ECOOP 2001)*.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. and Irwin, J. (1997), "Aspect-Oriented Programming," in *Proceedings of the 11th Europeen Conference on Object-Oriented Programming (ECOOP-97)*.

Krueger, C. W. (1992), "Software Reuse," in *ACM Computing Surveys*, vol. 24, p. 131-183.

Kulesza, U., Garcia, A. F., Lucena, C. J. P. d. and Alencar, P. S. C. (2005), "A Generative Approach for Multi-agent System Development," in *SELMAS 2004 - Software Engineering for Multi-Agent Systems III*.

Kulesza, U. and Lucena, A. G. C. (2004), "Towards a Method for the Development of Aspect-Oriented Generative Approaches," in *Early Aspects 2004: Aspect-Oriented Requirements Engineering and Architecture Design Workshop at OOPSLA 2004*.

Marchal, B. (2004), "UML, XMI, and code generation", http://www-128.ibm.com/developerworks/xml/library/x-wxxm23/.

Miller, J. and Mukerji, J. (2003), "MDA Guide Version 1.0.1." http://www.omg.org/docs/omg/03-06-01.pdf.

OMG MDA (2005), Model Driven Architecture, http://www.omg.org/mda/, accessed on 2005-11.

OMG MOF (2006), MetaObject Facility Specification, version 1.4, http://www.omg.org/mof/, accessed on 2006-02.

Monteiro, M. and Fernandes, J. (2005), "Towards a Catalog of Aspect-Oriented Refactorings," in *Proc. 4rd Int' Conf. on Aspect-Oriented Software Development (AOSD-2005)*.

Pawlak, R., Duchien, L., Florin, G., Legond-Aubry, F., Seinturier, L. and Martelli, L. (2002), "A UML Notation for Aspect-Oriented Software Design," in *Workshop on Aspect-Oriented Modeling with UML (AOSD-2002)*.

Pinto, M., Fuentes, L. and Troya, J. M. (2005), "A Component and Aspect Dynamic Platform", in *The Computer Journal* 48(4):401-420.

Piveta, E. K., Hecht, M., Pimenta, M. S. and Price, R. T. (2005), "Bad Smells em Sistemas Orientados a Aspectos," in *Simpósio Brasileiro de Engenharia de Software (SBES)*, Uberlandia - Brazil.

Poseidon (2005), http://gentleware.com/, accessed on 10/2005.

Rashid, A., Chitchyan, R., Sawyer, P., Garcia, A., Alarcon, M. P., Bakker, J., Tekinerdogan, B., Clarke, S. and Jackson, A. (2005), "Survey of Aspect-Oriented Analysis and Design Approaches," in *AOSD-Europe 2005-05*.

Rational Software (2005), http://www-306.ibm.com/software/rational/, accessed on 2005-11.

Stein, D., Hanenberg, S. and Unland, R. (2002), "An UML-based Aspect-Oriented Design Notation," in Proc. 1st Int' Conf. on Aspect-Oriented Software Development (AOSD-2002).

OMG UML (2004), Unified Modeling Language Specification, version 2.0, http://www.omg.org/technology/documents/formal/uml.htm, accessed on 2005-12-15.

Velocity (2005), http://jakarta.apache.org/velocity/, accessed on 2005-09.

OMG XMI (2005), XML Metadata Interchange Mapping Specification, version 2.1, http://www.omg.org/technology/documents/formal/xmi.htm, accessed on 2006-01.

XSLT 2.0 (2005), Extensible Stylesheet Language Transformations version 2.0, http://www.w3.org/TR/xslt20/, accessed on 2005-12.

Uma Investigação de Modelos de Estimativas de Esforço em Gerenciamento de Projeto de Software

Iris Fabiana de Barcelos Tronto, José Demísio Simões da Silva, Nilson Sant'Anna

Laboratório Associado de Computação e Matemática Aplicada – Instituto Nacional de Pesquisas Espaciais (INPE)
Caixa Postal 515 – 12.201-970 – São José dos Campos – SP – Brazil

{iris_barcelos,demisio,nilson}@lac.inpe.br

Abstract. *Accurate estimation in software project management is critical. Fundamental measurements are size, effort, resources, cost, and time spent in the software development process. Underestimates can lead to time pressures that compromise full functional development and thorough testing of software. Likewise, overestimates can result in noncompetitive bids. In this paper, predictive artificial neural network and stepwise regression based models are investigated, with the goal of offering an alternative for those who do not believe in estimation models. The results presented in this paper compare the performance of both methods and indicate that these techniques are competitive with the methods of APF, SLIM, and COCOMO.*

Resumo. *Estimativas precisas em gerenciamento de projetos é um fator crítico. Medidas de tamanho, esforço, recursos, custo, e tempo despendidos no desenvolvimento de software são fundamentais. Valores subestimados de esforço podem fazer com que pressões de tempo comprometam todo o desenvolvimento funcional e até mesmo o teste de software. Por outro lado, valores superestimados podem resultar em projetos não competitivos. Neste artigo, modelos como redes neurais e regressão, são apontados como alternativas para aqueles que não acreditam em modelos de estimativas. Os resultados apresentados comparam o desempenho desses métodos e indicam que estas técnicas são competitivas com os métodos APF, SLIM e COCOMO.*

1. Introdução

O desenvolvimento contínuo das áreas de hardware e de software e o fenômeno da globalização da economia tem contribuído para o aumento da competitividade entre empresas produtoras e prestadoras de serviços de software. Assim, há uma crescente necessidade de se produzir software com maior qualidade, em tempo hábil e com baixo custo.

Um nível de qualidade e produtividade internacional pode ser conseguido através da gestão efetiva de seus processos de software, focalizando pessoas, produto, processo e projeto. Em relação a projeto, é preciso que haja planejamento e acompanhamento através de um conjunto de atividades, dentre as quais as estimativas podem ser consideradas fundamentais, pelo fato de fornecerem um guia para as demais atividades. As estimativas estão relacionadas ao esforço, aos recursos e ao tempo de desenvolvimento, necessários para se construir um software [Agarval 2001].

É amplamente aceito o fato de que as estimativas de tamanho de software são importantes para se determinar o esforço de projeto de software [Jones 1986], [Lai and Huang 2003], [Hasting and Sajeev 2001], [Briand and Wieczorek 2002]. Porém, de acordo com última pesquisa relatada pelo Ministério da Ciência e Tecnologia, no Brasil, em 2001 apenas 29% das empresas realizam estimativas de tamanho e 45,7% realizavam estimativa de esforço de software [MCT 2001]. Não há um estudo específico que identifique as causas do baixo índice de realização de estimativas de esforço, mas o nível de confiabilidade dos modelos pode ser uma das possíveis causas. Diante destes dados apresentados pelo MCT é evidente a necessidade de se ter uma abordagem alternativa para realizar estimativas de esforço, através da qual possa ter estimativas confiáveis com modelos de implementação simples.

Tendo em vista que realizar estimativas consistentes e precisas é um grande desafio para os gerentes de projeto, muitas pesquisas têm sido conduzidas a fim de construir, avaliar e recomendar técnicas de predição [Boehm 1981], [Albrecht and Gaffney 1983], [Shepperd and Schofield 1997], [Zhong et al. 2004], [Sentas et al. 2005], [Bisio and Malabocchia 1995], [Myrtveit et al. 2005], [Samson et al. 1997]. Essas técnicas se enquadram em três categorias gerais [Shepperd et al. 1996]

1) Julgamento de Especialistas – está técnica tem sido amplamente utilizada. Entretanto, os meios de se derivar uma estimativa não são explícitos e conseqüentemente não repetíveis. A opinião do especialista, embora seja sempre difícil de se quantificar, ela pode ser uma ferramenta de estimativa efetiva como um fator de ajuste para modelos algorítmicos [Gray et al. 1999].

2) Modelos Algorítmicos – estes modelos são os mais populares na literatura e precisam ser calibrados ou ajustados a circunstâncias locais.. Eles tentam representar o relacionamento entre esforço e uma ou mais características. Geralmente, o principal direcionador de custo utilizado em tais modelos é aquele referente ao tamanho do software (por exemplo, o número de linhas de código fonte, o número de Pontos por Função, o número de Pontos por Caso de Uso, o número de páginas, dentre outros). Dentre os modelos algorítmicos tem-se o Modelo COCOMO [Boehm 1981] e o Modelo SLIM [Putnam,1978].

3) Aprendizagem de Máquina – Na última década, as técnicas de aprendizagem de máquina têm sido utilizadas como um complemento ou uma alternativa para as duas categorias anteriores. Exemplos incluem modelos de lógica nebulosa [Kumar et al. 1994], árvores de regressão [Selby and Porter 1998], redes neurais artificiais [Srinivasan and Fisher, 1995] e raciocínio baseado em casos [Shepperd et al. 1996]. Um resumo útil destas técnicas é apresentado em [Gray e MacDonell 1997].

Nos últimos anos, têm sido realizados vários estudos comparativos utilizando essas três categorias de técnicas de estimativa, baseando-se no poder de predição [Gray and MacDonell 1997], [Briand et al. 2000], [Jeffery et al. 2001], [Shepperd et al 1996] [Angelis and Stamelos 2000], [Finnie et al. 1997]. Porém, como os conjuntos de dados empregados têm diferentes características (*outliers,* colinearidade, número de variáveis, número de observações, dentre outras), diferentes medidas de precisão e diferentes projetos comparativos, ainda não se chegou a uma convergência e há a dúvida sobre qual técnica ou quais técnicas se deve utilizar.

Existem vários fatores que podem e devem ser considerados na seleção de uma técnica de predição. A seleção de técnica deve ser dirigida pela necessidade e capacidade organizacional. Em termos de necessidade, o objetivo mais comum é maximizar a precisão da estimativa, dentre outras. Por exemplo, uma técnica que produz estimativas ligeiramente menos precisas pode ser preferida em detrimento de modelos mais robustos, em organizações que não tenham acesso a uma calibração local ou um conjunto de dados bem comportado. Em termos de capacidade organizacional, algumas técnicas de modelagem são mais complexas que outras e requerem experiência significativa para que elas sejam utilizadas efetivamente. Há benefícios de se empregar técnicas mais sofisticadas (potencialmente mais útil) para construir modelos de estimativas, porém isso somente fornecerá benefícios reais se as técnicas forem usadas apropriadamente.

Em um trabalho anterior [Barcelos Tronto et al. 2006], foi realizado um estudo comparativo dos resultados de estimativa de esforço utilizando as técnicas de análise de regressão e redes neurais. O esforço foi estimado baseado em uma única variável independente: tamanho de software - que é representado por uma variável numérica, denominada KDSI, do conjunto de dados COCOMO. Os resultados mostraram que o modelo de redes neurais rendeu melhores resultados que regressão, com estimativas mais precisas. Diferentes medidas de erro têm sido utilizadas por vários pesquisadores, mas neste projeto a principal medida para avaliar a precisão do modelo é a magnitude média do erro relativo – *Mean Magnitude of Relative Error* – MMRE. MMRE é a percentagem média de erros absolutos:

$$MMRE = \frac{\left(\sum_{i=1}^{n} \left| \frac{M_{est} - M_{act}}{M_{act}} \right| * 100 \right)}{n} \qquad (2)$$

em que n é o número de projetos; M_{act} é o esforço real observado; e M_{est} é o esforço estimado. Neste artigo, o objetivo é investigar como estas técnicas se comportam quando são utilizadas outras variáveis independentes, que possuem características diferentes (por exemplo, variáveis categóricas).

Para atingir este objetivo, foi realizado um estudo para contrastar os resultados obtidos utilizando uma rede neural do tipo perceptron de múltiplas camadas com o agoritmo de aprendizagem por retro-propagação do erro, com os resultados obtidos através de uma regressão múltipla *stepwise*, sobre o conjunto de dados COCOMO [Boehm 1981]. Este artigo está estruturado em cinco seções. A Seção 2 apresenta um pequeno resumo de estudos empíricos relacionados à estimativa de software. Uma breve descrição sobre redes neurais e modelos de regressão é apresentada nas seções 3 e 4, respectivamente. A Seção 5 apresenta um caso de estudo, em que são mostrados os modelos de regressão e de redes neurais gerados e uma análise de confiabilidade deles. Finalmente, na Seção 6 é apresentada uma discussão da significância dos resultados obtidos e sobre perspectivas de trabalhos futuros.

2. Modelos de Estimativa Aplicados à Engenharia de Software

Estimativas precisas e consistentes de recursos necessários para o desenvolvimento de um projeto de software são componentes determinantes no gerenciamento efetivo de

projetos de software. Apesar da extensiva pesquisa realizada nos últimos vinte anos, a comunidade de software ainda é significativamente desafiada quando se trata de estimativas efetivas de recursos. De uma forma geral, grandes esforços têm sido investidos no desenvolvimento de técnicas para remover ou reduzir a subjetividade no processo de estimativa de custo e de esforço de software. Exemplos destes trabalhos incluem modelos paramétricos e baseados em regressão, como Análise de Pontos por Função [Albrecht 1979], o modelo COCOMO [Boehm 1981; Boehm et al. 2000] e o modelo de Regressão Ordinal [Sentas et al. 2005].

Porém, outras técnicas para a análise exploratória de dados, como *clustering,* raciocínio baseado em casos e redes neurais artificiais têm se mostrado eficiente para realizar predições. Zhong et al (2004) descreve o uso de *clustering* para realizar estimativas de qualidade de software. Uma abordagem baseada em casos, chamada ESTOR, foi desenvolvida para estimativa de esforço de software [Vicinanza et al. 1990]. Vicinanza et al. mostraram que *ESTOR* é comparável a um especialista e significativamente mais preciso que o modelo para estimativas COCOMO ou Pontos por Função, sobre amostras restritas de problemas. Há pesquisas importantes que utilizam redes neurais na tentativa de produzir estimativas mais precisas de recursos [Gray and MacDonnell, 1999],[Witting e Finnie, 1997].

Em [Karunanithi et al. 1992] redes neurais são utilizadas para predizer confiabilidade de software, sendo que os experimentos são realizados com redes de Jordan e com um algoritmo de aprendizagem de correlação cascata.

Outro estudo importante, realizado por Samson et al. (1997) utiliza uma arquitetura *perceptron* de múltiplas camadas de forma a estimar o esforço de software. Eles utilizam o conjunto de dados COCOMO implementado por Boehm. O trabalho compara regressão linear, com uma única variável dependente, com uma abordagem de redes neurais. Mas, as duas abordagens parecem executar pobremente com um MMRE de 520,7% e 428,1%, respectivamente.

Srinivasan e Fisher (1994) também relatam o uso de uma rede neural com um algoritmo de retro propagação do erro. Eles encontraram um MMRE = 70% resultante da aplicação da rede neural, o que significa um bom resultado quando comparado com outros modelos como COCOMO, SLIM e Pontos por Função. Entretanto, eles não deixam claro como o conjunto de dados foi dividido para treinar e validar o modelo. Aparentemente existe uma inconsistência no cálculo do MMRE para o modelo gerado.

Khoshgoftaar et al (2000) apresenta um estudo de caso realizado sobre dados de projetos de software de tempo real para estimar a testabilidade de cada módulo a partir de medidas estáticas de código fonte. Eles consideram redes neurais uma técnica promissora para construir modelos de predição, porque elas são capazes de modelar relacionamentos não lineares.

Finalmente, nos últimos anos, pôde-se perceber que o interesse pela utilização de redes neurais tem aumentado, em geral para projetar soluções para problemas de estimativas, classificação, controle, dentre outros. Redes Neurais têm sido aplicadas, com sucesso, em vários domínios de problemas, em áreas tais como medicina, engenharia, geologia e física. Elas podem ser usadas como modelos de predição porque são técnicas de modelagem capazes de modelar funções complexas.

Neste trabalho, a metodologia de redes neurais é utilizada com o objetivo de predizer o esforço de desenvolvimento de software (em homens-hora) a partir do tamanho do projeto (dado pela quantidade de linhas de código fonte) e de outros atributos direcionadores de custo e de esforço. Foi realizada uma análise comparativa entre o modelo de regressão e o modelo de redes neurais calibrados e testados neste estudo.

3. Redes Neurais

Redes Neurais artificiais – RNA, são sistemas paralelos inspirados na arquitetura de redes neurais biológicas, que compreende algumas unidades interconectadas (neurônios artificiais). O neurônio computa uma soma pesada de suas entradas e gera uma saída, se a soma excede um certo limite. Esta saída, então, torna uma entrada estimulante (positiva) ou inibitória (negativa) para outros neurônios na rede. O processo continua até que uma ou mais saídas sejam geradas.

A Figura 1 mostra um neurônio artificial que computa a soma pesada de suas *n* entradas e gera uma saída de *y*. A rede neural é o resultado de um arranjo de tais unidades em camadas, que são interconectadas umas as outras. As arquiteturas resultantes resolvem problemas através da aprendizagem das características dos dados disponíveis relacionados ao problema. Existem muitos algoritmos diferentes de aprendizagem. Os algoritmos *feed-forward multilayer perceptrons* são os mais comumente usados em RNA, embora existam redes neurais mais sofisticadas. Na maioria das vezes, as arquiteturas de múltiplas camadas são treinadas com o algoritmo de aprendizagem por retro-propagação do erro, que requer uma função de ativação diferençável.

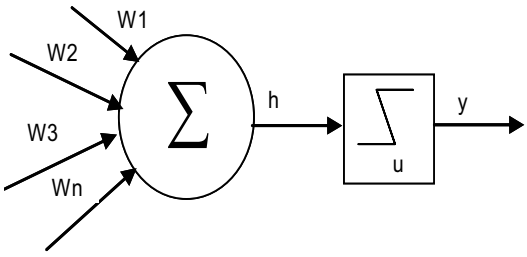


Figure 1. Um neurônio de McCulloch e Pitts

A rede é iniciada com pesos randômicos e, gradualmente, aprende os relacionamentos implícitos em um conjunto de dados de treinamento através do ajuste de seus pesos. Dentre os vários algoritmos de treinamento disponíveis, o de aprendizagem por retro-propagação do erro é o mais usado pelos pesquisadores de métricas de software.

Em geral, a maioria dos estudos relacionados com o uso de RNA para predizer esforço de desenvolvimento de software tem como principal objetivo comparar a precisão dos modelos. Por exemplo, Witting e Finnie (1997) exploram o uso de uma rede neural de múltiplas camadas sobre os conjuntos de dados de Desharnais e da Associação de Métricas de Software Australiana ASMA. Para o conjunto de dados de Desharnais, eles dividiram os projetos três vezes, randomicamente, em conjuntos de teste (contendo 10 projetos) e de treinamento (contendo 71 projetos). Os resultados obtidos dos três conjuntos de validação são agregados e mostram um alto nível de

precisão (MRE – *Magnitude of Relative Error* – para o conjunto de dados de Desharmais = 27% e o MRE para ASMA = 17%). Porém, alguns valores que representam possíveis *outliers* foram excluídos.

Entretanto, outros fatores também precisam ser investigados, por exemplo, o valor exploratório é igualmente importante. Neste trabalho, além de se analisar a precisão das predições, também se investiga a adequação da abordagem para construir sistemas de predição de esforço de software.

4. Regressão

Regressão tenta encontrar um relacionamento entre um ou mais parâmetros de predição (variáveis independentes) e uma variável dependente, minimizando o quadrado dos erros para a gama de eventos do conjunto de dados. Alguns pesquisadores têm defendido a construção de modelos locais simples, por exemplo, Kok et al.(1990), usa este tipo de abordagem. A filosofia é, essencialmente, resolver problemas de predição locais antes de tentar construir modelos universais. O sistema de predição resultante tem a forma:

$$Y_{ext} = \beta_0 + \beta_1 X_1,...,\beta_n X_n \quad (1)$$

Onde $\acute{Y}$ é o valor estimado e X_1, …,X_n são variáveis independentes, por exemplo, o tamanho do projeto (dado como a quantidade de linhas de código), o tamanho da base de dados, a capacidade do analista, a confiabilidade requerida do software. De acordo com os resultados encontrados, a variável que representa o tamanho do projeto influencia significativamente a estimativa de esforço.

A desvantagem com esta técnica é sua vulnerabilidade para valores extremos (*outliers)*, embora tenham sido utilizadas com sucesso, técnicas de regressão, que são menos sensíveis a tais problemas [Briand and Wieczorek 2002]. Outro problema é o impacto de co-linearidade – a tendência de variáveis independentes estarem fortemente correlacionadas umas com as outras – sobre a estabilidade de um sistema de predição do tipo regressão. Por exemplo, neste trabalho, verifica-se que a precisão das estimativas pode estar longe do desejado quando se aplica uma regressão linear múltipla sobre as variáveis categóricas do conjunto de dados COCOMO, sem um pré-processamento específico para as variáveis categóricas.

5. Dados, Experimentos e Resultados

A análise empírica abordada neste estudo, foi realizada sobre uma base de dados de projetos de software que tem sido utilizado em estudos anteriores na literatura de engenharia de software [Sentas et al. 2005], [Srinivasan e Fisher 1995], [Samson et al. 1997]. Este conjunto de dados foi selecionado porque ele está disponível para uso público e foi utilizado para descrever e testar um dos mais importantes métodos na área, que é o modelo COCOMO [Boehm,1981], [Boehm et al. 2000]. O conjunto de dados consiste de 63 projetos, incluindo instâncias de projetos de software de negócio, científico e de sistema, escritos em uma variedade de linguagens de programação que incluem COBOL, PLI, HMI e FORTRAN. As variáveis multiplicadoras de esforço de desenvolvimento de software (e uma breve descrição delas) consideradas neste estudo são apresentadas na Tabela 1. KDSI é uma variável do tipo numérico contínuo enquanto

que as demais são variáveis categóricas em que cada categoria é representada por um valor numérico.

Tabela 1. Multiplicadores de esforço de desenvolvimento de software

Variável	Descrição
RELY	Confiabilidade requerida do software
DATA	Tamanho da base de dados
CPLX	Complexidade do produto
TIME	Restrição de tempo de execução
STOR	Restrição de armazenamento principal
VIRT	Volatilidade da máquina virtual
TURN	Tempo de execução da máquina
ACAP	Capacidade do analista
AEXP	Experiência com aplicações
PCAP	Capacidade do programador
VEXP	Experiência com máquina virtual
LEXP	Experiência com linguagem de programação
MODP	Uso de práticas modernas de programação
TOOL	Uso de ferramentas de software
SCED	Cronograma de desenvolvimento requerido
RVOL	Volatilidade dos requisitos
KDSI	Número de linhas de código ajustado

Todos os 63 projetos foram utilizados na análise realizada. A variável dependente, considerada neste estudo, é ACT MM (quantidade de esforço total, em número de homens /hora, despendido no desenvolvimento do projeto). De uma maneira geral, para cada um dos experimentos os dados históricos são divididos em amostras usadas para treinar o sistema de aprendizagem e amostras separadas para testar a precisão do classificador treinado para estimar o esforço de desenvolvimento.

Estimativas de precisão das predições obtidas a partir de um conjunto de dados de treinamento são sempre otimistas. De forma a investigar a precisão dos modelos de redes neurais e de regressão múltipla construídos como parte deste estudo, utilizou-se o mesmo procedimento que em [Kitchenham 1998]. Kitchenham desenvolveu um procedimento simples para investigar a precisão das predições realizadas pelo seu modelo sobre a base de dados COCOMO. Baseando-se neste processo, omitiu-se um subconjunto de projetos (o conjunto de dados de teste), depois se desenvolveu um modelo com os projetos restantes (o conjunto de dados de treinamento) e finalmente avaliou-se a precisão das predições do modelo sobre o conjunto de dados de teste. Desta forma, foram criados seis diferentes pares de conjuntos de treinamento e de teste. Cada conjunto de treinamento foi construído removendo-se todo sexto projeto, a começar (da primeira vez) do primeiro projeto. Assim, o conjunto de dados de aprendizagem 1 foi

construído removendo-se os projetos: 1, 7, 13, 19, 25, 31, 37, 43, 49, 55 e 61; o conjunto de dados de aprendizagem 2 foi construído removendo-se os projetos: 2, 8, 14, 20, 26, 32, 38, 44, 50, 56, 62 e assim sucessivamente. Cada conjunto de dados removidos representa o respectivo conjunto de dados de teste.

Sabendo-se que todos os 63 projetos da base de dados COCOMO foram utilizados para construir o modelo, dos seis pares de conjuntos criados três conjuntos de dados de aprendizagem contêm 52 projetos e os outros três contêm 53 projetos.

Para um propósito comparativo entre os métodos algorítmicos e os métodos de predição de aprendizagem de máquina, foram conduzidos experimentos utilizando análise de regressão múltipla *stepwise* e redes neurais artificiais, sobre cada um desses conjuntos de dados.

A rede neural foi implementada com 17 entradas (a 18ª variável é o esforço a ser estimado, dado pela variável ACT MM), 5 neurônios na primeira camada, 3 neurônios na segunda camada e um neurônio de saída para estimar o esforço (variável de saída). Os dados foram todos normalizados em relação aos valores máximos que cada variável pode assumir. As variáveis de entrada são listadas na Tabela 1. A fase de treinamento foi repetida várias vezes, em uma busca pela melhor rede para resolver o problema. Por outro lado, várias arquiteturas de redes neurais foram experimentadas e os resultados apresentados neste artigo correspondem à rede neural com o melhor desempenho de generalização. As predições obtidas a partir de RNA (depois do treinamento sobre os seis conjuntos de dados de treinamento) usando os seis conjuntos de teste são mostradas na Tabela 2, sendo que "obs" representa os valores reais e "est" os valores estimados.

Tabela 2. Esforço estimado através de modelos de redes neurais

Conjunto1		Conjunto2		Conjunto3		Conjunto4		Conjunto5		Conjunto6	
obs	est	obs	est	obs	est	obs	est	obs	est	obs	est
2040	401,9	1600	2246,3	243	265,4	240	544,9	33	111,9	43	162,6
8	224,8	1075	858,4	423	258,1	321	249,5	218	165,9	201	135,1
79	254,5	73	269,5	61	399,8	40	466,9	9	165,0	11400	6077,8
6600	10570,3	6400	10543,9	2455	523,2	724	293,3	539	119,0	453	281,9
523	432,8	387	229,2	88	223,6	98	266,5	7,3	118,2	5,9	94,5
1063	448,3	702	3781,9	605	360,7	230	201,0	82	103,7	55	104,3
47	234,0	12	191,3	8	154,3	8	135,5	6	79,5	45	86,0
83	287,1	87	244,0	106	179,2	126	191,4	36	82,5	1272	2864,4
156	293,4	176	301,5	122	384,2	41	226,3	14	95,1	20	101,9
18	254,1	958	378,1	237	212,0	130	492,3	70	95,4	57	145,2
50	237,6	38	181,4	15	147,8						

Vários experimentos foram realizados e seis modelos foram calibrados, utilizando a técnica de regressão múltipla e o método *stepwise forward*, com um nível de significância igual a 0,05. Cada um desses modelos foi treinado sobre seu respectivo

conjunto de dados de treinamento, considerando ACT MM como a variável dependente e tomando-se todas as variáveis mostradas na Tabela 1 como variáveis independentes. A Tabela 3 mostra os modelos de regressão finais encontrados.

Os experimentos foram realizados usando o módulo GRM - General Regression Models, implementado pelo pacote de software *STATISTICA*. Foi implementado o algoritmo *subset model-building* para encontrar o melhor modelo a partir de um número de possíveis modelos. A estatística conhecida como R^2_ajustado, dos subconjuntos (modelos), permitiu comparações diretas e a escolha do "melhor" subconjunto entre os vários subconjuntos encontrados. Para cada conjunto de dados de treinamento escolheu-se o modelo (dentre dez modelos gerados) que tem o maior valor de R^2_ajustado. Conseqüentemente, foram obtidos seis modelos. Apesar dos seis conjuntos de treinamento serem derivados de uma mesma base de dados de projetos (a base de dados COCOMO), os respectivos modelos encontrados, são constituídos de variáveis diferentes entre si.

Todos os modelos encontrados possuem algumas variáveis com um valor de β considerado não significativo. Como o objetivo é obter uma maior precisão de estimativa, aplicou-se uma regressão múltipla *stepwise forward* tomando como variáveis independentes, aquelas contidas no modelo. Esta regressão foi utilizada para cada um dos seis modelos de forma que somente as variáveis com um valor de β significativo fossem consideradas no modelo. A regressão *stepwise forward* constrói um modelo de predição, adicionando-se ao modelo (a cada estágio) a variável com a maior correlação parcial com a variável dependente, considerando-se todas as variáveis presentes no modelo. O objetivo é encontrar um conjunto de variáveis independentes que maximizam a estatística F. F avalia se as variáveis independentes, quando consideradas juntas, são significativamente associadas com a variável dependente. O critério utilizado para adicionar uma variável é se ela aumenta o valor de F para a regressão, por alguma quantidade *k*. Quando uma variável reduz F, também por alguma quantidade *w*, ela é removida do modelo. As variáveis independentes que compõem cada modelo são apresentadas na Tabela 3.

Tabela 3. Modelos de regressão múltipla

Modelo	Equação de Regressão
Modelo 1	ACT MM = -2937,57453363+16,1182817KDSI+2410,01478707*CPLX
Modelo 2	ACT MM =-5558,849858+ 7,980057*KDSI+ 5654,065476*DATA
Modelo 3	ACT MM=-3709,2565+3757,96154*MODP+9,05218371*KDSI
Modelo 4	ACT MM=-8835,5111+5396,35474*DATA+3638,30337*MODP+7,77161528*KDSI
Modelo 5	ACT MM =-3682,0398+3802,92414*MODP+8,92926545*KDSI
Modelo 6	ACT MM=-4074,3481+1394,65051*RELY+2308,47763*VEXP +2535,60316*MODP-2129,0629*TOOL+7,29998110*KDSI

As estimativas dos modelos de regressão múltipla (obtidos da calibração sobre os conjuntos de treinamento) usando os conjuntos de teste são mostradas na Tabela 4.

Tabela 4. Esforço estimado através dos modelos de regressão múltipla

Conjunto1	Conjunto2	Conjunto3	Conjunto4	Conjunto5	Conjunto6
570,8	2986,9	905,4	2293,3	1176,5	-119,4
-416,3	-68,5	-17,9	15,3	299,5	286,4
582,3	-220,1	-254,2	227,9	137,8	2976,2
15042,7	2837,8	1991,6	718,8	790,6	661,3
807,9	478,3	-542,6	1095,7	-202,3	-154,2
278,4	2630,3	72,6	474,4	1149,7	-83,7
-702,5	214,9	-233,4	258,6	-174,1	-240,9
92,5	560,8	338,4	857,9	706,5	2161,0
577,7	286,7	1041,1	812,3	168,2	-373,7
-1149,1	-28,6	560,3	-250,1	-33,9	564,9
285,3	-171,4	-537,2			

Diferentes medidas de erro têm sido utilizadas por vários pesquisadores, mas neste projeto a principal medida para avaliar a precisão do modelo é a magnitude média do erro relativo – *Mean Magnitude of Relative Error* – MMRE. MMRE é a percentagem média de erros absolutos:

$$MMRE = \frac{\left(\sum_{i=1}^{n} \left| \frac{M_{est} - M_{act}}{M_{act}} \right| * 100 \right)}{n} \qquad (2)$$

em que n é o número de projetos; M_{act} é o esforço real observado; e M_{est} é o esforço estimado.

Neste artigo, o MMRE é adotado como indicador de desempenho de predição, uma vez que ele é amplamente utilizado e nos permite comparar os resultados aqui obtidos com aqueles obtidos por outros pesquisadores. A Tabela 5 apresenta os valores de MMRE obtidos através das estimativas realizadas sobre os seis conjuntos de dados de teste, utilizando os modelos de redes neurais artificiais, os modelos de regressão linear simples (somente com uma variável independente, KDSI) e os modelos de regressão múltipla gerados como resultado deste estudo.

Outros pesquisadores têm utilizado o R^2_ajustado ou o coeficiente de determinação para indicar a percentagem de variação na variável dependente que é "explicada" em termos das variáveis independentes. Neste estudo realizou-se uma análise de regressão linear para calibrar as predições realizadas utilizando redes neurais e regressão múltipla. Para tanto se considerou M_{est} como a variável independente e M_{act} como a variável dependente. O valor de R^2 indica a quantidade de variação nos valores reais de esforço, considerada por causa de um relacionamento linear com os valores estimados. Valores de R^2 próximos de 1.0 sugerem um forte relacionamento linear e aqueles próximos de 0.0 sugerem que não há relacionamento.

A Tabela 5 apresenta, além dos valores de MMRE, os valores de R^2 resultantes da regressão linear de M_{est} e M_{act} para os modelos de regressão múltipla, regressão

linear simples e de redes neurais. Também são mostrados os resultados obtidos por Srinivasan e Fisher (1995) utilizando um modelo de rede neural com retro-propagação do erro e os resultados obtidos por Kemerer (1987) com os modelos COCOMO-Básico, Pontos por Função e SLIM. Esses resultados indicam que as predições realizadas utilizando os modelos de redes neurais e de regressão linear simples têm um forte relacionamento linear com os valores de esforço de desenvolvimento reais observados para os projetos usados para teste. Sobre esta dimensão R^2, o desempenho do modelo de redes neurais é menor que o desempenho de SLIM nos experimentos de Kemerer e o de regressão linear simples usando somente a variável KDSI como dependente, mas melhor que os modelos de regressão múltipla *stepwise*. Em termos de MMRE, o modelo de redes neurais executa notavelmente melhor quando comparado com as outras abordagens e com os modelos de regressão múltipla. Regressão múltipla executou pobremente em todos os experimentos, com exceção daquele realizado utilizando o conjunto de dados 2, que é constituído de dados de teste bem comportados e que todos os outros modelos também tiveram seu melhor desempenho.

Para cada um dos experimentos, foram comparados os esforços estimados e os observados. Por exemplo, os resultados das estimativas utilizando a abordagem de regressão múltipla para o conjunto 1 mostram que a magnitude média do erro relativo MMRE é 1372. Enquanto que o valor de MMRE para as estimativas utilizando redes neurais, para este mesmo conjunto de dados de teste é igual a 506. Ao comparar esses resultados confirma-se que para projetos com valores maiores de KDSI o melhor ajuste é fornecido pelo modelo de redes neurais.

Tabela 5. Uma comparação de abordagens de estimativa de esforço

	Conjunto1		**Conjunto2**		**Conjunto3**		**Conjunto4**		**Conjunto5**		**Conjunto6**	
	MMRE	**R^2**	**MMRE**	**R^2**	**MMRE**	**R^2**	**MMRE**	**R^2**	**MMRE**	**R^2**	**MMRE**	**R^2**
Redes Neurais	506	0,89	278	0,89	353	0,46	384	-0,12	559	-0,05	278	0,87
Regressão Múltipla	1372	0,14	354	0,33	865	0,55	843	- 0,12	1526	-0,01	706	0,62
Regressão Simples	335	0,93	231	0,42	358	0,77	288,5	0,70	795	0,87	291	0,49
	MMRE						**R^2**					
APF	103						0.58					
SLIM	772						0.89					
COCOMO básico	610						0.70					

Porém, pode ser visto que alguns dos resultados obtidos com regressão múltipla para projetos menores predizem esforço negativo. Isto torna óbvio que estimativas são todas sujeitas a erros e devem ser interpretadas com cautela. É interessante notar que os resultados de regressão múltipla sobre estes conjuntos de dados executaram mais pobremente quando comparado com modelos anteriores utilizando somente uma única variável dependente: KDSI. Em termos de MMRE, modelos de redes neurais executam notavelmente melhor que a regressão múltipla e levemente melhor que regressão simples. Como pode ser visto na tabela acima, redes neurais executa bem e certamente melhor que a regressão sobre a maioria dos pontos. O ajuste de regressão para projetos

pequenos é pior que aqueles alcançados através de redes neurais. Isto não é surpreendente quando se considera as limitações de regressão linear.

Este experimento ilustra dois pontos. Por um lado, nenhum dos modelos executa particularmente bem para estimativas de esforço de desenvolvimento de software, particularmente sobre a dimensão MMRE. Mas, por outro lado, a abordagem de redes neurais é competitiva com modelos tradicionais. Em geral, embora MMRE seja alto no caso de todos os modelos, um valor alto de R^2 sugere que ao calibrar um modelo de predição em um novo ambiente, o modelo de predição ajustado pode ser usado com segurança. Considerando-se a dimensão R^2, o método de redes neurais fornece um ajuste significativo para os dados.

6. Conclusão

Este artigo compara o método de redes neurais com abordagens tradicionais para estimativa de esforço de software. Vários experimentos foram realizados, aplicando-se redes neurais e análise de regressão múltipla sobre o conjunto de dados COCOMO de Boehm, para estimar o esforço a partir do tamanho do software e de outros atributos multiplicadores de esforço. Os resultados das estimativas de redes neurais se comparam favoravelmente com aqueles obtidos a partir da regressão múltipla e até mesmo da regressão linear simples para estimativa de esforço a partir do tamanho [Barcelos Tronto et al. 2006].

De uma maneira geral, os resultados obtidos nesta pesquisa mostram que os modelos de redes neurais tiveram um desempenho melhor que os modelos de regressão múltipla para modelar as complexidades envolvidas no domínio de estimativa de esforço de software. A rede neural executou melhor que os modelos de regressão sobre estes conjuntos de dados e pode-se saber porque isso acontece. Sabe-se que o tamanho do software (dado pela variável KDSI) é a variável com maior significância na estimativa de esforço. Considerando o conjunto de dados COCOMO, existem duas observações com valores de esforço muito grandes, que estão fora de todas as proporções de seus tamanhos; além disso, existe um valor de esforço que é pequeno em relação ao tamanho do software. Assim, uma função linear de tamanho não teria muito sucesso para realizar estimativas para tais valores. Por outro lado, uma tentativa de resolver estes *outliers* poderia influenciar negativamente na precisão de regressão para realizar estimativas para outras observações. Uma vez que redes neurais não são limitadas a funções lineares, elas podem lidar melhor com observações que estão fora da reta idealizada.

O melhor dos modelos de regressão múltipla, calibrados nesta pesquisa teve um MMRE igual a 354 e o pior teve um MMRE igual a 1526. Já os modelos de redes neurais se mostraram capazes de capturar efetivamente os parâmetros que influenciam no esforço de desenvolvimento, sendo que o melhor modelo de redes neurais tem um MMRE igual a 277 e o pior tem um MMRE igual a 559. As variáveis categóricas do conjunto de dados COCOMO (todas aquelas da Tabela 1, exceto KDSI), consideradas nesta pesquisa, podem ser responsáveis por tal variação. Acredita-se que resultados melhores possam ser encontrados se for realizado um pré-processamento mais rigoroso dos dados, por exemplo, utilizando-se de recursos de inteligência artificial para realizar a transformação de dados e outras técnicas de seleção de atributos, análise de resíduos, dentre outros.

Possivelmente, um modelo de regressão se mostre mais vantajoso quando é utilizado um conjunto de dados mais homogêneo, sem *outliers.* Mas, rede neural também executa melhor sobre tais conjuntos de dados.

Embora a abordagem de redes neurais tenha demonstrado ter vantagens significativas em certas circunstâncias, ela não substitui regressão e deve ser considerada como outra poderosa ferramenta para ser usada na calibração de modelos de esforço de software. Conclui-se que existe um forte relacionamento entre o sucesso de uma técnica particular e o tamanho do conjunto de treinamento, a natureza da função custo e das características do conjunto de dados (*outliers*, colinearidade, número de características, dentre outras) e que a melhor técnica pode não ser a idéia certa a seguir. Essa premissa é investigada neste trabalho, em que uma variedade de experimentos revela que existe uma considerável variação no desempenho destes sistemas quando a natureza dos dados históricos muda. Por exemplo, para alguns dos conjuntos de dados apresentados neste artigo, a regressão linear simples tem melhor desempenho que os modelos de redes neurais para o mesmo conjunto de dados.

Conseqüentemente, novos experimentos estão sendo conduzidos de forma a combinar as técnicas de redes neurais e de regressão para treinar e testar modelos de predição de esforço de software, sobre outros conjuntos de dados. Por exemplo, sobre a base de dados ISBSG (*International Software Benchmarking Standard Group*), que é constituída de um conjunto de dados contendo informações de desenvolvimento de projetos, resultantes do uso de técnicas mais modernas de desenvolvimento de software. O objetivo é melhorar o desempenho do modelo de redes neurais obtido neste trabalho e obter um modelo que possa ser utilizado com segurança no desenvolvimento de software.

Referências

Agarval, R. (2001). Estimating software projects, Software Engineering Notes, v.26, p. 60-57.

Albrecht, A.J. (1979). Measuring application development productivity, Proc. IBM Application Development Symposium, p. 83-92.

Albrecht, A.J. and Gaffney, J.R. (1983). Software function, source lines of code and development effort prediction: a software science validation, IEEE Transactions on Software Engineering, v. 9, n. 6, p. 639-648.

Angelis, L. and Stamelos, I. (2000). A simulation tool for efficient analogy based cost estimation, Empirical Software Engineering, n.5, 35-68.

Barcelos Tronto, I. F., Simões da Silva, J.D. and Sant'Anna, N. (2006). Comparison of Artificial Neural Network and Regression Models in Software Effort Estimation, In: Symposium Brazilian of Neural Networks, Ribeirão Preto, Brazil (submitted).

Bisio, R. and Malabocchia, F. (1995). Cost estimation of software projects through case base reasoning, 1st Intl. Conf. on Case-Based Reasoning Research & Development, Springer-Verlag, p.11-22.

Boehm, B.W. (1981). *Software engineering economics*, Prentice-Hall, Englewood Cliffs, NJ.

Boehm, W., Horowitz, E., Madachy, R., Reifer, D., Clark, B.K., Steece, B., Brown, A.D. and Abts, C. (2000). *Software Cost Estimation with COCOMOII,* Prentice-Hall.

Briand, L.C., Langley, T. and Wieczorek, I. (2000). A replicated assessment and comparison of common software cost modeling techniques, Pro. ICSE 2000, Limerick, Ireland, 377-386.

Briand, L.C. and Wieczorek, I. (2002). Software resource estimation, Encyclopedia of Software engineering, n.2, p. 1160-1196.

Finnie, G.R. Witting, G.E. and Desharnais, J-M. (1997). A comparison of software effort estimation techniques: Using function points with neural networks, case-based reasoning and regression models, Journal of Systems and Software, 39, 281-289.

Gray, A.R. and MacDonell, S.G. (1997). A comparison of model building techniques to develop predictive equations for software metrics. Information and Software Technology, 39, 425-437.

Gray, A.R., MacDonell, S.G. and Shepperd, M. (1999). Factors systematically associated with errors in subjective estimates of software development effort: the stability of expert judgment, Proc. IEEE 6th Metrics Symposium.

Hasting, T.E. and Sajeev, A.S.M. (2001). A vector based approach to software size measurement and effort estimation," IEEE Transactions on Soft. Engineering, v. 27, n.4.

Jeffery, R., Ruhe, M. and Wieczorek, I. (2001). Using public domain metrics to estimate software development effort, Proc. IEEE 7th Metrics Symposium, London, UK, 16-27.

Jones, C. 1986. *Estimating Software Costs,* McGraw-Hill.

Karunanitthi, N., Whitley, D. and Malaiya, Y.K. (1992). Using neural networks in reliability prediction, IEEE Software, v. 9, n.4, p.53-59.

Kemerer, C.F. (1987). An empirical validation of software cost estimation models, Comunication of ACM, v.30, p.416-429.

Khoshgoftaar, T. M., Allen, E.B. and Xu, Z. (2000). Predicting testability of program modules using a neural network, Proc. 3rd IEEE Symposium on Application-Specific Systems and Sof. Eng. Technology, p. 57-62.

Kitchenham, B. (1998). A procedure for analyzing unbalanced datasets, IEEE Transactions on Software Engineering, 24, 278-301.

Kok, P., Kitchenham, B.A. and Kirakowski, J. (1990). The MERMAID approach to software cost estimation, Espirit Technical Week.

Kumar, S. Krishna, B.A. and Satsangi, P.S. (1994). Fuzzy systems and neural networks in software engineering project management. Journal of Applied Intelligence, 4: 31-52.

Lai, R. and Huang, S. (2003). A model for estimating the size of a formal communication protocol specification and its implementation, IEEE Transaction on Software Engineering, v.29, n. 1, p. 46-62.

MCT - Ministerio da Ciência e Tecnologia, (2001). Qualidade e Produtividade no setor de software," In: http://www.mct.gov.br/Temas/info/Dsi/Quali2001/2001Tab40.htm, Tabela 40 – Práticas de Engenharia de Software no Desenv. e Manutenção de Software.

Myrtveit, I., Stensrud, E. and Shepperd, M. (2005). Reliability and validity in comparative studies of software prediction models, IEEE Transaction on Soft. Engineering, v.31, n. 5, p. 46-62.

Putnam, L.H.(1978). A general empirical solution to the Macro Sizing and Estimating Problem, IEEE Transaction on Software Engineering, 4, 345-361.

Samson, B., Ellison, D. and Dugard, P. (1997). Software cost estimation using albus perceptron (CMAC), Information and Software Technology, v.39, p. 55-60.

Selby, R.W. and Porter, A.A. (1998). Learning from examples: generation and evaluation of decision trees for software resource analysis. IEEE Trans on Software Engineering, 14, 1743-1757.

Sentas, P., Angelis, L., Stamelos, I. and Bleris, G. (2005). Software productivity and effort prediction with ordinal regression, Journal Information and Software Technology, n. 47, p.17-29.

Shepeerd, M.J., Shofield, C. and Kitchenham, B. (1996). Effort Estimation Using Analogy. Proc. ICSE-18, Berlin.

Shepperd, M. and Schofield, C. (1997). Estimating software project effort using analogies, IEEE Transactions on Software Engineering, v.23, n.12, p.736-743.

Srinivazan, K. and Fisher, D. (1995). Machine learning approaches to estimating software development effort, IEEE Transactions on Software Engineering, v.21, n.2, p.126-137.

Vicinanza, S., Prietula, M.J. and Mukhopadhyay, T. (1990). Case-based reasoning in software effort estimation," In Proc.11th Int. Conf. Info. Syst, p.149-158.

Zhong, S., Khoshgoftaar, T.M. and Seliya, N. (2004). Analysing software measurement data with clustering techniques, IEEE Intelligent Systems, p.20-27.

Witting, G. and Finnie, G. (1994). Using artificial neural networks and function points to estimate 4GL software development effort, Journal Information and Systems, v. 1, n.2, p. 87-94.

Witting, G. and Finnie, G. (1997). Estimating software development effort with connectionist models, Information and Software Technology, v. 39, p. 369-476.

Aplicando uma Metodologia Baseada em Evidência na Definição de Novas Tecnologias de Software

Sômulo Nogueira Mafra[1], Rafael Ferreira Barcelos[1,2], Guilherme Horta Travassos[1]

{somulo, barcelos, ght}@cos.ufrj.br

[1]COPPE/UFRJ - Programa de Engenharia de Sistemas e Computação, Cx. Postal 68.511
CEP 21945-970, Rio de Janeiro - RJ – Brasil

[2]BENQ-SIEMENS – Research and Development Department, Manaus – AM – Brasil

Abstract. *The transfer of Software technologies to the industrial context could bring undesirable consequences if such technologies are not under an adequate maturity level. This paper aims at to describe the use of an evidence-based methodology that can support the reduction of this risk. The application of such methodology is illustrated through two concrete cases regarding the definition of a software requirements reading technique and a checklist based approach for inspecting software architectural models. Besides, some lessons learned by using the methodology are also described.*

Resumo. *A introdução de tecnologias de software recém-definidas no contexto industrial pode trazer conseqüências indesejáveis caso a tecnologia não possua um grau adequado de maturidade. Nesse sentido, o presente artigo visa a ilustrar como a utilização de uma metodologia baseada em evidência pode auxiliar a minimizar essa situação. A utilização de tal metodologia é ilustrada através de dois casos concretos relacionados à definição e à avaliação experimental de uma técnica de leitura de requisitos de software e de uma abordagem para inspeção de documentos arquiteturais. Além disso, são discutidas as principais lições aprendidas do uso de tal metodologia.*

1. Introdução

A busca pela melhoria da qualidade no desenvolvimento de software tem sido constante nos últimos anos. Para atingir esse fim, um programa de melhoria em uma organização geralmente caracteriza-se por ser realizado em duas frentes [Kitchenham *et al*. 2004]: por um lado, organizações visam à implantação de processos aderentes a modelos de maturidade, como o CMMI [Chrissis *et al*. 2003] e o MR mps [Weber *et al*. 2004]; por outro, engenheiros de software buscam aprimorar suas qualificações através da obtenção de certificados profissionais, como o PMP [PMP 2000], por exemplo.

Entretanto, ainda de acordo com Kitchenham *et al*. (2004), a utilização de processos de qualidade por engenheiros de software qualificados não é condição suficiente para a melhoria da qualidade no desenvolvimento. O desenvolvimento é dependente de diversas tecnologias muitas das quais não se possuem evidências suficientes sobre potenciais benefícios, limitações, custo de implantação e riscos associados. Em vista desse quadro, gerentes freqüentemente são confrontados com as

seguintes questões quando da adoção de novas tecnologias de software:

- Em qual tecnologia investir, quando todas elas prometem aprimorar a produtividade e a qualidade no desenvolvimento [Shull 1998]?
- Como saber o custo de implantação de uma determinada tecnologia?
- Como determinar o retorno de investimento da implantação de tal tecnologia?
- Sob quais circunstâncias a adoção de tal tecnologia pode ser recomendada?

Essa situação pode ser agravada pela baixa interação muitas vezes observada entre academia e indústria no que diz respeito à definição de novas tecnologias [Kaindl *et al.* 2002]. A alta demanda na indústria por soluções de curto prazo aliada à necessidade de justificar a continuidade de investimento em pesquisa têm pressionado cada vez mais a transferência para a indústria de tecnologias recém-definidas na academia. Como conseqüência, muitas tecnologias são transferidas para a indústria sem terem passado, durante seu processo de definição, por um processo adequado de avaliação que permitisse a caracterização do seu grau de maturidade.

Visando a minimizar a ocorrência desses problemas, Shull *et al*. (2001) propuseram uma metodologia para a introdução de tecnologias de software na indústria. Tal metodologia apresenta uma série de questões que devem ser tratadas durante a avaliação de uma tecnologia, assim como os tipos de estudos experimentais (estudos primários) que contemplam essas questões. A aplicação de tal metodologia possibilita determinar com níveis razoáveis de segurança as limitações e os pontos fortes da aplicação da tecnologia avaliada. Como resultado, eventuais dificuldades na aplicação da tecnologia poderiam ser exploradas, contribuindo para refinar a técnica antes de disponibilizá-la para a indústria.

Entretanto, a metodologia proposta em Shull *et al*. (2001) não contempla a etapa da definição inicial de uma tecnologia de software. A metodologia parte do princípio de que já existe uma versão inicial da tecnologia a ser avaliada. Nesse sentido, a identificação do conhecimento sobre a área disponível na literatura, que possui fundamental importância para a etapa de definição inicial de uma tecnologia, não é contemplada. Como conseqüência, esse processo de identificação de conhecimento geralmente é conduzido informalmente, aumentando a possibilidade de viés na pesquisa.

Portanto, o objetivo deste trabalho é apresentar uma extensão à metodologia proposta por Shull *et al*. (2001). Tal extensão é representada pela aplicação de uma abordagem formal para a identificação de evidência (estudos secundários) existente na literatura. A utilização da metodologia estendida é ilustrada através de dois casos concretos, que levam em consideração a definição e a avaliação experimental de uma abordagem para inspeção de documentos arquiteturais baseada em *checklist*, denominada ArqCheck [Barcelos e Travassos 2006a], e de uma técnica de leitura denominada OO-PBR [Mafra e Travassos 2006a].

O impacto da condução de estudos secundários na definição de novas tecnologias é discutido através de lições aprendidas oriundas tanto de projetos de pesquisas que utilizaram tal extensão quanto de projetos que não a utilizaram. A condução de estudos experimentais para caracterizar quantitativamente a contribuição da extensão não é tratada neste artigo, ficando como perspectiva futura de trabalho.

O texto encontra-se dividido em sete seções, incluindo esta que introduz o

artigo. Na seção 2, a metodologia para introdução de tecnologias de software na indústria é descrita. A seção 3 discute as lições aprendidas do uso de tal metodologia durante a definição de duas tecnologias. Essas lições motivaram a extensão proposta neste trabalho, que é descrita na seção 4. Na seção 5, a aplicação da metodologia estendida é ilustrada através da avaliação de ArqCheck e OO-PBR. As lições aprendidas da metodologia estendida são descritas na seção 6. Finalmente, a seção 7 conclui o artigo.

2. Metodologia Original: Estudos Primários

A metodologia experimental para introdução de tecnologias de software na indústria definida em Shull *et al*. (2001) é composta por quatro etapas, que contemplam a avaliação da tecnologia desde sua definição até sua transferência para a indústria. As quatro etapas são representadas por: (a) estudos de viabilidade, (b) estudos de observação, (c) estudo de caso (ciclo de vida), e (c) estudo de caso na indústria.

Os estudos sugeridos para cada uma das etapas estão diretamente relacionados às variáveis que se deseja controlar e os riscos possíveis de serem assumidos na fase em questão, conforme apontado por Silva (2004). Os quatro tipos de estudo são sucintamente descritos nas próximas subseções.

2.1. Estudo de Viabilidade

O objetivo principal de um estudo de viabilidade não é encontrar uma resposta definitiva, mas sim criar um corpo de conhecimento sobre a aplicação da tecnologia. Nesse sentido, seria possível ao pesquisador avaliar se a aplicação da tecnologia é viável, ou seja, se atende de forma razoável aos objetivos inicialmente definidos, de forma a justificar (ou não) a continuação da pesquisa. Além disso, espera-se que o corpo de conhecimento construído forneça subsídios que permitam: (a) o refinamento da tecnologia, e (b) a geração de novas hipóteses sobre sua aplicação a serem investigadas em estudos posteriores [Shull *et al*. 2004].

2.2. Estudo de Observação

Um estudo de observação geralmente é realizado em ambiente no qual a aplicação prática da tecnologia é observada por pesquisadores. Através dessa observação, é possível coletar dados sobre como a tecnologia é aplicada. Dessa forma, os pesquisadores podem adquirir uma compreensão refinada sobre a tecnologia, ao presenciarem eventuais dificuldades que os participantes possam apresentar.

2.3. Estudo de Caso (Ciclo de Vida)

Atingir esta fase pressupõe que haja indícios de que a tecnologia sendo avaliada é efetiva. Porém, a tecnologia foi avaliada de forma isolada, não sendo possível determinar qual o efeito de sua aplicação em conjunto com outras tecnologias. Para ter acesso a essa informação, um estudo de caso deve ser conduzido com o propósito de caracterizar a aplicação da tecnologia no contexto de um ciclo de vida de desenvolvimento.

2.4. Estudo de Caso (Indústria)

A condução das etapas anteriores permitiria: (a) determinar a viabilidade prática de aplicação da tecnologia, (b) aprimorar o entendimento dos pesquisadores resultando no

refinamento da tecnologia e (c) caracterizar a aplicação da tecnologia no contexto de um ciclo de vida. Como conseqüência, supõe-se que a tecnologia tenha atingido um grau de maturidade razoável que possibilite a sua avaliação no contexto industrial.

A condução de um estudo de caso na indústria tem como principal objetivo caracterizar a aplicação da tecnologia no ambiente industrial. Na próxima seção, são listadas algumas lições aprendidas sobre a aplicação da metodologia de introdução de tecnologias de software na indústria que motivaram a extensão proposta neste trabalho.

3. Utilizando a Metodologia Original

A metodologia de introdução de tecnologias de software na indústria tem sido tradicionalmente utilizada na pesquisa realizada pelo Laboratório de Engenharia de Software da COPPE/UFRJ [Silva 2004; Kalinowski e Travassos 2004; Lima 2005; Nunez 2005; Mafra 2006; Barcelos 2006]. Através da experiência adquirida com a aplicação dessa metodologia foi possível identificar oportunidades de melhoria que resultaram na proposição de sua extensão, discutida na seção 4.

As lições aprendidas decorrentes da aplicação da metodologia, e a necessidade de se estender tal metodologia, são discutidas através da experiência vivenciada durante a definição das OORT`s [Travassos *et al*. 1999], uma família de técnicas de leitura para modelos de projeto OO, e de ISPIS [Kalinowski e Travassos 2004], uma infra-estrutura computacional de apoio a inspeções de software. Ambas as tecnologias foram avaliadas através da condução das quatro etapas contempladas pela metodologia.

3.1. OORT`s: Técnicas de Leitura para Modelos de Projeto OO

Durante sua definição, os autores das OORT`s não tinham identificado na literatura abordagens de leitura para a inspeção de modelos de projeto OO. Essa carência de iniciativas representou um desafio considerável de pesquisa. Os autores tiveram de definir uma nova tecnologia contando apenas com suas próprias experiências no desenvolvimento de aplicações OO.

Porém, os autores das OORT's possuíam experiência na definição e avaliação experimental de outras técnicas de leitura, entre as quais PBR [Shull 1998], referente a inspeções de requisitos, por exemplo. A viabilidade de aplicação de técnicas de leitura em diferentes tipos de artefatos, aliada à demanda industrial, foi a principal motivação para a proposição de uma família de técnicas de leitura para inspeção de modelos de projeto OO. Contudo, diversas dúvidas permearam a definição das OORT's, entre elas:

- A falta de apoio à leitura de modelos de projeto OO representava de fato um problema para a indústria. Entretanto, como resolvê-lo?
- A abordagem de inspeção representada pela aplicação de técnicas de leitura (que tinha se mostrado adequada para a inspeção de requisitos, por exemplo) seria viável também para a inspeção de modelos de projeto OO?

O desafio representado por essas questões motivou a definição e a utilização de uma metodologia para a avaliação de tecnologias imaturas visando à sua transferência para a indústria. A utilização dessa metodologia no contexto das OORT's contribuiu para a caracterização do grau de maturidade da tecnologia. Resultados experimentais incluindo todos os estudos previstos na metodologia podem ser encontrados em Travassos *et al*. (1999), Shull *et al*. (2001), Melo *et al*. (2001) e Conradi *et al*. (2003).

Naquele momento, as dúvidas e incertezas presenciadas foram consideradas

naturais devido à carência de relatos na literatura sobre experiências anteriores que pudessem auxiliar a tomada de decisão durante o processo de definição das OORT's. Como conseqüência, a necessidade de se conduzir uma revisão sistemática ainda não tinha sido considerada.

3.2. ISPIS: Infra-estrutura Computacional para Inspeções de Software

Diferentemente do cenário presenciado durante a definição das OORT's, a pesquisa sobre inspeções de software possuía considerável pesquisa disponível na literatura. Como conseqüência, os autores de ISPIS puderam realizar uma revisão criteriosa da literatura levando em consideração resultados experimentais relacionados a processos de inspeção de software.

Através dos resultados obtidos, ISPIS foi definida e avaliada experimentalmente. A descrição e os resultados dos quatro tipos de estudo a que ISPIS foi submetida podem ser encontrados em Kalinowski e Travassos (2004) e Kalinowski e Travassos (2005).

Entretanto, mesmo a presença relativamente extensa na literatura de estudos experimentais sobre inspeções de software não contribuiu para a redução de incertezas durante a definição de ISPIS. Como exemplo das dificuldades encontradas pode ser citada a falta de consenso sobre a condução de inspeções síncronas ou assíncronas.

3.3. Lições Aprendidas

Como principal lição aprendida dessas pesquisas, verificou-se que a simples presença de estudos experimentais na literatura sobre uma determinada área de pesquisa não seria suficiente para minimizar a dificuldade na definição de uma nova tecnologia; resultados experimentais devem ser caracterizados: (a) para que sejam melhor entendidos, e (b) para que seja possível identificar tendências e oportunidades de pesquisa.

Para isso, um processo de caracterização de evidências deveria ser formalizado com o objetivo de se identificar e caracterizar eventuais resultados experimentais existentes relacionados à tecnologia a ser desenvolvida. A formalização desse processo permitiria que essa etapa fosse repetível e passível de ser melhorada.

Essa necessidade motivou a extensão da metodologia proposta em Shull *et al*. (2001) de forma a contemplar a adoção de uma abordagem baseada em evidência para a caracterização de resultados experimentais. Tal abordagem é representada pela condução de estudos secundários (revisões sistemáticas). A extensão é descrita na próxima seção.

4. Estendendo a Metodologia Original: Estudos Secundários

Esta seção descreve a importância da condução de estudos secundários no processo de geração de evidências na pesquisa científica. Além disso, é descrito como estudos secundários são complementares a estudos experimentais (estudos primários) nesse processo de geração de evidências. Na seqüência, é descrito como a condução de estudos secundários pode contribuir para a metodologia proposta em Shull *et al*. (2001).

4.1. Revisões Sistemáticas em Engenharia de Software

A importância da aplicação de uma metodologia baseada em evidência na pesquisa científica pode ser ilustrada pela experiência vivenciada na Medicina. Durante muito tempo, a área médica esteve repleta de revisões que não utilizavam métodos para

identificar, avaliar e sintetizar a informação existente na literatura [Cochrane 2003]. No final da década de 80, estudos conduzidos para avaliar a qualidade das publicações médicas chamaram a atenção para a baixa qualidade científica [Cochrane 2003].

Uma provável conseqüência dessa falta de rigor pôde ser observada no final dos anos 80s. Naquela ocasião, estudos apontaram que, de um lado, a falha em organizar a pesquisa médica em revisões sistemáticas podia custar vidas [Cochrane 1989]. Por outro lado, julgamentos clínicos de alguns especialistas foram considerados inadequados quando comparados aos resultados de revisões sistemáticas [Antman *et. al* 1992].

Desde então, o reconhecimento da necessidade da condução de revisões de forma sistemática e formal em Medicina tem crescido rapidamente. Este fato pode ser comprovado pela grande quantidade de revisões formais publicadas a cada ano na área médica [NHSCRD 2003]. Tais revisões, denominadas de revisões sistemáticas, são revisões rigorosas da literatura à procura de indícios que possam levar à identificação de evidências sobre um tema de pesquisa ou tópico na área em questão.

Foi o trabalho de Kitchenham *et al*. (2004) o primeiro a estabelecer um paralelo entre Medicina e Engenharia de Software, no que diz respeito à abordagem baseada em evidência. Segundo os autores, a Engenharia de Software Baseada em Evidência deve prover meios pelos quais melhores evidências provenientes da pesquisa possam ser integradas com experiência prática e valores humanos no processo de tomada de decisão considerando o desenvolvimento e a manutenção do software.

Em vista disso, para atingir um nível adequado de evidência a respeito da caracterização de uma determinada tecnologia em uso, a Engenharia de Software Baseada em Evidência deve fazer uso basicamente de dois tipos de estudos: estudos primários e estudos secundários [Mafra e Travassos 2006b].

Por estudos primários, entende-se a condução de estudos que visem a caracterizar uma tecnologia em uso dentro de um contexto específico. Nessa categoria encontram-se os estudos experimentais, entre os quais experimentos, estudos de caso e *surveys* [Wöhlin *et al*. 2000]. A metodologia de Shull *et al*. (2001) contempla a condução de estudos primários para a avaliação de uma tecnologia de software.

Por estudos secundários, entende-se a condução de estudos que visem a identificar, avaliar e interpretar todos os resultados relevantes a um determinado tópico de pesquisa, fenômeno de interesse ou questão de pesquisa. Revisão sistemática é um tipo de estudo secundário [Kitchenham *et al*. 2004; Biolchini *et al*. 2005]. Nesse sentido, resultados obtidos por diversos estudos primários correlatos atuam como fonte de informação a ser investigada por estudos secundários (Figura 1).

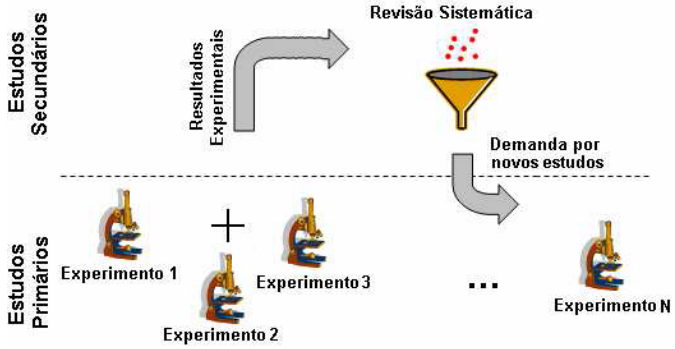


Figura 1 - Interação entre Estudos Primários e Secundários.

Além disso, é válido ressaltar que estudos secundários não podem ser considerados uma abordagem alternativa para a produção primária de evidências, representada pelos estudos primários [Biolchini *et al*. 2005]; estudos secundários são complementares a estudos primários nesse processo. A precisão e a confiabilidade proporcionadas pela condução de estudos secundários contribuem para a melhoria e para o direcionamento de novos tópicos de pesquisa, a serem investigados por estudos primários, num ciclo iterativo.

4.2. Proposta de Extensão à Metodologia Original

A proposta apresentada nesse trabalho consiste em estender a metodologia experimental de introdução de tecnologias de software na indústria, definida em Shull *et al*. (2001), através da condução de revisões sistemáticas (Figura 2). Nesse sentido, a metodologia resultante fica dividida em dois passos: Definição e Refinamento da Tecnologia.

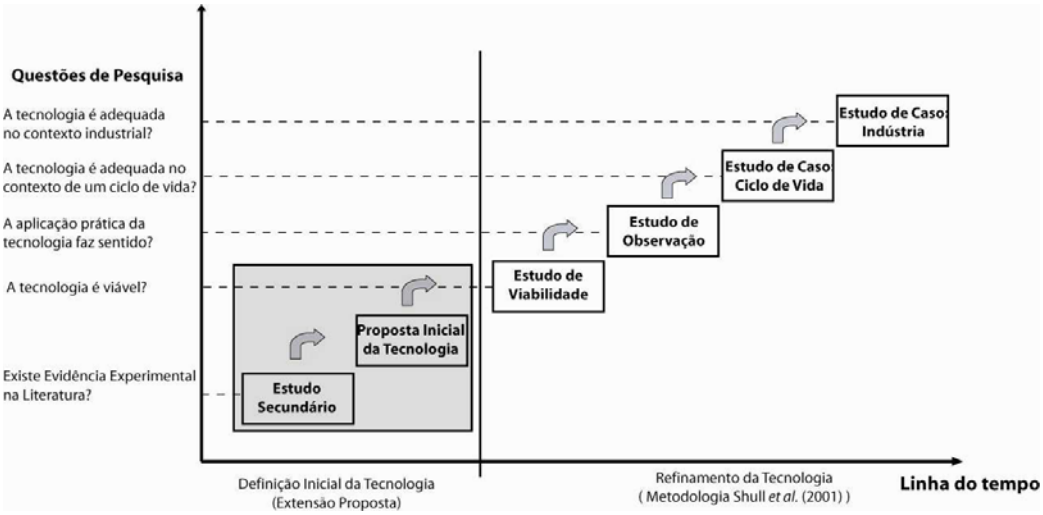


Figura 2 - Extensão da Metodologia Experimental para Introdução de Tecnologias de Software proposta em Shull *et al.* (2001).

O primeiro passo consiste na Definição Inicial da Tecnologia. Durante esse passo, a condução de uma revisão sistemática seria a primeira etapa a ser seguida. O planejamento da revisão sistemática representado pela definição de um protocolo de pesquisa ajudaria a delimitar o escopo da pesquisa. A presença de tal protocolo, definindo de forma explícita as questões de pesquisa, os critérios de seleção de fontes bibliográficas, os critérios de inclusão e exclusão de artigos, e os critérios de qualidade a serem observados nos estudos, ajudaria a amadurecer o problema sendo investigado.

Além disso, os procedimentos a serem seguidos durante a condução da revisão sistemática, no que se refere à identificação e à caracterização da evidência existente na área, poderiam reduzir drasticamente riscos associados à definição de uma tecnologia. Como conseqüência dessa caracterização, oportunidades de melhoria pertinentes à área de pesquisa poderiam ser identificadas. Além disso, tal caracterização possibilitaria identificar as lições aprendidas dos estudos identificados, minimizando a possibilidade de repetir erros passados.

Com base no conhecimento adquirido e nas evidências identificadas através da condução da revisão sistemática, o pesquisador define uma proposta inicial da

tecnologia. A partir dessa versão inicial da tecnologia, as etapas relacionadas à série de estudos de avaliação propostos na metodologia original são realizadas, visando a um refinamento sucessivo e a um amadurecimento da tecnologia em questão.

Na próxima seção, a aplicação da metodologia estendida é ilustrada através de dois casos concretos.

5. Aplicando a Metodologia Estendida

A metodologia estendida foi utilizada no contexto da definição e avaliação experimental de duas novas tecnologias de software. As subseções a seguir descrevem como a condução de revisões sistemáticas foi fundamental para a definição dessas tecnologias e para a posterior avaliação experimental à qual elas foram submetidas.

5.1. ArqCheck: Abordagem de Inspeção de Documentos Arquiteturais

Uma das motivações para a identificação de abordagens para inspeções arquiteturais na literatura foi decorrente da necessidade de um projeto acadêmico referente ao desenvolvimento de uma infra-estrutura para condução de estudos experimentais – Projeto eSEE [Mian *et al*. 2005]. Naquele momento, havia interesse na aplicação de uma abordagem para a avaliação da arquitetura eSEE em relação aos requisitos.

Além disso, a cada ano vem aumentando o interesse de empresas brasileiras de desenvolvimento de software por abordagens de melhoria de processos e de produtos [MTC/SEPIN 2005]. A necessidade por uma abordagem de inspeção arquitetural que fosse extensível, simples de ser aplicada, genérica, e adaptável [Barcelos e Travassos 2006a] visando à utilização no contexto dessas empresas foi outra motivação para a realização do trabalho. Portanto, uma revisão sistemática foi conduzida, e com base nos resultados obtidos, cada abordagem foi caracterizada de acordo com tais critérios. A revisão sistemática é sucintamente apresentada na próxima subseção.

5.1.1. Estudo Secundário: Revisão Sistemática sobre Abordagens de Inspeção Arquitetural

A revisão sistemática conduzida no contexto da definição de ArqCheck teve como principal propósito a obtenção de informações sobre o estado da arte em relação às abordagens de inspeções de documentos arquiteturais existentes na literatura.

O objetivo da revisão pode ser resumido como: *analisar* abordagens que avaliam documentos arquiteturais *com o propósito de* caracterizá-las *com respeito às* características presentes em técnicas de inspeção de software *do ponto de vista dos* pesquisadores *no contexto dos* estudos primários que descrevem tais abordagens.

A revisão sistemática foi inicialmente executada durante o mês de dezembro de 2004, e re-executada, durante o mês de dezembro de 2005. A decisão para re-executar a revisão sistemática deveu-se a dois motivos: (a) identificar novas publicações e abordagens arquiteturais que eventualmente tenham surgido durante esse intervalo de tempo, e (b) verificar a viabilidade de se identificar publicações na máquina de busca da ACM [ACM 2006], que sofreu evoluções após a primeira revisão ter sido executada.

Como resultado, foram encontrados 119 artigos, sendo 38 selecionados. Tais artigos representavam 27 diferentes abordagens de inspeção de documentos arquiteturais. Maiores detalhes sobre as revisões conduzidas, incluindo o protocolo de revisão utilizado e a análise dos resultados dos artigos identificados, podem ser

encontrados em Barcelos e Travassos (2006b) e Barcelos (2006).

5.1.1.1. Contribuição do Estudo Secundário para a Definição de ArqCheck

Os resultados obtidos pela revisão sistemática chamaram a atenção sobre a relativa imaturidade da área de arquitetura de software. Como exemplo de tal imaturidade pode ser citado a falta de consenso na comunidade acadêmica sobre a definição de conceitos básicos e sobre a forma como uma arquitetura de software deve ser representada.

Outro fato que chamou a atenção foi a falta de evidência experimental sobre o assunto. Nenhum dos artigos obtidos descrevia estudos experimentais com o objetivo de avaliar a aplicação das abordagens propostas. Essa falta de evidência experimental limitou as contribuições que a condução de uma revisão sistemática poderia fornecer para a definição de ArqCheck.

Entretanto, a condução de um estudo secundário possibilitou principalmente a caracterização das abordagens de avaliação arquitetural identificadas. Essa caracterização permitiu o entendimento das abordagens identificadas, assim como uma comparação entre elas. A partir dessa caracterização, foi possível identificar limitações que dificultavam a aplicação dessas abordagens em um contexto industrial.

Nesse sentido, a principal contribuição da revisão sistemática conduzida foi a identificação dessas limitações que influenciaram significativamente a definição de ArqCheck. A Tabela 1 resume as decisões de projetos tomadas visando a minimizar as limitações identificadas na aplicação de ArqCheck.

Tabela 1 - Decisões tomadas para minimizar as limitações identificadas

Limitação das Abordagens Identificadas	Solução para Minimizar as Limitações
Subjetividade na avaliação	Uso de um *checklist* composto por itens de avaliação que orientam o inspetor sobre o que ser avaliado, não deixando essa decisão somente a cargo de sua experiência.
Custo de aplicação	Uso de inspeção de software por não exigir a realização de tarefas complexas, como a definição de cenários. Além disso, ArqCheck não exige uma grande quantidade de profissionais ou especialistas em arquitetura para que a avaliação seja realizada.
Avaliação de múltiplas características de qualidade	Definição de itens no *checklist* para avaliar as características de qualidade de acordo com os requisitos de qualidade especificados pelo cliente e que possuem relevância em um contexto arquitetural
Contexto de aplicação limitado	Definição de um processo de configuração que permite adaptar o *checklist* às características do documento arquitetural a ser avaliado e aos objetivos da avaliação.

Na próxima subseção são descritos os estudos primários realizados com o objetivo de refinar ArqCheck.

5.1.2. Estudos Primários: Avaliação Experimental de ArqCheck

A abordagem ArqCheck ainda encontra-se em fase de definição, tendo sido avaliada em dois estudos experimentais. No momento (Abril de 2006), estudos de caso no contexto de um ciclo de vida e na indústria estão em fase de planejamento.

Estudo de viabilidade. O objetivo foi avaliar se os itens que compõem ArqCheck estavam claros para os participantes e se auxiliavam na detecção de defeitos [Barcelos e

Travassos 2006a]. O estudo foi executado em outubro de 2005 com quatro participantes escolhidos por conveniência. Os participantes eram alunos de pós-graduação com conhecimento sobre o domínio do problema e sobre técnicas de inspeção.

Para a realização desse estudo, foi utilizado um documento representando a arquitetura de software desenvolvido em um contexto acadêmico. Esse documento continha 18 defeitos conhecidos, identificados através de uma inspeção *ad hoc*. Além disso, o pesquisador semeou 10 defeitos que permitiam avaliar se os itens de ArqCheck estavam claros e se poderiam auxiliar na identificação de defeitos.

Devido à falta de uma abordagem padrão para representação arquitetural, foi realizado, antes da execução do estudo, um treinamento com os participantes sobre a abordagem de representação que foi utilizada para descrever o documento arquitetural avaliado. O principal objetivo na realização desse treinamento foi o de evitar eventuais problemas de entendimento dos participantes em relação à simbologia utilizada, o que poderia influenciar na tarefa de identificação de defeitos.

Após a realização do estudo, o pesquisador analisou os dados fornecidos por cada participante. Com base nesses dados, foi possível obter indícios de que ArqCheck auxilia na identificação de defeitos, um vez que 75% dos defeitos conhecidos foram encontrados. Além disso, os participantes indicaram que ArqCheck foi útil na identificação dos defeitos. Através desse estudo, foi possível identificar itens de ArqCheck que não foram compreendidos pelos participantes. Com isso, esses itens foram evoluídos, resultando numa nova versão de ArqCheck. Um detalhamento mais completo desse estudo e dos resultados obtidos pode ser encontrado em Barcelos e Travassos (2006a).

Estudo de Observação. O estudo foi conduzido em fevereiro de 2006 em um ambiente acadêmico utilizando estudantes de pós-graduação com conhecimentos sobre arquitetura de software e experiência na condução de inspeções. A condução desse estudo permitiu avaliar a aplicação de ArqCheck durante a inspeção de um documento arquitetural desenvolvido em ambiente industrial. Para permitir essa troca de informações academia-indústria, um acordo foi firmado entre a COPPE/UFRJ e a BENQ-SIEMENS.

A BENQ-SIEMENS é um instituto de pesquisa e de desenvolvimento de aplicações de software voltadas para a área de telefonia celular. O acordo firmado entre essas duas instituições previa o acesso dos pesquisadores a documentos de requisitos e de projeto de alguns projetos da BENQ-SIEMENS visando à realização do estudo.

Portanto, com o acesso a esses dados, foi possível caracterizar ArqCheck em relação a um *baseline*. Para esse estudo, o *baseline* escolhido foi uma abordagem de avaliação *ad hoc* utilizada pela BENQ-SIEMENS na melhoria da qualidade do documento arquitetural utilizado no contexto desse estudo.

Após os participantes terem aplicado ArqCheck, o pesquisador consolidou as discrepâncias encontradas e as enviou para os profissionais da BENQ-SIEMENS que tiveram a responsabilidade de identificar quais discrepâncias representavam defeitos reais. Baseado nas informações fornecidas por esses profissionais, foi possível constatar:

- A viabilidade de ArqCheck no que se refere à identificação de defeitos em documentos arquiteturais (47 discrepâncias representavam defeitos reais);

- A dificuldade de se utilizar ArqCheck na avaliação de documentos arquiteturais grandes devido a quantidade de avaliações realizadas durante sua aplicação.

Como resultado, ArqCheck foi evoluída resultando em nova versão a ser investigada em estudos de caso posteriores, em fase de planejamento. Uma descrição mais detalhada desse estudo pode ser encontrada em Barcelos (2006).

5.2. OO-PBR: Uma Técnica de Leitura de Requisitos de Software

A motivação para a pesquisa sobre técnicas de leitura de documento de requisitos de software foi decorrente da importância exercida por tal documento no contexto de um ciclo de desenvolvimento. Tal importância pode ser explicada pelo fato do documento de requisitos [Mafra e Travassos 2006a]: (a) representar o interesse de diferentes *stakeholders* e (b) servir como base para a construção de diversos artefatos em fases posteriores do desenvolvimento, como o modelo de projeto, por exemplo.

Nesse sentido, observou-se que a definição de uma técnica de leitura que contemplasse aspectos OO poderia ser útil ao fornecer ao engenheiro de software orientação sobre como obter um entendimento satisfatório sobre o problema e os conceitos representados nos requisitos. Através da obtenção desse entendimento, o engenheiro de software poderia ser orientado a: (a) revisar o documento de requisitos, e (b) construir um modelo de projetos OO com base nos requisitos. Na próxima subseção, a revisão sistemática sobre técnicas de leitura de software é descrita.

5.2.1. Estudo Secundário: Revisão Sistemática sobre Técnicas de Leitura de Requisitos de Software

No momento da definição do escopo do trabalho, não era do conhecimento dos pesquisadores a existência de técnicas de leitura que explorassem a construção de projetos OO durante a sua aplicação. A perspectiva de projetista utilizada pela técnica de leitura PBR [Shull 1998], por exemplo, explora a construção de diagramas de fluxo de dados, utilizando técnicas de análise estruturada. Além do mais, não sabia-se também da existência de todos os estudos experimentais dessa natureza. Nem sobre o grau de generalização das evidências geradas por eles, caso esses estudos existissem, e caso houvesse evidências significativas de seus resultados. Portanto, com o propósito de se obter um entendimento sobre a pesquisa experimental realizada na área, optou-se pela condução de uma revisão sistemática.

Porém, apesar do foco inicial da pesquisa estar na identificação de técnicas de leitura para documentos de requisitos de software que explorassem a construção de modelos de projeto OO, decidiu-se por não restringir a busca apenas a esse tipo de técnicas. Como justificativa, caso os resultados da revisão sistemática conduzida demandassem, uma nova técnica de leitura seria proposta. Nesse sentido, os resultados experimentais, as lições aprendidas, e os procedimentos envolvidos na condução dos estudos sobre todas as técnicas de leitura obtidas poderiam auxiliar de forma considerável uma eventual avaliação experimental da nova técnica de leitura proposta. Com isso, para a condução da revisão a seguinte estratégia foi seguida:

- Obter os diversos estudos experimentais realizados sobre técnicas de leitura de documento de requisitos de software descritos em linguagem natural;
- Avaliar os resultados experimentais obtidos, as lições aprendidas, e os procedimentos utilizados na condução desses estudos;

- Identificar, no universo desses estudos, a existência de estudos sobre técnicas de leitura que explorassem a construção de modelos de projeto orientado a objetos;
- Caso a revisão retornasse resultados positivos, esses estudos seriam analisados no sentido de avaliar o grau de evidência a ser extraído de seus resultados, visando a obter benefícios e limitações do uso das técnicas investigadas;
- Caso contrário, pretendia-se propor uma nova técnica de leitura e conduzir estudos no sentido de tentar responder eventuais hipóteses de interesse.

A revisão sistemática foi conduzida entre o final de 2004 e o início de 2005, durante um período de três meses. Ao todo foram analisados 278 artigos dos quais 38 foram selecionados e tiveram os seus dados extraídos e analisados, de acordo com as recomendações previstas no protocolo da revisão. O protocolo de revisão utilizado e os principais resultados experimentais obtidos por essa revisão sistemática podem ser encontrados em Mafra e Travassos (2005) e Mafra (2006).

5.2.1.1. Contribuição do Estudo Secundário para a Definição de OO-PBR

Através da análise dos resultados da revisão, concluiu-se que a definição de uma nova técnica PBR que explorasse a construção de modelos de projeto OO minimizaria potenciais riscos referentes à definição de uma tecnologia inócua ou inviável.

Estudos experimentais demonstraram que a construção de modelos durante a aplicação de PBR permitiu aos revisores uma compreensão refinada sobre o problema descrito nos requisitos [Denger *et al*. 2004; Belgamo e Fabbri 2005]. Como conseqüência a aplicação de PBR permitiu a identificação de defeitos críticos, muitas vezes não capturados por outras abordagens utilizadas na indústria, como *checklist* e leitura *ad hoc* [Biffl e Halling 2003; Berling e Thelin 2004]. Além disso, resultados obtidos da avaliação experimental das OORT's apontaram para a viabilidade de se explorar conceitos OO durante o processo de inspeção [Travassos *et al*. 1999].

A condução da revisão sistemática possibilitou também a caracterização dos procedimentos experimentais utilizados nos estudos executados sobre técnicas de leitura. A caracterização desses procedimentos permitiu o planejamento dos estudos sobre OO-PBR com relativa segurança, ao levar em consideração as lições aprendidas e as recomendações propostas. Como benefícios podem ser citados a identificação das hipóteses investigadas, o material de treinamento disponibilizado aos participantes, os questionários de avaliação de OO-PBR utilizados nos estudos, o projeto dos estudos experimentais, entre outros. A definição desses mecanismos exclusivamente a partir da própria experiência dos pesquisadores certamente consumiria tempo e esforço considerável de pesquisa.

5.2.2.Estudos Primários: Avaliação Experimental de OO-PBR

A técnica OO-PBR foi avaliada em dois estudos de observação precedidos por um estudo de viabilidade [Mafra 2006], todos executados em ambiente acadêmico. OO-PBR ainda encontra-se em fase de avaliação, necessitando de estudos na indústria.

Estudo de viabilidade. O estudo não teve como objetivo o teste de hipóteses; a formulação de quaisquer hipóteses durante essa etapa inicial de elaboração da técnica poderia ser vista como mera especulação, dado o grau de imaturidade de uma tecnologia recém-definida. O estudo de viabilidade de OO-PBR teve como objetivo proporcionar aos pesquisadores um entendimento sobre OO-PBR decorrente da aplicação prática da

técnica. Pretendia-se durante a condução do estudo avaliar principalmente a usabilidade e a utilidade de OO-PBR. Acreditava-se que a falha no atendimento satisfatório desses dois fatores poderia comprometer uma futura avaliação experimental de OO-PBR.

Para esse estudo foram utilizados dois participantes com larga experiência na condução de inspeções de software na indústria, e na elaboração de modelos de projeto OO. Além disso, os participantes utilizados possuíam experiência também na aplicação de outras técnicas de leitura, entre elas PBR perspectiva do usuário e OORT's. Os participantes foram estimulados a tecerem quaisquer comentários que julgassem pertinentes acerca dos materiais envolvidos no estudo. Nesse sentido, o relato de críticas (positivas ou negativas) e sugestões poderia fornecer um indicativo sobre os potenciais pontos fortes e fracos de OO-PBR.

Como principal resultado do estudo de viabilidade, verificou-se um efeito colateral de OO-PBR referente à orientação sobre a construção de um modelo de projeto. Revisores tendiam a investir a maior parte do tempo de inspeção na construção do modelo ao invés de revisarem o documento em busca de defeitos. Tal efeito pôde ser explicado devido ao alto nível de detalhamento dos passos para a construção do modelo. Nesse sentido, resolveu-se alterar OO-PBR para aumentar o foco na detecção de defeitos.

Primeiro Estudo de Observação. O primeiro estudo de observação foi executado no contexto de uma disciplina de Engenharia de Software OO da UFRJ. O estudo utilizou 15 alunos de graduação com baixa experiência industrial de desenvolvimento (média 2,5 anos) como participantes. Os participantes foram divididos em cinco equipes, sendo cada equipe responsável pelo desenvolvimento de um sistema seguindo um determinado ciclo de desenvolvimento. Nesse sentido, OO-PBR teve sua aplicação avaliada após a fase de definição dos requisitos e antes da fase de projeto de alto nível.

Como principal resultado do estudo, foi observado a inadequação do processo de leitura definido em OO-PBR. Tal processo exigia que o revisor lesse o documento diversas vezes, de forma a identificar defeitos. Como conseqüência, observou-se que a fadiga causada pela leitura em excesso prejudicou o desempenho dos revisores, ocasionando um baixo nível de conformidade com o processo. OO-PBR teve seu processo de leitura revisado, de forma a: (a) minimizar a quantidade de leitura necessária, e (b) aumentar o foco sobre a detecção de defeitos.

Segundo Estudo de Observação. O segundo estudo foi realizado no contexto de uma disciplina de pós-graduação da COPPE/UFRJ. O estudo utilizou quatro estudantes como participantes, sendo que dois deles possuíam mais de 15 anos de experiência industrial.

A análise dos resultados do estudo permitiu avaliar de forma positiva as alterações no processo de leitura de OO-PBR. A modificação no processo de leitura permitiu um aumento na freqüência de acesso às questões para identificação de defeitos, ocasionando num aumento de defeitos detectados. Além disso, a aplicação da técnica demandou um menor numero de leituras do documento, conforme esperado.

6. Lições Aprendidas do Uso da Metodologia Estendida

A utilização de uma metodologia baseada em evidência representada por revisões sistemáticas e por estudos experimentais contribui satisfatoriamente para a definição de uma nova tecnologia.

A condução de uma revisão sistemática como etapa do passo de definição inicial de uma tecnologia possibilita: (a) minimizar riscos e (b) acelerar o processo de definição. Por sua vez, a execução de estudos experimentais permite avaliar a aplicação da tecnologia sendo definida, evitando a introdução de uma tecnologia relativamente imatura no contexto industrial.

A identificação de evidências na literatura é o foco de uma revisão sistemática. Essa evidência pode ser caracterizada tanto pela presença de tópicos que reconhecidamente necessitam de maior investigação assim como por tópicos sobre os quais já se observa um grau de consenso na comunidade acadêmica e na indústria. Nesse sentido, a identificação de oportunidades de pesquisa decorrentes da caracterização da evidência na literatura minimiza o risco relacionado à definição de uma tecnologia inadequada, ou seja, que não represente solução para problemas vivenciados na indústria.

Além disso, a caracterização dos resultados experimentais obtidos por uma revisão sistemática pode acelerar o processo de definição de uma nova tecnologia. A caracterização desses resultados provê o pesquisador com lições aprendidas decorrentes de experiências anteriores. Tais lições são fundamentais para dissipar eventuais dúvidas e incertezas que caracterizam a definição de uma nova tecnologia. Além disso, os pesquisadores podem fazer uso de mecanismos comprovadamente eficazes, evitando o desperdício de tempo na elaboração de tais mecanismos.

7. Conclusões

A Engenharia de Software ainda faz pouco uso de métodos científicos na definição de novas tecnologias. Tecnologias de software são freqüentemente definidas em laboratórios sem que um processo adequado de transferência para a indústria seja conduzido. A falta de caracterização da tecnologia em uso, decorrente da falta de utilização de uma abordagem baseada em evidência durante sua etapa de definição, torna a Engenharia de Software palco para especulações sobre a eficácia e a qualidade de novas tecnologias.

De forma a tratar esses problemas, Shull *et al*. (2001) propuseram uma metodologia para transferência de tecnologias de software para a indústria. Tal metodologia prevê etapas que contemplam a avaliação experimental da tecnologia sendo definida, visando a caracterizar o seu grau de maturidade, antes de sua transferência para a indústria. Entretanto, a etapa de definição inicial de uma tecnologia representada pela revisão de literatura não é contemplada por tal metodologia.

Nesse sentido, o presente trabalho propôs como a metodologia de Shull *et al*. (2001) poderia ser estendida para contemplar a fase de concepção de uma nova tecnologia. Através da condução de revisões sistemáticas, seria possível identificar e caracterizar o grau de evidência experimental existente na área. Como conseqüência, eventuais dificuldades e incertezas que povoam o processo de definição de uma nova tecnologia poderiam ser minimizadas.

Com a adoção de uma abordagem baseada em evidência, caracterizada pela condução de estudos primários e secundários, a Engenharia de Software poderia atingir um elevado grau de maturidade. Dessa forma, conforme apontado por Juristo e Moreno (2002), evoluir-se-ia do desenvolvimento de software baseado em especulação para o desenvolvimento de software baseado em fatos. Essa evolução permitiria transformar o

processo de construção de software de qualidade em um processo de predição.

Agradecimentos. Os autores agradecem o apoio financeiro do CNPq e da FAPEAM para a realização deste trabalho e a colaboração da BENQ-SIEMENS durante os estudos de ArqCheck.

Referências

ACM (2006). "Association for Computing Machinery (ACM) Digital Library". In: http://www.acm.org, acessado em 01/12/2005.

Antman E., Lau, J., Kupelnick, B., Mosteller, F., Chalmers, T. (1992) "A comparison of results of meta-analysis of randomized controlled trials and recommendations of clinical experts", JAMA, 268(2):240-248, July 1992.

Barcelos, R. (2006) "Uma Abordagem para Inspeção de Documentos Arquiteturais Baseada em Checklist". Dissertação de Mestrado, Programa de Engenharia de Sistemas e Computação, COPPE/UFRJ, Rio de Janeiro, RJ, Brasil.

Barcelos, R., Travassos, G. (2006a) "ArqCheck: uma Abordagem para Inspeção de Documentos Arquiteturais Baseada em Checklist". In: V Simpósio Brasileiro de Qualidade de Software, Vila Velha, ES, Brasil.

Barcelos, R., Travassos, G. (2006b) "Evaluation Approaches for Software Architectural Documents: a Systematic Review". In: Proceedings of IDEAS'06, La Plata, Argentina.

Belgamo, A., Fabbri, S. (2005) "GUCCRA: Técnica de Leitura para apoiar a Construção Modelos de Casos de Uso e a Análise de Documentos de Requisitos" In: XIX SBES, Uberlândia, MG, Brasil.

Berling, T., Thelin, T. (2004) "A Case Study of Reading Techniques in a Software Company", 2004 International Symposium on Empirical Software Engineering (ISESE'04), August 19 - 20, 2004, Redondo Beach, California, pp. 229-238.

Biffl, S., Halling, M. (2003) "Investigating the defect detection effectiveness and cost benefit of nominal inspection teams", IEEE Trans. on Soft. Eng., Vol.: 29:5, May Pages: 385-397.

Biolchini, J., Mian, P.G., Natali, A.C., Travassos, G.H. (2005) "Systematic Review in Software Engineering: Relevance and Utility", Technical Report ES-679/05, PESC - COPPE/UFRJ, Rio de Janeiro, RJ.

Chrissis, M., Konrad, M., Shrum, S. (2003) "CMMI: Guidelines for Process Integration and Product Improvement", Addison Wesley.

Cochrane, Al. (1989) In Chalmers I, Enkin M, Keirse MJNC, eds. "Effective care in pregnancy and childbirth". Oxford University Press, Oxford, 1989.

Cochrane Collaboration (2003), Cochrane Reviewers' Handbook. Version 4.2.1. http://www.cochrane.dk/cochrane/handbook/hbook.htm, acessado em 04/01/2006.

Conradi, R., Mohagheghi, P., Arif, T. (2003) "Object-Oriented Reading Techniques for Inspection of UML Models – An Industrial Experiment". In: Proc. of the European Conference on Object-Oriented Programming (ECOOP 03), pp. 483-500, July.

Denger, C., Ciolkowski, M., Lanubile, F. (2004) "Investigating the Active Guidance Factor in Reading Techniques for Defect Detection", Proc. of ISESE'04, Redondo Beach, California.

Juristo, N., Moreno, A. (2002) "Reliable Knowledge for Software Development", IEEE Software, pp. 98-99, sep-oct, 2002.

Kaindl, H., Brinkkemper, S., Bubenko, J., Farbey, B., Greenspan, S., Heitmeyer, C., Leite, J.C.S.P., Mead, N., Mylopoulos, J., Siddiqi, J. (2002) "Requirements Engineering and Technology Transfer: Obstacles, Incentives and Improvement Agenda". Requirements Engineering, vol.7, pp. 113-123, 2002.

Kalinowski, M, Travassos, G. (2004) "A Computational Framework for Supporting Software Inspections". In: Proc. of the 19th IEEE Conference on Automated Software Engineering - ASE'04, pp. 46-55, Linz, Austria.

Kalinowski, M, Travassos, G. (2005) "Software Technologies: The Use of Experimentation to Introduce ISPIS – a Software Inspection Framework – Into the Industry". In: 2nd ESELAW, Uberlândia, MG, Brazil.

Kitchenham, B., Dybå, T., Jorgensen, M. (2004) "Evidence-based Software Engineering", Proceedings of the 26th International Conference on Software Engineering (ICSE'04).

Lima, G. (2005) "Heurísticas para Identificação da Ordem de Integração de Classes em Testes Aplicados a Software Orientado a Objetos". Dissertação de Mestrado, Programa de Engenharia de Sistemas e Computação, COPPE/UFRJ, Rio de Janeiro, RJ, Brasil.

Mafra, S. N., Travassos, G. H. (2005) "Técnicas de Leitura de Software: Uma Revisão Sistemática". In: XIX SBES, Uberlândia, MG, Brasil.

Mafra, S. N. (2006) "Definição de uma Técnica de Leitura Baseada em Perspectiva (OO-PBR) Apoiada por Estudos Experimentais". Dissertação de Mestrado, PESC, COPPE/UFRJ, Rio de Janeiro, RJ, Brasil.

Mafra, S., Travassos, G. (2006a) "Leitura Baseada em Perspectiva: A Visão do Projetista Orientada a Objetos". In: V SBQS, Vila Velha, ES, Brasil.

Mafra e Travassos (2006b) "Estudos Primários e Secundários Apoiando a Busca por Evidência em Engenharia de Software". Relatório Técnico ES-687/06, Programa de Engenharia de Sistemas e Computação, COPPE/UFRJ, Rio de Janeiro, RJ, Brasil.

MCT/SEPIN (2005) "Qualidade e Produtividade no Setor de Software". In: www.mct.gov.br/sepin/Dsi/Software/Menu_Qualidade.htm Acessado em Fevereiro de 2006.

Mian, P., Chapetta, W., Santos, P.S., Melo Jr., C., Natali, A.C.C., Biolchini, J., Rocha, A., Travassos, G. (2005) "eSEE: an Infrastructure for Supporting Experimental Software Engineering". In: Proceedings the 4th IEEE/ACM International Symposium on Empirical Software Engineering (ISESE) - Late Breaking Paper, Australia, November.

Melo, W., Shull, F., Travassos, G. (2001) "Software Review Guidelines". Technical Report ES-556/01, PESC - COPPE/UFRJ.

NHS Centre for Reviews and Dissemination. (2003), "Database of Abstracts of Reviews of Effectiveness". In: The Cochrane Library, Issue 1. Oxford: Updated quarterly.

Nunez, L. (2005) "Apoio Automatizado para Aplicação de Técnicas de Leitura Orientada a Objetos (OORTs)". Dissertação de Mestrado, Programa de Engenharia de Sistemas e Computação, COPPE/UFRJ, Rio de Janeiro, RJ, Brasil.

PMP (2000) "Project Management Professional: Credential Handbook". Disponível em http://www.pmi.org/info/PDC_PMP.asp. Acessado em 27/03/2006.

Shull, F. (1998) "Developing Techniques for Using Software Documents: A Series of Empirical Studies", PhD Thesis, Depart. of Computer Science, Univ. of Maryland, USA.

Shull, F., Carver, J., Travassos, G. (2001) "An Empirical Methodology for Introducing Software Processes", In: Proceedings of the Joint 8th European Software Engineering Conference (ESEC) and 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9), pp. 288-296.

Shull, F., Mendonça, M., Basili, V., Carver, J., Maldonado, J., Fabbri, S., Travassos, G., Ferreira, M. (2004) "Knowledge-Sharing Issues in Experimental Software Engineering", Empirical Software Engineering, Volume 9 Issue 1-2, March, 2004.

Silva, L., F. (2004) "Uma abordagem com apoio ferramental para aplicação de técnicas de leitura baseada em perspectiva". Dissertação de Mestrado, Programa de Engenharia de Sistemas e Computação, COPPE/UFRJ, Rio de Janeiro, RJ, Brasil.

Travassos, G., Shull, F., Fredericks, M., Basili, V. (1999) "Detecting defects in object-oriented designs: using reading techniques to increase software quality". Proc. of the OOPSLA'99, Volume 34 Issue 10, October 1999.

Weber, K., Rocha, A. R., *et al*. (2004) "Modelo de Referência para Melhoria de Processo de Software: uma abordagem brasileira". XXX Conf. Latino-americana de Informática, Arequipa - Peru, 2004.

Wöhlin, C., Runeson, P., Höst, M., Ohlsson, M., Regnell, B., Wesslén, A. (2000) "Experimentation in Software Engineering: An Introduction", The Kluwer International Series in Software Engineering, Norwell, USA, Kluwer Academic Publishers.

Uma Análise Comparativa de práticas de Desenvolvimento Distribuído de Software no Brasil e no exterior[i]

Rafael Prikladnicki, Jorge Luis Nicolas Audy

Faculdade de Informática (FACIN)
Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
90.619-900 – Porto Alegre – RS – Brasil
rafael@inf.pucrs.br, audy@pucrs.br

Abstract. *More than a decade ago, seeking lower costs and access to skilled resources, many organizations began to experiment with remotely located software development facilities. The purpose of this paper is to present the results of case studies conducted in the last four years with four companies located in Brazil and other countries, where those companies have some distributed development strategies. The analysis is presented according to the strategies, practices and challenges identified. The results are analyzed under the Software Engineering, Software Quality, and Software Project Management point of view.*

Resumo. *Nas últimas décadas, grandes investimentos têm permitido um movimento de transformação de um mercado local para mercados globais, em um processo que têm criado novas formas de colaboração e competição na área de Engenharia de Software, entre elas o Desenvolvimento Distribuído de Software. O objetivo deste artigo é apresentar os resultados de estudos de caso desenvolvidos nos últimos quatro anos com quatro empresas presentes neste cenário, localizadas no Brasil e no exterior. Apresenta-se a análise, levando-se em consideração as estratégias e práticas adotadas, e os desafios identificados. Os resultados são analisados do ponto de vista da Engenharia de Software, da Qualidade de Software e da Gerência de Projetos de Software.*

1. Introdução

Nas últimas décadas, grandes investimentos têm permitido um movimento de transformação de um mercado local para mercados globais, em um processo que têm criado novas formas de colaboração e competição (Herbsleb et. al., 2001) na área de Engenharia de Software (ES). Neste período, o mercado de desenvolvimento global de software vivenciou grandes crises: muitas falhas em projetos, e o aumento considerável da demanda, em um período onde não havia recursos com as habilidades necessárias para um bom desenvolvimento dos projetos. Assim, o Desenvolvimento Distribuído de Software (DDS) surgiu como uma oportunidade para minimizar as dificuldades encontradas com o crescimento do mercado global. Alguns fatores contribuíram para acelerar este cenário. Entre eles, podemos citar (Carmel, 1999):

- A necessidade de recursos globais para serem utilizados a qualquer hora;

- As vantagens de estar perto do mercado local, incluindo o conhecimento dos clientes e as condições locais para explorar as oportunidades de mercado;
- A grande pressão para o desenvolvimento *time-to-market*, utilizando as vantagens proporcionadas pelo fuso horário diferente, no desenvolvimento conhecido como *follow-the-sun* (24 horas contínuas, contando com as equipes fisicamente distantes).

As organizações visam obter vantagens competitivas associadas a custo, qualidade e flexibilidade no desenvolvimento de software, buscando um aumento de produtividade, assim como diminuição de riscos (Sengupta et. al., 2006). Muitas vezes, a busca por estas vantagens faz com que as organizações encontrem soluções em outros países (*offshore sourcing* ou *offshoring*). E os desafios existentes neste novo contexto estão geralmente relacionados não apenas com questões técnicas, mas também à questões estratégicas, culturais, e de gestão de conhecimento (Herbsleb et. al., 2001).

Segundo a IDC - *International Data Group* (2006), em média, pode se ter uma economia entre 25% a 50% em termos de custo, quando grandes projetos são transferidos para operações *offshore*. Mas custo não é o único benefício que torna o desenvolvimento distribuído atrativo. Especificamente em desenvolvimento *offshore*, outros benefícios tais como profissionais habilitados para trabalhar em um idioma diferente (geralmente a língua inglesa), baixas taxas de *turnover*, e o incentivo de governos locais contribuem para o crescimento desta nova forma de desenvolver software. Além disso, a maioria das organizações que tiveram algum tipo de experiência em *offshoring* reportou também benefícios em melhoria de processo. Desta forma, se for bem planejado, o desenvolvimento distribuído, seja inter ou intra-organizacional, local ou global, pode trazer benefícios consideráveis para as atividades envolvidas no desenvolvimento de software como um todo.

Por este motivo, em termos de estratégia, o *offshore outsourcing* é hoje uma das mais conhecidas para operacionalizar o DDS. Mas ao longo dos últimos anos, novas estratégias surgiram, devido a variedade de possibilidades existentes. Desta forma, o objetivo deste artigo é apresentar os resultados de estudos de caso desenvolvidos nos últimos quatro anos com quatro empresas que possuem alguma estratégia de DDS, analisando as práticas adotadas e as dificuldades encontradas. O principal método de pesquisa utilizado é o estudo de caso e a base empírica da pesquisa envolveu empresas de grande porte e suas unidades de desenvolvimento de software localizadas no Brasil e no exterior. Os resultados são apresentados e analisados do ponto de vista de qualidade de software (com foco no processo de desenvolvimento), engenharia de software (técnicas e ferramentas para o desenvolvimento), e gerência de projetos de software (estratégias de gestão dos projetos distribuídos).

Este artigo está estruturado da seguinte forma: a seção 2 apresenta o referencial teórico de desenvolvimento distribuído de software e os conceitos envolvidos em DDS. A seção 3 apresenta a descrição do estudo de caso realizado, a caracterização das empresas e os resultados encontrados e analisados. A seção 4 apresenta as conclusões, limitações do estudo e pesquisas futuras.

2. Desenvolvimento Distribuído de Software

O DDS tem se apresentado nos últimos anos como uma alternativa para o desenvolvimento de software. É um fenômeno que vem crescendo desde a última década, onde se observou um grande investimento na conversão de mercados nacionais em mercados globais, criando novas formas de competição e colaboração (Herbsleb et. al., 2001). Quando a distância física entre os atores em um ambiente de DDS envolve mais de um país, Carmel (1999) define uma instância do DDS chamada de desenvolvimento global de software (*Global Software Development*[1] – GSD).

Neste sentido, o DDS tem sido caracterizado pela colaboração e cooperação entre departamentos de organizações e pela criação de grupos de desenvolvedores que trabalham em conjunto, localizados em cidades ou países diferentes. Apesar de muitas vezes este processo ocorrer em um mesmo país, em regiões com incentivos fiscais ou de concentração de massa crítica em determinadas áreas, algumas empresas, visando maiores vantagens competitivas, buscam soluções globais, em outros países, o que potencializa os problemas e os desafios existentes. Segundo Carmel (1999), não existem dúvidas para qualquer profissional que trabalha na área de ES que tanto o desenvolvimento de software tradicional quanto o distribuído possuem diversas dificuldades. As principais características que os diferenciam são: **dispersão geográfica** (distância física); **dispersão temporal** (diferenças de fuso-horário); e **diferenças culturais** (idioma, tradições, costumes, normas e comportamento).

Estas diferenças refletem-se em diversos fatores. De acordo com Herbsleb et. al. (2001), destacam-se **questões estratégicas** (decisão de desenvolver ou não um projeto de forma distribuída, tendo por base análises de risco e custo-benefício); **questões culturais** (valores, princípios, etc., entre as equipes distribuídas); **questões técnicas** (fatores relativos à infra-estrutura tecnológica e ao conhecimento técnico necessário para o desenvolvimento dos projetos distribuídos, tais como redes de comunicação de dados, plataformas de hardware, ambiente de software, processo de desenvolvimento, etc.); e **questões de gestão do conhecimento** (fatores relativos à criação, armazenamento, processamento e compartilhamento de informações nos projetos distribuídos).

2.1. A operacionalização do DDS nas empresas

Nos últimos anos, a globalização tem proporcionado uma oportunidade para que empresas executem projetos de desenvolvimento de software utilizando equipes geograficamente distribuídas. À medida que as equipes passaram a se organizar desta forma, as empresas começaram a estudar meios para obter um melhor desempenho neste novo cenário. Desta forma, trabalhar em um projeto de DDS pode variar desde a simples distribuição em um mesmo país, ou então em uma distribuição em países ou continentes diferentes, criando assim o desenvolvimento global de software (Carmel, 1999).

Com isso, a pesquisa em DDS tem se tornado bastante comum nos últimos anos. Por este motivo, e devido à diversidade de possibilidades surgidas, existem muitos conceitos envolvidos. Para atuar em um ambiente de DDS, existem algumas características que precisam ser definidas. Entre elas, está o relacionamento das

[1] A maioria dos termos utilizados em DDS foi mantida em inglês, pois a literatura existente utiliza os temos tal qual está sendo utilizado neste artigo, e a tradução em alguns casos compromete o uso do termo.

empresas que participam de uma operação como esta, e a caracterização da equipe, em termos de localização geográfica. A união destas duas dimensões cria possibilidades para configurar um cenário de DDS, através de modelos de negócio específicos. Estes modelos variam de acordo com alguns critérios, e apresentam características diferentes em relação às atividades ligadas ao desenvolvimento de software. Desta forma, para melhor entender os desafios, vantagens e limitações dos tipos de DDS, apresenta-se a seguir uma padronização dos conceitos existentes, explorados através do relacionamento entre as dimensões citadas anteriormente. Considerando a relação entre as empresas, esta tem ocorrido através de três formas principais, segundo Robinson et. al. (2004):

- terceirização (*outsourcing*) ou comprar: neste cenário, uma empresa delega o controle sobre uma ou mais atividades para uma empresa externa à quem contratou o serviço. Geralmente é uma análise que envolve a decisão de *make-or-buy*, ou seja, desenvolver na própria empresa ou então delegar para um provedor externo de serviços. Tipicamente, segundo o autor, é uma das formas mais simples de ser implementada e a mais rápida de ser operacionalizada;

- *join-venture* ou colaboração: neste cenário, existe um acordo entre duas ou mais empresas onde, através da união de seus recursos, é criada uma nova entidade para executar um ou mais projetos por um certo período de tempo. Existe um controle nivelado sobre o projeto e sobre os recursos, impactando na redução de custos de todas as empresas envolvidas na *joint venture*;

- departamentos/subsidiárias da própria empresa (*wholly-owned subsidiaries* ou *insourcing*) ou desenvolver: neste último cenário, as empresas criam os seus próprios centros de desenvolvimento de software. Entre os motivos para que isto ocorra está um maior controle, maior flexibilidade, menores preços a longo prazo, além de manter a cultura interna da empresa. Segundo o autor, é o modelo que demanda uma maior complexidade de operacionalização e mais tempo.

Do ponto de vista de distância geográfica, a distribuição ocorre de duas formas:

- *offshore*, ou seja, em um país diferente da matriz da empresa contratante e do cliente. O desenvolvimento *offshore* pode ser realizado em algum centro de desenvolvimento da empresa contratada, ou através da contratação de serviços de uma empresa terceira, localizada em algum outro país. Esta forma de distribuição geográfica é recomendada para projetos que possuem um plano de projeto bem definido e um bom entendimento dos requisitos dos clientes;

- *onshore*, ou seja, no mesmo país onde estão localizados o cliente e a matriz da empresa. Neste caso, podem ocorrer duas situações: a primeira, onde o projeto é desenvolvido em um centro de desenvolvimento da empresa contratada, no mesmo país onde está o cliente, mas distante fisicamente do cliente (*offsite*). E a segunda, onde o desenvolvimento é realizado fisicamente no cliente (*onsite*). Neste caso, a empresa contratada utiliza seus recursos para suportar as atividades fisicamente no cliente (desde especificação de requisitos até implementação e testes). Enquanto que na primeira a vantagem é manter o desenvolvimento perto do cliente, a segunda garante ao cliente um maior controle, e é recomendada para projetos de missão crítica, que são sensíveis ao local onde são desenvolvidos, ou que necessitam uma constante atenção do cliente.

Buscando uma visão unificada das formas de relacionamento entre as empresas e a distribuição geográfica, a figura 1 traz um modelo adaptado de Robinson et. al. (2004).

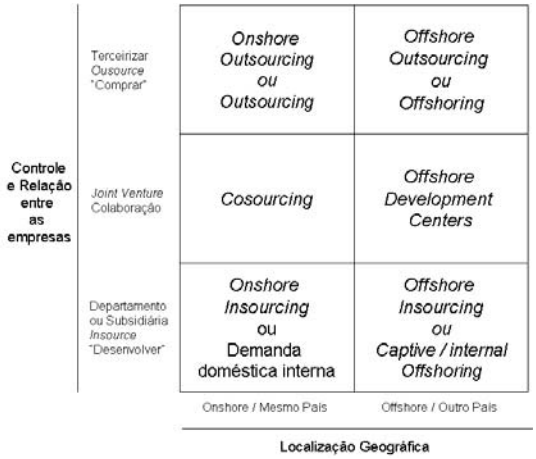


Figura 1. Modelos de Negócio para DDS, adaptado de Robinson et. al. (2004)

Como esta pesquisa teve seu foco direcionado para as relações de terceirização (*outsourcing* ou comprar) e de *insourcing* (desenvolver), as combinações relativas à *joint ventures* foram desconsideradas neste estudo, gerando assim um modelo simplificado, adaptado de Robinson et. al. (2004), representado na figura 2.

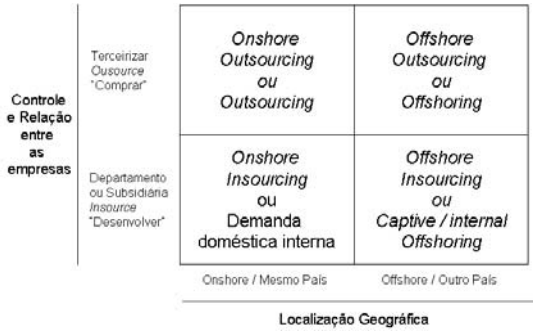


Figura 2. Modelos de Negócio estudados, adaptado de Robinson et. al. (2004)

Em relação aos modelos apresentados na figura 2, foco deste estudo, eles são assim definidos (Robinson et. al., 2004; Carmel et. al., 2005):

- *onshore insourcing* ou demanda doméstica interna: este é um dos mais simples e conhecidos pelos gerentes de projetos. Existe um departamento na própria empresa, ou uma subsidiária no mesmo país (*onshore*), que provê serviço de desenvolvimento de software, através de projetos internos (*insourcing*);

- *onshore outsourcing* ou *outsourcing*: este modelo de negócio indica a contratação de uma empresa terceira (*outsourcing*) para o desenvolvimento de determinados serviços ou produtos de software para uma empresa. A empresa terceira está localizada no mesmo país da empresa contratante (*onshore*);

- *offshore outsourcing* ou *offshoring*: este modelo de negócio indica contratação de uma empresa terceira (*outsourcing*) para o desenvolvimento de determinados serviços ou produtos de software, sendo que a empresa terceirizada está necessariamente localizada em um país diferente da contratante (*offshore*);

- *offshore insourcing* ou *captive / internal offshoring*: este úlltimo modelo de negócio indica criação de uma subsidiária da própria empresa para prover serviços de desenvolvimento de software (*insourcing*). Esta subsidiária está necessariamente localizada em um país diferente da matriz da empresa, ou empresa contratante (*offshore*).

2.2. Unificação dos conceitos envolvidos na operacionalização do DDS

O desenvolvimento *offshore* tem sido considerado uma das formas mais comuns para caracterizar o DDS. Segundo Schniederjans et. al. (2005), é também conhecido como *international sourcing* e não é algo novo, pois existe em áreas tais como manufatura há muito tempo. Mas ao longo dos últimos anos, percebeu-se um crescimento bastante significativo e acelerado em um âmbito global, do setor de serviços, tais como serviços de TI, e conseqüentemente o desenvolvimento de software (Bohem, 2006).

Isto faz com que as organizações se interessem por buscar soluções externas em outros países diferentes de onde está a matriz da empresa (*offshore sourcing* ou simplesmente *offshoring*), ou então buscar por soluções no mesmo país (terceirização ou *outsourcing*). Assim, o *international sourcing* também é conhecido e referenciado como *offshoring*. De acordo com Carmel et. al. (2005), *offshoring* é definido como a mudança de execução de um processo de negócio de uma empresa, que anteriormente era realizado no país onde está localizada a sua matriz (*onshore*), para um país estrangeiro, visando vantagens principalmente em relação ao baixo custo em um outro país. O *offshoring* de um processo de negócio independe de o mesmo continuar sendo executado pela mesma empresa (*insourcing*) ou por uma empresa terceira (*outsourcing*).

Freqüentemente, *offshoring* é confundido com *outsourcing*, terceirização, e *offshore outsourcing*. Além disso, estes quatro conceitos têm sido utilizados quase que com o mesmo significado, sem levar em consideração algumas importantes diferenças técnicas. Segundo Carmel et. al. (2005), *outsourcing*, em um contexto corporativo, representa a prática organizacional que envolve transferir uma função organizacional para um terceiro. Segundo Schniederjans et. al. (2005) e Morstead et. al. (2003), *outsourcing* não é um conceito novo, mas uma nova forma de se referenciar à terceirização de atividades e serviços para uma entidade externa especializada. E quando o terceiro está localizado em um outro país, configura o *offshore outsourcing*. Já o *offshoring* representa a transferência de uma função organizacional para um outro país, independente de ser executada dentro ou fora da organização.

Devido à quantidade de termos envolvidos neste cenário de DDS, muitas vezes um conceito é utilizado de forma equivocada. Um dos principais equívocos é pensar que todo o *offshoring* envolve *outsourcing*, ou que o *offshoring* é o *outsourcing* em um contexto global. Mas isto não é verdade. Enquanto *outsourcing* envolve terceirizar processos para uma entidade externa, *offshore* pode envolver tanto a terceirização quanto o desenvolvimento *in-house*, mas em outro país. Por isso, a definição de *offshoring* também inclui empresas que montam seus próprios centros de desenvolvimento, conhecidos como *dedicated captive centers*, em locais de baixo custo.

Em suma, *offshoring* ou *offshore sourcing* é similar ao *offshore outsourcing* quando as empresas contratam empresas terceiras em outros países, mas é diferente quando as empresas simplesmente transferem atividades para a mesma empresa em um

outro país, criando assim os centros de desenvolvimento dedicados (*dedicated captive centers*), conhecidos também como wholly-*owned subsidiary*. Desta forma, o *offshore insourcing* surge como uma opção onde existe a transferência de atividades ou operações para uma entidade interna. Geralmente é uma decisão realizada para manter o controle de certas competências consideradas críticas na empresa. Então, as empresas podem ter a terceirização ou o *outsourcing* sem que para isto elas precisem de uma estratégia *offshore*. Da mesma forma, é possível ter a estratégia *offshore* sem caracterizar a terceirização ou o *outsourcing*. Todas estas possibilidades têm seu reflexo nas atividades que compõem o desenvolvimento de software. Para cada cenário, novos desafios são identificados e novas estratégias para obter um melhor aproveitamento das técnicas de desenvolvimento de software são colocadas em prática.

3. Estudos de Caso: Análise de Práticas de DDS no Brasil e no Exterior

Nos últimos anos, diversas empresas estabeleceram estratégias de desenvolvimento distribuído de software no Brasil e no exterior. Alguns estudos realizados com estas empresas em separado (Sengupta et. al., 2006, Pilatti et. al., 2006; Lopes et. al., 2004; Prikladnicki et. al., 2006; Prikladnicki et. al., 2004; Prikladnicki et. al., 2003) constataram a existência de importantes desafios no processo de desenvolvimento. Mas nenhum destes estudos analisou em detalhes os desafios comparando-se os tipos de DDS implantados pelas empresas. Desta forma, este artigo procurou reunir dados de quatro empresas que possuem estratégias de DDS, procurando identificar desafios e estratégias para obter um melhor desempenho em projetos distribuídos. A análise foi feita sob a ótica da Engenharia de Software, da Qualidade de Software e da Gerência de Projetos de Software. O estudo e o método de pesquisa são detalhados a seguir.

3.1. Método de Pesquisa

Muito embora a ampla revisão teórica desenvolvida, não se tem conhecimento de que o problema apresentado tenha sido abordado sob a mesma perspectiva. Assim, esta pesquisa se caracteriza como um estudo predominantemente exploratório, sendo que o método de pesquisa principal foi o estudo de caso. Neste estudo, pode-se justificar o uso de métodos qualitativos, pois envolve o estudo de DDS no seu contexto real, com a compreensão do estado da arte onde a prática se antecipa à teoria (Creswell, 2003). O método de estudo de caso foi adotado conforme proposto por Yin (2001).

Foram identificados como elementos de análise desafios do DDS sob a ótica da Engenharia de Software, Qualidade de Software e Gerência de Projetos de Software. A coleta de dados foi realizada através de entrevistas *in loco* nas empresas com integrantes de equipes de projetos. O critério inicial para a definição dos entrevistados centrou-se na unidade de análise e nos objetivos do estudo. Neste sentido, foram entrevistados gerentes de projetos, gerentes de unidade e líderes técnicos, devido ao nível de interação destes papéis com colaboradores distribuídos geograficamente. A amostra foi não probabilística, por conveniência, e procurou-se, em conjunto com a organização, uma representatividade dos diversos grupos envolvidos.

Utilizou-se como instrumento de pesquisa um roteiro para uma entrevista semi-estruturada, com questões abertas. Ele foi desenvolvido a partir da teoria estudada e representada pelo protocolo de pesquisa gerado. As questões foram divididas em

tópicos, buscando uma abrangência de todo o processo de desenvolvimento. Além disso, foram desenvolvidos dois questionários, cada um explorando uma dimensão de informações para serem capturadas: dimensão organizacional, com informações da organização como um todo e dimensão de projetos, com informações relacionadas aos projetos vivenciados pelos entrevistados. Devido à limitação de páginas, os roteiros não foram anexos neste artigo, mas estão disponíveis mediante consulta aos autores.

As entrevistas foram realizadas em dois idiomas: inglês e português. Todas foram gravadas, transcritas e analisadas posteriormente, através de análise de conteúdo segundo Kripendorff (2004), buscando-se a replicabilidade e estabilidade dos dados. Para a análise, utilizaram-se as transcrições no idioma original. Buscou-se associar uma pergunta para cada trecho de resposta dada nas entrevistas. Por ser um roteiro semi-estruturado, em alguns casos houve perguntas adicionais, que foram incluídas nas análises. Para cada trecho de resposta, categorias foram identificadas, resultando em um conjunto de categorias, que por sua vez permitiram uma análise profunda do fenômeno.

As questões de qualidade de software basearam-se principalmente na existência e utilização de processos de desenvolvimento nas unidades distribuídas e na adoção de modelos de qualidade. Do ponto de vista da engenharia de software, as questões basearam-se nas técnicas e ferramentas utilizadas para melhorar o desempenho nos projetos distribuídos. Em relação à gerência de projetos, buscou-se avaliar as estratégias utilizadas para obter sucesso na gestão dos projetos distribuídos. Por tratar-se de uma pesquisa qualitativa, devem-se ter claras as limitações existentes, principalmente quanto ao número de empresas estudadas, restringindo a generalização dos resultados obtidos.

3.2. Caracterização das Empresas

O estudo foi realizado em quatro empresas ou respectivas unidades, sendo uma delas com matriz no Brasil, duas com matriz no Canadá e uma com matriz nos Estados Unidos. A tabela 1 caracteriza o estudo realizado e os tipos de relações de DDS que foram avaliadas em cada empresa, de acordo com a taxonomia proposta na figura 2.

Tabela 1. Caracterização das empresas

Empresa	País de origem	Subsidiária ou parceira	Local da coleta de dados	Modelos de Negócio analisados	Entrevistas
E1	Estados Unidos	Brasil Índia	Estados Unidos Brasil	*Offshore Insourcing* *Onshore Insourcing*	5
E2	Brasil	Brasil Estados Unidos	Brasil	*Onshore Outsourcing* *Offshore Outsourcing*	5
E3	Canadá	Índia	Canadá	*Offshore Outsourcing*	4
E4	Canadá	Índia França Inglaterra	Canadá	*Offshore Insourcing*	3

A empresa 1 (E1) é uma multinacional norte-americana de grande porte. O estudo foi aplicado tanto na matriz da empresa, nos Estados Unidos, quando na unidade brasileira, localizada no Estado do Rio Grande do Sul. Ela possui mais de 400 colaboradores trabalhando em projetos que atendem as necessidades da área de TI da empresa (demanda interna). A empresa possui dois modelos de negócio: o *offshore insourcig* e o *onshore insourcing*, pois a unidade no Brasil desenvolve projetos internos tanto para a matriz nos Estados Unidos quanto para uma filial no Brasil.

A empresa 2 (E2) é uma multinacional brasileira de desenvolvimento de software, com matriz em São Paulo. O estudo de caso foi desenvolvido na principal unidade de desenvolvimento de software, que possui em torno de 200 colaboradores. Com relação aos clientes, a unidade trabalha apenas com clientes externos à organização. Foram identificados dois modelos de negócio: o *onshore outsourcing*, pois a unidade estudada desenvolve projetos para clientes localizados no Brasil, e também o *offshore outsourcing*, pois a unidade desenvolve projetos para clientes no exterior.

A empresa 3 (E3) é uma multinacional canadense de desenvolvimento de software. O estudo de caso foi realizado na matriz da empresa no Canadá, que possui em torno de 100 colaboradores. Foi identificada a estratégia de *offshore outsourcing*, uma vez que a empresa terceiriza parte do desenvolvimento para uma empresa externa localizada na Índia. Esta empresa na Índia, quando os dados foram coletados, possuía em torno de 20 colaboradores trabalhando com parte da equipe no Canadá.

A empresa 4 (E4) é uma empresa de grande porte localizada no Canadá. O estudo foi realizado na unidade de desenvolvimento de software localizada na matriz da empresa, que possui 45 colaboradores. Foi identificada a estratégia de *offshore insourcing*, pois a empresa desenvolve produtos para vender no mercado, mas os clientes destes produtos são unidades de negócio da própria empresa, que identificam oportunidades nos escritórios de pelo menos quatro países diferentes (Tabela 1).

3.3. Resultados dos Estudos de Caso

Os estudos realizados consideraram os desafios de DDS sob o ponto de vista da Engenharia de Software, Qualidade de Software e Gerência de Projetos de Software. Além disso, considerou-se para análise o tempo de atuação das empresas em projetos de DDS. Desta forma, a empresa E1 atua há quatro anos, a empresa E2 há sete anos, a empresa E3 há três anos e a empresa E4 há um ano. A seguir são apresentados os dados coletados em cada uma das empresas, de acordo com o modelo de negócio analisado.

Engenharia de Software: durante as entrevistas, foram feitas perguntas específicas sobre técnicas e ferramentas utilizadas nas atividades de engenharia de software. Os principais resultados encontrados em cada empresa foram:

Empresa 1 (E1): do ponto de vista de *onshore insourcing*, foram comentadas dificuldades em relação ao entendimento dos requisitos. Neste modelo de negócio, a equipe de projeto não tinha muitos problemas de comunicação com o cliente devido ao idioma ser o mesmo e a distância geográfica não ser significativa. Por outro lado, existia uma tendência à falta de formalização de requisições de mudança, antes de a unidade investir na definição de processos de desenvolvimento e implantação de modelos de qualidade. Em relação ao modelo de *offshore insourcing*, foram apontados desafios bem diferentes. Como a equipe no Brasil interagia com um cliente que possuía conhecimento técnico nos Estados Unidos, houve conflitos em relação a padrões de código a serem utilizados, e diversos desentendimentos em relação aos requisitos a serem identificados, analisados, desenvolvidos e gerenciados. A arquitetura do software também foi uma dificuldade enfrentada pela equipe no Brasil, visto que exista a necessidade de se adequar a arquitetura já existente nos Estados Unidos. Por fim, identificou-se que a manutenção dos sistemas como um importante desafio, visto que a empresa estava recém iniciando um esforço de implantação de processos e gestão de conhecimento.

Empresa 2 (E2): do ponto de vista de *onshore outsourcing*, a unidade estudada terceirizou o desenvolvimento de certos projetos internos para empresas externas. Entre os principais desafios encontrados, está a necessidade de utilizar padrões de codificação definidos pela empresa. Como futuramente a unidade poderia decidir continuar o projeto internamente, ou ainda contratar uma outra empresa, foi necessário um treinamento inicial da empresa contratada nas técnicas e ferramentas de engenharia de software utilizadas. Em relação a outra estratégia existente, a de *offshore outsourcing*, a empresa tinha um cliente nos Estados Unidos, e decidiu desenvolver o projeto na unidade do Brasil. O principal desafio do projeto em relação à Engenharia de Software foi a engenharia de requisitos, principalmente na gestão dos requisitos. Houve diversas requisições de mudanças, motivadas principalmente por dificuldades de comunicação e de falta da correta documentação dos requisitos.

Empresa 3 (E3): nesta empresa foi analisado apenas o modelo de negócio *offshore outsourcing*, onde parte do trabalho foi terceirizado para uma empresa externa, na Índia. Foram citados como desafios a dificuldade em se unificar técnicas de desenvolvimento de software quando parte da equipe de projeto está distribuída. Especificamente, a empresa teve dificuldade com integração de diferentes módulos de um sistema, e com a padronização do desenvolvimento.

Empresa 4 (E4): nesta empresa foi analisado apenas o modelo de negócio *offshore insourcing*. E aqui o principal desafio em termos de Engenharia de Software foi o uso de ferramentas padrão para análise e projeto do sistema dentro da equipe distribuída. Além disso, como a empresa estava recém no início de sua experiência com projetos distribuídos, foi bastante comentada a falta de padronização entre as equipes.

Qualidade de Software: durante as entrevistas, foram feitas perguntas específicas sobre a existência de processos bem definidos, sobre como eram realizadas melhorias nos processos, caso existissem, sobre gerência de configuração, políticas da empresa para desenvolvimento de software, além de informações sobre a existência de práticas de gestão de conhecimento. Os resultados encontrados em cada empresa foram:

Empresa 1 (E1): para a estratégia de *onshore insourcing*, a empresa citou como desafio principal a dificuldade em definir um processo de desenvolvimento de software. Já para *offshore insourcing*, além da definição de um processo, surgiu com bastante importância a dificuldade de convergir dois ou mais processos diferentes em um conjunto de práticas para serem utilizadas em um projeto.

Empresa 2 (E2): tanto para a estratégia de *onshore outsourcing*, como para a estratégia de *offshore outsourcing*, os principais desafios encontrados em relação a qualidade de software foram idênticos. Eles se concentraram principalmente em questões de gerência de configuração, visto que as empresas contratada e contratante trabalhavam com ferramentas diferentes para tal finalidade. Além disso, a gestão do conhecimento também foi apontada como um desafio, visto que grande parte do conhecimento estava ficando na empresa contratada, e a empresa estudada recém estava criando estratégias para armazenar e disseminar as informações internamente.

Empresa 3 (E3): para a estratégia de *offshore outsourcing*, a empresa apontou como desafios principais a necessidade de convergir para um processo de desenvolvimento padrão entre as equipes distribuídas. Além disso, apesar de a

contratada na Índia possuir processos com base em modelos de qualidade de software tipo CMMI, os processos da empresa estudada no Canadá não eram baseados em modelos de qualidade, o que, segundo os entrevistados, contribuíram para as dificuldades vivenciadas. Além disso, comentou-se que a gerência de configuração foi um dos maiores desafios dos projetos desenvolvidos nesta estratégia, pois as empresas possuíam estruturas diferentes, e conseqüentemente ferramentas diferentes para desempenhar esta atividade. Além disso, as políticas das empresas para desenvolvimento de software eram sensivelmente diferentes.

Empresa 4 (E4): para a estratégia de *offshore insourcing*, a empresa identificou desafios relacionados principalmente a unificação dos processos nas unidades distribuídas. Além disso, citou que, para cada tipo e tamanho de projeto, um conjunto de processos era necessário, e isto necessariamente deveria ser feito no início da cada projeto. Na sua pouca experiência com projetos distribuídos, a empresa não possuía uma estratégia padrão de gerência de configuração, o que gerava problemas no desenvolvimento dos projetos. Um outro fator importante mencionado foi em relação ao ciclo de vida de desenvolvimento utilizado. A empresa acreditava que a estratégia de *offshore insourcing* era a única que permitia o uso de um ciclo iterativo e incremental, pois os colaboradores tinham um nível de interação maior do que uma estratégia de *offshore outsourcing*, por exemplo, e nesta última eles só enxergavam a possibilidade de desenvolver projetos seguindo um ciclo de vida do tipo cascata.

Gerência de Projetos de Software: durante as entrevistas, foram feitas perguntas sobre as atividades relacionadas com a gerência de projetos. Foram incluídas todas as dificuldades relacionadas com a integração da equipe de projeto, e fatores decorrentes da distribuição, entre eles os chamados aspectos não-técnicos (diferenças culturais, confiança, idioma, entre outros). Os principais resultados encontrados foram:

Empresa 1 (E1): para a estratégia de *onshore insourcing*, a empresa identificou desafios relacionados ao planejamento do projeto. Já para *offshore insourcing* os desafios se concentraram em questões tais como idioma diferente, diferenças culturais e falta de compartilhamento de contexto, além da necessidade de haver um processo de aquisição de confiança e gerenciamento de expectativas entre as equipes distribuídas.

Empresa 2 (E2): para a estratégia de *onshore outsourcing*, foram comentadas dificuldades e desafios relacionados diretamente às atividades da gerência do projeto, tais como planejamento, padronização de relatórios de acompanhamento entre outros. Já na estratégia de *offshore outsourcing*, surgiram alguns desafios adicionais, decorrentes da distribuição geográfica ser em outro país. Surgiram de forma bastante importante questões relacionadas à comunicação, e idioma diferente entre as equipes distribuídas, *awareness* (percepção do que cada colaborador está executando), além dos desafios já existentes relacionados às atividades da gerência de projetos. Adicionalmente, o desafio de adquirir confiança apareceu de forma idêntica ao encontrado na empresa 1. Um ponto interessante neste aspecto é que, mesmo estando a mais tempo trabalhando com DDS, o fato de trabalhar com clientes externos à empresa fez com que o processo de aquisição de confiança entre as equipes demorasse mais tempo em relação ao que ocorreu na empresa 1, cujos clientes eram internos.

Empresa 3 (E3): para a estratégia de *offshore outsourcing*, a empresa identificou alguns desafios importantes, tais como a necessidade de padronização de atividades

gerenciais, interfaces entre a empresa contratante e a empresa contratada e um maior controle das tarefas sendo realizadas. Além disso, comentou-se sobre as dificuldades com o idioma e a confiança entre as equipes distribuídas, e a necessidade de se realizar viagens no início dos projetos para aumentar a confiança das equipes, melhorando os processos de engenharia de requisitos e de planejamento do projeto.

Empresa 4 (E4): para a estratégia de *offshore insourcing*, a empresa citou como desafios principais a necessidade de ter ferramentas integradas de gestão de projeto, além da correta distribuição das atividades para os colaboradores e a percepção (*awareness*) sobre o que cada um está executando. Outro comentário foi a necessidade de caracterizar as equipes distribuídas como uma única equipe dentro de um projeto, o que não vinha ocorrendo devido o início recente da empresa neste cenário.

3.4. Análise dos Resultados Encontrados

Diversos desafios relacionados ao ambiente de DDS foram identificados nas empresas estudadas. A tabela 2 apresenta um resumo das principais características encontradas, de acordo com a estratégia avaliada, sob a ótica da Engenharia de Software.

Tabela 2. Resumo dos desafios de Engenharia de Software encontrados

	Empresa 1 (E1)	Empresa 2 (E2)	Empresa 3 (E3)	Empresa 4 (E4)
Onshore Outourcing		- Padrões de código - Definição de ferramentas		
Onshore Insourcing	- Gerência de Mudanças			
Offshore Outsourcing		- Engenharia de requisitos - Gerência de Mudanças	- Unificação de técnicas de desenvolvimento - Integração de sistema	
Offshore Insourcing	- Padrões de código - Engenharia de requisitos - Arquitetura do software - Manutenção de software			- Definição de ferramentas - Padrões de código e de desenvolvimento de software

A tabela 3 apresenta um resumo das principais características encontradas nas empresas, de acordo com a estratégia avaliada, sob a ótica da Qualidade de Software.

Tabela 3. Resumo dos desafios de Qualidade de Software encontrados

	Empresa 1 (E1)	Empresa 2 (E2)	Empresa 3 (E3)	Empresa 4 (E4)
Onshore Outourcing		- Gerência de configur. - Gestão do conhecimento		
Onshore Insourcing	- Definição de um processo desenvolvimento de software			
Offshore Outsourcing		- Gerência de configur. - Gestão do conhecimento	- Integração de processos - Gerência de configuração - Unificação de políticas de desenvolvimento - Ciclo de vida em cascata	
Offshore Insourcing	- Definição de um processo desenvolvimento de software - Integração de processos			- Integração de processos diferentes - Gerência de configur. - Ciclo de vida iterativo e incremental

A tabela 4 apresenta as principais características encontradas nas empresas, de acordo com a estratégia avaliada, sob a ótica da Gerência de Projetos de Software.

Tabela 4. Resumo dos desafios de Gerência de Projetos encontrados

	Empresa 1 (E1)	Empresa 2 (E2)	Empresa 3 (E3)	Empresa 4 (E4)
Onshore Outourcing		- Planejamento do projeto - Padronização de artefatos de gestão		
Onshore Insourcing	- Planejamento do projeto			
Offshore Outsourcing		- Comunicação - Idioma - Confiança - *Awareness*	- Padronização de artefatos de gestão - Idioma - Confiança - Controle do projeto	
Offshore Insourcing	- Idioma - Diferenças culturais - Contexto - Confiança			- Ferramentas integradas de gestão - *Awareness* - Alocação de atividades - Percepção de equipe

A partir da análise das tabelas apresentadas, algumas conclusões puderam ser identificadas. Do ponto de vista da Engenharia de Software, pode-se destacar que:

- A gerência de mudanças, a necessidade de padronização da atividade de codificação e a utilização de ferramentas para apoiar as atividades de ES distribuídas são desafios que não podem ser menosprezados, independente da estratégia de DDS;

- A engenharia de requisitos tem seus desafios mais acentuados quando a estratégia de DDS possui uma característica *offshore*;

Algumas conclusões também foram identificadas do ponto de vista da Qualidade de Software. Desta forma, pode-se destacar que:

- A gerência de configuração é um dos principais desafios, sendo que as dificuldades são acentuadas quando a estratégia envolve o *outsourcing*;

- De acordo com as empresas estudadas, a estratégia de DDS do tipo *outsourcing*, independente da localização geográfica, geralmente facilita o uso de um ciclo de vida em cascata. Por outro lado, a estratégia do tipo *insourcing* torna mais fácil a utilização de um ciclo de vida iterativo e incremental;

- Apesar de ser importante em todas as estratégias, a gestão de conhecimento do processo de desenvolvimento é mais crítica em estratégias do tipo *outsourcing*;

- A adoção de um processo único, ou a adaptação de processos dos diversos locais distribuídos é de extrema importância e faz a diferença em um projeto de DDS, independente da estratégia adotada pela empresa;

Por fim, também é possível destacar algumas conclusões do ponto de vista da Gerência de Projetos de Software, quais sejam:

- A necessidade de percepção (*awareness*) em relação ao que as equipes estão executando aumenta quando a estratégia tem a característica *offshore*. Nesta estratégia, questões relacionadas ao idioma, à cultura, à comunicação e à confiança são acentuadas;

- O planejamento do projeto, a padronização de artefatos, o processo de gestão, e a existência ferramentas são importantes, independente da estratégia de DDS adotada;

Além disso, um ponto bastante interessante que chamou a atenção durante a análise dos dados foi a relação entre o tamanho e a complexidade dos projetos com a necessidade de processos para obter melhores resultados na sua execução, no contexto de cada modelo de negócio ou estratégia de DDS. Em todas as empresas, questionou-se sobre a existência de processos durante todo o ciclo de desenvolvimento de software. Buscou-se então associar os processos com os tipos de projetos e respectivos modelos de negócios utilizados. Desta forma, como resultado, identificou-se que as empresas trabalhavam com dois tipos de projetos principais: projetos simples e projetos complexos. A classificação de cada tipo variava um pouco em cada empresa, mas era baseada em características idênticas. A complexidade dos projetos era avaliada em relação ao tamanho da equipe e ao número de horas contratadas. Com isso, foi possível desenvolver uma representação da necessidade dos processos de acordo com a complexidade dos projetos e com o modelo de negócio utilizado (Figura 3).

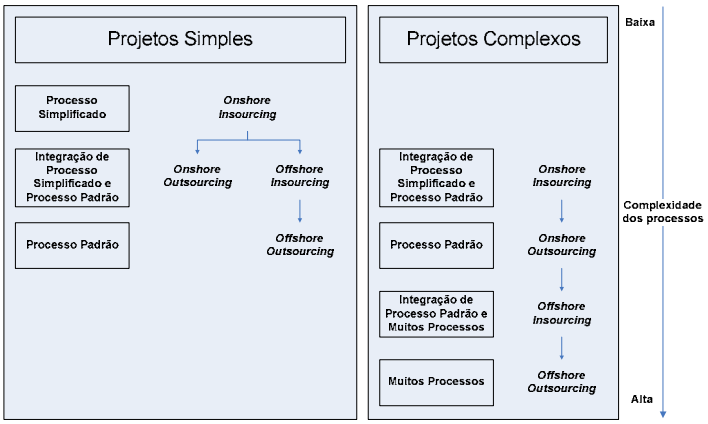


Figura 3. Complexidade dos Projetos X Modelo de Negócio X Processos

Identificaram-se três variações do conjunto de processos que eram utilizados nos projetos: processo simplificado (subconjunto de práticas do processo padrão), processo padrão (processo definido pela empresa), e muitos processos (criação de práticas adicionais além das existentes no processo padrão da empresa). Analisando a figura 3, é possível perceber que para projetos simples, as empresas utilizaram processos que variam de um processo simplificado para um modelo de *onshore insourcing*, até o uso de um processo padrão, no modelo de *offshore outsourcing*. Já em relação aos projetos complexos, identificou-se uma clara diferença no uso dos processos em cada modelo de negócio estudado, e uma necessidade de processos mais formalizados.

É interessante observar que para projetos simples não foi necessário o uso do que as empresas chamaram de muitos processos. Por outro lado, para projetos complexos, não se identificou o uso de um processo simplificado nos projetos estudados. Verificou-se ainda a necessidade do uso de uma grande carga de processos quando o modelo de negócio de desenvolvimento de software era caracterizado como *offshore outsourcing*. Então, do ponto de vista de necessidade de uso e complexidade do processo, tanto para projetos que as empresas consideraram simples, como para os projetos considerados complexos, o modelo de *onshore insourcing* representava o nível simples no uso dos processos, enquanto que o modelo de *offshore outsourcing* representou o nível mais crítico e com uma maior carga e necessidade de formalização de processos necessários para o bom andamento e conseqüente sucesso dos projetos.

4. Considerações Finais

O desenvolvimento de software sempre se apresentou de forma complexa. Existem diversos problemas e desafios inerentes ao processo. Em paralelo, a distribuição das equipes no tempo e no espaço tem tornado os projetos distribuídos cada vez mais comuns. O software é cada vez mais indispensável para a sociedade moderna, onde a globalização é uma característica fundamental. Neste contexto, o DDS surge como um grande desafio para a área de ES e tem levado os pesquisadores a defrontar-se com a necessidade de novos conhecimentos e uma abordagem mais multidisciplinar e até interdisciplinar. O estudo realizado buscou contribuir para aumentar a ainda pequena, mas crescente literatura da área, identificando desafios de quatro estratégias de DDS, comparando-se empresas no Brasil e no exterior, com diferentes tempos de experiência.

Os resultados deste estudo, assim como de estudos anteriores, apresentam oportunidades para aprofundar pesquisas voltadas para esta nova de classe de problemas que o DDS está trazendo (Sengupta et. al., 2006). Do ponto de vista científico, a legitimação dos resultados é decorrente do processo de pesquisa como um todo. Entre os principais resultados, constatou-se a semelhança de alguns desafios, independente da estratégia. Por outro lado, é clara a necessidade de um maior esforço durante todo o ciclo de vida de um projeto, quando a estratégia possui uma característica *offshore*.

Além disso, percebeu-se uma tendência cada vez maior de investimentos na criação de centros de desenvolvimento próprios, caracterizando assim uma operação do tipo *insourcing*. Isto ficou evidente ao analisar o tempo de experiência das empresas na área. Estudos anteriores (Carmel et. al., 2005) apontam o *offshore insourcing* como o último estágio a ser alcançado em uma representação de maturidade de modelos de negócio para DDS. Mas o estudo realizado mostrou que mesmo empresas atuando há apenas um ano neste contexto avaliam como positivo o investimento em *insourcing*.

Em relação à Engenharia de Software, cabe ressaltar que para cada modelo de negócio adotado, dificuldades diferentes podem surgir. Assim, recomenda-se estabelecer planos de ação no início de cada projeto, evitando surpresas ao longo do ciclo de vida. Especificamente em relação ao DDS no Brasil, Carmel et. al. (2005) identificam o Brasil como uma potência em ascensão. Segundo os autores, o país está no segundo time de possíveis destinos para operações distribuídas, mas em constante crescimento. Perde apenas para países tais como Índia e China. E em relação a práticas de DDS adotadas no Brasil e em outros países, nota-se uma maior experiência nos outros países. Por isto recomenda-se também o investimento na formação de profissionais para atuar globalmente e orientados para o DDS, evitando o foco apenas na formação técnica.

Como estudo futuro, pretende-se expandir a pesquisa para outras empresas e outros modelos de negócio. Pretende-se também identificar características essenciais que as empresas que investem em DDS possuem ao longo do tempo, desde o momento em que iniciam uma operação deste tipo, seja ela qual for, até atingir um estágio onde se pode perceber um amadurecimento significativo nas práticas de desenvolvimento de software neste contexto. Além disso, outras oportunidades envolvem o aprofundamento em temas específicos de DDS, associados a cada modelo de negócio explorado. Entre os temas mais críticos identifica-se engenharia de requisitos, gerência de risco, gerência de configuração, gestão de conhecimento, definição de processos globais, manutenção de software, projetos distribuídos em micro e pequenas empresas e teste de software.

Referências Bibliográficas

Bohem, B. (2006), "A View of 20th and 21st Century Software Engineering". In: 28th ICSE, Xangai, China.

Carmel, E., Tija, P. (2005), "Offshoring Information Technology: Sourcing and Outsourcing to a Global Workforce". UK: Cambridge.

Carmel, E. (1999), "Global Software Teams – Collaborating Across Borders and Time-Zones". Prentice Hall, EUA, 269p.

Creswell, J. W. (2003), "Research Design: Qualitative, Quantitative, and Mixed Methods Approaches". EUA: SAGE Publications.

Herbsleb, J. D., Moitra, D. (2001), "Global Software Development", IEEE Software, March/April, EUA, p. 16-20.

IDC – International Data Group (2006), Disponível em http://www.idc.com/, abril/2006.

Krippendorff, K. (2004), "Content analysis: an introduction to its methodology". EUA: Sage Publications.

Lopes, L., Prikladnicki, R., Audy, J. L. N. (2004), "Distributed Requirements Specification: Minimizing the Effect of Geographic Dispersion". In: VI ICEIS, Cidade do Porto, Portugal, p. 531-534.

Morstead, S., Blount, G. (2003), "Offshore Ready: Strategies to Plan & Profit from offshore IT-Enabled Services". EUA: ISANI Press.

Pilatti, L., Audy, J. L. N., Prikladnicki, R. (2006), "Software Configuration Management over a Global Software Development Environment: Lessons Learned from a Case Study". In: Workshop on Global Software Development for the Practitioners at ICSE, Xangai, China.

Prikladnicki, R., Evaristo, R., Audy, J. L. N., Yamaguti, M. H. (2006), "Risk Management in Distributed IT Projects: Integrating Strategic, Tactical, and Operational Levels". Aceito para publicação no International Journal of e-Collaboration, special issue on Collaborative Project Management.

Prikladnicki, R., Audy. Jorge L. N. (2004), "MuNDDoS: Um Modelo de Referência para Desenvolvimento Distribuído de Software", In: 18º SBES, Brasília, Brasil.

Prikladnicki, R., Audy, J. L. N., Evaristo, R. (2003), "Global Software Development in Practice: Lessons Learned", Journal of Software Process: Improvement and Practice – Special Issue on Global Software Development, Wiley, 8(4), Oct-Dec, p. 267-281.

Robinson, M., Kalakota, R. (2004), "Offshore Outsourcing: Business Models, ROI and Best Practices". EUA: Mivar Press.

Schniederjans, M. J., Schniederjans, M. A., Schniederjans, D. G. (2003), "Outsourcing and Insourcing in an International Context". EUA: M. E. Sharpe.

Sengupta, B., Chandra, S., Sinha, V. (2006) "A Research Agenda for Distributed Sofwtare Development". In: 28th ICSE, Xangai, China.

Yin, Robert. (2001), "Estudo de Caso: planejamento e métodos". SP: Bookman, 2001.

---

[i] Estudo realizado pelo grupo de pesquisa em Desenvolvimento Distribuído de Software, do PDTI, financiado pela Dell Computadores do Brasil Ltda., com recursos da Lei Federal Brasileira nº 8.248/91.

Aspectos para Construção de Aplicações Distribuídas

Cristiano Amaral Maffort e Marco Túlio de Oliveira Valente

Instituto de Informática
Pontifícia Universidade Católica de Minas Gerais
{maffort,mtov}@pucminas.br

Resumo. *Plataformas de middleware são largamente empregadas por engenheiros de software para tornar mais produtivo o desenvolvimento de aplicações distribuídas. No entanto, os sistemas atuais de middleware são, em geral, invasivos, pervasivos e transversais ao código de negócio de aplicações distribuídas. Neste artigo, apresenta-se um sistema de apoio à programação distribuída que encapsula em aspectos os serviços de distribuição providos por duas plataformas de middleware: Java RMI e Java IDL. O sistema proposto é suficientemente flexível e expressivo de forma a permitir o uso das principais abstrações providas por estas duas tecnologias de middleware, além de suportar diversas arquiteturas e configurações típicas de sistemas distribuídos.*
Palavras-chave: distribuição; middleware; orientação por aspectos; separação de interesses.

Abstract. *Software engineers often rely on middleware systems to improve the productivity in the design of distributed systems. However, middleware functionality is usually invasive, pervasive and tangled with business-specific concerns. In this paper, we describe a distributed programming system that encapsulates services provided by two middleware technologies: Java RMI and Java IDL. The proposed system is flexible and has enough expressive power to support common abstractions provided by object oriented middleware platforms. The system is also compatible with different distributed systems architectures and configurations.*
Keywords: distribution; middleware; aspect orientation; separation of concerns.

1 Introdução

Atualmente, plataformas de *middleware* são largamente empregadas por engenheiros de *software* para simplificar e tornar mais produtivo o desenvolvimento de aplicações distribuídas. Basicamente, estes sistemas encapsulam diversos detalhes inerentes à programação em ambientes de rede, incluindo protocolos de comunicação, heterogeneidade de arquiteturas, serialização de dados, sincronização, localização de serviços etc. No entanto, os sistemas atuais de *middleware* são, em geral, invasivos, pervasivos e transversais ao código de negócio de aplicações distribuídas [4, 14, 16, 11]. Normalmente, usuários de um sistema de *middleware* devem seguir uma série de convenções de programação, como, por exemplo, estender classes internas da plataforma, tratar exceções geradas pelo sistema, codificar interfaces remotas, invocar geradores de *stubs* etc. Como resultado, sistemas distribuídos baseados em *middleware* não apresentam graus esperados de reusabilidade, separação de interesses e modularidade, o que dificulta o desenvolvimento, manutenção e evolução dos mesmos.

Neste artigo, apresenta-se um sistema de apoio à programação distribuída, chamado DAJ (*Distribution Aspects in Java*). O sistema DAJ permite isolar a funcionalidade de distribuição – provida por plataformas de *middleware* – da lógica de negócio de aplicações implementadas em Java. Mais especificamente, DAJ inclui um *framework* para implementação de distribuição, o qual define classes e aspectos que encapsulam diversos detalhes de programação usualmente requeridos por plataformas de *middleware*. Além disso, o sistema DAJ inclui uma ferramenta de geração de código, responsável por gerar aspectos que especializam os aspectos abstratos do *framework* de forma a introduzir distribuição em uma determinada aplicação alvo. Ou seja, em DAJ, a adaptação e instanciação do *framework* de distribuição é feita de modo automático. O gerador de aspectos de DAJ gera código para duas plataformas de *middleware* nativas de ambientes de desenvolvimento Java: Java RMI [18] e Java IDL [5]. Java RMI é um *middleware* utilizado com freqüência por aplicações distribuídas cujos módulos são integralmente implementados em Java. A plataforma Java IDL, por sua vez, é uma implementação do padrão CORBA [13] e, portanto, permite que aplicações distribuídas implementadas em Java acessem sistemas desenvolvidos em outras linguagens de programação.

Em aplicações distribuídas implementadas com o apoio do sistema DAJ, classes de negócio não precisam seguir quaisquer protocolos de codificação, isto é, não precisam estender outras classes ou interfaces, não precisam implementar métodos `get` e `set`, não precisam ativar ou tratar exceções pré-definidas etc. Em vez disso, no sistema proposto, descritores de distribuição são usados para descrever o papel desempenhado por tais classes em um sistema de objetos distribuídos baseado em uma determinada tecnologia de *middleware*. Na solução proposta, descritores de distribuição são usados para declarar propriedades de um serviço (ou objeto) remoto, incluindo a interface implementada pelo serviço, a classe usada para instanciar o serviço, a plataforma de *middleware* usada para comunicação com o serviço e informações necessárias para registrar o serviço junto a um servidor de nomes. A partir das informações contidas em um descritor de distribuição, a ferramenta de geração de código integrante do sistema DAJ se encarrega de gerar automaticamente aspectos e classes que modularizam o código de distribuição requerido pela aplicação base. Os aspectos gerados por DAJ são implementados em AspectJ.

Acredita-se que o sistema proposto proporciona pelo menos três benefícios a um engenheiro de sistemas distribuídos. Primeiro, e mais importante, este engenheiro pode se concentrar exclusivamente no desenvolvimento do código funcional de sua aplicação, o qual não será mais entrelaçado com código de distribuição. Segundo, este engenheiro não precisa dominar detalhes e convenções de codificação requeridos por plataformas de *middleware*. Terceiro, este engenheiro não precisa dominar conceitos de programação orientada por aspectos, já que não será responsável por codificar de forma manual aspectos responsáveis por modularizar código de distribuição, o que quase sempre é uma tarefa tediosa e sujeita a erros.

O restante deste artigo está organizado conforme descrito a seguir. Inicialmente, na Seção 2, apresenta-se uma aplicação distribuída destinada a controlar e monitorar o preço das ações negociadas em uma bolsa de valores. Esta aplicação será usada ao longo de todo o artigo para motivar, apresentar e e validar a arquitetura do sistema DAJ. Em seguida, na Seção 3, descreve-se a interface de programação proposta por DAJ, com ênfase nos parâmetros que podem ser configurados por meio de descritores de distribuição. A

Seção 4 apresenta de forma detalhada as classes e aspectos do *framework* de distribuição proposto por DAJ, bem como as classes e aspectos gerados automaticamente pelo sistema de forma a especializar este *framework* e, conseqüentemente, introduzir código de distribuição em uma determinada aplicação. A Seção 5 avalia o projeto de DAJ. Nesta seção, compara-se também o sistema com uma outra solução para implementação de distribuição em AspectJ, proposta por Soares, Borba e Laureano [14]. A Seção 6 discute outros trabalhos relacionados e a Seção 7 conclui o presente artigo.

2 Exemplo Motivador: Sistema para Controle de Ações

A fim de ajudar a descrever e avaliar o funcionamento do sistema DAJ, será utilizado como exemplo no restante deste artigo uma aplicação distribuída para monitoramento e controle do preço de ações negociadas em uma determinada bolsa de valores. Este sistema inclui um servidor que armazena o preço corrente das ações negociadas nesta bolsa de valores. Clientes acessam este servidor com dois objetivos: (i) comunicar uma alteração no preço de venda de uma determinada ação e (ii) manifestar interesse em receber notificações de alterações no preço de determinada ação. Neste segundo caso, o servidor notifica o cliente por meio de um *callback*.

Apesar de simples, esta aplicação faz uso das principais abstrações providas por sistemas de *middleware* orientados por objetos. Essencialmente, ela utiliza chamadas remotas de métodos, passando objetos tanto por valor como por referência. Além disso, ela admite diversas configurações de implantação. Por exemplo, um cenário de implantação pode incluir um único servidor, utilizando Java RMI, e diversos clientes Java. Em outro cenário, este servidor deve utilizar CORBA, já que o mesmo será acessado por sistemas implementados em outras linguagens de programação. Em um terceiro cenário, pode existir um servidor RMI e um servidor CORBA, cada um deles controlando um subconjunto das ações negociadas na bolsa.

Descreve-se a seguir as principais interfaces e classes utilizadas na implementação deste sistema.

Interfaces: O serviço provido pelo servidor que armazena o preço corrente das ações negociadas na bolsa de valores possui a seguinte interface:

```
interface StockMarket {
  void update(StockInfo info);
  void subscribe(String stock, StockListener obj);
  void unsubscribe(String stock, StockListener obj);
}
```

Clientes publicam uma alteração no preço das ações de uma determinada companhia chamando o método `update`. A classe `StockInfo`, mostrada a seguir, é usada para armazenar o valor de uma ação em uma determinada data e hora.

```
class StockInfo {
  String stock;
  double value;
  String date_time;
}
```

Clientes podem ainda registrar interesse em serem notificados de alterações no preço das ações de uma determinada companhia. Para isso, devem utilizar o método `subscribe`, fornecendo como argumento o nome da ação e um objeto que receberá via *callback* a notificação. O método `unsubscribe` é usado para cancelar a assinatura realizada por meio do método `subscribe`. Em ambos os métodos, o objeto usado para receber a notificação deve implementar a interface `StockListener`, mostrada a seguir:

```
interface StockListener {
  void update(String stock, double value);
}
```

Implementação: Na implementação do sistema, a classe `StockMarketImpl` fica responsável por implementar a interface `StockMarket`. Por sua vez, a classe `StockListenerImpl` implementa a interface `StockListener` e, portanto, o método `update`. Ao contrário do que aconteceria se o sistema tivesse sido implementado diretamente sobre Java RMI ou Java IDL, as interfaces e classes de negócio descritas não possuem qualquer código de distribuição entrelaçado. Em outras palavras, com a utilização do sistema DAJ, permite-se que um projetista de *software* se concentre totalmente na lógica de negócios de sua aplicação.

3 Interface de Programação

Descreve-se nesta seção a interface de programação proposta por DAJ. Particularmente, são descritas as convenções que devem ser seguidas para especificação de descritores de distribuição e para codificação de clientes e servidores.

3.1 Descritores de Distribuição

Um dos objetivos do *framework* de distribuição proposto é ocultar do projetista de *software* a camada de *middleware* subjacente à aplicação distribuída que está sendo desenvolvida. Com isso, este projetista pode se concentrar na lógica de negócio do seu sistema, deixando a cargo do *framework* a implementação dos aspectos relacionados a distribuição. No entanto, este projetista precisa explicitar quais objetos da aplicação serão acessados remotamente. Para isso, utiliza-se um arquivo XML, chamado *descritor de distribuição*, por meio do qual são definidos diversos parâmetros de distribuição da aplicação.

Em um descritor de distribuição, são configurados os servidores que compõem a aplicação. Além disso, são definidos os tipos que, em chamadas remotas de métodos, são passados por referência e por valor. No caso de servidores (nodo `server` do arquivo XML), deve-se definir o identificador do servidor, o nome de sua interface de negócio, o nome da classe que implementa esta interface, o protocolo de comunicação a ser utilizado e o nome e a porta do servidor de nomes onde o servidor será registrado. No caso de tipos passados por referência (nodo `remote`), deve-se informar a interface e a classe deste tipo. No caso de tipos passados por valor (nodo `serializable`), deve-se informar a classe deste tipo[1].

Exemplo: Mostra-se a seguir um exemplo de descritor de distribuição para o Sistema de Controle de Ações.

[1] Por questões de espaço, este artigo não apresenta o esquema do documento XML usado como descritor de distribuição. No entanto, acredita-se que o exemplo a seguir transmite uma idéia bastante completa das informações que devem ser codificadas neste arquivo.

```
 1:  <server id="StockMarketA">
 2:    <interface>stockMarket.StockMarket</interface>
 3:    <class>stockMarket.StockMarketImpl</class>
 4:    <protocol>corba</protocol>
 5:    <nameserver>skank.pucminas.br</nameserver>
 6:  </server>
 7:  <server id="StockMarketB">
 8:    <interface>stockMarket.StockMarket</interface>
 9:    <class>stockMarket.StockMarketImpl</class>
10:    <protocol>javarmi</protocol>
11:    <nameserver>patofu.pucminas.br:1530</nameserver>
12:  </server>
13:  <remote>
14:    <interface>stockMarket.StockListener</interface>
15:    <class>stockMarket.StockListenerImpl</class>
16:  </remote>
17:  <serializable>
18:    <class>stockMarket.StockInfo</class>
19:  </serializable>
```

Por meio deste descritor, configura-se uma implantação do sistema com dois servidores: o primeiro deles utilizará Java IDL para comunicação e será registrado com o nome `StockMarketA` (linhas 1 a 6); o segundo utilizará Java RMI para comunicação e será registrado com o nome `StockMarketB` (linhas 7 a 12). Além disso, define-se que, em chamadas remotas de métodos destes servidores, objetos do tipo `StockListenerImpl` serão passados por referência (linhas 13 a 16) e que objetos do tipo `StockInfo` serão passados por valor (linhas 17 a 19).

3.2 Codificação de Clientes

Em sistemas distribuídos apoiados por plataformas de *middleware* orientadas por objetos, aplicações cliente obtêm referências para objetos remotos e então chamam métodos dos mesmos. Nestas chamadas, parâmetros podem ser passados por cópia (via serialização) ou então por referência (quando se passa o *stub* do parâmetro de chamada). Em DAJ, um cliente deve utilizar o método `getReference` do *framework* para obter uma referência para um dos servidores configurados no descritor de distribuição do sistema. A partir da referência obtida, o cliente pode chamar métodos remotos deste servidor, sem precisar estar ciente do *middleware* que suporta esta comunicação.

Exemplo: Mostra-se a seguir um fragmento de código de um cliente do Sistema de Controle de Ações.

```
1: StockMarket s1,s2;
2: s1=(StockMarket)ServiceLocator.getReference("StockMarketA");
3: s2=(StockMarket)ServiceLocator.getReference("StockMarketB");
4:   ....
5: StockInfo info;
6: info= new StockInfo("cvrd", 124.60, "10/04/2006 18:21");
7: s1.update(info);
8: .......
9: StockListener listener= new StockListenerImpl();
```

```
10: s1.subscribe("bb", listener);
11: s2.subscribe("cef", listener);
```

Nas linhas de 1 a 3, por meio do método `getReference`, este cliente obtém referências para os servidores de nome `StockMarketA` e `StockMarketB`, definidos no descritor de distribuição mostrado no exemplo da Seção 3.1. Em seguida, o cliente chama o método `update` do primeiro servidor passando um objeto do tipo `StockInfo` por valor (linhas 5 a 7). Por fim, o cliente manifesta seu interesse em receber notificações sobre alterações nos preços de duas ações (linhas 9 a 11). Veja que em ambas chamadas do método `subscribe`, o cliente informa o mesmo objeto do tipo `StockListener`. Este objeto, passado sempre por referência, receberá notificações do primeiro servidor via Java IDL e do segundo servidor via Java RMI.

3.3 Codificação de Servidores

Conforme afirmado na Seção 2, as interfaces e classes de negócio de uma aplicação distribuída construída sobre a plataforma DAJ não possuem código de distribuição. Assim, seus desenvolvedores não precisam se preocupar com convenções de programação requeridas por uma determinada plataforma de *middleware*. Além disso, para cada servidor definido em um descritor de distribuição, o sistema DAJ gera automaticamente uma classe de ativação, a qual possui um método `main` contendo código para instanciar, ativar e registrar o respectivo objeto remoto. Esta classe possui o nome do servidor, prefixado com a palavra `Server`.

Exemplo: Suponha o descritor de distribuição mostrado na Seção 3.1. Para instanciar, ativar e registrar os servidores especificados neste descritor, o sistema DAJ gera automaticamente as seguintes classes: `ServerStockMarketA` e `ServerStockMarketB`.

4 Arquitetura Interna

Descreve-se nesta seção a arquitetura interna do sistema DAJ, enfatizando os aspectos abstratos propostos pelo sistema e os aspectos concretos, gerados para estender estes últimos a partir das informações contidas em descritores de distribuição. Inicia-se a seção descrevendo-se o processo de geração de interfaces remotas e, em seguida, são descritos os aspectos do *framework* responsáveis por transformar classes de negócio em classes remotas, obter referências para objetos remotos e ativar servidores.

4.1 Interfaces Remotas

Em DAJ, interfaces remotas são interfaces com código de distribuição entrelaçado. A partir das informações lidas do descritor de distribuição, o sistema DAJ gera automaticamente interfaces remotas conforme requerido pelas plataformas Java RMI ou Java IDL. Em ambos os casos, estas interfaces têm o mesmo nome das interfaces de negócio especificadas no descritor de distribuição. No entanto, para se evitar colisões, interfaces remotas são geradas em um pacote separado.

Interfaces Remotas em Java RMI: As interfaces remotas requeridas por Java RMI são semelhantes às suas respectivas interfaces de negócio, exceto pelo fato de estenderem `java.rmi.Remote` e de todos os seus métodos declararem que podem ativar uma exceção do tipo `java.rmi.RemoteException`. O código destas interfaces remotas deve ser gerado pelo sistema DAJ já que os recursos de transversalidade estática de

AspectJ não permitem acrescentar uma cláusula `throws` na assinatura de métodos. Esta mesma dificuldade já foi reportada em outros trabalhos [14, 16].

Outra diferença entre interfaces de negócio e interfaces remotas diz respeito a parâmetros passados por referência. Mais especificamente, se um método de uma interface de negócio possui um parâmetro de um tipo `T` declarado como remoto no descritor de distribuição do sistema, então, na interface remota gerada, o tipo `T` deve ser trocado pelo tipo de sua respectiva interface remota. Em chamadas de métodos, o *run-time* de Java RMI se encarrega de passar o *stub* do parâmetro de chamada quando o parâmetro formal é de um tipo remoto. Como o *stub* gerado automaticamente pela implementação de Java RMI implementa a interface remota, ele não seria compatível para atribuição com um parâmetro formal declarado como sendo do tipo de uma interface de negócio. Daí a necessidade da substituição descrita neste parágrafo[2].

A fim de ilustrar a geração de interfaces em RMI, mostra-se a seguir a interface remota gerada para a interface de negócio `StockMarket`, apresentada na Seção 2:

```
interface StockMarket extends Remote {
  void update(StockInfo info) throws RemoteException;
  void subscribe(String stock, daj.sca.rmi.StockListener obj)
       throws RemoteException;
  void unsubscribe(String stock, daj.sca.rmi.StockListener obj)
       throws RemoteException;
}
```

A interface remota apresentada estende `Remote`. Além disso, todos os seus métodos ativam `RemoteException`. Por fim, o tipo do parâmetro `obj` dos métodos `subscribe` e `unsubscribe` foi substituído pelo tipo da interface remota associada (a qual, apesar de não mostrada no exemplo, também é gerada automaticamente por DAJ).

Interfaces Remotas em Java IDL: Como usual em aplicações baseadas em CORBA, interfaces remotas devem ser codificadas em IDL (isto é, em uma linguagem neutra proposta por CORBA especificamente para definição de interfaces). Portanto, o sistema DAJ assume que já existe uma especificação em IDL das interfaces de negócio definidas no descritor de distribuição do sistema. Esta suposição é bastante razoável, pois a definição de interfaces IDL é normalmente o ponto de partida de qualquer desenvolvimento baseado em CORBA. Em seguida, o sistema se encarrega de chamar o compilador `idlj`, o qual integra a plataforma Java IDL. Este compilador gera diversas classes e interfaces utilizadas na implementação de aplicações distribuídas baseadas em Java IDL. Em implementações que não utilizam DAJ, o uso destas classes fica entrelaçado com a implementação das classes de negócio do sistema.

A fim de ilustrar a geração de interfaces em Java IDL, mostra-se a seguir a interface `StockMarketOperations`, gerada pelo compilador `idlj`. Em sistemas baseados em Java IDL, esta interface deve ser implementada por uma classe de negócio.

[2]Suponha que `A` e `ARMI` são, respectivamente, interfaces de negócio e remotas. Uma possível alternativa para a referida substituição seria, via declaração intertipos, fazer `A` estender `Remote` e `ARMI` estender `A`. No entanto, esta alternativa é inviável, pois em Java não se permite que um método redefinido em uma sub-interface acrescente exceções verificadas à cláusula `throws` do método sobrescrito.

```
interface StockMarketOperations {
 void update(daj.sca.corba.StockInfo info);
 void subscribe(String stock, daj.sca.corba.StockListener obj);
 void unsubscribe(String stock,daj.sca.corba.StockListener obj);
}
```

É interessante notar que, apesar de criada automaticamente pelas ferramentas de geração de código de Java IDL, esta interface também adota a substituição de tipos de negócio por tipos remotos, conforme proposto para interfaces remotas em Java RMI.

4.2 Classes Remotas

Em DAJ, uma classe remota é aquela resultante da introdução, em uma classe de negócio, de código de distribuição requerido por uma determinada plataforma de *middleware*. Dentre este código, por exemplo, encontra-se aquele responsável por informar a interface remota que é implementada pela classe. Na implementação de DAJ, faz-se uma distinção entre classes remotas cujas instâncias serão registradas em um servidor de nomes e classes remotas cujas instâncias não são registradas em um servidor de nomes. As primeiras são sempre declaradas em um nodo `server` de um descritor de distribuição. Como exemplo, podemos citar a classe `StockMarketImpl`. As últimas são declaradas em um nodo `remote`, sendo usadas, portanto, para instanciar objetos que serão passados por referência em chamadas remotas de métodos. Como exemplo, podemos citar a classe `StockListenerImpl`.

A seguir, descreve-se como estes dois tipos de classes remotas são instrumentadas com código de distribuição requerido por Java IDL. A introdução de código de comunicação requerido por Java RMI segue metodologia semelhante à que será proposta para Java IDL e, portanto, não será discutida no presente artigo.

Classes Remotas usadas para Instanciação de Servidores em Java IDL: Como qualquer classe remota, estas classes devem implementar a interface remota gerada por DAJ. Além disso, para cada método desta interface que tenha um parâmetro `T` que seja do tipo de uma interface remota, deve-se introduzir na classe um método correspondente. O código deste método deve chamar o método já implementado na classe contendo como parâmetro o tipo da interface de negócio associada a `T`.

Para ilustrar, mostra-se abaixo o aspecto gerado por DAJ para transformar a classe de negócio `StockMarketImpl` em uma classe remota.

```
 1: aspect RemoteStockMarketImpl {
 2:  declare parents:
 3:     StockMarketImpl implements StockMarketOperations;
 4:
 5:  public void StockMarketImpl.subscribe(String stock,
 6:       daj.sca.corba.StockListener obj) {
 7:    subscribe(stock, new StockListenerAdapter(obj));
 8:  }
 9:  ...
10: }
```

Nas linhas 2 e 3, indica-se que a classe de negócio `StockMarketImpl` deve implementar a interface remota `StockMarketOperations` (mostrada na Seção 4.1).

Como o método `subscribe` desta interface possui um parâmetro formal `obj` cujo tipo é uma interface remota, cuida-se de introduzir uma implementação para este método na classe `StockMarketImpl` (linhas 5 a 8). Esta introdução é necessária, pois o método de negócio `subscribe` implementado pelo usuário na classe `StockMarketImpl` espera um parâmetro do tipo de negócio `StockListener` (e não do tipo remoto associado a `StockListener`, conforme especificado em `StockMarketOperations`).

Na linha 7 do aspecto mostrado anteriormente, o código do método `subscribe` redireciona a chamada para o método de negócio de mesmo nome (o qual foi codificado pelo desenvolvedor da classe `StockMarketImpl`). Para que esse redirecionamento ocorra, no entanto, deve-se instanciar um objeto adaptador de uma classe similar à mostrada abaixo:

```
1: class StockListenerAdapter implements StockListener {
2:   daj.sca.corba.StockListener adaptee;
3:   StockListenerAdapter (daj.sca.corba.StockListener adaptee){
4:     this.adaptee= adaptee;
5:   }
6:   public void update(String stock, double value) {
7:      adaptee.update(stock,value);
8:   }
9: }
```

Este adaptador adapta a interface `StockListener` à sua respectiva interface remota.

Classes Remotas usadas para Instanciação de Objetos Remotos em Java IDL: Neste caso, realiza-se o mesmo processo de introdução de requisitos transversais descrito anteriormente. No entanto, adicionalmente, são introduzidos dois novos membros nas classes de negócio associadas: um método responsável por realizar a ativação de objetos destas classes e uma referência para o *stub* retornado pelo processo de ativação[3]. Basicamente, o método introduzido consulta o valor desta referência. Se ela for nula, realiza-se a ativação do objeto remoto e o *stub* associado ao mesmo é atribuído à referência.

Para ilustrar, mostra-se a seguir o aspecto gerado por DAJ para transformar a classe de negócio `StockListenerImpl` em uma classe remota.

```
 1: aspect RemoteStockListenerImpl {
 2:   declare parents:
 3:      StockListenerImpl implements StockListenerOperations;
 4:
 5:   StockListener StockListenerImpl.refIDL= null;
 6:
 7:   StockListener StockListenerImpl.export2IDL() {
 8:     // ativa objeto segundo as convenções de Java IDL
 9:   }
10: }
```

[3]Em Java IDL, objetos remotos devem ser sempre ativados após serem instanciados, independentemente de serem ou não registrados em um servidor de nomes. Objetos são ativados chamando-se o método `activate`, o qual retorna uma referência para o *stub* associado ao objeto. Esta referência deve, no restante do programa, ser usada no lugar da referência original para o objeto.

Nas linhas 2 e 3, indica-se que a classe de negócio `StockListenerImpl` deve implementar a interface remota `StockListenerOperations` (gerada automaticamente pelo compilador `idlj`). Na linha 5, introduz-se a referência para o *stub* associado a instâncias desta classe. Nas linhas 7 a 9, introduz-se o método `export2IDL`, o qual é responsável por realizar a ativação de objetos desta classe.

4.3 Obtenção de Referências Remotas

Conforme afirmado na Seção 3.2, clientes conseguem referências para servidores remotos declarados em um descritor de distribuição chamando o método `getReference`. A implementação deste método realiza o *parser* do documento XML que representa o descritor de distribuição, identifica o servidor para o qual está sendo solicitada uma referência e efetua uma consulta ao servidor de nomes de Java RMI ou Java IDL.

No entanto, o método `getReference` devolve para o cliente não a referência para o objeto retornado pela consulta ao servidor de nomes, mas uma referência para um *proxy* deste objeto. Este *proxy*, cuja classe é gerada pela implementação de DAJ, funciona como um representante local do objeto remoto e têm duas funções: tratar exceções remotas lançadas pela platatorma de *middleware* em caso de falhas de comunicação e ativar o objeto remoto, quando o mesmo é utilizado como argumento de uma chamada remota de método. Para realizar esta ativação, o *proxy* utiliza o método `export2IDL` ou `export2RMI` introduzido em classes remotas pelo aspecto descrito na Seção 4.2.

Mostra-se a seguir o *proxy* gerado quando o cliente solicita uma referência para um servidor da classe `StockMarketImpl` e que utiliza Java IDL para comunicação.

```
 1: class StockMarketImplProxy implements StockMarket {
 2:   daj.sca.corba.StockMarket server;
 3:   .....
 4:   void subscribe(String stock, StockListener obj) {
 5:     try {
 6:       daj.sca.corba.StockListener _obj;
 7:       _obj= ((StockListenerImpl) obj).export2IDL();
 8:       server.subscribe(stock, _obj)
 9:     }
10:     catch(Exception e) { ... }
11:   }....
12: }
```

No método `subscribe` da classe *proxy* mostrada, ativa-se o parâmetro `obj` (linha 7), antes de efetivamente utilizá-lo na chamada remota (linha 8). Esta ativação é requerida por Java IDL, já que este parâmetro é remoto.

4.4 Ativação de Objetos Servidores

Conforme afirmado na Seção 3.3, em DAJ existem aspectos responsáveis pela instanciação, ativação e registro de objetos servidores, de forma que usuários do sistema não precisam se preocupar com a codificação destas tarefas.

No caso específico de Java RMI, o seguinte aspecto abstrato é usado para definir *pointcuts*, *advices* e métodos relacionados com a implementação de tais funções:

```
 1: abstract aspect RMIServer {
 2:   abstract pointcut ServerMainExecution();
 3:   abstract String getServerName();
 4:   abstract String getRegistry();
 5:   abstract Remote getInstance();
 6:
 7:   void around(): ServerMainExecution() {
 8:       // instancia, ativa e registra objetos remotos em RMI
 9:   }
10: }
```

Neste aspecto, o *pointcut* abstrato `ServerMainExecution` denota os pontos do programa onde deve-se instanciar um objeto servidor (linha 2). O método abstrato `getServerName` é usado para obter o nome com o qual o objeto servidor será registrado no servidor de nomes de Java RMI (linha 3). O *host* do servidor de nomes é retornado pelo método `getRegistry` (linha 4). Já o método `getInstance` é usado para se obter uma instância do servidor (linha 5). Por fim, um *advice* associado ao *pointcut* `ServerMainExecution` se encarrega de instanciar, registrar e ativar um objeto remoto segundo as convenções de Java RMI (linhas 7 a 9).

Em DAJ, existe ainda um aspecto abstrato chamado `CORBAServer`, o qual é semelhante ao aspecto `RMIServer`. No entanto, neste aspecto, o *advice* associado ao *pointcut* `ServerMainExecution` instancia e registra um objeto remoto segundo a interface de programação de Java IDL.

Exemplo: Mostra-se a seguir o aspecto concreto `RMIServerStocketMarketB` gerado automaticamente pelo sistema DAJ para instanciar e registrar o servidor `StockMarketB` (definido no descritor de distribuição mostrado na Seção 3.1).

```
 1: aspect RMIServerStockMarketB extends RMIServer {
 2:  pointcut ServerMainExecution:
 3:   execution(public static void ServerStockMarketB.main(..));
 4:  String getServerName() {
 5:    return "StockMarketB";
 6:  }
 7:  String getRegistry() {
 8:    return "patofu.pucminas.br:1530";
 9:  }
10:  StockMarket getInstance() {
11:    return new StockMarketImpl();
12:  }
13: }
```

Como pode ser observado, os parâmetros que diferenciam este código (tais como, nome do servidor, endereço do *registry* e nome da classe do servidor) são todos obtidos do descritor de distribuição.

5 Avaliação

O desenvolvimento de DAJ foi inspirado no trabalho de Soares, Borba e Laureano, visando a implementação de aspectos de distribuição em AspectJ. Em [14], estes autores

descrevem uma experiência bem sucedida de reestruturação de uma aplicação real, chamada *HealthWatcher*, usada por cidadãos para enviar reclamações sobre restaurantes e similares a órgãos públicos responsáveis pela vigilância sanitária destes estabelecimentos. No referido trabalho, os autores mostram como este sistema foi reestruturado de forma a modularizar em aspectos o interesse de distribuição, representado por código de comunicação via Java RMI. Na versão original do sistema, este código encontrava-se espalhado e entrelaçado nas classes de negócio da aplicação. Além de distribuição, os autores apresentam também uma solução para modularização de persistência.

Assim, inicia-se esta seção de avaliação com uma descrição das principais contribuições que DAJ introduz no método de implementação proposto em *HealthWatcher*:

- DAJ oferece uma solução para modularização de código de distribuição requerido tanto por Java RMI como por Java IDL, enquanto que em *HealthWatcher* propõe-se uma solução apenas para Java RMI. Do ponto de vista de projeto, o principal desafio enfrentado para prover suporte a uma segunda plataforma de *middleware* foi evitar incompatibilidades e colisões originadas por aspectos específicos de cada uma das plataformas. Por exemplo, no caso de declarações intertipos, optou-se sempre por usar declarações que definem que classes da aplicação alvo devem implementar interfaces e não estender classes, já que Java não possui herança múltipla. O resultado final foi uma solução simétrica e ortogonal para implementação modularizada de distribuição.

- DAJ permite passagem de objetos por referência em chamadas remotas de métodos. Permite também que chamadas remotas retornem referências para objetos remotos. Estes recursos são normalmente usados por aplicações distribuídas interessadas em serem notificadas via *callback* da ocorrência de determinados eventos[4]. Já o método de implementação proposto em *HealthWatcher* inclui suporte apenas a passagem de objetos por valor.

- DAJ permite a geração de aspectos de distribuição independentemente da arquitetura de *software* adotada pela aplicação base. Por outro lado, a solução proposta em *HealthWatcher* é restrita e limitada pela arquitetura deste sistema. Por exemplo, assume-se em *HealthWatcher* que existe um único objeto servidor (do tipo `HWFacade`). Além disso, chamadas remotas codificadas em clientes são sempre capturadas usando-se este tipo. O resultado é que o *framework* proposto em *HealthWatcher* não é compatível, por exemplo, com clientes que possuam múltiplas referências remotas do mesmo tipo, cada uma delas referenciando objetos remotos distintos. Em DAJ, esta situação pode ser modelada facilmente em um descritor de distribuição.

Em [15], Soares descreve uma ferramenta que automaticamente gera aspectos que concretizam os aspectos abstratos propostos em *HealthWatcher*. No entanto, a ferramenta

[4] A fim de avaliar a freqüência com que aplicações distribuídas utilizam passagem por referência, foi realizada uma pesquisa com os oito projetos de maior atividade que utilizam Java RMI hospedados no repositório de código livre `sourceforge.net`. Esta pesquisa mostrou que quatro destes aplicativos fazem uso de passagem por referência: BlackJackRMI, ChatX, Java3D Distributed Environment e RMI-MessageQueue. Não fazem uso de passagem por referência os seguintes projetos: Load4J, m-e-c Scheduler, RMIJavaChat e Virtual Sharing. Esta pesquisa acessou a versão mais recente destes projetos em 17/04/2006.

proposta é específica para sistemas que seguem a mesma arquitetura de *software* adotada em *HealthWatcher*.

A seguir, são discutidas e avaliadas outras características consideradas como contribuições importantes em DAJ:

- Suporte simultâneo a duas plataformas de *middleware*: DAJ permite que uma determinada classe servidora seja instrumentada tanto com código de comunicação Java RMI quanto com código requerido por Java IDL. Assim, uma mesma instância desta classe pode responder a chamadas remotas provenientes tanto de sistemas RMI como de sistemas CORBA. Um exemplo desta situação foi mostrado no exemplo da Seção 3.2, onde um mesmo objeto do tipo `StockListener` é usado para receber *callbacks* provenientes de um servidor Java RMI e de um servidor Java IDL. Este suporte simultâneo a múltiplos protocolos de comunicação foi alcançado graças ao projeto dos aspectos abstratos do sistema, no qual se conseguiu evitar interferências entre aspectos responsáveis por interesses transversais demandados por plataformas de *middleware* distintas.

- Reconfigurações em clientes e servidores: DAJ permite que parâmetros de distribuição usados por clientes e servidores sejam reconfigurados sem requerer manutenções ou recompilações dos módulos envolvidos. As reconfigurações possíveis incluem alterações no endereço do servidor de nomes e no protocolo de comunicação utilizado por um servidor. Estas reconfigurações afetam apenas o descritor de distribuição da aplicação.

- *Obliviousness* e Transparência: Plataformas de *middleware* têm como objetivo tornar transparentes detalhes de programação em redes. Já o sistema DAJ tem como objetivo modularizar o código invasivo e intrusivo, requerido pelas próprias plataformas de *middleware*. Portanto, a pergunta que resta responder é a seguinte: DAJ também não introduz novas convenções invasivas de programação, que tornem aplicações distribuídas dependentes do próprio sistema DAJ? Felizmente, o grau de dependência e entrelaçamento do sistema é nulo, no caso de módulos servidores. No caso de módulos clientes este grau é mínimo, resumindo-se ao uso do método `getReference`.

No entanto, a dependência introduzida por DAJ em clientes, mencionada no último item, pode ser eliminada criando-se um aspecto adicional, conforme ilustra o seguinte exemplo:

```
 1: class MyClient {
 2:   StockMarket s1,s2;
 3:   ....
 4: }
 5: aspect MyClientDependencyInjection {
 6:   after(MyClient c):
 7:     execution(void MyClient.new(..)) && this(s) {
 8:     c.s1= ServiceLocator.getReference("StockMarketA");
 9:     c.s2= ServiceLocator.getReference("StockMarketB");
10:   }
11: }
```

O aspecto proposto, a exemplo do que ocorre em *frameworks* de injeção de dependência [8], é responsável por obter as referências remotas associadas às variáveis de instância `s1` e `s2`. Com isso, a classe cliente `MyClient` não possui mais qualquer referência a componentes do sistema DAJ.

6 Trabalhos Relacionados

Diversos projetos de pesquisa têm investigado o uso de aspectos para modularizar serviços transversais providos por plataformas de *middleware*. Por exemplo, já foram desenvolvidas ferramentas baseadas em aspectos para auxiliar na implementação de aplicações EJB, tais como Gotech [16] e AspectJ2EE [3]. Gotech é um *framework* que gera aspectos capazes de transformar classes Java em componentes EJB. AspectJ2EE é uma implementação orientada por aspectos de um *container* que disponibiliza serviços não-funcionais típicos de servidores EJB.

Outros trabalhos têm como objetivo incorporar abstrações para programação distribuída em linguagens orientadas por aspectos. DJcutter [9] é uma linguagem que acrescenta em AspectJ o conceito de *pointcuts* remotos, os quais viabilizam a captura de pontos de junção pertencentes à execução de programas em máquinas remotas. GluonJ [2] é uma extensão de AspectJ com suporte a injeção de dependência.

Ghosh e colegas descrevem uma metodologia para implementar de forma independente interesses de negócio e interesses de distribuição providos por plataformas de *middleware* [4]. Assim, considera-se que esta metodologia e o sistema DAJ compartilham dos mesmos objetivos. A idéia dos autores é incorporar conceitos e técnicas de desenvolvimento orientado por modelos e, mais especificamente, conceitos propostos pela arquitetura MDA [10], ao desenvolvimento de aplicações distribuídas. Na metodologia proposta, designa-se como MTD (*middleware transparent design*) um modelo que captura apenas os interesses funcionais de um sistema. Este modelo é então combinado com aspectos específicos de plataformas de *middleware*, a fim de se obter um MSD (*middleware specific design*), isto é, um modelo incluindo requisitos funcionais e de distribução. Consideramos que DAJ é um sistema que põe em prática os princípios e conceitos da proposta de Ghosh e colegas, disponibilizando uma ferramenta que automatiza e torna transparente detalhes de programação distribuída requeridos por Java IDL e Java RMI.

Os descritores de distribuição propostos em DAJ podem ser considerados como uma linguagem orientada por aspectos de domínio específico para representação de requisitos transversais introduzidos por tecnologias de *middleware*. Alguns autores, como Wand [17] e Lieberherr [12], consideram que linguagens de domínio específico são veículos mais adequados para modelar e expressar aspectos, já que as mesmas utilizam vocabulários e abstrações mais próximos do interesse transversal que pretendem modularizar. Além disso, as mesmas são menos sujeitas às críticas normalmente feitas a linguagens orientadas por aspectos de propósito geral, como violação de encapsulamento e impossibilidade de se raciocinar localmente sobre módulos [17]. É interessante observar ainda que as primeiras gerações de linguagens orientadas por aspectos eram linguagens de domínio específico. Dentre elas, podemos citar COOL (para sincronização e coordenação) e RIDL (para especificação de interfaces remotas), ambas propostas por Lopes em seu trabalho de doutoramento [7].

Ceccato e Tonella descrevem um *framework* que inclui geração automática de

aspectos de distribuição e interfaces remotas [1]. No entanto, a solução proposta é restrita a Java RMI. Além disso, a especificação dos parâmetros de distribuição é feita por meio de uma simples lista com o nome das classes remotas do sistema.

RMIX [6] é um sistema que permite a instanciação de servidores que aceitam chamadas remotas provenientes de Java RMI, CORBA ou SOAP. Para prover esta flexibilidade, RMIX reimplementa diversos componentes internos comuns a sistemas de *middleware*. Em DAJ, a mesma flexibilidade é obtida de forma mais simples e não-invasiva.

7 Conclusões

Atualmente, distribuição é um interesse presente na grande maioria dos projetos de desenvolvimento de *software*. Daí o sucesso de tecnologias de *middleware*, as quais têm como objetivo encapsular detalhes inerentes à programação em ambientes de redes. No entanto, paradoxalmente, os sistemas atuais de *middleware* também acrescentam detalhes, convenções e protocolos próprios de programação, os quais devem ser dominados por engenheiros de *software*. Mais importante, estes detalhes, via de regra, possuem um comportamento transversal, se entrelaçando e se espalhando pelo código funcional de uma aplicação. O resultado final é que aplicações distribuídas baseadas em *middleware* não apresentam graus desejados de modularidade, reusabilidade e separação de interesses.

Portanto, desde o final da década de 90, distribuição vêm sendo citada e estudada como um dos principais interesses que podem se beneficiar de implementações orientadas por aspectos. O presente trabalho contribui com estas iniciativas ao propor uma ferramenta orientada por aspectos cujo objetivo final é automatizar e encapsular código de distribuição requerido por duas plataformas de *middleware* bastante utilizadas por desenvolvedores de aplicações em Java. Para alcançar este objetivo, a ferramenta DAJ se beneficia da sinergia gerada pela combinação de três tecnologias utilizadas na sua implementação: aspectos (para modularização de código transversal de distribuição), linguagens de domínio específico (para descrição e configuração de parâmetros de distribuição) e geração e transformação de código (para automatizar a criação de aspectos).

As principais contribuições e funcionalidades da ferramenta DAJ foram validadas por meio de uma aplicação distribuída simples, mas que faz uso de diversas abstrações normalmente providas por plataformas de *middleware*. Com o uso de DAJ, foi possível encapsular todo código de distribuição deste sistema. Com isso, as classes de negócio da aplicação permaneceram livres de código entrelaçado demandado pelas plataformas de *middleware* subjacentes. Os descritores de distribuição utilizados em DAJ se mostraram ainda flexíveis e expressivos para modelar diversas (re)configurações de implantação deste sistema.

Como trabalho futuro, pretende-se investigar o emprego de DAJ no desenvolvimento e/ou reestruturação de outras aplicações distribuídas. Pretende-se ainda adicionar suporte a aplicações baseadas em serviços *Web*.

Agradecimentos: Este trabalho foi desenvolvido como parte de um projeto de pesquisa financiado pela FAPEMIG (processo CEX-817/05 - Edital Universal 2005).

Referências

[1] Mariano Ceccato and Paolo Tonella. Adding distribution to existing applications by means of aspect oriented programming. In *4th IEEE International Workshop on Source Code Analysis*

and Manipulation, pages 107–116. IEEE Computer Society, 2004.

[2] Shigeru Chiba and Rei Ishikawa. Aspect-oriented programming beyond dependency injection. In *19th European Conference on Object-Oriented Programming*, volume 3586 of *Lecture Notes in Computer Science*, pages 121–143. Springer, 2005.

[3] Tal Cohen and Joseph Gil. AspectJ2EE = AOP + J2EE. In *18th European Conference on Object-Oriented Programming*, volume 3086 of *LNCS*, pages 219–243. Springer-Verlag, 2004.

[4] S. Ghosh, R. B. France, A. Bare, B. Kamalalar, R. P. Shankar, D. M. Simmonds, G. Tandon, P. Vile, and S. Yin. A middleware transparent approach to developing distributed applications. *Software Practice and Experience*, 35(12):1131–1154, October 2005.

[5] Java IDL. http://java.sun.com/products/jdk/idl.

[6] Dawid Kurzyniec, Tomasz Wrzosek, Vaidy S. Sunderam, and Aleksander Slominski. RMIX: A multiprotocol RMI framework for Java. In *17th International Parallel and Distributed Processing Symposium*, pages 140–146. IEEE Computer Society, April 2003.

[7] Cristina Videira Lopes. *D: A Language Framework for Distributed Programming*. PhD thesis, College of Computer Science, Northeastern University, December 1997.

[8] Martin Fowler. Inversion of control containers and the dependency injection pattern. http://www.martinfowler.com/articles/injection.html.

[9] Muga Nishizawa, Shigeru Chiba, and Michiaki Tatsubori. Remote Pointcut: A Language Construct for Distributed AOP. In *3rd International Conference on Aspect-Oriented Software Development*, pages 7–15. ACM Press, 2004.

[10] OMG Model Driven Architecture. http://www.omg.org/mda.

[11] Roman Pichler, Klaus Ostermann, and Mira Mezini. On aspectualizing component models. *Software Practice and Experience*, 33(10):957–974, August 2003.

[12] Macneil Shonle, Karl J. Lieberherr, and Ankit Shah. Xaspects: an extensible system for domain-specific aspect languages. In *OOPSLA Companion*, pages 28–37. ACM Press, October 2003.

[13] Jon Siegel. *CORBA 3 Fundamentals and Programming*. John Wiley & Sons, 2nd edition, 2000.

[14] Sergio Soares, Paulo Borba, and Eduardo Laureano. Distribution and persistence as aspects. *Software Practice and Experience*, 36(7):711–759, 2006.

[15] Sérgio Soares. *An Aspect-Oriented Implementation Method*. PhD thesis, Federal Universisty of Pernambuco, Recife, Brazil, October 2004.

[16] Eli Tilevich, Stephan Urbanski, Yannis Smaragdakis, and Marc Fleury. Aspectizing server-side distribution. In *Automated Software Engineering Conference*, pages 130–141. IEEE Press, October 2003.

[17] Mitchell Wand. Understanding aspects: extended abstract. In *8th International Conference on Functional Programming*, pages 299–300. ACM Press, August 2003.

[18] A. Wollrath, R. Riggs, and J. Waldo. A distributed object model for the Java system. In *2nd Conference on Object-Oriented Technologies & Systems*, pages 219–232, 1996.

Utilizando Ontologias e Serviços Web na Computação Ubíqua

Marcos Forte[+], Wanderley Lopes de Souza, Antonio Francisco do Prado

Departamento de Computação (DC) - Universidade Federal de São Carlos (UFSCar)
Caixa Postal 676 - 13565-905 – São Carlos (SP)

[+] Centro Universitário Fundação Santo André – Av. Príncipe de Gales, 821
09060-650 - Santo André (SP)

{marcos_forte, desouza, prado}@dc.ufscar.br

Abstract *In the near future, most users will access the Internet by means of small mobile devices. This context of ubiquitous computing is highly volatile because the diversity of device characteristics and access networks extends each day. The flexibility necessary for software reuse in this environment is not provided by the current programming paradigms. Aiming to offer this flexibility this paper proposes the use of ontologies and Web services to extend a framework of components to the domain of content adaptation, which facilitates the development of software based on reuse. A case study illustrates the use of the proposed solution.*

Resumo *Num futuro próximo a maioria dos usuários acessarão a Internet via pequenos dispositivos móveis. Esse contexto da computação ubíqua é altamente volátil, pois a diversidade de características de dispositivos e redes de acesso se amplia a cada dia. A flexibilidade necessária para o reuso de software neste ambiente não é provida pelos atuais paradigmas de programação. Visando oferecer esta flexibilidade este artigo propõe a utilização de ontologias e serviços Web para estender um framework de componentes para o domínio da adaptação de conteúdo, que facilita o desenvolvimento de software baseado no reuso. Um estudo de caso ilustra o uso da solução proposta.*

1. Introdução

Pode-se classificar a história da Computação, em função dos ambientes computacionais predominantes, em três eras: a passada dos *mainframes*; a atual dos computadores pessoais; a futura da Computação Ubíqua, denominada por alguns autores de *Pervasive Computing* [Hansmann et al. 2003], que tem por meta permitir ao usuário, usando qualquer dispositivo, a qualquer momento e de qualquer lugar, o fácil acesso e processamento da informação.

A comunicação móvel é um dos fatores que está impulsionando o salto da Computação para essa nova era. A possibilidade de conexão a qualquer momento e de qualquer lugar outorgou aos usuários uma escolha e uma liberdade sem precedentes, permitindo que estes busquem por novas e recompensadoras formas para o tratamento de assuntos pessoais e profissionais. Em apenas uma década as redes móveis permitiram atingir um ritmo de crescimento que demandou quase um século às redes fixas e os

avanços nas tecnologias móveis têm propiciado uma transição de serviços exclusivos de voz para serviços de conteúdo baseados em *Web*.

Esta mobilidade globalizada demanda por novas arquiteturas e protocolos, que permitam a fácil conexão das redes móveis aos diversos tipos de provedores de serviços e de conteúdo espalhados pela Internet. Uma visão futuristica da *Internet Móvel* assume usuários com diferentes perfis utilizando diversos tipos de redes de acesso e uma grande variedade de dispositivos móveis, exigindo serviços personalizados que atendam da melhor forma possível as suas necessidades, disponibilidades e localizações. Nesses contextos da computação ubíqua são descritas informações sobre pessoas, lugares, dispositivos e outros objetos que são considerados relevantes para a interação entre usuários e serviços, incluindo os próprios usuários e serviços.

De forma a facilitar à descrição dessas informações de contexto, a comunidade da *Web* semântica [*Semantic Web Activity*] produziu um conjunto de linguagens e ferramentas para o uso, desenvolvimento, manutenção e compartilhamento de ontologias. Entre as vantagens apresentadas por essas linguagens destacam-se [Knublauch et al. 2006]: são otimizadas para a representação de conhecimento estruturado em um nível elevado de abstração; os modelos de domínio podem ser disponibilizados na Internet e compartilhados entre múltiplas aplicações; é baseada em dialetos não ambíguos da lógica formal, permitindo a utilização de serviços inteligentes baseados no raciocínio (*reasoning*), possibilitando assim, em tempo de execução, a definição dinâmica de classes, a re-classificação de instâncias e a execução de consultas lógicas complexas. Além disso, essas linguagens operam sobre estruturas similares às linguagens orientadas a objeto, e podem ser eficazmente integradas aos componentes de software tradicionais.

Dentre os diferentes domínios da computação ubíqua destaca-se o da adaptação de conteúdo para dispositivos móveis, que é o enfoque desta pesquisa. A adaptação de conteúdo consiste em adaptar conteúdo e aplicações de modo automático, respeitando as preferências dos usuários e as características do contexto da computação ubíqua. Como existe uma infinidade de adaptações possíveis, quanto maior a quantidade de serviços de adaptação disponíveis, maior a chance de satisfazer as necessidades dos usuários. Neste caso o uso de serviços *Web* e seus padrões facilitam o desenvolvimento de novos servidores, pois permitem independência de linguagens e plataformas.

Prover soluções adaptáveis a esses contextos tão diversos, que facilite o desenvolvimento de aplicações baseado no reuso, constitui um grande desafio para a Engenharia de Software. Assim, motivados por estas idéias, este artigo propõe a utilização de ontologias e serviços *Web* no contexto da computação ubíqua para facilitar o desenvolvimento e incentivar o reuso de software. Ontologias e serviços *Web* foram usados para reconstruir um *framework*, denominado FACI (***F****ramework* para **A**daptação de **C**onteúdo na **I**nternet) [Claudino, Souza e Prado 2005], com o propósito de apoiar o desenvolvimento de aplicações no domínio da adaptação de conteúdo, baseado no reuso da modelagem e do código. A reconstrução do FACI resultou na adaptação e adição de novos componentes mais genéricos, o que ampliou o seu reuso para diferentes aplicações desse domínio de adaptação de conteúdo. A seqüência deste artigo está estruturada da seguinte forma: a seção 2 versa sobre o tema ontologias e sobre algumas linguagens, desenvolvidas para a especificação das mesmas, e trata também da integração com serviços *Web*; a seção 3 concentra-se no uso de ontologias e serviços

Web na computação ubíqua; a seção 4 apresenta a reconstrução do FACI e um estudo de caso baseado no reuso dos componentes dessa nova versão; a seção 5 descreve os trabalhos correlatos; e finalmente a seção 6 apresenta as conclusões.

2. Ontologias

Para alcançar a interoperabilidade entre sistemas heterogêneos, envolvidos na execução de um determinado domínio de aplicações, é fundamental que estes possam compartilhar informações de maneira automática, com um entendimento comum e não ambíguo dos termos e conceitos usados nessas aplicações. Neste sentido as ontologias constituem-se em artefatos importantes na viabilização do tratamento dessa heterogeneidade.

Berners-Lee propôs a *Web*-Semântica [Berners-Lee, Hendler e Lassila 2001] como uma evolução natural da *Web* vigente, a fim de viabilizar a manipulação do conteúdo por parte das aplicações com capacidade de interpretar a semântica das informações. O conteúdo da *Web*, que não passa de informação sem significado para os computadores, pode então ser interpretado por máquinas através do uso de ontologias, tornando a recuperação da informação na *Web* menos ambígua e fornecendo respostas mais precisas às consultas dos usuários.

Ontologias são usadas para diversos fins, variando do apoio aos processos de desenvolvimento de software, ou qualquer outra atividade executada por equipes geograficamente distribuídas, ao apoio em tempo de execução aos sistemas de informação [Guarino 1998]. Neste artigo o termo ontologia refere-se a uma descrição explícita de conceitos e relações de um domínio de aplicação, incluindo um vocabulário de termos empregado nesse domínio e um conjunto de axiomas que expressam as restrições para a interpretação desse vocabulário.

Para a construção de ontologias é essencial dispor-se de uma linguagem apropriada. Essa linguagem deve possuir uma semântica bem-definida, ser expressiva o suficiente para descrever inter-relacionamentos complexos e restrições entre os objetos, ser capaz de manipular e inferir automaticamente, isso tudo dentro de limites aceitáveis de tempo e recursos [Martin e Mcllraith 2003].

O *World Wide Web Consortium* (*W3C*) orienta o desenvolvimento, a organização e a padronização de linguagens para promover a interoperabilidade entre aplicações na Web. Dentre essas linguagens destacam-se: *Resource Description Framework (RDF)* [Monola e Miller 1999] e mais recentemente a *Web Ontology Language (OWL)* [McGuinness e Harmelen 2004].

2.1. *Web* Ontology Language (*OWL*)

A *OWL* é uma linguagem de marcação semântica usada na publicação e compartilhamento de ontologias na *Web*. Na *OWL* ontologia é um conjunto de definições de classes, propriedades e restrições em relação aos modos com que essas classes e propriedades podem ser empregadas.

Uma ontologia *OWL* pode incluir: relações de taxonomia entre classes; propriedades dos tipos de dados e descrições dos atributos de elementos das classes; propriedades do objeto e descrições das relações entre elementos das classes; instâncias das classes; e instâncias das propriedades. As propriedades de tipos de

dados e as propriedades de objeto são coletivamente as propriedades de uma classe.

Uma das vantagens na utilização da *OWL* é sua base em um dialeto não ambíguo da lógica formal chamado *Descrição Lógica* [Baader et al. 2003]. O suporte formal torna possível a utilização de serviços inteligentes baseados em raciocínio, tais como a classificação e verificação automática de consistência. Esses serviços podem ser usados em tempo de criação, de modo a facilitar a construção de modelos reusáveis e bem-testados do domínio, ou em tempo de execução, possibilitando a definição dinâmica de classes, re-classificação de instâncias e execução de consultas lógicas complexas.

2.2. Serviços *Web* Semânticos

Os serviços *Web*, que há anos são a base das arquiteturas orientadas a serviços, começam a apresentar deficiências principalmente nas áreas de descrição, descoberta e composição de serviços. O problema está na falta de suporte semântico na linguagem de descrição de serviços (*Web Services Description Language - WSDL*) e no mecanismo de armazenamento e pesquisa de serviços (*Universal Description, Discovery and Integration - UDDI*).

Com o propósito de integrar a *Web* semântica aos serviços *Web*, foram desenvolvidas novas linguagens para descrição e composição de serviços apresentadas na Figura 1. Essa integração ainda está em curso, e as atuais extensões semânticas do *UDDI* [Naumenko et al. 2005],[Srínívasan, Paoluccí e Sycara 2005] não garantem a compatibilidade e performance necessárias.

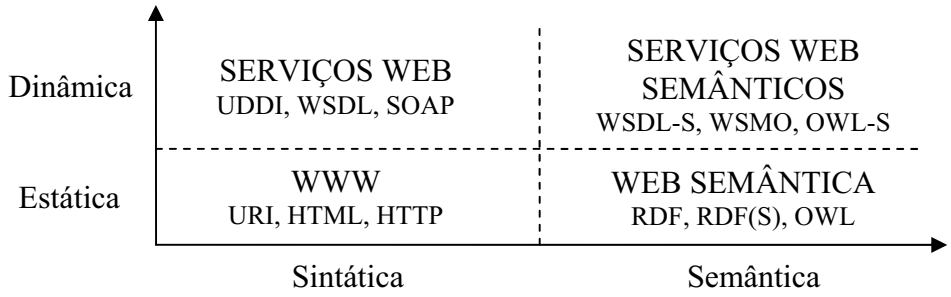


Figura 1. Evolução das tecnologias *Web*

2.2.1. *Web Ontology Language for Services* (*OWL-S*)

OWL-S [Martin 2004] é uma linguagem de descrição de serviços *Web*, que une a *WSDL* [WSDL] com a informação semântica do *OWL*, sendo organizada em três módulos: *Profile*; *Process Model* e *Grounding*. O *Profile* descreve as capacidades e características adicionais de serviços *Web*, com base nas transformações que esses serviços produzem, auxiliando na decisão do requisitante quando este utiliza um determinado serviço *Web*.

O *Process Model* especifica o protocolo de interação, que possibilita ao requisitante saber, num dado momento da transação, que informação enviar e receber do fornecedor. Esse módulo distingue dois tipos de processos: atômicos e compostos. Os atômicos fornecem descrições abstratas das informações trocadas com os requisitantes, correspondendo às operações que o fornecedor pode executar diretamente. Os

compostos são usados para descrever coleções de processos (atômicos ou compostos), organizados através de algum tipo de estrutura de controle de fluxo.

O *Grounding* descreve como processos atômicos são transformados em mensagens concretas, de modo que possam ser trocados via rede ou através de uma chamada de procedimento. É definido como um mapeamento "um para um" de processos atômicos para a especificação de descrições *WSDL*.

3. Uso de Ontologias e Serviços *Web* na Computação Ubíqua

Baseado nos estudos apresentados pesquisou-se o uso de ontologias para descrever características do contexto da computação ubíqua, concentrando-se na solução de problemas relacionados com a adaptação de conteúdo em dispositivos móveis, tais como celulares e *palmtops*.

Informações sobre as características do contexto são elementos fundamentais no desenvolvimento de soluções para a adaptação de conteúdo e compõe a política de adaptação. Essa política decide quais serviços de adaptação serão oferecidos, quais adaptadores locais ou remotos executarão essas adaptações e quando estas deverão ser solicitadas.

Visando aperfeiçoar o processo de tomada de decisão, a política de adaptação considera ainda as regras de adaptação, que indicam as condições que devem ser atendidas para que as ações correspondentes a uma determinada adaptação sejam executadas.

Finalmente, é necessário disponibilizar uma infra-estrutura de servidores de adaptação distribuídos pela Internet. O uso da tecnologia de serviços *Web* oferece uma solução consistente, com uma grande quantidade de ferramentas e padrões bem definidos. Além disso, o uso de ontologias e a incorporação de semântica em seus padrões possibilitam uma fácil migração das atuais soluções proprietárias, de descoberta e composição de serviços, para uma arquitetura distribuída baseada na *Web* Semântica.

3.1. Perfis

Para descrever o contexto da computação ubíqua, visando à eficácia da política de adaptação, as seguintes informações são necessárias: características e capacidades dos dispositivos de acesso; dados pessoais e preferências dos usuários; condições da rede de comunicação; características dos conteúdos requisitados; resoluções contratuais entre o provedor de serviços e o usuário final. Essas informações estão contidas, respectivamente, nos seguintes perfis: *dispositivo*, *usuário*, *rede*, *conteúdo* e *Service Level Agreement (SLA).* Esses perfis são descritos em *OWL*.

Também são necessárias informações sobre os serviços de adaptação disponíveis. Neste caso, além das características dos serviços oferecidos, devem ser incluídas informações sobre comunicação (e.g., protocolos, endereçamento) e as condições para a execução de determinada adaptação. Essas condições são baseadas em regras e utilizadas pela política de adaptação. Os perfis de serviço são descritos em *OWL-S.*

3.1.1. Perfis de dispositivo, usuário, rede, conteúdo e *SLA*

O perfil de dispositivo contém características de hardware e software do dispositivo. O conhecimento das capacidades de um dispositivo orienta o processo de adaptação, a fim de que apenas as adaptações necessárias sejam aplicadas ao conteúdo (e.g., remoção de som de um conteúdo requisitado por um dispositivo incapaz de reproduzir som). O campo *UserAgent*, do cabeçalho *HTTP*, é empregado para determinar o dispositivo de acesso utilizado pelo usuário, permitindo assim consultar o perfil desse dispositivo que está armazenado na base de dados. A Figura 2 apresenta um dos modelos de ontologia, baseado no *EMF Ontology Definition Metamodel* (*EODM*) [EODM 2006], para a descrição de perfis de dispositivo. Para que se tenha uma melhor idéia das vantagens do uso de ontologias para a descrição de perfis, o modelo permite a inserção de restrições que auxiliam na consistência e validação do perfil. Por exemplo, a restrição *Supported_ImageRestriction* define que a classe *Supported_Image* só possa ser instanciada com os indivíduos declarados na classe enumerada *Image_Format*.

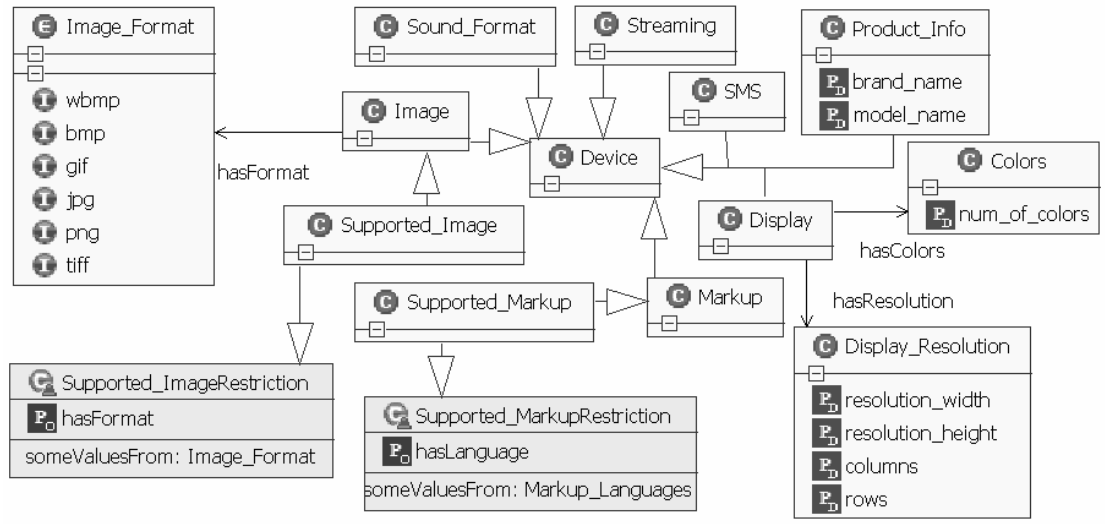


Figura 2. Modelo de ontologia para a descrição de perfis de dispositivo.

Atualmente os padrões de perfis de dispositivos móveis mais adotados pelos fabricantes são o *User Agent Profile (UAPROF)* [OMA 2003] e o *Composite Capability /Preference Profiles (CC/PP)* [Klyne et al. 2004], sendo que ambos deveriam produzir especificações equivalentes, já que o *UAPROF* seria um vocabulário específico com base na estrutura do *CC/PP* [Cannistrà 2003]. Infelizmente os desenvolvedores do *UAPROF* acabaram criando um novo esquema (www.openmobilealliance.org/tech/ profiles/UAPROF/ccppschema-20030226), que redefiniu alguns elementos do *CC/PP* (www.w3.org/2002/11/08-ccpp-schema). Por esse motivo a equivalência lógica entre elementos das diferentes especificações não pode ser reconhecida em termos de *RDF*. Para resolver esse problema, o uso de ontologias, baseadas em *OWL*, permite o mapeamento dos elementos de diferentes padrões, possibilitando o reuso de perfis que utilizem essas especificações.

O perfil de usuário contém informações pessoais deste e de suas preferências de adaptação de conteúdo. Diferentes usuários podem desejar que diferentes adaptações sejam aplicadas a um conteúdo requerido (e.g., enquanto um usuário tem preferência pela redução da resolução de imagens, um outro pode preferir pela redução do número

de cores). Adaptações, não baseadas nas preferências dos usuários, podem ser inconvenientes ou até mesmo indesejadas.

A Figura 3 apresenta um dos modelos de ontologia para a descrição de perfis de usuário. Esse modelo representa a classe *USER* com as propriedades *ID*, que é usada para recuperar as informações do usuário na base de dados, e *Name,* que identifica o nome do usuário. Nas sub-classes de *Services* estão descritas as propriedades dos serviços de adaptação, que podem ser configuradas pelo usuário de acordo com as suas preferências. Para exemplificar, na classe *Classification_and_Filtering* há a propriedade *Filter_Profile*, onde são definidos quais tipos de conteúdo serão bloqueados (e.g., sexo, shopping, jogos, violência).

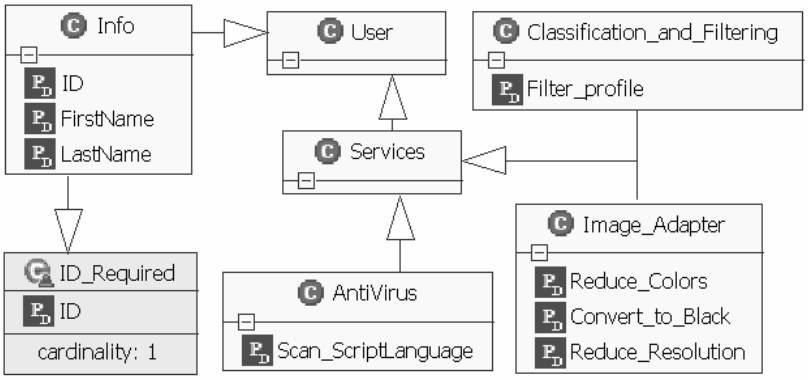


Figura 3. Modelo de ontologia para descrição de perfis de usuário.

O perfil de rede é obtido dinamicamente através de agentes, que monitoram parâmetros da rede de comunicação entre o provedor e o usuário. As informações contidas nesse perfil orientam alguns processos de adaptação (e.g., imagens, vídeo e áudio sob demanda), a fim de que o conteúdo adaptado esteja otimizado para as condições da rede num determinado momento.

O perfil de conteúdo também é obtido dinamicamente, sendo baseado em características do próprio conteúdo requisitado. A partir de informações extraídas do cabeçalho *HTTP* (e.g., se o conteúdo dispõe de texto e/ou imagem, idioma) e do conjunto de meta dados do conteúdo, se disponível, são determinadas as alterações necessárias e aplicáveis ao conteúdo.

No perfil *SLA* estão descritas as resoluções contratuais entre o provedor de acesso e o usuário. Atualmente esses provedores oferecem diferentes planos aos seus usuários, incluindo largura de banda, tempo de conexão e vários serviços de valor agregado, permitindo assim que os usuários escolham o plano mais adequado as suas necessidades.

3.1.2. Perfis de Serviços

O perfil de serviço descreve as características de um serviço de adaptação, incluindo as condições necessárias para a sua execução. Essas características podem ser divididas em duas classes: as diretamente associadas ao processo de adaptação, representadas pelas Entradas e Saídas; e as de apoio, representadas pelas Pré-condições e Efeitos, que auxiliam a decisão da política de adaptação em executar, ou não, determinado serviço, conforme ilustra a Figura 4.

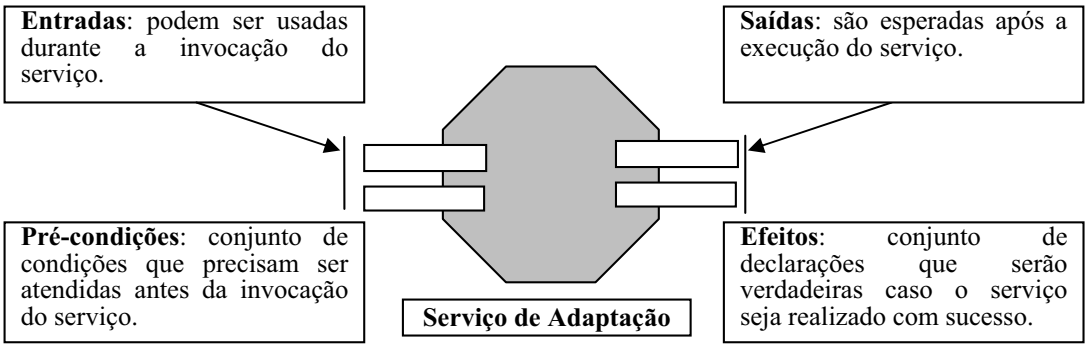


Figura 4. Características de perfis de serviço.

Existem outras funções que destacam esse perfil em relação aos outros, e que determinaram o uso do *OWL-S* para sua descrição. A principal delas, é que nesse perfil é realizado o mapeamento das descrições semânticas (*OWL*) dos perfis, com as descrições sintáticas (*WSDL*) dos serviços *Web,* permitindo a integração de ontologias com serviços *Web.* Além disso, ele também descreve as características de comunicação (e.g., protocolos, endereços) dos serviços *Web*, facilitando o acesso remoto a esses serviços.

3.2. Descoberta e Composição de Serviços

A partir da combinação das informações do ambiente ubíquo, descritas nos perfis de *dispositivo*, *usuário*, *rede*, *conteúdo* e *SLA*, e das informações dos perfis de serviço, são definidos os serviços necessários para a execução de uma determinada adaptação. Caso seja necessário mais de um serviço para realizar a adaptação necessária, deve-se gerenciar a ordem de suas execuções.

Para auxiliar na determinação dos serviços necessários, os perfis de serviço podem conter regras especificadas em *Semantic Web Rule Language (SWRL)* [Horrocks 2003]. Essa abordagem permite que os componentes responsáveis pela política de adaptação, sejam independentes das variações provocadas pelas mudanças e inclusões de novos serviços e regras.

Com o propósito de exemplificar a especificação, baseada em *OWL-S*, dos perfis de serviço, é apresentado na Figura 5 o esqueleto do perfil de um serviço de conversão de linguagens de marcação. Inicialmente estão descritas as ontologias que serão importadas, de modo a incluir as informações semânticas já definidas (1). Nesse caso são reutilizados: uma ontologia pública, *semwebglossary.owl,* que define termos da *Web* semântica com o propósito de evitar ambigüidade de definições; os perfis de conteúdo (*content.owl*) e dispositivo (*device.owl*). Na seqüência é definida uma classe, que descreve as linguagens de marcação suportadas (2), e uma propriedade *canBeConvertedTo* (3), que será utilizada na definição das possíveis conversões realizadas pelo serviço (4). Em (5) é declarado o parâmetro que descreve a linguagem de marcação a ser convertida. Essa informação é obtida da classe *Content_Type* do perfil de conteúdo, que é previamente instanciada com informações sobre o conteúdo requisitado. Do mesmo modo, a linguagem de marcação desejada é obtida através do parâmetro, definido na classe Supported_*Markup* do perfil de dispositivo, que informa as linguagens suportadas (6). As entradas, as saídas, as precondições e os efeitos estão definidos na descrição do processo (7). A pré-condição baseia-se em uma regra em

SWRL (8), que retorna verdadeiro quando a conversão necessária está entre as definidas em (4). A seção de *Grounding*, entre outras que não estão apresentadas neste exemplo por limitações de espaço, é gerada automaticamente a partir da descrição *WSDL* do serviço.

```
...
<!ENTITY gloss "http://www.personal-reader.de/rdf/semwebglossary.owl">
<!ENTITY device "http://www.adaptationsrv.org/device.owl">
<!ENTITY content "http://www.adaptationsrv.org/content.owl">
...
<owl:Class rdf:ID="SupportedMarkupLanguage">
  <owl:oneOf rdf:parseType="Collection">
     <factbook:Language rdf:about="&glossary;#HTML"/>
     <factbook:Language rdf:about="&glossary;#XHTML"/>
     <factbook:Language rdf:about="&glossary;#XML"/>
     <factbook:Language rdf:about="&glossary;#CHTML"/>
     <factbook:Language rdf:about="&glossary;#WML"/>
  </owl:oneOf>
</owl:Class>
<owl:ObjectProperty rdf:ID="canBeConvertedTo">
  <rdfs:domain rdf:resource="#SupportedMarkupLanguage"/>
  <rdfs:range rdf:resource="#SupportedMarkupLanguage"/>
</owl:ObjectProperty>
<rdf:Description rdf:about="&glos;#HTML"><canBeConvertedTo rdf:resource="&glos;#WML"/></rdf:Description>
<rdf:Description rdf:about="&glos;#HTML"><canBeConvertedTo rdf:resource="&glos;#XHTML"/></rdf:Description>
<rdf:Description rdf:about="&glos;#HTML"><canBeConvertedTo rdf:resource="&glos;#XML"/></rdf:Description>
<rdf:Description rdf:about="&glos;#HTML"><canBeConvertedTo rdf:resource="&glos;#cHTML"/></rdf:Description>
<rdf:Description rdf:about="&glos;#XML"><canBeConvertedTo rdf:resource="&glos;#XHTML"/></rdf:Description>
...
<process:Input rdf:ID="InputMarkupLanguage">
  <process:parameterType  rdf:datatype="&xsd;#anyURI">&content;#Content_Type
  </process:parameterType>
</process:Input>
<process:Input rdf:ID="OutputMarkupLanguage">
  <process:parameterType  rdf:datatype="&xsd;#anyURI">&device;#Supported_Markup
  </process:parameterType>
</process:Input>
...
<process:AtomicProcess rdf:ID="MarkupConverterProcess">
  <process:hasInput rdf:resource="#InputMarkupLanguage"/>
  <process:hasInput rdf:resource="#OutputMarkupLanguage"/>
  <process:hasInput rdf:resource="#InputString"/>
  <process:hasOutput rdf:resource="#OutputString"/>
  <process:hasPrecondition rdf:resource="#SupportedConvertion"/>
  <process:hasEffect rdf:resource= ="#MarkupLanguageConverted"/>
</process:AtomicProcess>
...
<expr:SWRL-Condition rdf:ID="SupportedConvertion">
  <rdfs:label>canBeConvertedTo(InputMarkupLanguage, OutputMarkupLanguage)</rdfs:label>
  <expr:expressionLanguage rdf:resource="&expr;#SWRL"/>
  <expr:expressionObject><swrl:AtomList><rdf:first>
       <swrl:IndividualPropertyAtom>
         <swrl:propertyPredicate rdf:resource="#canBeConvertedTo"/>
         <swrl:argument1 rdf:resource="#InputMarkupLanguage"/>
         <swrl:argument2 rdf:resource="#OutputMarkupLanguage"/>
       </swrl:IndividualPropertyAtom>
    </rdf:first></swrl:AtomList></expr:expressionObject>
</expr:SWRL-Condition>
...
```

Figura 5. Esqueleto do perfil de serviço de conversão de linguagens de marcação.

Algumas vezes não é possível encontrar um único serviço que satisfaça todos os objetivos necessários. Uma solução para este problema pode ser obtida a partir da composição de serviços, combinados da maneira apropriada para atender as metas pretendidas.

Inicialmente é necessário identificar os serviços *Web*, que farão parte da composição, assumindo que os mesmos são definidos pelos seguintes elementos: Entradas *I*, Saídas *O*, Pré-Condições *P* e Efeitos *E*. Seja S = {S_1, S_2,..., S_n} um conjunto de serviços, onde cada S_i é definida por uma quádrupla $\left(I_s^i, O_s^i, P_s^i, E_s^i\right)$, e seja M = {$M_1$, M_2,...,M_n} um conjunto de metas, onde cada meta M_j é definida por uma quádrupla $\left(I_m^j, O_m^j, P_m^j, E_m^j\right)$. Se existe um S_i tal que:

1) $\bigcup_{i=1}^{m} I_s^i \subseteq \bigcup_{j=1}^{n} I_m^n$ e $\bigcup_{i=1}^{m} P_s^i \subseteq \bigcup_{j=1}^{n} P_m^n$; e,

2) $\bigcup_{i=1}^{m} O_s^i \supseteq \bigcup_{j=1}^{n} O_m^n$ e $\bigcup_{i=1}^{m} E_s^i \supseteq \bigcup_{j=1}^{n} E_m^n$

são incluídos na lista de possíveis serviços a serem aplicados na adaptação de um conteúdo. Caso somente a primeira parte seja satisfeita, significando que não foi encontrado um serviço que realize sozinho a adaptação necessária, é ativado o algoritmo de composição. Uma das técnicas utilizadas para a composição de serviços é a *forward chaining* [Lara et al. 2005]. Essa técnica baseia-se na seleção de um possível serviço *S*, que contenha as entradas requeridas por *O* e verifica se os elementos de saída são satisfeitos. Caso *S* não atinja o objetivo, um novo objetivo *O'* será definido a partir de *O* e das saídas e efeitos geradas por *S* e todo o processo se repete.

4. Extensão do FACI

Para o domínio de aplicações alvo desta pesquisa, os casos de uso foram modelados e implementados no *Framework para Adaptação de Conteúdo na Internet (FACI)* [Claudino, Souza e Prado 2005], estendido de modo a suportar ontologias e serviços Web, aproveitando a sua estrutura básica para o desenvolvimento de diferentes aplicações de adaptação de conteúdo. Esse *framework* foi organizado em dois pacotes principais: *Adaptation Proxy*, que desempenha o papel do dispositivo de borda (e.g., proxy); e *Adaptation Server*, que desempenha o papel do módulo de serviços de adaptação. No *FACI,* os atores, usuário da Internet (*User*) e servidor de origem (*Origin Server*) interagem com o *Adaptation Proxy* através das requisições de conteúdo e suas respostas. O *Adaptation Proxy* oferece recursos para analisar essas requisições e respostas, implementa a política de adaptação e pode realizar adaptações localmente ou solicitá-las aos *Adaptation Servers*, que as realizarão remotamente.

No nível de componente a preocupação é com a estrutura interna do sistema para atender as suas funcionalidades. Para tal o *FACI* foi estruturado num conjunto de componentes combinados, que disponibilizam seus serviços através de suas interfaces. A Figura 6 apresenta os componentes do FACI, reconstruído com base em ontologias e serviços *Web*. No modelo estão ressaltados os componentes adaptados e adicionados com a extensão do FACI.

O *Callout Protocol Clien*t, que é responsável pela comunicação com o *Adaptation Server*, foi adaptado com a adição do *Simple Object Access Protocol (SOAP)* [SOAP], proporcionando a compatibilidade com serviços *Web*. O *Internet Content Adaptation Protocol (ICAP)* [Elson e Cerpa 2003] foi mantido de modo a suportar servidores já desenvolvidos. O perfil de serviço descreve qual protocolo será utilizado para a comunicação com o respectivo serviço.

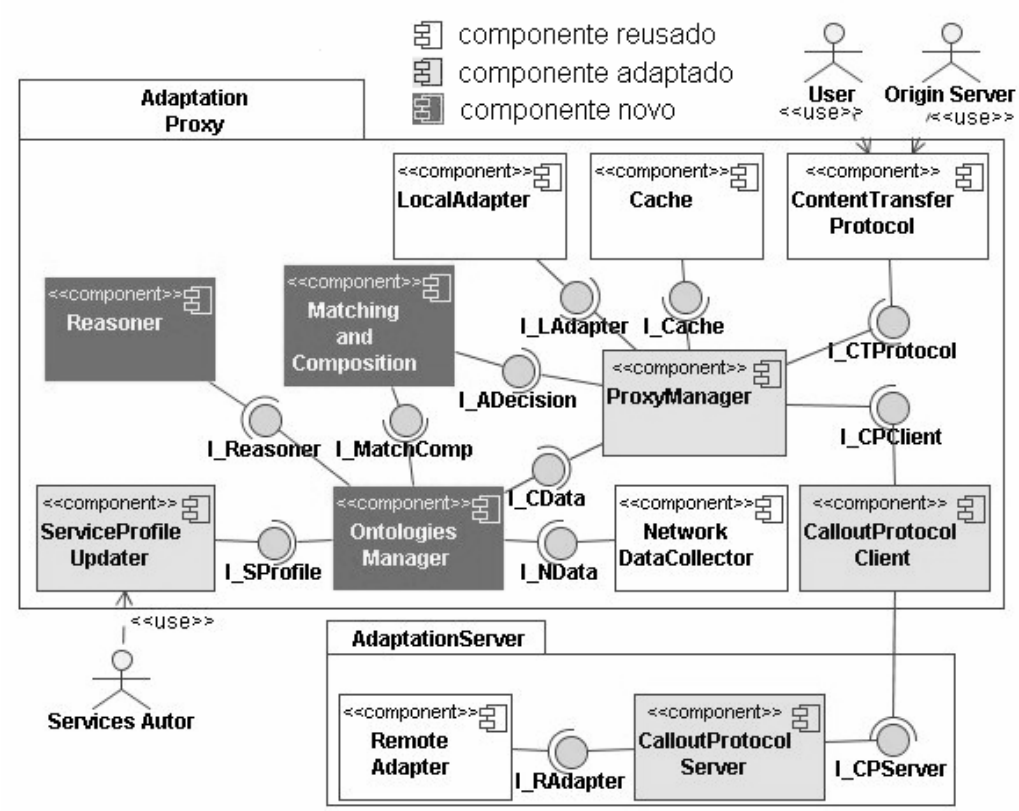


Figura 6. Componentes do *Framework para Adaptação de Conteúdo na Internet*.

Ontologies Manager, um novo componente que foi introduzido para gerenciar o armazenamento, a recuperação e o processamento das ontologias estáticas (perfis de dispositivo, usuário, *SLA* e serviços) numa base de dados relacional. Esse componente também trata os perfis de serviços, inseridos através do *Service Profile Updater*. Este último, que foi adaptado para realizar as funções descritas, disponibiliza uma interface Web para a inserção dos perfis de serviços, onde o provedor de serviço deverá inserir informações sobre os serviços de adaptação oferecidos, que serão convertidos para a especificação OWL-S. Para possibilitar o suporte a serviços *Web*, o *Service Profile Updater* converte as descrições de serviço *Web*, baseadas em *WSDL*, em um módulo *Grounding* do *OWL-S*. As ontologias dinâmicas, armazenadas em memória, são geradas sob demanda a partir das informações obtidas pela interface *I_NData*, para a geração do perfil de rede, e *I_CData*, para a geração do perfil de conteúdo. A interface I_CData foi adicionada a partir da adaptação do componente ProxyManager.

Matching and Composition, outro componente adicionado, realiza a função da política de adaptação no *Framework* a partir do cruzamento das informações do contexto de computação ubíqua com as características dos perfis de serviços. Para realizar essa tarefa são realizadas consultas baseadas em *SPARQL* [Prud'hommeaux e Seaborne 2006], utilizando além das informações declaradas nos perfis, as inferidas pelo *Reasoner,* para localizar os serviços que possuam entradas e saídas compatíveis com o contexto. Os resultados obtidos passam por uma segunda filtragem que verifica a conformidade com as pré-condições e os efeitos. Caso o resultado final aponte a necessidade da composição de serviços, esse componente também gerenciará a ordem de execução dos serviços de adaptação. Através da interface *I_ADecision* são enviadas

as informações, sobre o serviço de adaptação a ser invocado, ao *Proxy Manager* que irá, através do *Callout Protocol Client,* requisitar o serviço ao respectivo *Adaptation Server***.** A invocação de serviços é executada recursivamente até que o último serviço da composição seja realizado, liberando o *Proxy Manager* para o envio da resposta adaptada ao *User.*

No pacote *Adaptation Server*, o componente *Callout Protocol Server* foi adaptado, para suportar a tecnologia de serviços *Web, com a adição do SOAP*. Porém, com as alterações feitas no *Adaptation Proxy*, qualquer servidor de aplicações compatível com os padrões de serviços *Web* pode assumir a função de Adaptation Server.

Para que se tenha uma melhor compreensão do emprego de ontologias e serviços *Web* na política de adaptação, a Figura 7 apresenta um contexto de adaptação com perfis de uma aplicação, utilizando a notação perfil.classe, onde um usuário acessa a *Web* via celular. Quando o usuário conecta-se a *Web*, o provedor de acesso envia seu *User_ID*, conjuntamente com a requisição, para o *Adaptation Proxy*. A partir do *User_ID* o *Ontologies Manager* busca, na sua base de dados, os perfis de usuário e *SLA*. Da requisição do usuário é extraído o *User Agent (UA)*, que identifica o modelo do dispositivo de acesso, possibilitando assim a busca do perfil de dispositivo. O perfil de rede é criado dinamicamente a partir das informações obtidas pelo *Network Data Collector* e o perfil de conteúdo é gerado a partir de informações extraídas dos campos do cabeçalho da mensagem de resposta. Neste exemplo o usuário contratou o serviço pago de classificação e filtragem de conteúdo, tendo a propriedade *Pay_for_Filtering*, do perfil *SLA*, configurada como *true*.

User.Filter_Profile – 001	**Network**.WAN – GPRS
SLA.Services – Pay_for_Filtering	**Network**.RTT – 10
Device.DisplayResolution – 320x200	**Content**.Type – HTML
Device.Supported_Markup – XHTML	**Content**.URL – www.test.com

Figura 7. Exemplo de um contexto de adaptação.

Uma vez coletadas as informações dos perfis, é definido o objetivo, *O = (HTML, XHTML, Pay_ for_Filtering*, _), e são iniciadas as buscas de serviços para a realização das adaptações necessárias. O componente *Matching and Composition* descobre que o serviço de classificação e filtragem de conteúdo*,* S_x *= (HTML, HTML*, *Pay_for_Filtering*, _), atende às entradas e pré-condições, mas não suporta a saída em *XHTML*. Como todos objetivos pretendidos não foram atingidos, é criado um novo objetivo, *O' = (HTML, XHTML*, _ , _), sendo que em uma nova busca o serviço de tradução de linguagem de marcação, S_y *= (HTML, XHTML*, _ , _), atendeu aos requisitos e a composição é concluída.

A extensão do FACI foi implementada em Java, tendo sido utilizadas várias *APIs* disponíveis publicamente. No componente *Ontologies Manager* foram usadas a *API OWL-S* [*OWL-S*], que estende a *API JENA* [*Jena*], e o banco de dados relacional *MySQL*. No *Reasoner* foi usada a *API Pellet* [*Pellet*] e no *Matching and Composition* foi usado o módulo *SPARQL* do *API OWL-S.*

4.1. Estudo de Caso

Para testar a performance e a capacidade de carga do FACI com suporte a ontologias e serviços *Web*, foi realizado um estudo de caso envolvendo dois computadores. Um com sistema operacional *Linux Fedora Core 2* (2Ghz – 256MB) e o outro com *Windows XP* (3Ghz – 1GB). Para evitar que alterações no tempo de resposta dos servidores de origem (incluindo variações no *throughput* da Internet) afetassem o resultado final, um servidor *Apache 2* foi instalado, empregando-se o recurso *hosts virtuais*. Isto permitiu a clonagem das páginas testadas, restringindo o fluxo de dados aos computadores envolvidos no estudo de caso. No *Linux* também foi instalado um servidor *Tomcat/Axis*, com os respectivos *remote adapters*, que exerceu o papel de *adaptation server*. Na plataforma *Windows XP* foram instalados o *adaptation proxy* com os perfis de serviços de classificação e filtragem de conteúdo e de conversão de linguagens de marcação.

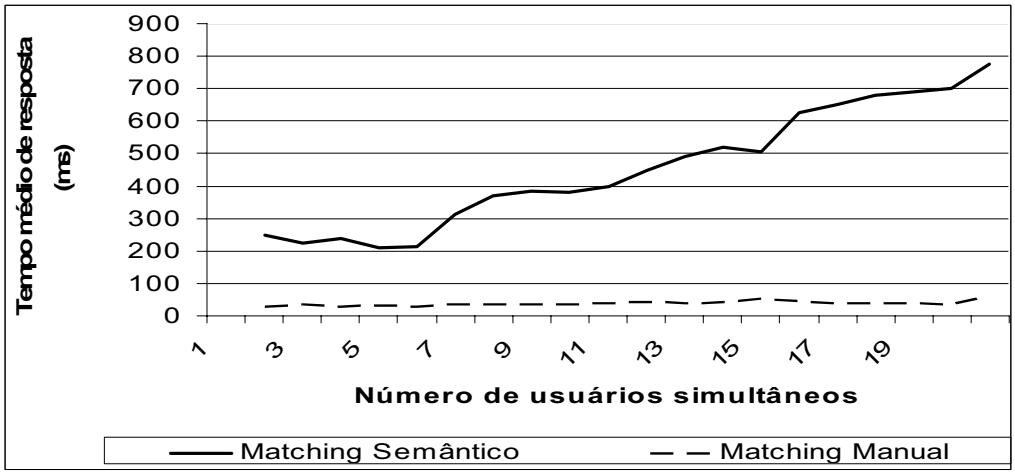


Figura 8. Tempo Médio de Resposta x Número de Usuários

A experiência consistiu em acessar 5 *links* pré-definidos durante 5 minutos. Desses *links* 3 eram bloqueados pelos seus endereços de domínio. Aleatoriamente foram atribuídos aos usuários dispositivos de acesso de mesa (e.g., *desktop*) e celulares, de modo a demandar, em alguns casos, a conversão da linguagem de marcação. Foi aplicada uma carga progressiva de usuários até o limite de 20 usuários, com cada usuário clicando num link a cada 5 segundos.

Foram definidos dois cenários de teste: o primeiro, utilizando-se ontologias, com a descoberta e a composição de serviços realizados sob demanda, onde se obteve um tempo médio de resposta de 453 ms; o segundo, sem a utilização de ontologias, com a descoberta e composição de serviços definidos manualmente, com um tempo médio de 39 ms.

Do ponto de vista quantitativo, analisando os resultados da Figura 8, fica evidente o aumento no tempo de resposta e na demanda por processamento, que foram causados pelo uso de ontologias no FACI reconstruído. Entretanto, do ponto de vista qualitativo, fica evidente também a flexibilidade e precisão, que o uso de semântica no mecanismo de descoberta e composição proporciona ao engenheiro de software [Yao, Su e Yang 2006]. É necessário destacar também que esse tempo de resposta tende a diminuir com o amadurecimento das *API´s* semânticas. Além disso, esse estudo de caso demonstra que a opção, de realizar a pesquisa e a composição de serviços internamente,

apresentou um ganho significativo no tempo de resposta (de 40 a 50%) em relação a outros estudos de caso que usam *UDDI's* adaptadas [Naumenko et al. 2005][Srínívasan Paoluccí e Sycara 2005].

5. Trabalhos Correlatos

O uso de ontologias, para a descrição de contexto e de serviços de adaptação, é uma linha de pesquisa recente, devendo receber um grande impulso quando novas especificações de padrões, que suportam as tecnologias de Serviços *Web* (e.g., *UDDI*, *WSDL*), proverem suporte nativo para as descrições semânticas.

Dentre os trabalhos correlatos destaca-se o de [Berhe, Brunie e Pierson 2004], que trata da especificação formal do processo de adaptação e da utilização de *proxies* de adaptação de conteúdo e o do protocolo *ICAP*, mas que não emprega ontologias. Em [Zahreddine e Mahmoud 2005] é apresentada uma solução baseada em *OWL-S* e *WSDL*, para adaptação de conteúdo, com foco no desenvolvimento de um módulo que substitui o *UDDI*, visando ganhos de desempenho. Em [Srínívasan, Paoluccí e Sycara 2005] é proposta uma *API*, que proporciona um mapeamento entre as informações de registro da *UDDI* e as descrições semânticas do *OWL-S*. Apesar das vantagens proporcionadas por essa API que permite o uso, de semântica pelos servidores *UDDI* distribuídos pela Internet, o estudo de caso apresenta valores de tempo de resposta que inviabilizam, no atual estágio, o seu uso no domínio da adaptação de conteúdo. Em [Geyter e Soetens 2005] é proposto um modelo, baseado em redes de tarefa hierárquica, para a composição e a descoberta de serviços, e uma nova especificação para a descrição de regras muito similar às definidas pelo *SWRL* e *OWL-S*.

Apesar da qualidade desses trabalhos correlatos, a pesquisa apresentada neste artigo destaca-se pelo uso de ontologias para a descrição de perfis, proporcionando uma definição mais precisa do contexto, que resulta em uma melhoria na busca de serviços de adaptação. A utilização de serviços *Web* simplifica o desenvolvimento de servidores de adaptação e facilitará a migração da solução proposta para os futuros padrões de *Web* semântica. Ambos, ontologias e serviços *Web*, foram incorporados a uma abordagem baseada em um *framework* de componentes. A sua característica modular oferece uma solução flexível, previamente testada, para o domínio de adaptação de conteúdo na Internet, facilitando assim a reutilização e a manutenção das aplicações.

6. Conclusão

Este artigo propõe o emprego de ontologias e serviços *Web* em um ambiente ubíquo, concentrando-se na solução de problemas relacionados com a adaptação de conteúdo em dispositivos móveis, tais como celulares e *palmtops*.

Tendo em vista que é essencial a definição de uma política de adaptação e que esta é fundamentada num conjunto de informações, expressas através de perfis, e num conjunto de regras de adaptação, foi proposto que esses perfis e essas regras fossem descritos através de ontologias, utilizando-se uma linguagem apropriada para as suas especificações. Essa linguagem apresenta uma fácil integração com linguagens orientadas a objeto, auxiliando na implementação e interoperabilidade da proposta. Também foi adicionado um suporte a serviços *Web*, com a integração do *SOAP e WSDL*, de modo a utilizar a infra-estrutura de ferramentas e serviços disponíveis, facilitando assim o crescimento do número de servidores de adaptação.

Para viabilizar esta proposta foi desenvolvida uma extensão do FACI, a partir da adição, adaptação e reuso de componentes. Essa extensão resultou em um *framework* mais genérico, facilitando o desenvolvimento de novas soluções no domínio da adaptação de conteúdo.

Para trabalhos futuros sugere-se: o aprimoramento do módulo de *Matching and Composition* para melhorar a performance na descoberta e composição de serviços e também possibilitar que essa tarefa possa ser realizada externamente, utilizando-se as novas especificações de serviços *Web* Semânticos que estão sendo desenvolvidas (e.g., *OWL-S/UDDI*, *WSDL-S*); a adição de um componente que gerencie a troca de perfis de serviços entre *Adaptation Proxies*; e o desenvolvimento de novos *local* e *remote adapters* de modo a ampliar a variedade de adaptações disponíveis.

7. Referências

Baader, F. et al. (2003) "Volume Handbook on Ontologies in Information Systems of International Handbooks on Information Systems", pp. 3-31, Steffen Staab and Rudi Studer, Eds., Springer.

Berhe, G.; Brunie, L.; Pierson, J. (2004) "Modeling Service-Based Multimedia Content Adaptation in Pervasive Computing", Conference of Computing Frontiers, pp. 60-69.

Berners-Lee, T.; Hendler, J.; Lassila, O. (2001) "The Semantic Web", Scientific American, edição de maio de 2001.

Cannistrà, F. (2003) "Exploiting Ontologies to Achieve Semantic Convergence Between Different CC/PP-like RDF Schemes for Representing Device´s Capabilities: the SADiC Approach", Second International Semantic Web Conference (ISWC-03).

Claudino, R. A. T.; Souza, W. L.; Prado, A. F. (2005) "Um framework baseado em componentes para o domínio de adaptação de conteúdo na Internet", Anais do 19º Simpósio Brasileiro de Engenharia de Software, pp. 88-103, Uberlândia, MG.

Elson, J.; Cerpa, A. (2003) "Internet Content Adaptation Protocol", IETF Request for Comments 3507, <http://www.isi.edu/in-notes/rfc3507.txt>.

EODM. (2006) "IBM Integrated Ontology Development Toolkit". <http://www.alphaworks.ibm.com/tech/semanticstk>.

Geyter, M.; Soetens, P. (2005) "A Planning Approach to Media Adaptation within the Semantic Web", DMS 2005, Banff, Canada.

Guarino, N. (1998) "Formal Ontology and Information Systems" in: N. Guarino, (Ed.) Formal Ontology in Information Systems. pp. 3-15, IOS Press, Netherlands.

Hansmann, U. et al. (2003) "Pervasive Computing", Springer-Verlag, Second Edition.

Horrocks, I. et al. (2003) "SWRL: A Semantic Web Rule Language Combining OWL and RuleML", DAML, <http://www.daml.org/2003/11/swrl/>.

Jena. <http://jena.sourceforge.net/>.

Klyne, G. et al. (2004) "Composite Capability/Preference Profiles (CC/PP): Structure and Vocabularies 1.0", W3C, <http://www.w3.org/TR/CCPP-struct-vocab/>.

Knublauch, H. et al. (2006) "A Semantic Web Primer for Object-Oriented Software Developers", <http://www.w3.org/TR/2006/NOTE-sw-oosd-primer-20060309/>.

Lara, R. et al. (2005) "Deliverable 2.4. Semantics for Web Service Discovery and Composition", KnowledgeWeb.

Martin, D. (2004) "OWL-S: Semantic Markup for Web Services", DAML, <http://www.daml.org/services/owl-s/1.1/>.

Martin, D. L.; Mcllraith, S. A. (2003) "Bringing Semantics to Web Services", IEEE Intelligent Systems, IEEE Press, pp. 90-93, USA.

McGuinness, D. L.; Harmelen, F. V. (2004) "OWL Web Ontology Language Overview", W3C, < http://www.w3.org/TR/owl-features/>.

Monola, F.; Miller, E. (1999) "Resource Description Framework (RDF) Model and Syntax Specification", W3C , <http://www.w3.org/TR/REC-rdf-syntax/>.

Naumenko A. et al. (2005) "Using UDDI for Publishing Metadata of the Semantic Web**"**, In: M. Bramer and V. Terziyan (Eds.): Industrial Applications of Semantic Web, Proceedings of the 1st International IFIP/WG12.5 Working Conference IASW-2005, Jyväskylä, Finland, Springer, IFIP, pp. 141-159.

OWL-S. <http://www.mindswap.org/2004/owl-s/api/>.

Pellet. <http://www.mindswap.org/2003/pellet/>.

Prud'hommeaux, E.; Seaborne, A. (2006) "SPARQL Query Language for RDF", W3C, <http://www.w3.org/TR/2006/WD-rdf-sparql-query-20060220/>.

Semantic Web Activity. <http://www.w3.org/2001/sw/>.

SOAP. "Simple Object Access Protocol", W3C, <http://www.w3.org/TR/soap/>.

Srínívasan N., Paoluccí M., Sycara K., (2005) "Adding OWL-S to UDDI, implementation and throughput". Robotics Institute, Carnegie Mellon University, USA.

WSDL. "Web Service Definition Language", W3C, <http://www.w3.org/TR/wsdl>

Yao, Y.; Su, S.; Yang, F. (2006) "Service Matching Based on Semantic Descriptions", In: Advanced International Conference on Telecommunications and International Conference on Internet and Web Applications and Services (AICT/ICIW), Guadeloupe, French Caribbean, pg. 126-131.

Zahreddine, W.; Mahmoud Q. H. (2005) "A Framework for Automatic and Dynamic Composition of Personalized Web Services", AINA 2005, pp. 513-518.

Uma proposta para a geração semi-automática de aplicações adaptativas para dispositivos móveis

Windson Viana[1, i], Rossana M. C. Andrade[2]

[1]LSR-IMAG, équipe SIGMA, Université Joseph Fourier (UJF), France

[2]Departamento de Computação (DC), Grupo de Redes, Engenharia de Software e Sistemas (GREat), Universidade Federal do Ceará (UFC)

Windson.Viana-de-Carvalho@imag.fr, rossana@ufc.br

Abstract. *Ubiquitous Computing promises seamless access to information anytime, anywhere with different and heterogeneous devices. When executing an application, this kind of environment demands adaptation of its user interface in order to provide transparent and consistent access. This paper proposes a construction-time approach for semi-automatic generation of adaptive applications to pervasive computers such as mobile phones and PDAs. The approach includes a device-independent user interface framework and a code generation tool for providing fast development of multi-platform and adaptable applications according to device and platform features. A case study is also presented to illustrate how the approach can be used for constructing an application to heterogeneous devices.*

Resumo. Computação Ubíqua promete acesso transparente à informação através de dispositivos diferentes e heterogêneos. A concepção de uma aplicação que executa neste tipo de ambiente apresenta como um dos desafios a adaptação da sua interface com usuário. Com o objetivo de prover o rápido desenvolvimento de aplicações para esse ambiente, este artigo propõe uma abordagem para a geração semi-automática de aplicações para dispositivos móveis multi-plataformas e com diferentes níveis de adaptação. Ela é composta por um *framework* de componentes de interface independentes de dispositivo e por uma ferramenta de geração de código. No artigo, é também apresentado um estudo de caso para ilustrar como o ambiente pode ser utilizado para a construção de uma aplicação para dispositivos heterogêneos e como o código gerado pode ser integrado com outros *frameworks* de desenvolvimento para DMs.

[i] Este trabalho foi financiado pela CAPES no Programa de Mestrado em Ciência da Computação da Universidade Federal do Ceará durante dois anos e pela LG Electronics, convênio UFC/FCPC N^{o} 1472 de março/2005 a setembro/2005.

1. Introdução

A evolução na capacidade de processamento e na miniaturização dos dispositivos computacionais, assim como, o aumento da disponibilidade dos meios de comunicação sem fio para esses dispositivos (e.g., *WiFi*, WiMax, UMTS) torna possível uma visão ubiqüitária da computação [19], na qual os dispositivos computacionais estão imersos no ambiente e permitem a um usuário o acesso à informação, em qualquer lugar e através de diferentes tipos de modalidades de interação (e.g., visual, auditiva).

Um exemplo de uma aplicação ubíqua é um sistema inteligente para transportes urbanos, onde um usuário, José, utiliza um painel interativo de uma estação de metrô para obter informações sobre as linhas e seus horários e, em seguida, para escolher sua estação de destino. A aplicação do painel, por sua vez, detecta a presença dos dispositivos pessoais de José, por exemplo, um *smartphone*, que são celulares com maior capacidade de processamento e recursos multimídia, e um relógio de pulso, ambos equipados com Bluetooth e transfere para estes dispositivos uma aplicação para ajudá-lo em seu itinerário. José, ao entrar em um vagão, consulta seu relógio que informa o horário previsto para sua chegada. Ele, então, decide utilizar seu *smartphone* para consultar informações sobre as linhas que fazem correspondência na estação destino e modifica a parada desejada. Durante o trajeto, José usa seu *smartphone* para ouvir MP3 e dedica-se a leitura de um jornal. Após vinte minutos, a música é interrompida e o seu relógio vibra para indicar que a parada desejada está próxima.

Aplicações como estas já se encontram em discussão em projetos europeus [7,11] e um dos aspectos críticos da construção desses sistemas é a premissa de que as aplicações e os serviços devem ser capazes de executar e de se adaptar à heterogeneidade dos dispositivos computacionais do usuário e do ambiente no qual ele está imerso [9,13]. No caso especial dos dispositivos móveis (DMs) como PDAs, celulares e *smartphones*, essa heterogeneidade é caracterizada por diferentes restrições de processamento, memória, bateria e largura de banda de comunicação. Eles possuem ainda diferenças nas formas de interação com o usuário e nas dimensões e na quantidade de cores dos *displays*. Além disso, o suporte a plataformas de programação varia de um dispositivo para outro, dificultando a adoção de uma única plataforma para o desenvolvimento das aplicações [13].

Dessa forma, a concepção e o desenvolvimento de aplicações que executam em dispositivos heterogêneos tornam-se desafios para o engenheiro de software [9]. Por exemplo, no sistema ubíquo descrito anteriormente, diferentes versões da aplicação de auxílio ao itinerário teriam que ser criadas, de forma a adequar sua interface, seu comportamento de execução e suas funcionalidades ao relógio e ao *smartphone* de José. Contudo, a criação de versões pode se tornar inviável e não escalável para codificação humana devido à variedade de plataformas e dispositivos disponíveis. Assim, existe a necessidade da construção de ferramentas que possam automatizar o processo de criação dessas aplicações [9].

Dentro desse contexto, este artigo propõe uma abordagem para a geração semi-automática de aplicações adaptativas para dispositivos móveis, com o objetivo de reduzir o tempo de prototipagem e facilitar a construção de aplicações com diferentes modos de conectividade à redes. Esta abordagem é composta por um *framework* de componentes de interface e por uma ferramenta para geração de código, que em

conjunto permitem o rápido desenvolvimento de aplicações multi-plataformas com diferentes níveis de adaptação de interface.

O restante deste artigo está organizado da seguinte forma: a seção 2 apresenta conceitos sobre geração interfaces adaptativas; a seção 3 apresenta a abordagem proposta; as seções 4 e 5 apresentam, respectivamente, o *framework* XformUI e o gerador de interface; a seção 6 discute os trabalhos relacionados; a seção 7 apresenta um estudo de caso que ilustra como o ambiente pode ser utilizado para construção de uma aplicação para dispositivos heterogêneos; e, finalmente, na seção 8 são apresentadas as considerações finais e possíveis trabalhos futuros.

2. Geração de Interfaces Adaptativas

A geração automática de interfaces de usuários (UIs) é extremamente útil no desenvolvimento de aplicações, pois proporciona a separação entre a descrição da interface e a lógica da aplicação [4]. Em geral, os ambientes para construção e geração de UIs utilizam uma hierarquia de camadas ou de modelos de descrição da interface para dissociar sua definição da forma como ela é renderizada [16]. Esses modelos de interface permitem a modelagem com níveis distintos de abstração, a reutilização das especificações das interfaces em diferentes fases de um projeto de aplicação, além de prover uma infra-estrutura para a construção de métodos e ferramentas para geração automática da apresentação final da interface [18]. Por exemplo, uma interface pode ser construída em três modelos: um modelo mais abstrato para descrever os componentes e as funcionalidades da interface; um segundo modelo para descrever a composição e o *layout*; e um terceiro modelo concreto no qual os componentes são mapeados para *widgets* reais de uma plataforma.

Em computação ubíqua, o uso da descrição da interface em níveis de abstração permite que para uma mesma interface possam ser definidos mais de um modelo de composição e *layout*, assim como modelos distintos de mapeamentos para *widgets* reais. A escolha de quais modelos são utilizados pode ser associada a diferentes contextos de utilização do usuário e, desta forma, a interface pode ser renderizada de forma adaptada ao dispositivo, à plataforma de programação, às preferências do usuário e às características do ambiente onde ele se encontra.

2.1. Linguagens de Descrição de Interface

Na literatura, são encontradas várias linguagens baseadas em XML concebidas com o objetivo de dissociar a definição da interface da sua apresentação final, tais como: UIML [22], XIML [16], UID [17], AWD [2], Teresa XML [10]. Essas linguagens possuem formas distintas de estruturação da interface e diferem entre si em relação ao poder de expressão, ao conjunto de componentes disponíveis e às técnicas de mapeamento da descrição abstrata para a apresentação final da interface. Por exemplo, UIML divide a definição da interface em cinco níveis: *description*, *structure*, *data*, *style* e *events*. Em <*description*>, <*structure*> e <*data*> descrevem-se os componentes, a sua composição e os dados a serem exibidos. Em <*style*> e <*events*> definem-se os mapeamentos dos componentes e seus eventos em uma linguagem real (e.g., Java). Já XIML utiliza uma estrutura de representação de ontologias, que é o par atributo-valor, para descrever os componentes, as relações e os atributos da interface, assim como as regras de mapeamento para uma determinada plataforma.

Neste trabalho, iremos utilizar XForms [24], um esforço do W3C para substituir formulários HTML por um formato mais adequado para execução em dispositivos heterogêneos. Seus componentes foram criados independentes do paradigma de acesso via *mouse* e da exibição visual em um monitor, o que torna a descrição do formulário independente da modalidade de interação (e.g., visual ou auditiva) e das características do dispositivo onde a aplicação executa. Por exemplo, os campos de entrada de dados, são descritos como "*inputs*" e não como campos de texto que ocupam determinadas posições na tela e os elementos que disparam eventos de ação são descritos como gatilhos (i.e., *triggers*) ao invés de descrevê-los como botões que recebem eventos de *click*. A estrutura de um documento XForms é baseada no modelo MVC (i.e., *Model/Viewer/Controller*) e é dividida em três partes: *Instance, Model* e *User Interface*. Essa estrutura é composta pelos dados iniciais, pelos componentes de interface e pelos eventos. Dessa forma, uma interface XForms já é construída com a separação dos dados, da lógica e da apresentação. Outra vantagem do XForms é permitir a integração com outras tecnologias como CSS, XML Schema, XPath e SVG, ampliando e enriquecendo as possibilidades de descrição da interface.

2.2. Estratégias de Geração de Código

Os trabalhos para geração automática de interfaces adaptativas para dispositivos móveis em geral utilizam duas estratégias para mapear a definição abstrata da interface na interface final a ser apresentada:

- **Geração em tempo de construção**. Neste caso as interfaces descritas são mapeadas antes da execução da aplicação e várias versões da interface da aplicação são geradas para diferentes contextos de execução. Exemplos de trabalhos que utilizam essas estratégias são MultiMad [1], SEFAGI [2], UID [17] e Teresa [10].

- **Geração em tempo de execução**. Neste caso, a definição da interface é transformada durante a execução da aplicação. Essa abordagem é usada, sobretudo, para adaptação de sistemas Web baseados em requisição/resposta, nos quais as interfaces são mapeadas para tecnologias Web como WML, XHTML e VoiceXML dependendo do dispositivo que faz a requisição. Os trabalhos LiquidUI [22], e GIA [6] utilizam essa estratégia.

A geração em tempo de execução limita os tipos de aplicações que podem ser construídas, já que exige que a aplicação esteja sempre conectada. E, em geral, com a utilização das tecnologias WML, XHTML e VoiceXML, essa estratégia restringe o acesso a funcionalidades dos dispositivos móveis [15]. No exemplo do sistema de auxílio a itinerário citado na Seção 1, usando essas tecnologias não seria possível acionar a função de vibração do relógio de José, assim como interromper a execução de música em seu *smartphone*. Neste trabalho é utilizada a estratégia de geração em tempo de construção, visto que o uso de geração em tempo de construção permite ao desenvolvedor acrescentar funcionalidades específicas pra cada versão da aplicação. Além disso, ela permite aos engenheiros de software a concepção de aplicações com tecnologias como J2ME MIDP, SuperWaba, Mophun e Doja I-Mode. Essas tecnologias apresentam maior interatividade, com a customização da entrada de dados e do acesso a diversas novas funcionalidades dos DMs (e.g.; *bluetooth*, câmera fotográfica, gps, mp3). Finalmente, elas também oferecem diferentes modos de

conectividade (e.g.; sempre conectado, desconectado com sincronização a posteriori) e diversos protocolos de comunicação (e.g.; HTTP, HTTPs, XML-RPC, SOAP).

3. Abordagem Proposta

Com o objetivo de auxiliar o processo de desenvolvimento de aplicações para DMs e fornecer mecanismos para amenizar o problema da heterogeneidade das plataformas de programação e dos dispositivos, foi proposta uma abordagem para geração semi-automática de aplicações adaptativas. Essa abordagem foi concebida para ser utilizada em processos de desenvolvimento orientados a prototipação e desta forma permitir rapidamente testes de usabilidade das aplicações.

3. 1. Requisitos

Para a concepção da abordagem foram utilizados alguns requisitos apontados em [8,14,5] como fundamentais para a concepção de ambientes de construção de interfaces adaptativas, a seguir:

A. Orientação ao processo de desenvolvimento e Interoperabilidade. Uma ferramenta de construção de interface deve ser facilmente inserida dentro de um processo de desenvolvimento de software. A sua forma de utilização, suas funções, seus documentos de entrada e saída não devem ser divergentes dos processos de engenharia de software. Além disso, a ferramenta tem que ser integrável com outros ambientes de desenvolvimento, pois a heterogeneidade dos dispositivos e das plataformas de programação torna impraticável a concepção de uma única ferramenta para todo o desenvolvimento da aplicação.

B. Riqueza de componentes e extensibilidade. É importante para um engenheiro de software ter a sua disposição uma variedade de componentes de interface, um conjunto de mecanismos de restrição e validação da entrada de dados, funções para navegação entre interfaces e exibição de mensagens, assim como mecanismos para o gerenciamento de eventos. Contudo, todas essas funcionalidades devem ser implementadas suportando a sua extensibilidade, já que as plataformas de programação e os dispositivos móveis evoluem rapidamente.

C. Independência de plataforma e dispositivo. O ambiente deve permitir a descrição da interface de forma dissociada de uma modalidade, de uma plataforma ou de um dispositivo para facilitar o processo de adaptação. Entretanto, essa descrição não pode ser restringida a um conjunto mínimo de funcionalidades comuns entre os dispositivos ou entre as plataformas de programação, pois isto limitaria o processo da adaptação da interface.

3.2. Componentes

A abordagem é composta pelo *framework* XFormUI de componentes de interface e pela ferramenta User Interface Generator (UIG) de geração de código.

O *framework* XFormUI fornece uma infra-estrutura para a construção de aplicações adaptativas baseadas em formulários. O XFormUI foi projetado para conter componentes que tivessem o mesmo nome e comportamento dos componentes do XForms, facilitando o seu aprendizado pelo engenheiro de software, assim como, auxilia a geração de código realizada pela UIG. Esta ferramenta, por sua vez, permite

ao engenheiro de software a definição das interfaces, dos estilos e da validação dos dados usando as seguintes normas técnicas (i.e.; *standards*) do W3C [24]: XForms, XMLSchema e CSS. A ferramenta utiliza esses documentos para gerar automaticamente código adaptável das interfaces das aplicações. O uso dos *standards* W3C facilita a aprendizagem e proporciona a integração do ambiente XMobile com outras ferramentas de desenvolvimento.

Na Figura 1, é apresentada uma visão geral da abordagem e do relacionamento entre o *framework* XFormUI e a ferramenta UIG. A abordagem permite ao engenheiro de software descrever os formulários de uma aplicação utilizando o XForms para definir os componentes, o CSS para definir o estilo (e.g., cores e *layout*) e o XML Schema para definir restrições aos campos de entrada de dados do formulário. O engenheiro, utilizando um plugin do Eclipse, cria um documento XML, o Manifest.xml, que define a navegabilidade entre as interfaces e as diretivas de mapeamento para um código executável de uma plataforma de programação. Ele dispara a ferramenta UIG e esta, por sua vez, gera o código dos formulários utilizando o *framework* XFormUI.

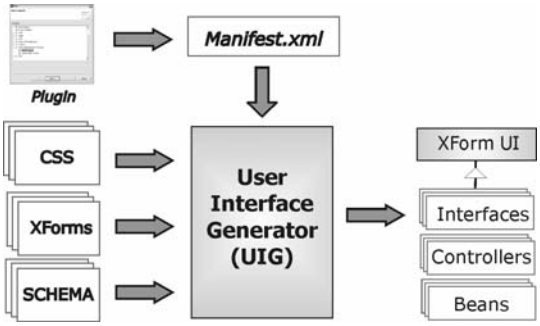


Figura 1 - Visão geral dos componentes

3.3. Tipos de Adaptação Utilizados

As plataformas de programação para DMs, como Superwaba e J2ME MIDP, oferecem algumas funcionalidades de adaptação dinâmica de interface que permitem uma certa independência de um dispositivo específico. Por exemplo, a disposição dos componentes na interface pode ser descrita de maneira relativa entre os componentes (um componente A à esquerda de um componente B). Além disso, os componentes de seleção e de ação podem alterar o estilo de apresentação de acordo com a forma de interação do dispositivo (e.g., via *stylus* ou via teclado do celular). No entanto, como mencionado na Seção 1, essas plataformas de programação não são suportadas de maneira homogênea pelos sistemas de operacionais de DMs, o que dificulta a adoção de uma única plataforma para o desenvolvimento das aplicações. Com o objetivo de solucionar este problema e de reutilizar as funções de adaptação já existentes nessas plataformas, a abordagem proposta neste artigo fornece três tipos de adaptação de interface: i) adaptação às classes de dispositivo, ii) adaptação à plataforma, e iii) adaptação ao dispositivo. Na Figura 2, são apresentados esses três tipos de adaptação e como elas são encadeadas durante o processo de geração de código.

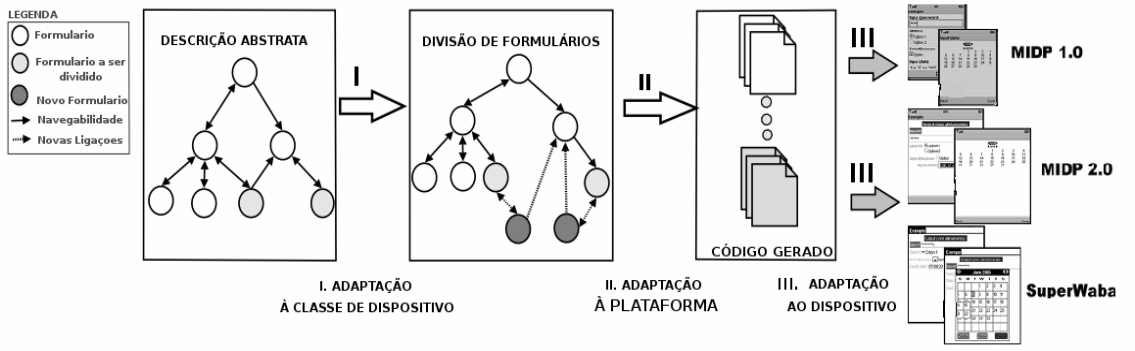


Figura 2 - Os três tipos de adaptação permitidos pelo ambiente

Os dois primeiros tipos de adaptação são fornecidos diretamente pela UIG através da geração de diferentes versões da aplicação. Contudo, **a adaptação ao dispositivo** provém da forma como o código é gerado pela ferramenta UIG. As interfaces geradas utilizam o *framework* XformUI que foi especialmente projetado para reutilizar as funcionalidades de adaptação dinâmica ao dispositivo disponível nas plataformas de programação.

A adaptação à classe de dispositivo consiste em limitar o número de componentes que podem ser apresentados simultaneamentes na interface. Para cada classe de dispositivo, a UIG estabelece um número máximo de componentes por interface. Quando uma descrição em XForm da interface possui mais componentes que o número máximo admitido, uma divisão automática da interface é realizada (o formulário é transformado em vários formulários para que todos os componentes sejam exibidos). Para permitir essa divisão, a descrição do formulário XForm é particionada em diferentes formulários através da utilização do padrão *Wizard Dialog Pattern* [21]. Esse padrão permite a construção de uma seqüência de formulários interligados e foi concebido inicialmente para adequar formulários Web a interfaces de aparelhos celulares. Além disso, aos novos formulários são acrescidos Triggers (i.e.; próximo e anterior) para realizar a interligação entre eles. E os Triggers iniciais contidos na descrição XForm original são colocados no último formulário da seqüência do *Wizard*. Dessa forma, o código gerado garante a similaridade e mantém a consistência da interface descrita pelo engenheiro de software. No ambiente, três classes de dispositivo foram definidas: "mobilephone", "smartphone" e "PDA". Para cada classe de dispositivo, um número máximo de componentes foi definido através de testes de utilização. Por exemplo, para a classe de dispositivo "mobilephone", o número máximo de componentes definido é oito. Assim se uma interface contém mais de oito componentes e se a classe de dispositivo escolhida para a transformação é "mobilephone", então a interface é dividida. Vale ressaltar que o desenvolvedor pode alterar, se desejar, o número máximo de componentes de cada classe de dispositivo.

A adaptação à plataforma de programação é realizada de duas maneiras na ferramenta UIG. A primeira forma consiste também na divisão de interfaces e é utilizada quando a plataforma de programação alvo da geração de código não permite a apresentação simultânea de certos componentes. Por exemplo, J2ME MIDP não permite apresentar numa mesma interface um componente de exibição de texto (TextBox) e outro campo de exibição de dados. Para resolver este problema, a

interface é dividida usando o padrão descrito anteriormente. A segunda forma de adaptação à plataforma é ligada às funcionalidades de estilo de apresentação (cores, *layout*). A ferramenta utiliza as informações descritas pelo CSS, apenas se a plataforma alvo da geração de código suporta funcionalidades de mudança de cores. Esta modificação de regras visa otimizar o código gerado a fim de utilizar da melhor forma possível os recursos de definição de interface disponíveis em cada plataforma.

4. Framework XFormUI

O *framework* XFormUI consiste em um *toolkit* de componentes de interface independente de uma plataforma de programação. Além dos componentes, o XFormUI disponibiliza funções para navegação entre interfaces, funções de restrição de dados de entrada (e.g., limitação de tipos numéricos e quantificadores existenciais), funções para validação automática de dados com exibição de mensagens de alerta e mecanismos para definição de *layout* e estilo dos formulários. Construído através de uma estratégia *bottom-up*, o *framework* oferece abstração das APIs de diferentes plataformas de programação para DMs e utiliza os recursos de adaptação de interface encontrados nestas plataformas. Durante a concepção do *framework* quatro estratégias foram utilizadas para atender os requisitos de riqueza de componentes, extensibilidade, adaptabilidade e independência de plataforma de programação (i.e., partes dos requisitos B e C mencionados na Seção 3.1.). Essas estratégias são listadas a seguir:

- **Generalização**. Para que o XformUI fosse implementado em várias plataformas, um estudo das principais plataformas de programação para DMs foi realizado e as características comuns foram capturadas e generalizadas. A generalização acontece principalmente em relação à estrutura das classes principais das plataformas que são instanciadas no início da aplicação (e.g., Midlet em J2ME MIDP e MainWindow em Superwaba). Outro exemplo de generalização usado no framework é o gerenciamento de eventos que contém apenas os eventos comuns às plataformas investigadas.

- **Transparência**. Todos os métodos contidos nas classes do *framework* abstraem da aplicação o modo pelo qual são instanciadas as APIs de cada plataforma para criar os componentes, montar a interface, disparar eventos e exibir mensagens de alerta. Essa estratégia permite às aplicações construídas executarem em qualquer plataforma na qual o *framework* seja implementado.

- **Especificidade**. Para evitar que o *framework* se limitasse a um conjunto mínimo de características comuns das plataformas de programação investigadas, funções que não correspondem a funcionalidades comuns foram adicionadas ao *framework*. Estas funções estão principalmente relacionadas ao gerenciamento de *layout*, que não é disponibilizado em certas plataformas de programação como J2ME MIDP 1.0. No caso em que funcionalidades não são possíveis de serem implementadas por completo em uma plataforma de programação, as implementações dos métodos correspondentes no *framework* podem não executar nada ou executar aquilo que se predispõem de forma parcial. Um exemplo dessa estratégia são os métodos para mudança de cores dos componentes que, na implementação em J2ME MIDP 1.0, não têm nenhum código para execução. Isto acontece devido a essa plataforma não suportar mudança de cores para os componentes da API. A vantagem do uso dessa estratégia é que a

aplicação é executada da melhor forma em cada uma das plataformas (e.g., se a plataforma suporta cores, então os componentes da aplicação ficam coloridos).

• **Combinação de Componentes.** Nem todos os componentes existentes no XForms têm componentes correspondentes que executam o mesmo comportamento nas plataformas de programação. Nesses casos, componentes das plataformas são combinados para construir o componente do XFormUI e garantir, em parte, a similaridade entre as implementações.

Na Figura 3, é apresentada uma visão geral do XFormUI que é dividido em quatro partes: Formulários; Núcleo e Eventos; Componentes; e Validação e Restrição. Três tipos de formulários podem ser construídos com o *framework* XFormUI: *TextArea*, *Select1Full* e *XForm.* O *TextArea* corresponde a um formulário com uma única caixa de texto que deve ocupar toda a tela do dispositivo. O *Select1Full* é um formulário com um menu para escolha de uma única opção. O *XForm*, por sua vez, corresponde ao formulário padrão ao qual pode-se adicionar vários componentes (e.g., campos de entrada de dados, componentes de escolha de opções). A parte de Núcleo e Eventos contém as três principais classes do *framework*: *MainXFform, XFormItem* e *Trigger.* A classe MainXForm é uma classe abstrata que contém os métodos *onStart()* e *onExit()* que devem ser reescritos pelo engenheiro de software para definir o que ocorre ao iniciar aplicação e ao sair dela. Essa classe implementa os métodos obrigatórios das estruturas de cada plataforma e mapeia suas chamadas para os seus métodos abstratos.

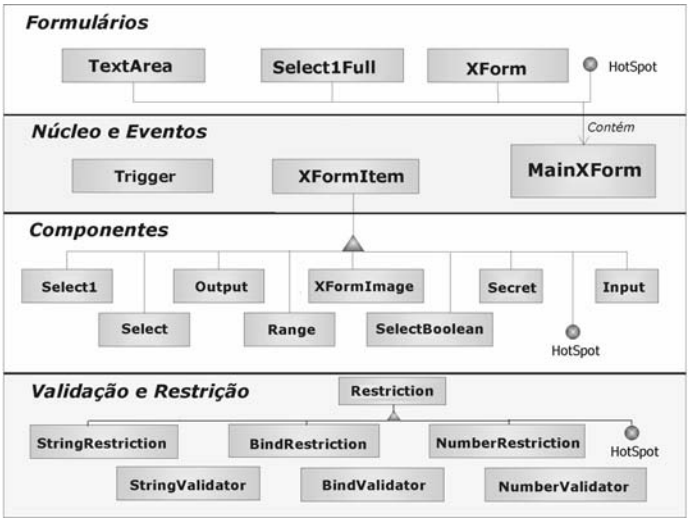


Figura 3 - *Framework* XFormUI

A parte Componentes é composta pelas oito classes disponibilizadas pelo *framework* para construção de formulários do tipo *XForm*. Essas classes em conjunto permitem a entrada de dados, como texto, datas e senhas; a exibição de textos e imagens, e a seleção de opções. A parte Validação e Restrição é composta pelas classes que permitem ao engenheiro de software definir restrições à entrada de dados nos componentes e testar se os dados não violam essas restrições. As classes *StringRestriction, StringValidator, BindRestriction, BindValidator*, *NumberRestriction* e *NumberValidator* compõem esta parte. Por exemplo, restrições como quantidades de caracteres máximas e mínimas de um texto e intervalo de validade para um valor numérico podem ser expressas usando métodos dessas classes. Na Tabela 1 pode ser

visto como algumas classes e métodos do *framework* são mapeados para as plataformas de programação.

XFormUI	J2ME MIDP 1.0	J2ME MIDP 2.0	Superwaba
MainXFform	*Midlet*	*Midlet*	*MainWindow*
TextArea	*TextBox*	*TextBox*	*Container* com um *ListBox*
XFormItem	*Item*	*Item* e *ColorLabel*	*Control* e *Label*
Input	Pode conter um *TextField* ou um *DateField*	*ColorLabel e* pode conter um *TextField* ou um *DateField*	*Edit* e *Label*. No caso da restrição de datas, um *Calendar* é associado
Output	*StringItem*	*StringItem* e *ColorLabel*	*Label*
Select1	*ChoiceGroup* com restrição *Exclusive*	*ColorLabel* e *ChoiceGroup* com restrição *Exclusive* ou *Popup*	*Label* e *ComboBox* ou *RadioGroup* ou *um ListBox*
triggerEvent()	*commandAction()*	*commandAction()*	*onEvent()*
setForeColor()	-	*ColorLabel.setForeColor()*	*Control.setForeColor()*

Tabela 1- Mapeamento das classes do *framework*

5. Ferramenta de Geração da Interface do Usuário

A ferramenta User Interface Generator (UIG) foi construída com o objetivo de mapear os documentos que descrevem uma aplicação (i.e., XForms, CSS, XMLSchema e Manifest.xml) em um código executável. A ferramenta realiza essa geração de código utilizando XSL [24] (i.e., eXtensible Stylesheet Language). Essa linguagem de marcação baseada em XML permite a criação de folhas de estilo ou documentos de transformação chamados XSLT (i.e., XSL Transformation). O XSLT permite a definição de regras de mapeamento das tags de um documento XML para uma outra descrição. A vantagem do uso do XSLT decorre da existência de vários processadores ou parsers XSL, capazes de realizar a transformação a partir de dois documentos: o documento XML original e o documento XSLT de transformação.

Na Figura 4 é apresentado o processo de geração de código da ferramenta UIG. Como mencionado anteriormente, a ferramenta utiliza XSLT para realizar as transformações dos documentos XMLs para código Java que estende o *framework* XFormUI. Contudo, uma das entradas da ferramenta é um documento CSS que não é baseado no padrão XML. Assim, para realizar o processo de mapeamento, o primeiro passo é transformar o CSS em um documento XML. Para isso, foi criado um *Document Type Definition* (DTD) que especifica um documento XML para suportar os parâmetros de um documento CSS, o XCSS. Após a geração do XCSS, esse documento, o documento XForms e o XML Schema são transformados para um documento intermediário: o XMLMiddleForm.

O XMLMiddleForm é a forma canônica definida pelo UIG para facilitar o mapeamento para as plataformas de programação. Os documentos XForms, XCSS e XMLSchema são primeiramente transformados para XmlMiddleForm e depois de XMLMiddeForm para as plataformas de programação. Essa abordagem é a aplicação do padrão de geração de código Meta-Model, descrito em [23], que tem como vantagem facilitar a extensibilidade da ferramenta para a adição de novos tipos de mapeamentos (e.g., para a plataforma de programação BREW).

Após a geração do XMLMiddleForm, o Manifest.xml é lido e são identificadas a plataforma e a classe de dispositivo desejadas pelo engenheiro de software para realizar a transformação. Assim, a UIG passa para o processador XSLT o *template*

específico da classe do dispositivo que verifica a necessidade da realização da divisão dos formulários. Em seguida, os *templates* que mapeiam de XMLMiddleForm para classes Java que instanciam XFormUI são processados. A construção da ferramenta UIG foi realizada em Java e utilizou o JAXP (i.e., Java API for XML Processing). O JAXP contém várias classes que facilitam a manipulação de documentos XML e um processador XSLT capaz de realizar as transformações desejadas.

O código dos formulários gerados pela ferramenta segue um modelo semelhante ao MVC (i.e., *Model/Viewer/Controller*). Para cada formulário, três classes são geradas: *Interface*, *Bean* e *Controller*. A classe *Interface* define a visão do formulário na qual estão descritos os componentes gráficos e a sua composição. Esses componentes gráficos utilizados são instâncias das classes que fazem parte do *framework* XFormUI. Além disso, diferentes construtores são gerados para a classe *Interface* permitindo a sua instanciação com dados estáticos e dados dinâmicos. A classe *Bean* contém atributos relacionados aos campos de seleção e de entrada de dados do formulário, assim, essa classe corresponde ao modelo no qual os dados do formulário estão relacionados. Por sua vez, a classe *Controller* contém métodos de execução para cada *Trigger* existente nos formulários, além dos métodos para validação de dados e de navegabilidade. Esse modelo de código gerado permite o desenvolvimento mais rápido das aplicações, pois o engenheiro se concentra na escrita dos métodos da classe controladora referentes às regras de negócio da aplicação.

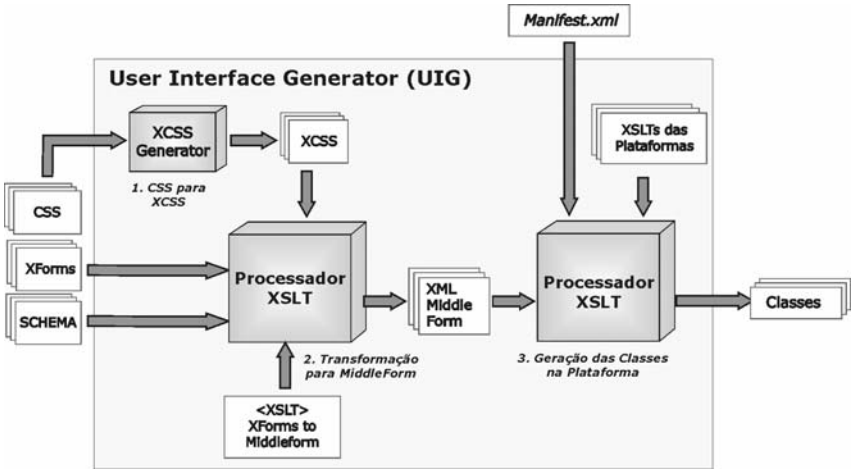


Figura 4 - Processo de mapeamento realizado pela ferramenta UIG

6. Trabalhos Relacionados

Nesta seção, são apresentados os trabalhos que possuem estreita relação com a abordagem apresentada neste artigo. Na Tabela 2, é apresentada uma comparação desses trabalhos com a abordagem discutida nas seções anteriores. Nesta tabela podem ser vistos os tipos de adaptação de interface suportados pelos ambientes e as principais diferenças entre as abordagens, a seguir: tipos de aplicações, plataformas, componentes de interfaces e outras características específicas.

O MultiMad, por exemplo, é uma ferramenta visual para geração de aplicações multi-modelos para dispositivos móveis [1]. Ela permite a construção de quatro tipos de formulários com um conjunto limitado de componentes genéricos. Contudo, a ferramenta não oferece outros mecanismos de adaptação, além da geração de código multi-plataforma. O SEFAGI, por sua vez, é uma arquitetura para geração automática

de interface para aplicações baseadas em Web-Services [2]. As janelas das aplicações são definidas em XML AWD (*Abstract Window Description*) através da composição de painéis (i.e., painel de exibição de vídeo, painel de exibição de uma tabela), na qual cada painel é associado a um serviço Web. Regras de adaptação são definidas para gerar as interfaces em código executável. Essas regras definem o mapeamento de *widget* descrito em AWD para um componente de uma plataforma, assim como, o *layout* da interface (e.g.; posição dos componentes, divisão da janela). A principal desvantagem do uso do SEFAGI decorre da necessidade de criar, para cada novo componente construído, um componente correspondente na descrição do AWD. Além de limitar a aplicação à um cliente que acessa um Web-Service.

	Tipos de Aplicações	***Plataformas***	***Componentes de Interface***	***Tipos de Adaptação de Interface***	***Características Específicas***
MultiMad	Aplicações *online/offline*	WML 1.0/2.0 e J2ME MIDP 1.0 e 2.0	Listas, Textos, Formulários com campos de entrada de dados e datas	Adaptação estática à plataforma	Ferramenta Visual, Passagem de parâmetros e código MobileVC
SEFAGI	Aplicações *online* baseadas em *Web-Services*	J2SE e J2ME	Painéis com grupos de componentes (grid, lista, imagens)	Adaptação estática à plataforma e dinâmica à classe de dispositivos	Tabelas de regras de adaptação para mapear AWD em componentes específicos. Divisão de formulários
Roam	Aplicações moveis	J2SE, Personal Java e J2ME Doja	Componentes semelhantes a API Java Swing	Adaptação dinâmica à classe de dispositivos	Migração de código, recuperação de estado da aplicação
UIG + XFormUI	Aplicações *online/offline*	Superwaba, J2ME MIDP 1.0 e 2.0	10 componentes do XForms e o *image* do XHtml	Adaptação estática à plataforma e à classe de dispositivos. Adaptação dinâmica ao dispositivo	XSLT para geração de código. Suporte a CSS e XML Schema. Plugin Eclipse e Manifest.xml para a navegação entre as interfaces. Divisão de formulários

Tabela 2 –Trabalhos relacionados com a geração de interfaces adaptativas

Roam é um *framework* baseado em componentes para construção de aplicações que executam em dispositivos heterogêneos e que podem migrar entre eles [13]. Roam permite ao desenvolvedor definir a interface da aplicação através de um toolkit de componentes de interface independentes de dispositivos. Esses componentes assemelham-se aos componentes gráficos da API Java Swing. Este *framework* utiliza uma estrutura de agentes para garantir a captura e a transferência do estado da aplicação quando esta migra entre os dispositivos. O engenheiro de software descreve como os componentes devem ser adaptados para um par plataforma-dispositivo e se a lógica da aplicação deve migrar para um dispositivo de maior capacidade (e.g., um computador desktop). Durante a migração da aplicação, um agente Roam é responsável por escolher como a interface será exibida no novo dispositivo através das estratégias de adaptação descritas.

A nossa abordagem tem como principal diferencial o suporte tanto de adaptação estática, com a ferramenta UIG, quanto de adaptação dinâmica, com o *framework* XFormUI. E ao contrário dos trabalhos Roam e SEFAGI, não existe uma limitação dos tipos de aplicação que podem ser construídas. Além disso, o uso de normas técnicas do W3C facilita o aprendizado e interoperabilidade da ferramenta UIG.

7. Estudo de Caso

Para validar a abordagem, foi construído um *mobile fotolog* que permite aos usuários publicarem suas fotos e comentários utilizando um PC ou um dispositivo móvel. O portal, chamado de M-Flog, pode ser acessado por uma grande quantidade de usuários e deve responder em um tempo satisfatório às requisições. Durante a concepção do portal, requisitos não funcionais foram identificados:

- **Adaptação da aplicação cliente**. Diferentes usuários irão utilizar o M-Flog em dispositivos com características diferentes entre si. Desta forma, a aplicação deve ser capaz de ser executada em uma grande diversidade de dispositivos e plataformas de programação.

- **Adaptação das imagens acessadas.** A adaptação das imagens é necessária devido à dificuldade que um dispositivo móvel pode ter para conseguir acessar uma imagem fotográfica da *Web*, por causa do seu tamanho e formato.

- **Diferentes modos de conectividade.** A aplicação deve permitir o armazenamento e o acesso das imagens preferidas dos usuários no próprio dispositivo de forma a evitar a troca excessiva de pacotes.

- **Acesso a funcionalidades multimídia.** O cliente do M-Flog pode utilizar, em caso de existência, a câmera fotográfica do dispositivo móvel e o protocolo MMS para envio de fotos.

Este exemplo simples de aplicação não pode ser construído com os trabalhos [6][22] para adaptação de interface baseados somente em tecnologias Web para celulares (i.e., WML e XHTML) e com os trabalhos baseados em modelo de conectividade *online* [2][13], já que os dois últimos requisitos não podem ser satisfeitos. Desta forma, a proposta deste artigo mostra-se mais adequada. Na Figura 5, uma visão geral do M-Flog é apresentada. Ele é composto de uma aplicação cliente que executa em DMs e de um servidor Web. Para construir o cliente, os formulários foram especificados em XForms, CSS e XML Schema.

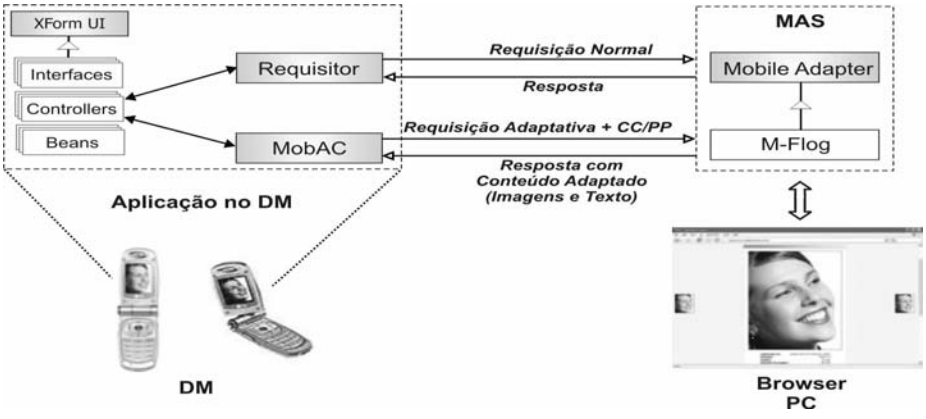


Figura 5 - Visão geral do M-Flog

Após a escrita dos formulários, um Manifest.xml foi criado com as diretivas de mapeamento e a descrição da navegação. Foram realizadas três gerações de código: J2ME MIDP 1.0 com a classe de dispositivo "mobilephone", J2ME MIDP 2.0 com a classe de dispositivo "smartphone" e SuperWaba com a classe de dispositivo "PDA". O tempo máximo decorrido para a geração de código de todos formulários para as três

plataformas foi de 9,52 segundos. Vale ressaltar que a UIG foi executada em uma máquina AMD Atlhon 2400Hz, 512 MB de RAM com Windows XP Pro.

O servidor do M-Flog foi construído com a arquitetura MobileAdapter [3] que proporciona a construção de aplicações para a adaptação da apresentação de conteúdo baseada nas preferências do usuário e nas características do dispositivo e do ambiente de execução. Assim, foi construído um módulo para a adaptação de imagens do M-Flog que utiliza as dimensões do dispositivo e as preferências de exibição descritas pelo usuário, mais detalhes sobre esse processo de adaptação são descritos em [3].

Para realizar a comunicação entre as aplicações geradas e o servidor do M-Flog, as classes controladores do código gerado foram associadas aos *frameworks* Requisitor e MobAC [3] que permitem a comunicação e aquisição de contexto nas três plataformas alvo. Esses códigos escritos nos controladores executam nas três plataformas de programação, já que elas são baseadas em Java e os códigos utilizam os *frameworks* que são também implementados nas plataformas. Desta forma, todas as funcionalidades de validação dos dados, troca de formulários e comunicação com o servidor do M-Flog foram escritas apenas uma vez pelo engenheiro de software.

O armazenamento local das imagens preferidas do usuário foi realizado utilizando o modelo de serialização, persistência e busca proposto em [12], que permite sua implementação nas três plataformas através do mapeamento das classes do FramePersist para as APIs RMS e Catalog respectivamente de J2ME MIDP e SuperWaba. O acesso a câmera foi realizado apenas na plataforma J2ME MIDP 2.0 com uso da Mobile Media API (MMA). As versões do cliente M-Flog foram executadas em três dispositivos, um HP iPaq 4100 com SuperWaba, o celular motorola A388 com MIDP 1.0 e no *smartphone* P900 com MIDP 2.0. Na Figura 6, é apresentado um fluxo de execução da aplicação cliente do M-Flog no emulador do P900. Neste fluxo, podem ser vistos os formulários criados para entrada de dados de *login*, visualizações de imagens e comentários, escolha de opções, busca de imagens, mudança das preferências do usuário e adição de novos comentários.

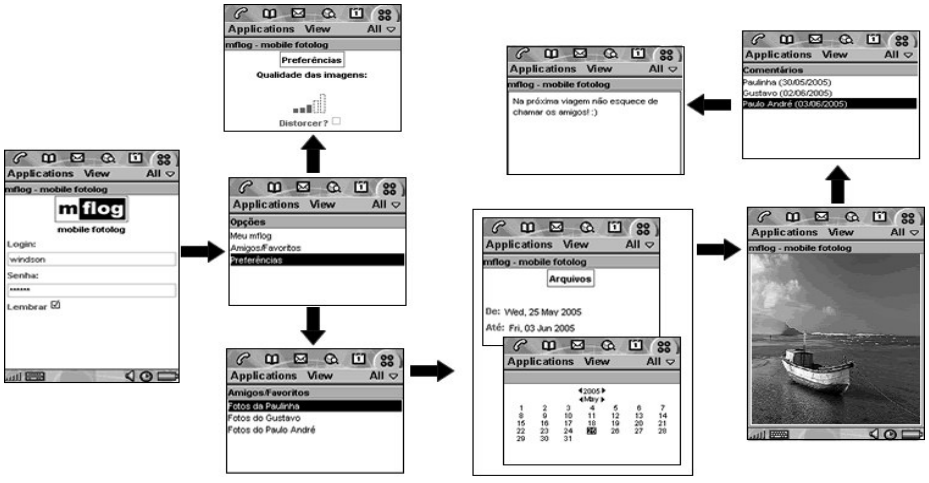


Figura 6 - Fluxo de execução do cliente do M-Flog no emulador do P900

8. Conclusão e Trabalhos Futuros

A abordagem proposta deve facilitar a rápida prototipagem das interfaces de aplicações para aplicações pervasivas em dispositivos móveis heterogêneos e proporcionar

diferentes níveis de adaptação. Por um lado, o *framework* XFormUI possibilita a construção de interfaces gráficas com recursos de validação de dados e *layout* de componentes. Suas implementações permitem ao engenheiro de software escrever uma única vez as interfaces gráficas e executá-las de forma adaptativa em três plataformas. Por outro lado, a ferramenta UIG de geração de código permite a definição das interfaces, dos estilos e da validação dos dados usando as linguagens declarativas XForms, CSS e XML Schema. Esta ferramenta proporciona ainda a adaptação da interface em relação a classes de dispositivos, com a divisão de formulários, e gera automaticamente o código para navegação entre eles. Outra importante contribuição dessa ferramenta é o modelo de código gerado que possibilita ao engenheiro de software escrever apenas os controladores dos formulários e, dessa forma, concentrar-se nas regras de negócio da aplicação, uma vez que as interfaces já estão construídas.

O estudo de caso apresentado ilustra a capacidade da abordagem de construir uma aplicação para dispositivos heterogêneos com diferentes modos de conectividade. Além disso, ela demonstra como o código gerado pela UIG pode ser facilmente integrado a outros *frameworks* de desenvolvimento.

Como trabalhos futuros, a abordagem pode ser estendida com a adição de componentes com outras modalidades de interface (e.g., entrada/saída de dados em áudio). Além disso, um estudo da usabilidade das interfaces geradas pela ferramenta deve ser realizado para identificar o grau de interferência do uso da adaptação na usabilidade da aplicação.

Referências

1. FIGUEIREDO, Thiago H., LOUREIRO, A. A. F. “MultiMAD: Uma Ferramenta Multimodelo de Desenvolvimento de Aplicações para Dispositivos Móveis”. Anais do Salão de Ferramentas, XXIII Simpósio Brasileiro de Redes de Computadores, Fortaleza-CE, Brasil, 2005.

2. CHAARI, T.; LAFOREST, F. “SEFAGI: Simple Environment For Adaptable Graphical Interfaces”. Anais do VII International Conference on Enterprise Information Systems (ICEIS), Miami, USA, 2005.

3. VIANA, Windson; FERNANDES, Paula; TEIXEIRA, Robson; ANDRADE, R. M. C. "Mobile Adapter: Uma abordagem para a construção de Mobile Application Servers adaptativos utilizando as especificações CC/PP e UAProf". Anais do XXXII Seminário Integrado de Software e Hardware (SEMISH 2005). São Leopoldo-RS, Brasil, 2005.

4. NICHOLS, Jeffrey; FAULRING, Andrew. “Automatic Interface Generation and Future User Interface Tools”. Anais do Workshop: The Future of User Interface Design Tools, ACM CHI 2005, Portland, USA, 2005.

5. PUERTA, A. “A Better Future for UI Tools through Engineering”. Anais do Workshop: The Future of User Interface Design Tools, ACM CHI 2005, Portland, USA, 2005.

6. ITO, G.; ROCHA, R.; GONÇALVES, M.; SANTANNA, N. “Uma Arquitetura para Geração de Interfaces Adaptativas para Dispositivos Móveis”. Anais do 4th International Information and Telecommunication Technologies Symposium (I2TS), Florianopolis, Brasil, 2005.

7. COUTAZ, J.; CROWLEY, J. L.; DOBSON, S.; GARLAN, D. “Context is key”. *Communications of the ACM*, vol.48, nº 8, p.49-53. 2005.

8. GAJOS, Krzysztof; WU, Anthony; WELD, Daniel S. "Cross-Device Consistency in Automatically Generated User Interfaces". Anais do Workshop on Multi-User and Ubiquitous User Interfaces (MU3I'05). San Diego, CA, 2005.

9. GAJOS, Krzysztof; WELD, Daniel S. "SUPPLE: automatically generating user interfaces". Anais do Intelligent User Interfaces (IUT'2004). Funchal, Portugal, 2004.

10.PATERNO, F.; MORI, Giulio; SANTORO, Carmen. "Design and Development of Multidevice User Interfaces through Multiple Logical Descriptions". *IEEE Transactions on Software Engineering*, vol. 30, nº 8, p. 507-520, 2004.

11. RIOULT, Jean; GRANSART,Christophe; AMBELLOUIS, Sebastien. « Zut, j'ai loupé mon arrêt ! Un nouveau service d'aide aux déplacements ». Anais do II Colloque Les nouvelles technologies dans la cité (em françês). Lille, França, 2004.

12.MAGALHÃES, Katy; VIANA, Windson; LEMOS, Fabrício; CASTRO, Javam de; ANDRADE, Rossana. "FramePersist: An Object Persistence Framework for Mobile Device Applications". Anais do Brazilian Symposium on Databases (SBBD), Brasilia-DF, 2004.

13.CHU, H.; SONG, H.; WONG, C.; KURAKAKE, S.; KATAGIRI, M.. "Roam, a seamless application framework". *Journal of Systems and Software*, vol. 69, nº 3, p.209-231, 2004.

14.WEGSCHEIDER,Florian; DANGL,Thomas; JANK, Michael; SIMON, Rainer. "A Multimodal Interaction Manager for Device Independent Mobile Applications". Anais do XIII ACM International World Wide Web Conference, New York, USA, 2004.

15.READ, Kris; MAURER, Frank. "Developing Mobile Wireless Applications". *IEEE Internet Computing*, vol. 07, no. 1, p. 81-86, 2003.

16.PUERTA, Angel; EISENSTEIN, Jacob. "XIML: A Universal Language for User Interfaces". Anais do IUI 2002 - Intelligent User Interfaces. San Francisco, USA, 2002.

17.MUELLER, A.; MUNDT, T.; LINDNER, W. "Using xml to semi-automatically derive user interfaces". Anais do Uidis 2001 - II International Workshop on User Interfaces to Data Intensive Systems, Zurique, Suiça, 2001.

18.DA SILVA, P.P. User Interface Declarative Models and Development Environments: a Survey. Anais do DSV-IS'2000, 2000.

19.WEISER, M. The computer for the 21st century. Scientific American, 1991.

20.Open Mobile Aliance. Criadores do Wap e do UAProf. Disponivel em: http://www.openmobilealliance.org/. Acessado em Dezembro de 2005.

21.HUI, Ben. Big Design for Small Devices. Design Patterns for J2ME. Disponivel em: http://www.javaworld.com/javaworld/jw-12-2002/jw-1213-j2medesign.html. Acessado em Agosto de 2005.

22.Site do LiquidUI, ferramenta de UIML. Disponível em: <http:// www.harmonia.com>. Acessado em Março de 2006.

23.VOELTER, M., "A Catalog of Patterns for Program Generation". Anais do, EuroPlop'2003. Alemanha, 2003.

24.W3C- WWW CONSORTIUM, Site do fórum de desenvolvimento de tecnologias para Web. Disponível em:<http::\\www.w3c.org >. Acessado em Janeiro de 2006.

Sistemas de Software Transparentes[1]

Julio Cesar Sampaio do Prado Leite

PUC-Rio – Pontifícia Universidade Católica do Rio de Janeiro

julio@inf.puc-rio.br

Abstract

Transparency has been, for long, a general requirement for democratic societies. The right to be informed and to have access to the information has been an important issue on modern societies. People want to be informed on a proper way. As such, transparency is a well regard characteristic for organizations. However, as software permeates several aspects of our society, at some point in the future, software engineers will need to deal with yet another demand: transparency. In such foreseen environment, engineers will need to have methods, techniques and tools to help make transparent software.

Resumo

Transparência foi, por muito tempo, um requisito geral para sociedades democráticas. O direito a ser informado e a ter acesso à informação tem sido uma questão importante nas sociedades modernas. Pessoas querem ser informadas de maneira apropriada. Por isso, transparência é uma característica bem vista nas organizações. Entretanto, à medida que o software permeia vários aspectos de nossa sociedade, em algum momento do futuro, engenheiros de software terão que cuidar de mais uma demanda: transparência. Nesse ambiente imaginado, engenheiros deverão ter métodos, técnicas e ferramentas para auxiliar a produção de software transparente.

[1] O presente resumo é uma edição do que foi publicado em/This summary is na edited version of a text published in: http://jcspl.wordpress.com/2006/05/19/transparencia-desafios-para-a-engenharia-de-software

Linhas de Produto de Software: Reuso que faz Sentido para Negócios

Software Product Lines: Reuse That Makes Business Sense

Linda Northrop

Software Engineering Institute

Abstract

Traditionally, software-intensive systems have been acquired, developed, tested, and maintained as separate products, even if these systems have a significant amount of common functionality and code. Such an approach wastes technical resources, takes longer, and costs more than necessary. A product line approach to software can reduce development cycles, improve return on software investments, improve software system integration, and give an organization more future options. Building a new product or system becomes more a matter of assembly or generation than creation, of integration rather than programming. Organizations of all types and sizes are discovering that a product line strategy, when skillfully implemented, can improve productivity, quality, and time to market. Software product lines present at long last a reuse strategy with real economic benefit.

Making the move to product lines, however, is a business and technical decision and requires considerable changes in the way organizations practice software engineering, technical management, and organizational management. This talk will explore the basic concepts of software product lines, share experience reports from companies employing the paradigm, and identify the software engineering and management practices necessary to develop a successful software product line.

Resumo

Tradicionalmente, sistemas que fazem uso intensivo de software têm sido adquiridos, desenvolvidos, testados e mantidos como produtos separados, mesmo se estes sistemas tem uma grande quantidade de funcionalidades e códigos em comum. Esta abordagem desperdiça recursos técnicos, leva mais tempo e custa mais do que o necessário. Uma abordagem de linha de produto em software pode reduzir os ciclos de desenvolvimento, aumentar o retorno dos investimentos em software, melhorar a integração de sistemas de software e dar a uma organização mais opções futuras. Construir um novo produto ou sistema torna-se mais uma matéria de montagem ou geração do que criação, e de integração ao invés de programação. Organizações de todos os tipos estão descobrindo que uma estratégia de linha de produto, quando

implementada com competência, pode melhorar a produtividade, qualidade e o tempo para o mercado. Linhas de produto de software apresentam uma estratégia de reuso com reais benefícios econômicos.

Mudar para linhas de produtos, entretanto, é uma decisão técnica e de negócio que requer mudanças consideráveis na maneira como as organizações praticam engenharia de software, gerenciamento técnico, e gerenciamento organizacional. Esta palestra vai explorar os conceitos básicos de linhas de produtos de software, compartilhar relatos de experiência de companhias que empregam o paradigma e as práticas gerenciais necessárias para desenvolver uma linha de produtos de software de sucesso.

Evolução de Software

Software Evolution

Júlio César S. do P. Leite

PUC-Rio – Pontifícia Universidade Católica do Rio de Janeiro

julio@inf.puc-rio.br

Abstract

I believe that there is no way to speak about software construction without taking into account the concept of evolution as a central issue. To focus on the concept of evolution is a paradigm shift. The great challenge for software evolution is how to maintain coherence and consistency of the evolving artifacts. Our mini-tutorial characterizes the concept of evolution and its problems. It will focus on the idea of maintaining traceability among the many software artifacts. When characterizing evolution we will use the Lehman´s laws of evolution, and their effects on software construction. Special attention will be given to the concepts of version control and configuration management: key pieces to adequately approach software evolution.

Resumo

Acredito que hoje não exista uma maneira de falar da construção de software, se a mesma não for centrada no conceito de evolução. Centrar-se no conceito de evolução é uma mudança de paradigma no processo de produção de software. O grande desafio da evolução é o de manter coerência e consistência nos artefatos gerados pelo processo. Nosso mini-tutorial vai caracterizar o conceito de evolução, seus problemas, e vai concentrar-se na idéia da manutenção de rastros entre os vários artefatos de software. Na caracterização de evolução usaremos as leis de Lehman e falaremos sobres seus efeitos na produção de software. Especial atenção será destinada aos conceitos de controle de versões e gerência de configuração: peças chave para tratar a evolução de software

Dois tutoriais em um: O Método ATAM e Atributos de Qualidade em Arquiteturas Orientadas a Serviços

Two Talks In One: The ATAM Method and Quality Attributes of Service-Oriented Architectures

Paulo Merson

Software Engineering Institute

Abstract

Software engineers today understand how a sound software architecture is critical to successful implementation. Nonetheless, few organizations apply systematic design evaluations in their software process. The Architecture Tradeoff Analysis Method[SM] (ATAM[SM]) is the leading method in the area of software architecture evaluation. This tutorial starts with a description of the ATAM method, its goals and benefits, and its nine steps. The ATAM is based on the principle that quality attributes have a significant influence on the architecture of a system. This assumption makes the second part of the tutorial a nice follow up, because it describes how different qualities are impacted by the architecture decision of using the Service-Oriented Architecture (SOA) approach. While there are significant benefits with respect to interoperability and modifiability, other qualities such as performance, security and testability are concerns.

Resumo

Engenheiros de software compreendem hoje em dia como uma arquitetura de software consistente é crítica para uma implementação de sucesso. Apesar disso, poucas organizaçÕes aplicam avaliações de projeto sistematicamente eu seu processo de desenvolvimento de software. O Método ATAM[SM] (Architecture Tradeoff Analysis Method[SM]) é o principal método na área de avaliação de arquitetura de software. Este tutorial inicia com uma descrição do método ATAM, suas metas e benefícios e seus nove passos. ATAM baseia-se no princípio de que atributos de qualidade têm uma influência significativa na arquitetura de um sistema. Esta hipótese faz com que a segunda parte do tutorial seja uma boa seqüência, porque ela descreve como diferentes qualidades são impactadas pela decisão arquitetural de usar a abordagem de Arquitetura Orientada a Serviços (SOA). Enquanto há benefícios significativos com respeito à interoperabilidade e modificabilidade, outras qualidades tais como performance, segurança e testabilidade são preocupações.

Técnicas de Gestão da Evolução de Software

Nicolas Anquetil

Universidade Católica de Brasília

anquetil@ucb.br

Resumo

Apesar da importância comprovada da evolução de software na prática, ela continua sendo uma atividade pouco estudada, pouco ensinada e geralmente mal considerada. Este tutorial tem por objetivo mostrar essa importância real da evolução para os futuros profissionais que são alunos de graduação e para os profissionais que a praticam. Passado este primeiro momento de "quebra de paradigma", o tutorial pretende ver algumas técnicas básicas, mas infelizmente muitas vezes desconhecidas, de boa gestão da evolução de software.

Métodos Estatísticos aplicados em Engenharia de Software Experimental

Márcio de Oliveira Barros[1], Guilherme Horta Travassos[2], Leonardo Gresta Paulino Murta[2], Marco Antônio Pereira Araújo[2]

[1]UNIRIO

[2]COPPE/UFRJ

marcio.barros@uniriotec.br, {ght, murta, maraujo}@cos.ufrj.br

Resumo

Este tutorial tem como objetivo apresentar as principais técnicas estatísticas utilizadas no planejamento e análise de estudos experimentais em Engenharia de Software. A abordagem do tutorial será prática, buscando sempre apresentar as técnicas estatísticas no contexto de exemplos práticos e, sempre que for possível, reais. Utilizaremos informações de estudos experimentais já realizados pelos autores e publicados na literatura técnica para apoiar as discussões e apresentação de aplicação das abordagens para a audiência. Entendemos que um tutorial desta natureza deva apresentar, além de conceitos básicos de experimentação e de estatística, indicações concretas da aplicabilidade das técnicas apresentadas. Por isto, a procura por apoiar as apresentações em situações e casos concretos que permitirão aos atendentes obter informações complementares sobre o tema.

Engenharia de Requisitos Orientada pelos Aspectos

Ana Moreira, João Araújo

Departamento de Informática
Faculdade de Ciências e Tecnologia – Universidade Nova de Lisboa

{amm, ja}@di.fct.unl.pt

Resumo

As abordagens existentes para Engenharia de Requisitos, por exemplo, *viewpoints* e casos de uso, oferecem mecanismos para particionar os requisitos em especificações parciais, suportando a separação de assuntos. Todavia, certos requisitos (e.g. Segurança, Extração de Informação) atravessam estas especificações parciais e não podem ser modularizadas em casos de uso ou *viewpoints*, por exemplo. O objetivo deste tutorial é oferecer técnicas para modularizar e analisar tais requisitos transversais usando a Engenharia de Requisitos Orientada pelos Aspectos.

Índice por Autor
Author Index