UNIVERSIDADE FEDERAL FLUMINENSE
INSTITUTO DE COMPUTAÇÃO
GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

Wallace da Silva Ribeiro

Monitor de Métricas de Software

NITERÓI
2011

WALLACE DA SILVA RIBEIRO

MONITOR DE MÉTRICAS DE SOFTWARE

Monografia apresentada ao Curso de Graduação em Ciência da Computação da Universidade Federal Fluminense, como requisito parcial para obtenção do Grau de Bacharel em Ciência da Computação.

Orientador: Prof. Dr. Leonardo Gresta Paulino Murta
Co-Orientador: Prof. Dr. Alexandre Plastino de Carvalho

Niterói
2011

WALLACE DA SILVA RIBEIRO

MONITOR DE MÉTRICAS DE SOFTWARE

Monografia apresentada ao Curso de Graduação em Ciência da Computação da Universidade Federal Fluminense, como requisito parcial para obtenção do Grau de Bacharel em Ciência da Computação.

Aprovada em dezembro de 2011.

BANCA EXAMINADORA

Leonardo Gresta Paulino Murta, D.Sc. - Orientador
IC-UFF

Alexandre Plastino de Carvalho , D.Sc. - Co-Orientador
IC-UFF

Teresa Cristina de Aguiar, D.Sc.
IC-UFF

Gleiph Ghiotto Lima de Menezes, M.Sc.
IC-UFF

Niterói
2011

RESUMO

A sub-área da engenharia de software chamada gerência de configuração de software é responsável pelo estudo e controle das mudanças que ocorrem durante o desenvolvimento de projetos de software. Gerência de configuração de software estuda o que mudou, onde mudou e quais as consequências destas mudanças no projeto. Por outro lado, métrica de software é a medida quantitativa do grau em que um sistema, componente ou processo se encontra em relação a um determinado atributo. Portanto, o objetivo deste trabalho é empregar métricas de software no estudo de gerência de configuração de software. Para tal, desenvolvemos coletores para vinte e duas diferentes métricas básicas, um mecanismo para definir métricas compostas de outras métricas já existentes através de fórmulas e gráficos que permitem analisar tais métricas. Finalmente, fizemos uma experiência inicial da análise de alguns desses indicadores sobre o IdUFF, o sistema de gestão acadêmica da Universidade Federal Fluminense.

Palavras-chave: métricas de software, gerência de configuração.

ABSTRACT

The software engineering sub-field called software configuration management is responsible for the study and control of changes that occur during the development of software projects. Software configuration management studies what changed, where the change occurred, and what are the consequences of the modifications in the project. On the other hand, software metric is a quantitative measure of the degree to which a system, component, or process is in relation to a particular attribute. Therefore, the objective of this work is to employ software metrics in the study of software configuration management. To do so, we developed collectors to twenty two different basic metrics, a mechanism to define composed metrics from other existing metrics via formulas, and graphs that allow analyzing such metrics. Finally, we ran an initial experiment analyzing some of these metrics over IdUFF, the academic management system of the Fluminense Federal University.

Keywords: software metrics, software configuration management.

LISTA DE ILUSTRAÇÕES

Fig. 1 Descrição dos níveis do QMOOD ….. 8
Fig. 2 Diagrama de classe das classes que implementam o sistema métricas …..................... 18
Fig. 3 Diagrama de sequência da implementação da métrica Abstractness …......................... 21
Fig. 4 Diagrama de sequência para métricas retiradas com o Dependency Finder ….............. 22
Fig. 5 Diagrama de sequência da implementação da métrica Lack Of Cohesion
of Methods ….. 24
Fig. 6 Diagrama de sequência da implementação da métrica Number Of Interfaces …......... 26
Fig. 7 Árvore representando uma métrica composta …... 29
Fig. 8 Códigos dos métodos extractMetric com duas assinaturas diferentes …...................... 30
Fig. 9 Diagrama de classes das métricas derivadas ….. 31
Fig. 10 Implementação do método getDoubleValue da classe MetricManagerExpression … 32
Fig. 11 Códigos das implementação do método getDoubleValue nas classes Add, Sub
e Pow 33
Fig. 12 Códigos das implementação do método getDoubleValue nas classes Sqrt
e UnarySub …... 33
Fig. 13 Exemplo de hierarquia de classes em métricas derivadas …...................................... 34
Fig. 14 Exemplo de funcionamento da estratégia recursiva, de obter o extractsFrom da
métrica derivada …... 35
Fig. 15 Implementação do método setup(Metric metric) na classe DeriveMetric …............... 37
Fig. 16 Lista de objetos Token ….. 37
Fig. 17 Funcionamento do método expressionReckonigze ….. 38
Fig. 18 Conjunto de Tokens ligados em uma lista …... 40
Fig. 19 Primeira recursão do método expressionReckonigze ….. 40
Fig. 20 Segunda recursão do método expressionReckonigze ….. 41
Fig. 21 Resolução do operador * na segunda recursão ….. 41
Fig. 22 Retorno a primeira recursão …... 41
Fig. 23 Resolução do operador ^ na primeira recursão ….. 41
Fig. 24 Resolução do operador + na primeira recursão …... 42
Fig. 25 Retorno à instância original do método expressionReckonigze …............................. 42
Fig. 26 Resolução do UnarySub …... 43
Fig. 27 Página de criação de métricas derivadas …... 44

Fig. 28 Preenchimento dos dados para a criação de métrica derivada …............................... 45
Fig. 29 Erro sendo retornado ….. 45
Fig. 30 Preenchimento dos campos ….. 46
Fig. 31 Métrica criada e mostrada na lista de métricas …... 47
Fig. 32 Página de monitoramento gráfico de um projeto ….. 49
Fig. 33 Gráfico de controle de valores absolutos da métrica *Lines of Code* …...................... 50
Fig. 34 Gráfico de valores absolutos de métrica Design Size in Classes …........................... 51
Fig. 35 Histograma da métrica *Lines of Code* …... 52
Fig. 36 Diagrama de sequência da criação dos gráficos de controle …................................ 54
Fig. 37 Diagrama de sequência da criação do histograma …... 55
Fig. 38 Gráfico de valores absolutos da métrica Number Of Static Attributes …................... 56
Fig. 39 Gráfico delta da métrica Number Of Static Attributes …... 57
Fig. 40 Histograma da métrica Number Of Static Attributes ….. 57
Fig. 41 Histograma com os valores de limite modificados ….. 58
Fig. 42 Histograma com os valores de limite modificados novamente …............................. 58
Fig. 43 Gráfico do valor absoluto da métrica Number Of Methods …................................. 61
Fig. 44 Histograma da métrica Number Of Methods …... 62
Fig. 45 Gráfico do valor absoluto da métrica Lines Of Code ….. 62
Fig. 46 Histograma da métrica Lines Of Code …... 63
Fig. 47 Gráfico do valor absoluto da métrica Total Cyclomatic Complexity …..................... 64
Fig. 48 Histograma da métrica Total Cyclomatic Complexity …... 64

LISTA DE TABELAS

Tabela. 1 Relação entre métricas de projeto e propriedades de projeto …............................... 10

Tabela. 2 Relação entre propriedades de projeto e atributos de qualidade de projeto …........ 10

Tabela. 3 Métricas implementadas …... 15

SUMÁRIO

1 INTRODUÇÃO ….. 1
2 FUNDAMENTOS DE MEDIÇÃO E INFRAESTRUTURA BASE DO PROJETO …... 4
2.1 MÉTRICAS DE SOFTWARE ….. 4
2.2 A QUALIDADE DE UM SOFTWARE ….. 5
2.2.1 QUALIDADE PARA PROJETOS DE SOFTWARE ORIENTADO A OBJETO …... 8
2.3 O PROJETO OCEANO …... 11
2.3.1 OSTRA …... 12
2.3.2 IMPORTÂNCIA DA OSTRA ….. 12
2.4 CONSIDERAÇÕES FINAIS …... 13
3 MÉTRICAS DESENVOLVIDAS NO PROJETO ….. 14
3.1 INTRODUÇÃO ….. 14
3.2 DESCRIÇÃO DAS MÉTRICAS …... 16
3.3 INFRAESTRUTURA PARA MEDIÇÃO …... 17
3.4 IMPLEMENTAÇÃO DAS MÉTRICAS …... 20
3.4.1 JDEPEND ….. 20
3.4.2 DEPENDENCY FINDER ….. 21
3.4.3 BCEL …... 22
3.4.4 CKJM …... 24
3.4.5 JAVANCSS …... 24
3.4.6 LOCC …... 25
3.4.7 SEM USO DE BIBLIOTECAS AUXILIARES ….. 26
3.5 CONSIDERAÇÕES FINAIS …... 26
4 MÉTRICAS DERIVADAS ….. 28
4.1 APRESENTAÇÃO DAS MÉTRICAS DERIVADAS …... 28
4.2 ESTRUTURA DAS MÉTRICAS DERIVADAS …... 29
4.3 CONSTRUINDO NOVAS MÉTRICAS INTERATIVAMENTE …............................... 43
4.4 EXEMPLOS …... 45
4.5 CONSIDERAÇÕES FINAIS …... 47
5 MONITORAMENTO GRÁFICO ….. 48

5.1 MONITORAMENTO VIA GRÁFICOS DE CONTROLE E HISTOGRAMA ... 49
5.2 FUNCIONAMENTO DO MONITOR ….. 52
5.3 EXEMPLOS DE UTILIZAÇÃO ….. 56
5.4 CONSIDERAÇÕES FINAIS …... 59
6 AVALIAÇÃO EXPERIMENTAL ….. 60
6.1 AVALIAÇÕES REALIZADAS ….. 60
6.2 RESULTADO DAS AVALIAÇÕES ….. 60
6.3 CONSIDERAÇÕES FINAIS …... 66
7 CONCLUSÃO …... 67

1 INTRODUÇÃO

A sub-área da engenharia de software chamada de gerência de configuração trata do estudo e controle das modificações ocorridas em um projeto de software. Gerência de configuração [1] estuda o que mudou, onde mudou e quais as consequências das modificações no projeto. Contudo, para que esse estudo possa ser feito de forma objetiva, é necessário o emprego de métricas de software. Métrica é a medida quantitativa do grau em que um sistema, componente ou processo se encontra em relação a um determinado atributo [1]. Quantidade de linhas de código e número de métodos são exemplos de métricas de software.

Em um projeto de software hospedado em um repositório de um sistema de controle de versões [2], os programadores inserem suas modificações no código fonte desse projeto em diversos momentos. A cada momento que um projeto de software é modificado, cria-se uma nova entidade chamada de revisão. Cada revisão representa uma versão do software em um determinado momento do seu ciclo de vida.

Este trabalho tem como objetivo principal desenvolver as ferramentas que possam tornar possível o estudo das modificações ocorridas em diferentes revisões de um software, tendo como foco as modificações dos valores de suas métricas. Para tal objetivo, é necessária então a construção de um módulo que possa extrair as métricas, chamadas de métricas básicas.

Além disso, para viabilizar análises mais complexas que correlacionem diferentes métricas, torna-se necessária a construção de um módulo que possa, utilizando métricas previamente implementadas, criar outras métricas. O processo de criação de novas métricas consiste em combinar métricas através de uma fórmula matemática e gerar assim uma nova forma de se medir o software. Métricas criadas a partir desse método são chamadas de métricas derivadas ou métricas compostas.

Um terceiro passo consiste na necessidade de desenvolver um módulo que possa exibir graficamente os valores das métricas e suas modificações com o passar do tempo para que o

usuário possa verificar o comportamento das métricas nos diversos momentos do ciclo de vida do software. A principal motivação para esse módulo é o poder de observação dado por uma ferramenta gráfica em comparação a uma tabela com diversos valores.

Dessa forma, a contribuição deste trabalho consiste na combinação desses três módulos, permitindo ao usuário extrair as métricas de seu projeto, criar métricas caso sinta a necessidade de estudar seu software de outras maneiras e, finalmente, enxergar o comportamento das métricas nas diferentes etapas de construção de seu software de uma forma simples e intuitiva.

Esses módulos foram implementados na linguagem Java (no contexto do Projeto Oceano [3]) e utilizados para fins de experimentação sobre um software auxiliar do sistema acadêmico da Universidade Federal Fluminense, denominado IdUFF [4]. Esse software auxiliar, chamado Publico Core, foi desenvolvido em conjunto com o sistema IdUFF e serve tanto para auxiliar o IdUFF quanto para auxiliar outros sistemas mantidos pela Universidade Federal Fluminense. Do Publico Core foram extraídas métricas, visualizados seus gráficos e inferidas conclusões sobre o comportamento dessas métricas no ciclo de vida do software. Diversas conclusões puderam ser obtidas desse estudo, que focou principalmente na relação entre o tamanho de um projeto e sua complexidade.

Este trabalho está organizado em sete capítulos. O Capítulo 2 faz uma revisão sobre métricas, mostrando o que são, para que servem e alguns estudos sobre métricas. Também apresenta o projeto Oceano, que é o projeto principal onde estão inseridas as ferramentas desenvolvidas neste trabalho de conclusão de curso, ressaltando sua importância e como está organizado.

O Capítulo 3 aborda as métricas implementadas neste projeto, mostrando quais são essas métricas, as características de cada uma, como são calculadas e como cada uma foi implementada.

O Capítulo 4 aborda as métricas derivadas. Esse capítulo explica o que são métricas derivadas, como foi construído o sistema que permite sua criação e de que forma o usuário as constrói. Adicionalmente, nesse capítulo são apresentados alguns exemplos de construção de métricas derivadas.

O Capítulo 5 aborda o módulo de exibição dos gráficos das métricas. Esse capítulo mostra os tipos de gráficos que são exibidos, como são apresentados os valores das métricas nos gráficos e as informações que podem ser observadas nesses gráficos. Esse capítulo também descreve o funcionamento interno desse módulo, como foi implementada a geração dos gráficos e alguns exemplos de gráficos de métricas retiradas de um projeto real.

O Capítulo 6 apresenta uma avaliação da proposta. Para viabilizar essa avaliação foi escolhido um projeto real, o software Publico Core, do qual foram extraídas algumas de suas métricas e foram exibidos alguns gráficos dessas métricas. Ao final, são feitas algumas observações sobre as possíveis razões sobre os resultados observados.

No Capítulo 7, são feitas considerações finais sobre este trabalho, resumindo o que foi apresentado, mostrando suas contribuições, suas limitações e algumas sugestões de trabalhos futuros para expandir as funcionalidades deste projeto.

2 FUNDAMENTOS DE MEDIÇÃO E INFRAESTRUTURA BASE DO PROJETO

Este capítulo aborda alguns conceitos e explicações que são necessários ao entendimento dos capítulos seguintes. Seu conteúdo divide-se em explicações sobre os conceitos no quais se apóia este projeto e explicações sobre a infraestrutura na qual este projeto foi implementado. A Seção 2.1 exibe uma visão geral sobre métricas de software, suas aplicações, características e importância no contexto da engenharia de software. A Seção 2.2 aborda a qualidade de software e os fatores que a influenciam. A Seção 2.3 aborda a infraestrutura deste projeto, que consiste no ambiente Oceano, onde este projeto está inserido. A Seção 2.4 apresenta algumas considerações finais.

2.1 MÉTRICAS DE SOFTWARE

Métrica de software é a medida quantitativa do grau em que um sistema, componente ou processo se encontra em relação a um determinado atributo [1]. Atualmente, a engenharia de software não possui um conjunto de métricas padrão aceito por todos e ainda há muita discussão sobre quais métricas devem ser extraídas ou que conclusões podem ser obtidas a partir de valores de medição [1].

Pode-se dividir as métricas em métricas de medidas diretas e de medidas indiretas [1]. As métricas de medidas diretas podem ser obtidas através de uma forma direta de medição. Dessa maneira, essas medidas podem ser obtidas diretamente no código fonte ou no processo de criação do software. Exemplos de métricas diretas são: o número de linhas de código e a complexidade ciclomática (uma forma de se medir a complexidade de um método). Já métricas como capacidade de manutenção somente podem ser medidas de forma indireta. Essas são diretamente ligadas aos atributos de qualidade de um software, também conhecidos como requisitos não-funcionais de um sistema.

Requisitos não-funcionais referem-se a propriedades, comportamentos e restrições que um software apresenta [1]. Como exemplo de requisitos não-funcionais têm-se o seu desempenho, segurança e usabilidade. A partir do estudo desses atributos de qualidade, é possível obter informações relevantes sobre o software. Atualmente, há discussões onde se tenta mapear métricas obtidas de forma indireta a partir das métricas de medidas direta [5]. O objetivo é que, a partir de um conjunto de valores obtidos de medidas diretas, seja possível, através de fórmulas matemáticas, inferir atributos de qualidade do software.

De acordo com *Pressman* [1], no passado, alguns projetistas de software acreditavam que se deveria esperar mais anos até que se pudesse realmente entender a natureza do software, antes de começar a medi-lo. Alguns outros chegaram até mesmo a acreditar que talvez as características que compõem um software não fossem mensuráveis. *Pressman* afirma que esse pensamento é um erro e, portanto, deve-se haver medição no software para que haja melhores resultados em sua construção.

Assim como em todas as áreas das engenharias e das ciências, medir é uma necessidade primordial para gerenciar qualquer projeto. As métricas de software possuem um grande número de aplicações na engenharia de software. Métricas são utilizadas basicamente para quatro objetivos [1]:

- Entender – Por intermédio das métricas, pode-se conhecer melhor as características dos projetos. Entender como o projeto está configurado é primordial para a engenharia de software.
- Avaliar – Métricas podem ser usadas para avaliar certas características e, dessa forma, auxiliar na tomada de decisões.
- Controlar – Métricas podem ser utilizadas para o controle de processos de produção de software.
- Obter Previsões – Através de um conjunto de medidas, é possível inferir características e, dessa forma, estimar prazos ou até mesmo prever eventos que podem ocorrer no processo de desenvolvimento do software.

2.2 A QUALIDADE DE UM SOFTWARE

Uma das principais metas da utilização de métricas é conseguir medir atributos de qualidade de um software [1]. Entretanto, antes de efetuar medições, deve-se ter em mente que é necessário definir o que é qualidade de um software. Para solucionar esse problema,

vários autores identificaram fatores que influenciam na qualidade de software.

McCall, *Richards* e *Walters* [6] propõem uma categorização de fatores que influenciam a qualidade do software e, para cada fator, dão descrições. Os seguintes fatores foram considerados: (i) Confiabilidade – representa o quanto um programa realiza as funções para qual foi projetado com a precisão exigida para essas funções; (ii) Correção – representa o quanto um programa satisfaz suas especificações e os objetivos do cliente; (iii) Eficiência – representa a quantidade de recursos de hardware ou até mesmo de software necessária para que um programa realize determinada função; (iv) Flexibilidade – refere-se ao esforço necessário para se modificar um programa quando o mesmo já se encontra em produção; (v) Integridade – refere-se ao controle do acesso a dados ou funções de um software por pessoas não autorizadas; (vi) Interoperabilidade – refere-se ao esforço necessário para acoplar um sistema a outro; (vii) Manutenibilidade – refere-se ao esforço necessário para encontrar e consertar um erro em um programa; (viii) Portabilidade – representa o esforço necessário para transferir um programa de um ambiente para outro (seja ambiente de software ou de hardware); (ix) Reutilização – representa o quanto um programa (ou partes dele) pode ser utilizado para se construir outros programas; (x) Testabilidade – representa o esforço necessário para testar um programa; (xi) Usabilidade – representa o esforço necessário para aprender a utilizar e entender um software.

A norma ISO 9126 [7] é um padrão internacional para estimar a qualidade de um software. Essa norma leva em consideração os seguintes fatores: (i) Funcionalidade – representa o grau em que o software satisfaz as necessidades para as quais foi projetado; (ii) Confiabilidade – representa o período em que o software está disponível para o uso; (iii) Usabilidade – representa o grau de facilidade para se operar o sistema; (iv) Eficiência – representa o grau em que o software faz uso otimizado dos recursos do sistema; (v) Manutenibilidade – representa a facilidade com a qual se pode fazer reparos e ajustes no software; (vi) Portabilidade – representa a facilidade com a qual um software pode ser transportado de um ambiente para outro. Cada um desses fatores é subdividido em sub-características.

A Funcionalidade possui como sub-características: (i) Adequação – mede o quanto as funcionalidades do sistema são adequadas ao usuário; (ii) Precisão – representa a capacidade do software de fornecer resultados com a precisão exigida; (iii) Interoperabilidade – indica como o software interage com outros sistemas; (iv) Segurança – mede a capacidade do sistema de proteger as informações do usuário e fornecê-las apenas a pessoas autorizadas.

A Confiabilidade possui como sub-características: (i) Maturidade – a capacidade do

software em evitar falhas; (ii) Tolerância a falhas – a capacidade do software em manter o funcionamento adequado mesmo quando ocorrem defeitos; (iii) Recuperabilidade – a capacidade do software se recuperar após uma falha.

A Usabilidade possui como sub-características: (i) Inteligibilidade – representa a facilidade com que o usuário pode compreender as funcionalidades do software; (ii) Apreensibilidade – representa a facilidade de aprendizado do sistema para o usuário; (iii) Operabilidade – é como o produto facilita a sua operação por parte do usuário, incluindo a maneira como ele tolera erros de operação; (iv) Atratividade – envolve características que possam atrair o usuário, como por exemplo interfaces gráficas intuitivas.

A Eficiência possui como sub-características: (i) Comportamento em relação ao tempo – avalia se os tempos de resposta e de processamento estão dentro das especificações; (ii) Utilização de recursos – a capacidade do sistema em utilizar os recursos disponíveis.

A Manutenibilidade possui como sub-características: (i) Analisabilidade – representa a facilidade em se diagnosticar problemas e identificar suas causas; (ii) Modificabilidade – representa a facilidade com que o comportamento do software pode ser modificado; (iii) Estabilidade – representa a capacidade do software de se manter estável após a ocorrência de modificações; (iv) Testabilidade – representa a capacidade de se testar o sistema quando modificado, tanto as novas funcionalidades quanto as não afetadas diretamente pela modificação.

A Portabilidade possui como sub-características: (i) Adaptabilidade – representa a capacidade do software se a adaptar a diferentes ambientes sem a necessidade de configurações adicionais; (ii) Capacidade para ser instalado – representa a facilidade com que o sistema pode ser instalado; (iii) Coexistência – representa o quão facilmente um software convive com outros instalados no mesmo ambiente; (iv) Capacidade para substituir – representa a capacidade que o sistema tem de substituir outro sistema especificado, em um contexto de uso e ambiente específicos.

Essas duas taxonomias possuem semelhanças e diferenças. Ambas possuem, por exemplo, preocupações com a facilidade de uso do sistema pelo usuário, com a portabilidade, com a eficiência e com a manutenibilidade do sistema. Adicionalmente, nenhuma das duas fornece especificações claras sobre a aplicação de métricas para se encontrar qualquer um dos fatores de qualidade citados. Entretanto, como diferença, pode-se apontar a estrutura hierárquica do ISO 9126, que é dividida em fatores e cada um desses fatores em sub-características. Essa hierarquização não ocorre em *McCall*, que organiza seus fatores em apenas um nível. Outra diferença entre os modelos é a preocupação com a reutilização do

software para se criar outros softwares apresentada por *McCall*, a qual não é levada em conta pela norma ISO 9126.

2.2.1 QUALIDADE PARA PROJETOS DE SOFTWARE ORIENTADO A OBJETO

Tanto no modelo de qualidade de software de *McCall* quanto no modelo ISO 9126, o mapeamento entre os atributos de qualidade e métricas ainda é uma lacuna. Não há uma descrição de quais métricas podem fazer uma avaliação quantitativa da qualidade do software. Para as características de mais baixo nível, que estão próximas do código, não há uma definição de como, nem de quanto, essas influenciam na qualidade do software.

Pensando em resolver esse problema, *Bansiya* e *Davis* [5] definiram um modelo hierárquico para qualidade de projetos orientados a objeto (do inglês *Quality Model for Object-Oriented Design*), também conhecido pela sigla QMOOD. Esse modelo trabalha com softwares orientados a objeto e tem, como objetivo principal, mostrar a relação entre diferentes níveis de abstração, do mais básico ao mais complexo.

O QMOOD divide o modelo em quatro níveis: atributos de qualidade de projeto, propriedades de projetos orientados a objeto, métricas de projetos orientados a objeto e componente de projetos orientados a objeto. A Figura 1 exibe como esses níveis se relacionam.

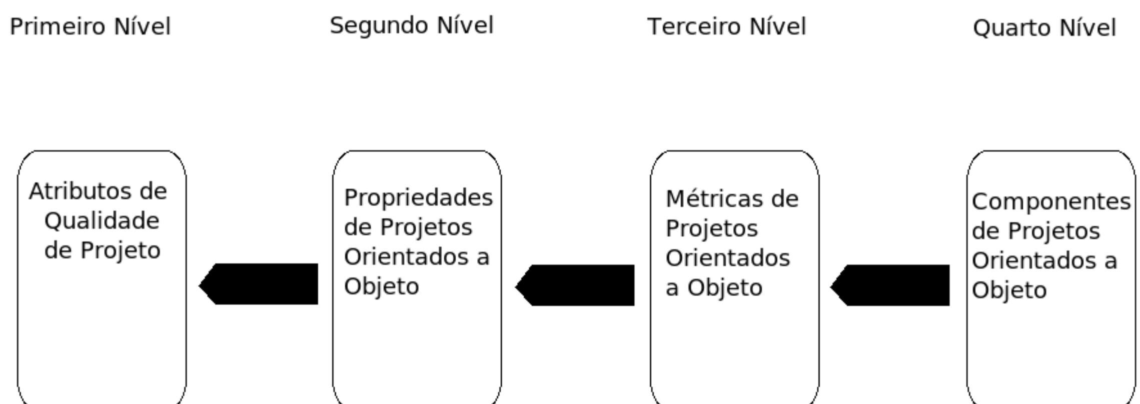

Figura 1. Descrição dos níveis do QMOOD (adaptada de [5]).

O nível de atributos de qualidade de projeto refere-se aos atributos de qualidade do software, ou seja, nesse nível são apresentados os fatores de qualidade que influenciam positivamente em um software orientado a objeto. É o nível de abstração mais alto. O

QMOOD especifica os seguintes fatores: reusabilidade, flexibilidade, compreensibilidade, funcionalidade, extensibilidade e eficácia.

No segundo nível, estão as propriedades de projetos orientados a objeto. Nesse nível apresentam-se características do software que podem ser obtidas por meio do exame da estrutura do software, seus componentes e o relacionamento entre esses diversos componentes. O QMOOD especifica as seguintes propriedade: tamanho do projeto, hierarquias, abstração, encapsulamento, acoplamento, coesão, composição, herança, polimorfismo, mensagens e complexidade.

O terceiro nível refere-se às métricas de projeto. O QMOOD especifica um conjunto de métricas utilizadas para fins de medição. Os valores dessas métricas definem o grau de qualidade de software representados pelos níveis superiores. As métricas escolhidas pelo QMOOD são: Tamanho do Projeto em Classes (*Design Size In Classes)*, Número de Hierarquias (*Number Of Hierarchies)*, Número Médio de Ancestrais (*Average Number of Ancestors)*, Métrica de Acesso a Dados (*Data Access Metric)*, Acoplamento Direto Entre Classe (*Direct Class Coupling)*, Coesão Entre Métodos de Classe (*Cohesion Among Methods of Class)*, Medida de Agregação (*Measure of Aggregation)*, Medida de Abstração Funcional (*Measure of Functional Abstraction)*, Número de Métodos Polimórficos (*Number of Polymorphic Methods)*, Tamanho de Interface de Classe (*Class Interface Size)* e Número de Métodos (*Number of Methods)*. Essas métricas são descritas em detalhes no Capítulo 3, capítulo relativo às métricas implementadas neste projeto.

Por último, o nível mais baixo refere-se aos componentes de projetos orientados a objeto, que são as classes, os objetos e os seus relacionamentos. Existe uma ligação entre cada nível e o nível superior. Essa ligação representa a influência de um nível em outro. Por exemplo, no terceiro nível, a propriedade acoplamento é influenciada pela métrica do segundo nível Acoplamento Direto entre Classe.

O nível dos componentes de projetos influenciam no nível de métricas de projeto no sentido em que as métricas são calculadas a partir da medição dos componentes do projeto. A métrica Tamanho do Projeto em Classes, por exemplo, representa o número de classes de um projeto. A descrição de como as métricas do QMOOD são calculadas é mostrado no Capítulo 3.

O nível de métricas de projeto influencia no nível de propriedade de projeto no sentido em que cada métrica de projeto mensura uma propriedade de projeto. Essa relação é mostrada na Tabela 1.

Tabela 1. Relação entre métricas de projeto e propriedades de projeto (adaptada de [5]).

PROPRIEDADE	MÉTRICA
Tamanho do Projeto	Tamanho do Projeto em Classes
Hierarquias	Número de Hierarquias
Abstração	Número Médio de Ancestrais
Encapsulamento	Métrica de Acesso a Dados
Acoplamento	Acoplamento Direto entre Classe
Coesão	Coesão de Métodos entre Classe
Composição	Medida de Agregação
Herança	Medida de Abstração Funcional
Polimorfismo	Número de Métodos Polimórficos
Mensagens	Tamanho de Interface de Classe
Complexidade	Número de Métodos

Já o nível de propriedades de projeto influencia o nível de atributos de qualidade de projeto no sentido em que as propriedades do projeto influenciam positivamente ou negativamente os seus atributos de qualidade. Uma mesma propriedade pode ter um peso positivo em relação a um atributo de qualidade e um peso negativo em relação a outro. A Tabela 2 mostra o quanto cada propriedade influencia nos atributos de qualidade, mostrando para cada atributo uma equação com as propriedades.

Tabela 2. Relação entre propriedades de projeto e atributos de qualidade de projeto (adaptado de [5]).

ATRIBUTO DE QUALIDADE	EQUAÇÃO DE PROPRIEDADES
Reusabilidade	-0,25 * Acoplamento + 0,25 * Coesão + 0,5 * Mensagens + 0,5 * Tamanho do Projeto
Flexibilidade	0,25 * Encapsulamento - 0,25 * Acoplamento + 0,5 * Composição + 0,5 * Polimorfismo
Compreensibilidade	-0,33 * Abstração + 0,33 * Encapsulamento -0,33 * Acoplamento + 0,33 * Coesão - 0,33 * Polimorfismo - 0,33 * Complexidade – 0,33 * Tamanho do Projeto
Funcionalidade	0,12 * Coesão + 0,22 * Polimorfismo + 0,22 * Mensagens + 0,22 * Tamanho do Projeto + 0,22 * Hierarquias
Extensibilidade	0,5 * Abstração – 0,5 * Acoplamento + 0,5 * Herança – 0,5 * Polimorfismo
Eficácia	0,2 * Abstração + 0,2 * Encapsulamento + 0,2 * Composição + 0,2 * Herança + 0,2 * Polimorfismo

Diferentemente dos modelos de *McCall* e ISO 9126, pode-se, com o QMOOD, aferir atributos de qualidade a partir da medição de métricas de medida direta. Por outro, lado vários dos fatores de *McCall* e do ISO 9126 podem ser mapeados através do QMOOD. Entretanto, existem fatores que não podem ser mapeado com o uso das métricas do QMOOD. Características como Atratividade, mostrada na norma ISO 9126, não possuem correspondência nos fatores mostrados pelo QMOOD e provavelmente essa característica tão peculiar não possua métricas que possam mensurá-la.

2.3 O PROJETO OCEANO

O projeto Oceano [3] é um ambiente de apoio a tarefas de gerência de configuração. É constituído até o presente momento por quatro ferramentas: Peixe-Espada, Ostra, Polvo e Ouriço. O Oceano, inicialmente com o nome de Peixe Espada, era uma ferramenta que realizava mudanças automáticas em projetos de software a fim de melhorar características no mesmo. Um exemplo de característica que pode ser melhorada através da ferramenta é a facilidade de manutenção. A ideia é que após, o expediente de trabalho (geralmente de madrugada), robôs entrem em ação e comecem a gerar relatórios de métricas e fazer alterações automáticas no código visando refatorar partes que podem ser melhoradas segundo um determinado objetivo.

Como o projeto Peixe Espada tornou-se grande e englobava várias ferramentas, surgiu a necessidade de transformá-lo em um ambiente, o qual poderia ser utilizado por outros projetos que tivessem a necessidade de utilizar as facilidades já implementadas pelo mesmo. O Oceano possui as seguintes características:

- É escrito na linguagem de programação Java [8]. A parte de interação com o usuário foi desenvolvida como uma interface *web*. É utilizado, para tanto, *JSF* 1.2 [9].
- O projeto utiliza o Subversion [10] para controle de versão. Existem módulos que realizam controle de versão programaticamente. Para desenvolver esse módulo, foi utilizada a biblioteca SVNKit [11].
- A persistência de dados é realizada através de JPA [12]. O JPA é implementado pelo Hibernate [13].
- Utiliza o Maven [14] para gerenciamento de construção.

Na versão atual, o ambiente Oceano é baseado em dois módulos principais: o Oceano-

Core e o Oceano-Web. Dentro do Oceano-Core estão os códigos que implementam o modelo de negócios. Dentro desse módulo está, por exemplo, a implementação das métricas de software. O Oceano-Web implementa a interface Web com o usuário. Nela estão implementadas as páginas e os controladores das paginas (foi utilizado o estilo arquitetural MVC [15] no projeto).

2.3.1 OSTRA

Um dos projetos que constituem o Oceano é a Ostra. A Ostra é uma ferramenta que extrai métricas para avaliação e, através da mineração de dados, tenta encontrar padrões a partir dos resultados das métricas extraídas.

Para um projeto de software ser analisado pela Ostra, este necessita ser escrito na linguagem de programação Java, utilizar o sistema de construção Maven e estar hospedado em um repositório Subversion. Satisfeitas essas pré-condições, a Ostra, automaticamente, realiza *checkouts* das várias revisões do projeto. Para cada revisão, a Ostra calcula métricas e armazena seus resultados em um banco de dados. Terminada a fase de extração de métricas, a Ostra pode então minerar os dados para tentar encontrar padrões e exibi-los ao usuário.

O projeto de conclusão de curso aqui descrito implementa ferramentas que são utilizadas principalmente no projeto Ostra. Apesar de poder ser utilizado por outros projetos, a grande motivação para criação das ferramentas aqui descritas foi fornecer alguns dos principais recursos para que a Ostra pudesse ser implementada.

2.3.2 IMPORTÂNCIA DA OSTRA

A mineração de dados consiste em explorar grandes quantidades de dados à procura de padrões [16]. No que se refere à Ostra, os dados a serem explorados são as métricas extraídas dos projetos nas várias revisões e os padrões costumam ser o relacionamento entre a variação dos valores das diversas métricas.

Por exemplo, imaginando que existam duas métricas (facilidade de manutenção e desempenho), que estão relacionadas a características muito importantes do software e que valores altos em ambas significam uma característica desejável, suponha que através da mineração de dados descubra-se, que quando o valor de uma métrica cresce, o valor da outra diminui. Portanto, descobriu-se que não se consegue otimizar as duas características do

software relacionadas às duas métricas, sendo que para se melhorar uma das características é necessário sacrificar a outra.

Finalizando, a Ostra auxilia a gerência de configuração, relacionada ao estudo das modificações do projeto a cada revisão, com o foco nas métricas e como essas métricas variam de acordo com o tempo.

2.4 CONSIDERAÇÕES FINAIS

Neste capítulo, foram apresentados os principais conceitos para o entendimento do funcionamento deste projeto. Foi explicado o que são métricas e quais são suas aplicações. Métricas de software são utilizadas em diversas situações, porém este capítulo concentrou-se no seu estudo para se aferir a qualidade do software. Sendo o modelo QMOOD um modelo que mapeia os atributos de qualidade de software para as métricas, as métricas descritas nesse modelo formam a base das métricas implementadas neste trabalho de conclusão de curso. Suas características e detalhes de implementação são discutidos no Capítulo 3.

Este capítulo também apresentou um resumo do ambiente Oceano e principalmente do projeto Ostra. Essa apresentação serve para mostrar onde se encaixa este trabalho de conclusão de curso nesse ambiente. Tendo essa visão do Oceano, os capítulos posteriores mostram como as ferramentas desenvolvidas neste trabalho estão inseridas no Oceano.

3 MÉTRICAS DESENVOLVIDAS NO PROJETO

Neste capítulo são apresentadas as métricas implementadas neste projeto. Essas métricas são chamadas aqui de métricas básicas por serem as métricas implementadas sem fazer o uso de outras métricas, em oposição às métricas derivadas que são criadas pelo usuário pela combinação de outras métricas já existentes. Todas as métricas derivadas usam direta ou indiretamente essas métricas básicas. A Seção 3.1 apresenta uma introdução sobre as métricas básicas. A Seção 3.2 mostra a descrição das métricas básicas. A Seção 3.3 mostra como é a infraestrutura do Oceano que permite a implementação dessas métricas. A Seção 3.4 mostra como é implementada cada métrica. Finalmente, a Seção 3.5 apresenta algumas considerações finais.

3.1 INTRODUÇÃO

Foram implementadas vinte e duas métricas de software. Essas métricas são extraídas dos projetos de software e seus valores são armazenadas em um banco de dados. Para se extrair a métrica de um projeto de software é necessário que esse projeto seja escrito na linguagem de programação Java [8], que seja gerenciado pelo Maven [14] e que esteja hospedado em um repositório Subversion [10].

Foi escolhida a implementação das métricas do QMOOD, por ser um modelo que consegue mapear as métricas básicas para atributos de qualidade. Além do QMOOD, foram também implementadas algumas outras métricas populares, como, por exemplo, o número de linhas de código. A Tabela 3 mostra quais são as métricas implementadas e algumas de suas características.

Essa tabela de métricas exibe o nome da métrica, sua sigla e uma descrição em alto nível. As métricas aqui implementadas, em sua maioria, medem características básicas do

software, como, por exemplo, número de linhas, número de atributos e número de métodos.

Tabela 3. Métricas implementadas.

NOME	SIGLA	DESCRIÇÃO	TIPO DE RETORNO	ELEMENTO MEDIDO	EXTRAI DE CÓDIGO	DOMÍNIO
Abstractness [17]	RMA	Indica a razão entre as classes abstratas e o número total de classes de um pacote.	real	pacote	não	entre 0 a 1
Average Number Of Ancestors [18]	ANA	Indica o número médio de classes que cada classe do projeto herda informações.	real	projeto	não	maior ou igual a 1
Class Interface Size [17]	CIS	Indica o número de métodos públicos em uma classe.	inteiro	classe	não	maior ou igual a 0
Cohesion Among Methods in Class [18]	CAM	Indica a coesão entre os métodos de uma classe.	real	classe	não	entre 0 e 1
Cyclomatic Complexity [19]	VG	Indica basicamente o número de caminhos independentes que o programa pode tomar durante a execução.	inteiro	classe	sim	maior ou igual a 1
Data Access Metric [18]	DAM	Indica a razão entre os atributos privados (e protegidos) e o número total de atributos.	real	classe	não	entre 0 e 1
Design Size in Classes [18]	DSC	Indica o número de classes de um projeto.	inteiro	projeto	não	maior ou igual a 0
Direct Class Coupling [18]	DCC	Indica o número de classes diferentes com que uma classe se relaciona.	inteiro	classe	não	maior ou igual a 0
Lack Of Cohesion Of Methods [19]	LCOM	Indica quanto os métodos de uma classe se relacionam.	inteiro	classe	não	maior ou igual a 0
Lines Of Code [19]	LOC	Indica o somatório de linhas de código de uma classe, incluindo comentários.	inteiro	classe	sim	maior ou igual a 0
Measure Of Aggregation [18]	MOA	Indica o número de declarações de dados, cujos dados são definidos pelo usuário.	inteiro	projeto	não	maior ou igual a 0
Measure Of Functional Abstraction [18]	MFA	Indica a razão entre os métodos herdados e todos os métodos acessíveis de uma classe.	real	projeto	não	entre 0 e 1

Methods Lines Of Code [18]	MLOC	Indica o número de linhas de código por método.	real	classe	não	maior ou igual a 0
Number Of Attributes [17]	NOA	Indica o número de atributos de uma classe.	inteiro	classe	não	maior ou igual a 0
Number Of Hierarchies [18]	NOH	Indica o número total de hierarquias de um projeto.	inteiro	projeto	não	maior ou igual a 0
Number Of Interfaces [17]	NOI	Indica o número de interfaces de um pacote.	inteiro	pacote	sim	maior ou igual a 0
Number Of Methods [18]	NOM	Indica o número de métodos de uma classe.	inteiro	classe	não	maior ou igual a 0
Number of Overridden Methods [17]	NORM	Indica o número de métodos sobrescritos de um projeto.	inteiro	projeto	não	maior ou igual a 0
Number of Polymorphic Methods [18]	NOP	Indica o número de métodos que apresentam comportamento polimórfico.	inteiro	projeto	não	maior ou igual a 0
Number Of Static Attributes [17]	NSF	Indica o número de atributos estáticos de uma classe.	inteiro	classe	não	maior ou igual a 0
Number Of Static Methods [17]	NSM	Indica o número de métodos estáticos de uma classe.	inteiro	classe	não	maior ou igual a 0
Total Cyclomatic Complexity [17]	WMC	Indica a soma da Complexidade Ciclomática de todas os métodos de uma classe.	inteiro	classe	sim	maior ou igual a 0

3.2 DESCRIÇÃO DAS MÉTRICAS

Nesta seção são descritas algumas das métricas que foram implementadas. Algumas métricas têm descrições suficientemente simples para que a própria Tabela 3 possa informar do que elas tratam (como, por exemplo, a métrica *Number of Lines* que representa número de linhas de código). Entretanto, outras possuem definições mais complexas, e, portanto, necessitam de uma explicação mais aprofundada para o entendimento do que estas representam. Esta seção descreve cada uma destas métricas.

Cohesion Among Methods In Class – Métrica do QMOOD que representa o grau de coesão existente entre os métodos de uma classe. Esta métrica é calculada da seguinte maneira: existe um conjunto T que possui todos os tipos de objetos dos parâmetros dos

métodos de uma classe. Para cada método cria-se um conjunto M que possui todos os tipos de objetos dos parâmetros do respectivo método. Tendo em determinada classe N métodos, o valor desta métrica será a razão entre o somatório do tamanho de todos os conjuntos M e a multiplicação do tamanho de T por N.

Cyclomatic Complexity – Complexidade Ciclomática é a métrica que indica a quantidade de caminhos de execução independentes. Esta métrica é de grande importância no que se refere à manutenibilidade do código. Métodos com grandes quantidades de *while* e *if* são muitas vezes difíceis de ler, fazer manutenção ou criar testes. Verificam-se as estruturas de controle dos métodos para desta forma encontrar a quantidade de caminhos independentes e, consequentemente, calcular o valor dessa métrica. Por exemplo, se um método não possui nenhuma estrutura de controle então sua complexidade ciclomática é 1, já que existe apenas um caminho através deste método. Se por outro lado, um método possui uma estrutura de seleção com uma condição, existem então dois caminhos possíveis, aquele quando a condição é verdadeira e aquele quando a condição é falsa. Portanto, este método tem 2 de complexidade ciclomática.

Direct Class Coupling – Métrica do QMOOD que indica o número de classes diferentes com que uma classe se relaciona. Esta métrica entende como relacionamento entre classes, a declaração de atributos e passagem de parâmetros. Contam-se apenas as classes que estão no projeto (excluem-se daí as classes de bibliotecas importadas).

Lack of Cohesion of Methods – A falta de coesão entre os métodos indica quanto os métodos de uma classe relacionam-se. Considera-se para o cálculo dessa métrica todos os pares de métodos de uma classe. Existem pares que acessam ao menos um mesmo atributo dessa classe e outros que não acessam nenhum atributo em comum. *Lack of Cohesion of Methods* é então calculado subtraindo-se o número total de pares que compartilham ao menos um acesso a um mesmo atributo do número total de pares que não compartilham.

Number of Polymorphic Methods – Métrica do QMOOD que indica o número de métodos que apresentam comportamento polimórfico. Um método que possui comportamento polimórfico é um método que foi sobrescrito ou sobrecarregado por um ou mais descendentes.

3.3 INFRAESTRTURA PARA MEDIÇÃO

A infraestrutura para medição das métricas é baseada nas classes *Metric*, *Revision, MetricValue* e *MetricManager*. A classe *Metric* representa uma métrica, ou seja, para cada

métrica implementada neste projeto existe um objeto da classe *Metric*. A classe *Revision* representa uma revisão de um projeto de software, armazenando informações como número da revisão e quem a criou. A classe *MetricValue* representa o valor de uma determinada métrica em uma determinada revisão. A classe *MetricManager* é uma classe abstrata, a qual define a base para implementação dos métodos de extração das métricas. A Figura 2 exibe o diagrama de classes que mostra a relação entre estas classes.

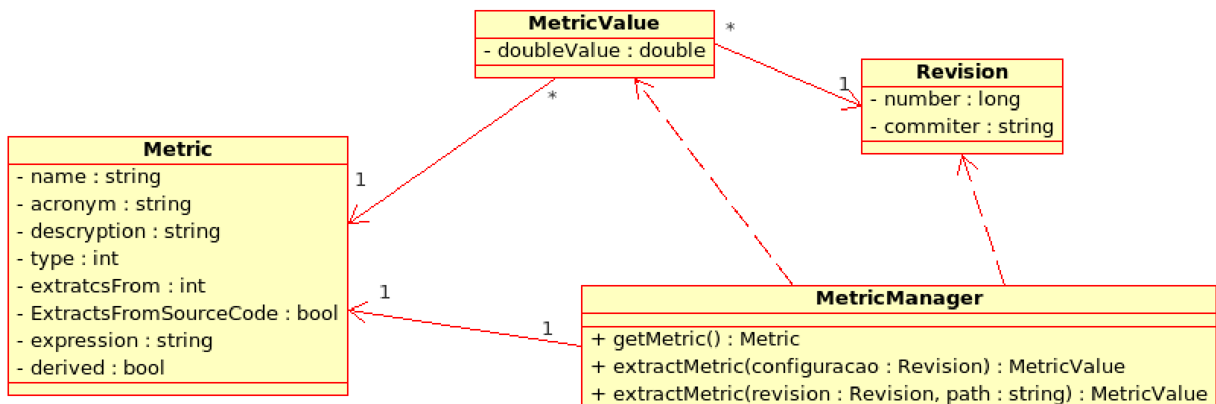

Figura 2. Diagrama de classe das classes que implementam o sistema métricas.

Uma métrica é implementada pela classe *Metric*. A classe *Metric* possui os atributos *name* (nome da métrica), *acronym* (sigla da métrica), *descryption* (descrição da métrica), *type*, *extratcsFrom*, *extractsFromSourceCode*, *expression* e *derived*. O atributo *type* descreve que tipo de resultado aquela métrica retorna (i.e., número inteiro, ponto flutuante ou valor booleano). O atributo *extratcsFrom* define de onde a métrica é extraída (se de um arquivo, de um pacote, ou de todo o projeto). O atributo booleano *extractsFromSourceCode* indica se a métrica extrai seus valores de um código fonte ou de um arquivo compilado. Quando as métricas necessitam ser extraídas de um arquivo compilado, é necessário que o sistema pré-compile os fontes do Java (geralmente guardam-se nos repositórios somente os arquivos fontes do projeto). Como os projetos devem ser gerenciados pelo software Maven, o processo de compilação dos projetos é relativamente simples, podendo ser feito automaticamente pelo sistema.

O atributo *derived* indica se uma métrica é ou não derivada. O campo *expression* é utilizado somente nas métricas derivadas, e será discutido no Capítulo 4, que aborda as métricas derivadas.

A classe *Metric* é persistida em banco de dados, ou seja, cada métrica é definida pela classe *Metric* e todas as métrica que o sistema possui estarão armazenadas em um banco de

dados, sejam elas métricas derivadas ou não. Os objetos das classes *Revision* e *MetricValue* também são persistidos em banco de dados. Outras classes que representam informações sobre projetos de software também são persistidas em banco de dados, porém, por motivos de simplificação, estas não estão representadas no diagrama da Figura 2. Um exemplo é a classe SoftwareProject que representa um projeto de software.

A classe *Metric* apenas define as características da métrica. Entretanto, existe outra classe chamada *MetricManager* que implementa a forma como cada métrica é extraída de um projeto. Essa classe possui os seguintes métodos abstratos:

- *public abstract Metric getMetric():* retorna a métrica que aquele *MetricManager* implementa.
- *public abstract MetricValue extractMetric(Revision configuracao) throws MetricException:* extrai uma métrica através de um objeto *Revision.* É utilizado principalmente por métricas que possuem um único valor para todo projeto.
- *public abstract MetricValue extractMetric(Revision revision, String path) throws MetricException:* extrai uma métrica através de um objeto *Revision* e um caminho. É utilizado pelas métricas que possuem um valor por pacote ou por classe. O argumento *path* refere-se ao caminho do pacote ou arquivo da classe. *Revision* é a classe que representa uma revisão de um projeto de software. Essa classe possui informações como o projeto a que essa pertence, quem criou essa revisão e o caminho do diretório raiz do projeto quando foi realizado o *checkout* daquela revisão para a máquina local. Por medidas de simplificação, todas as métricas foram implementadas utilizando o método *extractMetric(Revision revision, String path)*, sendo que o *path* é ignorado nas métricas de projeto.

Dessa forma, os métodos onde são implementadas realmente as extrações das métricas são *extractMetric(Revision revision, String path)* e *extractMetric(Revision revision)*, dependendo de onde a métrica é extraída. São nesses métodos que estão localizadas todas as implementações que coletam dos projetos de softwares medidas numéricas. Esses métodos devolvem como resultado o objeto *MetricValue*, que representa o valor de uma determinada métrica em uma determinada revisão.

Para auxiliar a retirada de certas métricas é utilizado a classe *ClassLoader*. Essa classe é padrão da biblioteca Java e consegue carregar classes dinamicamente durante a execução do programa. Existe no projeto Oceano-Core uma classe chamada *MavenUtil* localizada no pacote br.uff.ic.oceano.core.util. Dentro dessa classe existe um método de nome

getProjectClassLoaderByCommandLine(Revision revision). Esse método retorna um objeto *ClassLoader*, o qual pode carregar todas as classes do projeto que está sendo medido e de todas as bibliotecas as quais esse projeto depende.

3.4 IMPLEMENTAÇÃO DAS MÉTRICAS

A implementação das métricas utilizou-se de vários softwares. Umas necessitam de apenas algumas classes padrão de Java, enquanto outras necessitam de bibliotecas externas. Algumas puderam aproveitar implementações de códigos de software livre, enquanto outras tiveram que ser implementadas desde o início. Neste capítulo são apresentadas as ferramentas que auxiliaram na implementação e como cada métrica foi implementada utilizando tais ferramentas.

3.4.1 JDEPEND

O software JDepend [20] é utilizado para calcular algumas métricas, principalmente de pacotes. Foi utilizado algumas de suas classes para poder implementar a métrica *Abstracness.*

Para se extrair *Abstractness*, primeiramente cria-se uma coleção de arquivos pertencentes ao pacote do qual se deseja extrair a métrica. Esses arquivos são processados pelo método *buildClasses(File file)*, que os transformam em uma coleção de objetos da classe *JavaClass.*

Os objetos *JavaClass* são então adicionados a um objeto da classe *JavaPackage* através do método *addClass(JavaClass clazz).* O método *abstractness() da* classe *JavaPackage* retorna o valor da métrica. A Figura 3 mostra o diagrama de sequência para a implementação desta métrica.

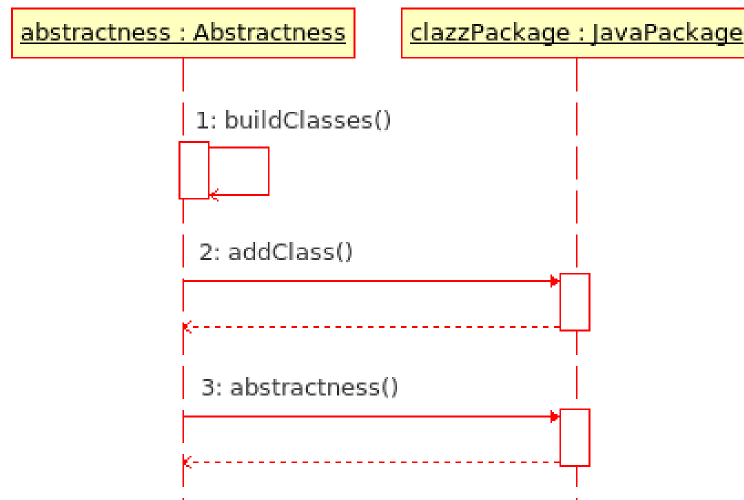

Figura 3. Diagrama de sequência da implementação da métrica *Abstractness*.

3.4.2 DEPENDENCY FINDER

O software Dependency Finder [21] é um conjunto de ferramentas que pode extrair diversas métricas de arquivos compilados Java. Neste projeto ele é utilizado para extrair as métricas *Class Interface Size*, *Data Access Metric*, *Methods Lines Of Code*, *Number Of Attributes*, *Number of Hierarchies*, *Number Of Static Attributes* e *Number Of Static Methods.*

Para extrair essas métricas, é armazenado o caminho de uma classe (em métricas de classe), ou são armazenados todos os caminhos das classes do projeto inteiro (para métricas de projeto) em uma estrutura *Collection*. Essa estrutura é carregada em um objeto *ClassfileLoader* através do método *load(Collection<String> filenames)*. Através de um conjunto de operações entre diversas classes do *Dependencyfinder* as métricas são extraídas. A Figura 4 exibe o diagrama de sequência que mostra o conjunto das principais classes e métodos envolvidos para a extração das métricas.

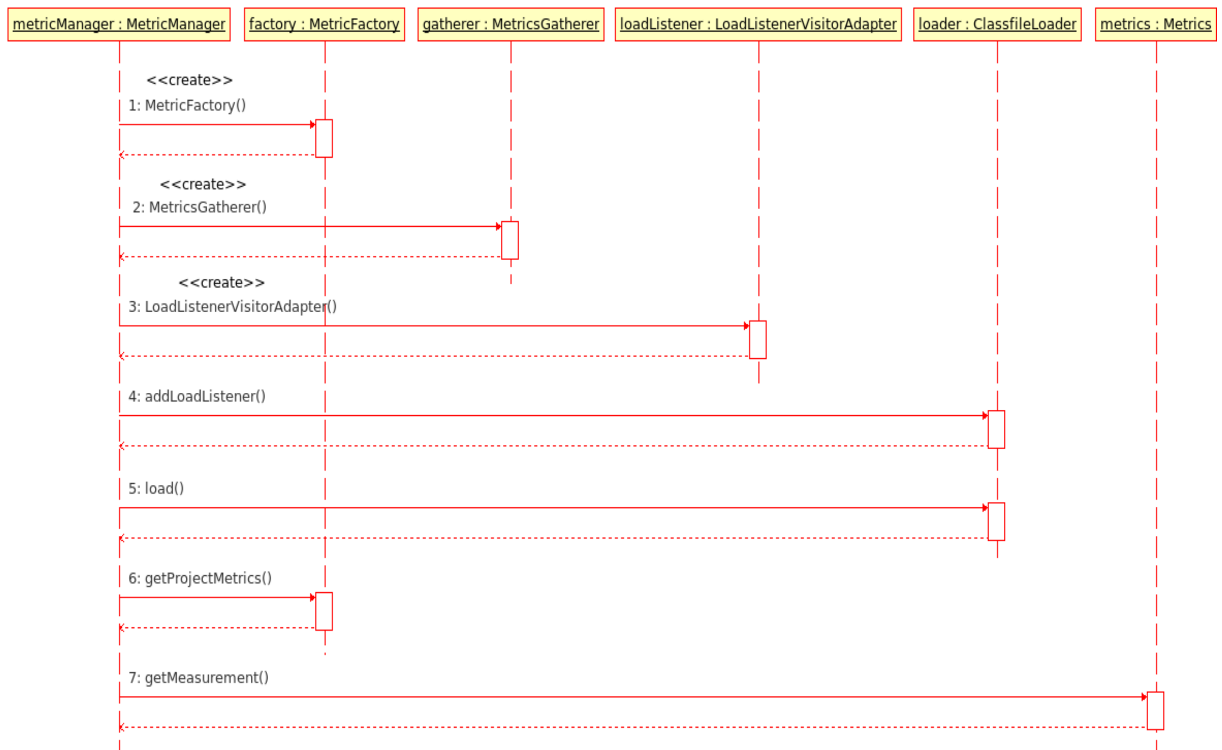

Figura 4. Diagrama de sequência para métricas retiradas com o Dependency Finder.

As métricas de classe e projeto são extraídas de forma bastante semelhante. A diferença básica é que para as métricas de classe utiliza-se o método *getClassMetrics()* para se obter os objetos *Metrics* e para métricas de projeto utiliza-se o método *getProjectMetrics()*. Estes métodos são o de número de sequência 6 no diagrama. No caso do diagrama da Figura 4, este representa a extração de uma métrica de projeto e, portanto, utiliza o método *getProjectMetrics()*. Uma vez obtido o objeto da classe *Metrics* utiliza-se o seu método *getMeasurement(String name)* para se obter o valor da métricas, sendo *name* a sigla da respectiva métrica que deseja-se extrair. Importante notar que a classe *Metrics* é diferente da classe *Metric* apresentadas na Seção 3.3, pois a primeira pertence ao software Dependency Finder, e a segunda à infraestrutura proposta neste trabalho.

3.4.3 BCEL

O software Byte Code Engineering Library (BCEL) [22] é capaz de manipular classes compiladas Java (i.e., *bytecodes*). Com este software é possível obter várias informações de uma classe, como, por exemplo, seus métodos, seus parâmetros e seus ancestrais. Utilizando esse software foram extraídas as métricas *Cohesion Among Methods In Class*, *Direct Class*

Coupling, Measure of Aggregation, Measure Of Functional Abstraction, Number Of Methods, Number of Overridden Methods e *Number of Polymorphic Methods.*

Para se calcular *Cohesion Among Methods In Class s*ão obtidos todos os métodos da classe a ser medida através das funcionalidades do BCEL. De cada método são obtidos os seus tipos de parâmetros. Através desses tipos de parâmetros é calculado o número de tipos de parâmetros diferentes que cada método possui. Utilizando essas informações, essa métrica é calculada através de sua definição (descrita na Seção 3.2).

Para calcular *Direct Class Coupling* primeiro armazenam-se todas as assinaturas dos nomes das classes do projeto. Adicionalmente obtêm-se todos os métodos e atributos de uma classe. Então se calcula o número de atributos e de parâmetros cujas suas classes tenham a mesma assinatura das classes do projeto.

Para calcular *Measure of Aggregation* primeiro armazenam-se todas as assinaturas dos nomes das classes do projeto. De cada classe do projeto obtêm-se os seus atributos. São então calculados quantos atributos cujo o tipo tem mesma assinatura das classes do projeto.

Para se calcular Measure Of Functional Abstraction calculam-se o número de métodos herdados e o número de métodos acessíveis de uma classe. O número de métodos herdados é calculado pela subtração do número de métodos acessíveis pelo número de métodos declarados naquela classe. O BCEL consegue obter o número de métodos declarados de cada classe.

Contudo, para saber o numero total de métodos acessíveis de uma classe, são visitados todos os ancestrais desta classe e calculado o numero de métodos declarados nestes que não estão sobrescritos naquela. Somando-se esse número ao número de métodos declarados de uma classe obtém-se o número de métodos acessíveis daquela classe. Com essas informações pode-se então calcular o valor desta métrica.

Para se calcular a métrica *Number of Overridden Methods* são comparadas a assinaturas de cada método de cada classe, com as assinaturas dos métodos de suas classes ancestrais. Se tiverem as assinaturas iguais, indicará que houve uma sobrescrita de método. São então somados todos os números de sobrescritas de métodos e assim é calculado o valor dessa métrica.

Para calcular *Number of Polymorphic Methods* são analisados cada um dos métodos de cada uma das classes, comparando os nomes de cada método de uma classe com os nomes dos métodos das classes ancestrais desta. Desta forma, encontram-se os métodos que foram sobrescritos por uma ou mais classes. O número total de métodos que foram sobrescritos é o valor desta métrica.

3.4.4 CKJM

O CKJM [23] é um software que calcula algumas métricas através dos bytecodes da linguagem Java, de forma similar ao BCEL. Com a utilização de classes deste software foi extraída a métrica *Lack Of Cohesion of Methods*.

Para implementar esta métrica é criado um objeto da classe *ClassMetricsContainer*. É então passado esse objeto e o caminho de uma classe para o método *processClass(ClassMetricsContainer cm, String path)*. Este método retorna uma *String* com o nome da classe que foi processada.

Através do método de *ClassMetricsContainer* chamado *getMetrics(String arg)*, o qual possui como parâmetro o nome da classe processada, obtém-se todas as métricas de classe. Essas métricas de classe são representadas por uma classe chamada *ClassMetrics*. Essa classe possui um método chamado *getLcom()* que devolve o valor que representa a falta de coesão de métodos da classe processada. A Figura 5 exibe o diagrama de sequência da implementação desta métrica.

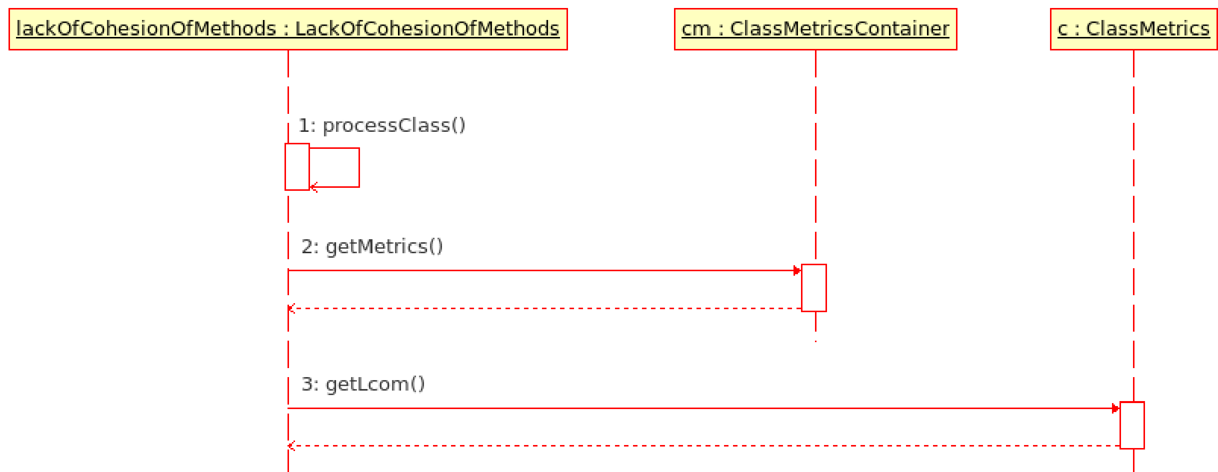

Figura 5. Diagrama de sequência da implementação da métrica *Lack Of Cohesion of Methods*.

3.4.5 JAVANCSS

O software JavaNCSS [24] é utilizado para extrair medidas de software, assim como todos os anteriores. Através deste foram extraídas as métricas *Cyclomatic Complexity*, *Total Cyclomatic Complexity* e *Lines of Code*.

Para se extrair as métricas *Cyclomatic Complexity e Total Cyclomatic Complexity* um objeto da classe *Javancss* é criado passando o caminho de uma classe para seu construtor. Através do método *getFunctionMetrics()* é obtida uma lista de objetos da classe *FunctionMetric*, sendo que cada objeto desse tipo representa um método da classe passada inicialmente. Cada objeto *FunctionMetric* possui o atributo *ccn*, que representa a complexidade ciclomática. Para obter a métrica *Cyclomatic Complexity*, é calculada a média do *ccn*. Para obter a métrica *Total Cyclomatic Complexity*, é calculada a soma do *ccn*.

Para se extrair a métrica *Lines of Code*, constrói-se um objeto do tipo *Javancss* passando como parâmetro do construtor um objeto *File*, que representa a classe de onde se quer extrair a métrica. Chama-se então o método *getLOC()* de *Javancss*, e assim obtêm-se a quantidade de linhas de um arquivo fonte Java.

3.4.6 LOCC

O software *Locc* [25] é utilizada para coletar a métrica *Number Of Interfaces*. Em sua implementação é criado um objeto da classe *DirTree*, enviando a esse objeto, pelo seu construtor, o caminho de um diretório e a *String* que contem o valor ".java". Essa ação adiciona todos os nomes dos arquivos fontes de um diretório.

Então são adicionados todos esses nomes de arquivos fontes em um vetor. A partir de cada arquivo, é criado um objeto da classe *JavaParser*. Através do método *TopLevel()* da classe *JavaParser* é extraído um objeto da classe *CompilationUnit*. Através do método *getNumInterfaces()* dessa classe é obtido o número de interfaces. São somados, então, todos os números de interfaces de todos os objetos *CompilationUnit* associados aos arquivos armazenados no vetor. Importante salientar que esta métrica determina o número de interfaces de um pacote, excluindo-se desse cálculo os sub-pacotes. A Figura 6 mostra o diagrama de sequência da implementação desta métrica.

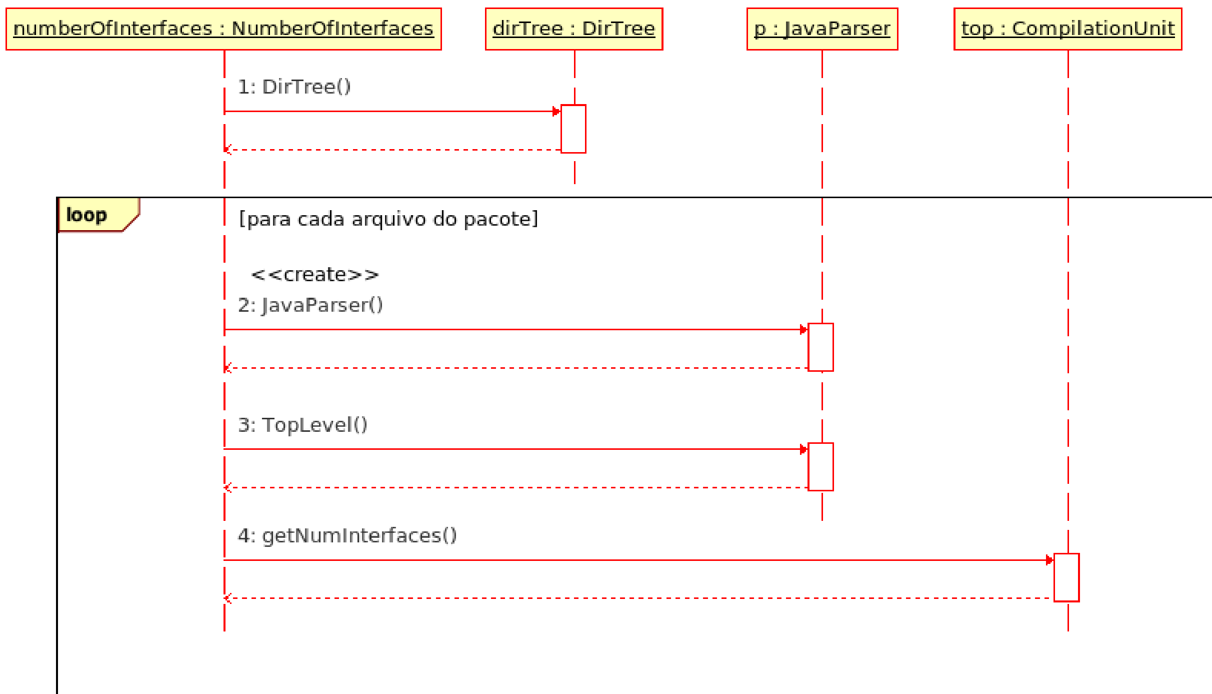

Figura 6. Diagrama de sequência da implementação da métrica *Number Of Interfaces*.

3.4.7 SEM USO DE BIBLIOTECAS AUXILIARES

A métrica *Average Number Of Ancestors* não fez uso de softwares auxiliares para ser implementada. A única necessidade foi a classe *ClassLoader* que guarda diversas informações sobre o projeto. Através da classe *ClassLoader* são obtidas as classes pertinentes à revisão específica do projeto. Em seguida, são percorridos de forma ascendente os ancestrais de cada classe até a classe *Object* (superclasse de todas as classes Java). Então são calculados quantos ancestrais cada classe possui e, consequentemente, o número médio de ancestrais que cada revisão do projeto possui.

Outra métrica que não fez uso de softwares externos é a *Design Size In Classes*. Esta é calculada através do somatório do número de classes de um projeto (excluindo as bibliotecas importadas). Basicamente consiste em obter a raiz do diretório do projeto o qual o Maven dispõe as classes, e então calcular, recursivamente pelos diretórios, o número de classes totais.

3.5 CONSIDERAÇÕES FINAIS

Este capítulo tratou da implementação das métricas básicas. Como pôde ser observado, essas métricas podem ser utilizadas diretamente pelo usuário de sistema e constitui a base do

sistema de métricas. Estas métricas focam muitas vezes em medidas bastante simples dos programas em Java, como número de linhas de código, número de métodos e número de atributos. O objetivo é que medidas simples sejam usadas como base para que sejam elaboradas métricas derivadas, que é o assunto do Capítulo 4.

4 MÉTRICAS DERIVADAS

Neste capítulo é apresentado o conceito de métricas derivadas. Através desse conceito foi criado um sistema que pôde, então, expandir o poder das métricas implementadas no Capítulo 3. A Seção 4.1 descreve o que são as métricas derivadas, também chamadas aqui de métricas compostas. A Seção 4.2 explica como foi implementado o sistema que torna possível a criação de uma métrica derivada. A Seção 4.3 mostra como o usuário cria as métricas derivadas. A Seção 4.4 mostra alguns exemplos de utilização do sistema. A Seção 4.5 faz algumas considerações finais sobre o capítulo.

4.1 APRESENTAÇÃO DAS MÉTRICAS DERIVADAS

Foi apresentado no Capítulo 3, as vinte e duas métricas implementadas no projeto. Estas são todas as onze métricas do QMOOD, mais onze métricas bastante populares. Por exemplo, número de linhas de código, apesar de não estar no QMOOD, talvez seja a métrica mais popular, sendo um dos indicadores mais requisitados de um projeto para normalização.

Com essas métricas pode-se investigar características sobre projetos de software. Porém, ainda assim, a utilização dessas métricas no exato formato em que foram implementadas pode significar um desperdício do poder que a abordagem proposta por este trabalho pode oferecer.

O que ocorre é que essas métricas representam características muito básicas de um software e, portanto, podem em alguns casos não dizer muito sobre o mesmo. Já combinações de diferentes métricas poderiam construir uma melhor caracterização dos projetos de software, em especial se essas combinações pudessem ser criadas e refinadas pelo usuário do sistema. Portanto, métricas derivadas ou compostas são métricas que podem ser criadas a partir da combinação de outras métricas através de fórmulas matemáticas.

Toma-se como exemplo a métrica *Lines Of Codes*. De grande importância para saber o tamanho do projeto, ela poderia ser utilizada também em outras análises. Tome-se também a métrica *Total Cyclomatic Complexity*. Essa métrica indica o quão complexa (no que tange a uso de laços e desvios) é uma determinada classe. Quanto maior o valor dessa métrica, em tese pior está codificado o programa. Entretanto é de se esperar que quanto mais linhas de código tiver uma classe, mais difícil será evitar o uso de laços e desvios. Nesse caso surge a necessidade de achar uma razão entre essas duas métricas, a fim de criar uma nova métrica que possa mensurar essa realidade. Como exemplo, essa métrica poderia se chamar *Total Cyclomatic Complexity per Line*, a qual o valor seria definido pela divisão da métrica *Total Cyclomatic Complexity* pela métrica *Lines Of Code*.

Para poder realizar esse processo, criou-se um sistema que pudesse montar expressões matemáticas de métricas com o objetivo de formar novas métricas. Uma vez criada uma nova métrica, pode-se utilizar dela para criar novas métricas derivadas. Deste modo tem-se um sistema flexível para criação de métricas a partir de métricas, recursivamente.

4.2 ESTRUTURAS DAS MÉTRICAS DERIVADAS

O objetivo da configuração das diversas classes que montam esse sub-sistema de criação de métricas derivadas foi o de simular uma árvore de expressão matemática. Através dessa árvore é que se obtém os valores para métricas derivadas. O exemplo da Figura 7 demonstra como é a árvore de uma expressão matemática.

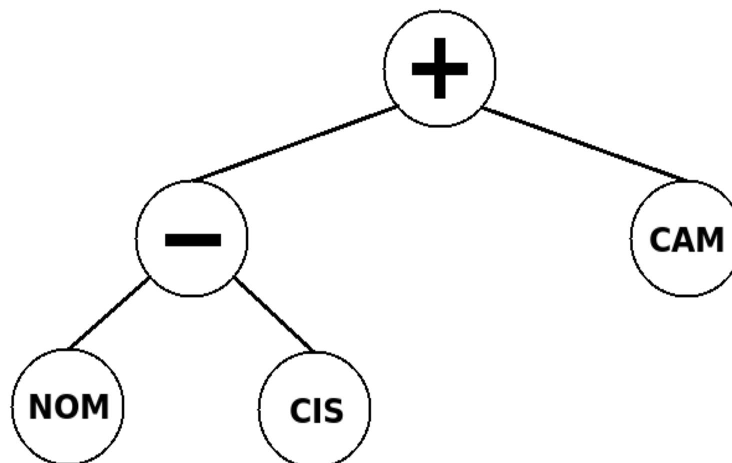

Figura 7. Árvore representando uma métrica composta.

A árvore descrita na Figura 7, representa uma métrica composta formada pelas métricas básicas *Number of Methods* (sigla NOM), *Class Interface Size* (sigla CIS) e *Cohesion Among Methods In Class* (sigla CAM). A árvore representa a expressão matemática (NOM – CIS) + CAM.

A estrutura da métrica derivada está centralizada na classe *DerivedMetric*. A classe *DerivedMetric* herda da classe abstrata *MetricManager*. Ela possui um atributo do tipo *Metric* assim como todas as outras métricas. A classe *Metric* define as características de cada métrica. Também possui os métodos *public MetricValue extractMetric(Revision configuration)* e *public MetricValue extractMetric(Revision revision, String path)*.

O que difere a classe *DerivedMetric* das demais classes de métrica é um atributo do tipo *MetricExpression* de nome *metricExpression*. A classe *MetricExpression* representa uma fórmula matemática envolvendo métricas. Os métodos polimórficos *extractMetric* da classe *DerivedMetric* apenas devolvem o valor da expressão matemática resolvida por essa classe. Os códigos da Figura 8 ilustram bem como esses métodos são implementados.

```
@Override
public MetricValue extractMetric(Revision configuration) throws
MetricException {
    double metricValue = metricExpression.getDoubleValue(configuration);
    return new MetricValue(configuration, metric, metricValue);
}
@Override
public MetricValue extractMetric(Revision revision, String path) throws
MetricException {
    double metricValue =
Double.valueOf(metricExpression.getDoubleValue(revision, path));
    return new MetricValue(revision, metric, metricValue);
}
```

Figura 8. Códigos dos métodos *extractMetric* com duas assinaturas diferentes.

A classe *MetricExpression* consiste em uma classe abstrata que possui os métodos: (i) *abstract public double getDoubleValue(Revision revision) throws MetricException;* (ii) *abstract public double getDoubleValue(Revision revision, String path) throws MetricException*; e (iii) *abstract public int getExtractsFrom()*. Os dois métodos de nome *getDoubleValue* são métodos abstratos que retornam o valor de uma métrica segundo um objeto *Revision* ou um objeto *Revision* e uma *String*. Essa classe deve ser estendida por outras

classes e de acordo com a classe que a estenda, esses métodos devem ser implementados de uma determinada maneira. O objetivo é que, através das várias extensões da classe *MetricExpression*, possa-se formar uma árvore que represente uma expressão matemática. Cada classe estendida de *MetricExpression* representa um nó da árvore da expressão matemática (operandos e operadores). Existem, então, nove classes que estendem *MetricExpression*. São elas *Add*, *Div*, *DoubleValue*, *MetricManagerExpression*, *Mult*, *Pow*, *Sqrt*, *Sub*, *UnarySub*. A Figura 9 representa o diagrama de classes das métricas derivadas.

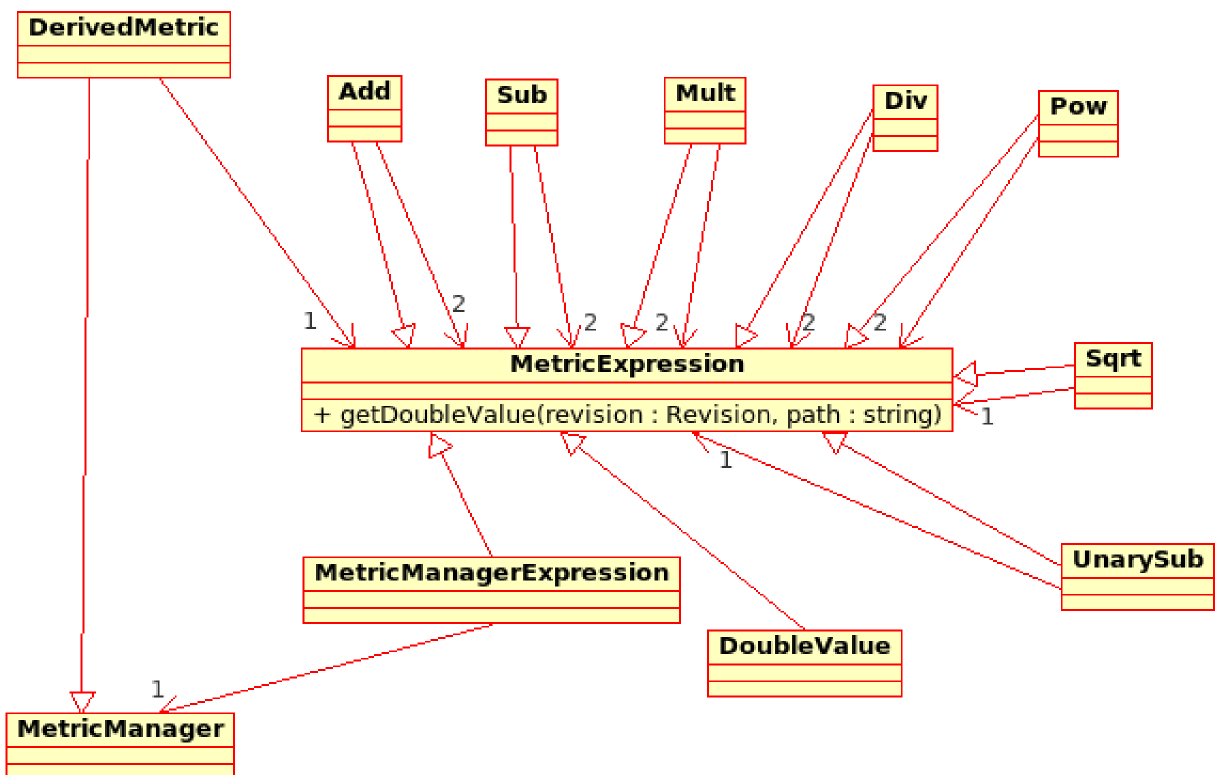

Figura 9. Diagrama de classes das métricas derivadas.

A classe *MetricManagerExpression* possui um atributo *MetricManager metricManager* que define uma métrica (podendo ser ela derivada ou não). Essa classe representa uma métrica na expressão matemática. Portanto, os seus métodos *getDoubleValue* retornam o valor da métrica *metricManager*. A Figura 10 mostra o código da implementação do seu método *getDoubleValue(Revision revision, String path)*.

```
public double getDoubleValue(Revision revision, String path) throws
MetricException {
    return metricManager.extractMetric(revision, path).getDoubleValue();
}
```

Figura 10. Implementação do método *getDoubleValue* da classe *MetricManagerExpression*.

A classe *DoubleValue* possui um atributo *double doubleValue*. Essa classe representa um número real na expressão matemática. Em algumas vezes, a expressão que representa a métrica derivada pode possuir constantes em sua fórmula. É para representar essas constantes que foi criado a classe *DoubleValue*. O seu valor de retorno (para os métodos *getDoubleValue*) é o próprio valor de *doubleValue*.

Tanto a classe *DoubleValue* quanto *MetricManagerExpression* são as folhas da árvore da expressão matemática. Ambas não precisam, necessariamente, de outros *MetricManagerExpression* para que se obtenham o seus valores. A única exceção fica por conta de *MetricManagerExpression*, pois o seu atributo *MetricManager* pode ser um *DerivedMetric*, o que implicaria em resolver uma outra árvore de expressão para saber o seu valor. Entretanto, se a sua classe *MetricManager*, for uma das vinte e duas classes de métricas básicas, apenas será necessário calcular o valor para essa métrica para se obter seu resultado.

As classes *Add*, *Sub*, *Mult*, *Div* e *Pow* representam respectivamente as operações de soma, subtração, multiplicação, divisão e exponenciação. Cada um delas possuem dois objetos *MetricExpression*, o *rigth* e o *left*. Dessa maneira, essas classes representam uma operação matemática binária sobre os dois objetos *MetricExpression*, os quais podem ser *DoubleValue*, *MetricManagerExpression* ou uma outra expressão matemática.

O resultado dos métodos *getDoubleValue* de cada uma dessas classes é então, a operação matemática que ela representa sobre os objetos *left* e *right*. A Figura 11 exibe respectivamente os códigos dos métodos das classe *Add*, *Sub* e *Pow*.

```
public double getDoubleValue(Revision revision, String path) throws
MetricException {
  return left.getDoubleValue(revision, path) +
right.getDoubleValue(revision, path);
}
public double getDoubleValue(Revision revision, String path) throws
MetricException {
  return left.getDoubleValue(revision, path) -
right.getDoubleValue(revision, path);
}
public double getDoubleValue(Revision revision, String path) throws
MetricException {
    return Math.pow(left.getDoubleValue(revision, path),
right.getDoubleValue(revision, path));
}
```

Figura 11. Códigos das implementação do método *getDoubleValue* nas classes *Add*, *Sub* e *Pow*.

Por ultimo, as classes *Sqrt* e *UnarySub* representam respectivamente as operações unárias raiz quadrada e negativo. Ambas as classes possuem somente um único atributo *MetricExpression* de nome *metricExpression*. O resultado dos métodos *getDoubleValue* de cada classe é então, a operação unária sobre *metricExpression*. A Figura 12 mostra respectivamente o código do método *getDoubleValue* da classe *Sqrt* e *UnarySub*.

```
public double getDoubleValue(Revision revision, String path) throws
MetricException {
    return Math.sqrt(metricExpression.getDoubleValue(revision, path));
}

public double getDoubleValue(Revision revision, String path) throws
MetricException {
    return (-1) * metricExpression.getDoubleValue(revision, path);
}
```

Figura 12. Códigos das implementação do método *getDoubleValue* nas classes *Sqrt* e *UnarySub*.

A estratégia que foi utilizada para a criação das métricas derivadas é conhecida como o padrão de projetos *Interpreter* [15]. Esse padrão resolve a interpretação de gramáticas (geralmente gramáticas simples) através da criação de uma hierarquia de classes. A Figura 13 demonstra um exemplo de expressão matemática utilizando algumas das classes apresentadas até aqui, mostrando a relação entre as classes que compõe uma métrica derivada. A expressão matemática é a mesma da Figura 7, ou seja (NOM – CIS) + CAM.

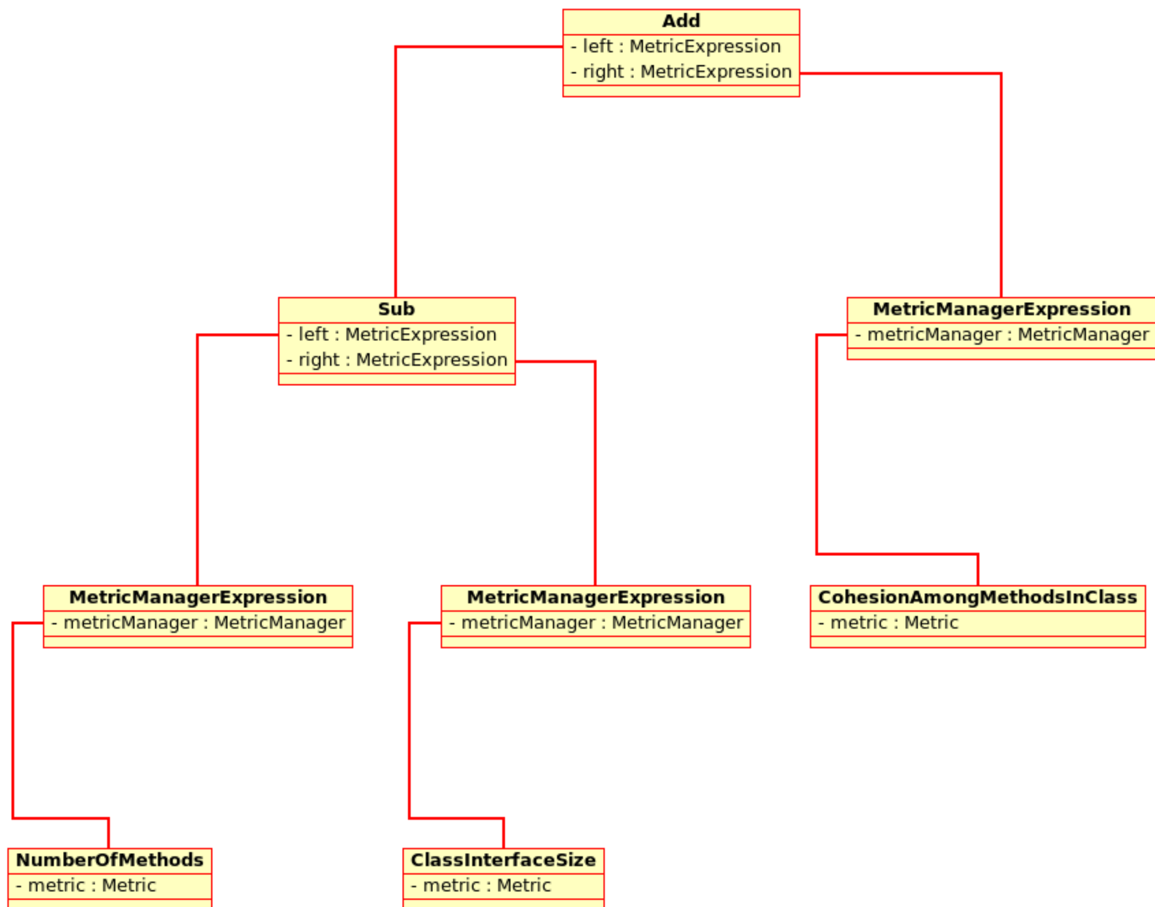

Figura 13. Exemplo de hierarquia de classes em métricas derivadas.

Como foi exposto no Capítulo 3, cada classe *Metric* possui o atributo *extractsFrom*, que indica de onde a métrica é extraída (projeto, pacote ou classe). Surge então a necessidade de definir o atributo *extractsFrom* das métricas derivadas. Convenciona-se então que uma métrica composta terá o seu *extractsFrom* igual ao menor (grão mais fino) *extractsFrom* das métricas que a compõe. Por exemplo, se uma métrica derivada é composta por três outras métricas, sendo que uma de projeto, outra de pacote e outra de classe, então esta métrica

derivada será uma métrica de classe. Se, por outro lado, for composta por três métrica de projeto, então essa métrica derivada será de projeto. Se for composta por duas de projeto e outra de pacote, será uma métrica de pacote.

Para saber qual o menor grão em uma métrica derivada é implementado o método *public int getExtractsFrom()* localizado na classe abstrata *MetricExpression*. A classe *MetricManagerExpression* implementa esse método de forma que ele retorne o *extractsFrom* da métrica que ela representa (a métrica localizada no atributo *metricManager*), executando o método *getExtractsFrom()* da classe *Metric*.

As outras classes descendentes de *MetricExpression*, com exceção da classe *DoubleValue*, implementam esse método obtendo informações dos seus atributos do tipo *MetricExpression*, com o objetivo de que a operação, recursivamente, chegue a alguma folha e retorne o valor desejado. O objetivo final é chegar às folhas da árvore, as quais retornarão um valor *extractsFrom*. Uma classe de operação binária, então, retorna o menor *extractsFrom* de seus dois atributos *MetricExpression* (*left* e *right*). Já uma operação unária simplesmente retorna o *extractFrom* do seu único atributo *MetricExpression*.

A classe *DoubleValue* também será uma folha da árvore de expressão e como não possuirá nós abaixo dela, terá que retornar algum valor para os nós ascendentes. Por esse motivo, esse nó sempre retorna o *extractsFrom* com valor de métrica de projeto. O objetivo é que, caso haja alguma métrica com *extractsFrom* de menor granularidade, esse valor seja perdido quando chegar em algum nó de maior ascendência. Um exemplo de como essa estratégia funciona é mostrado na Figura 14.

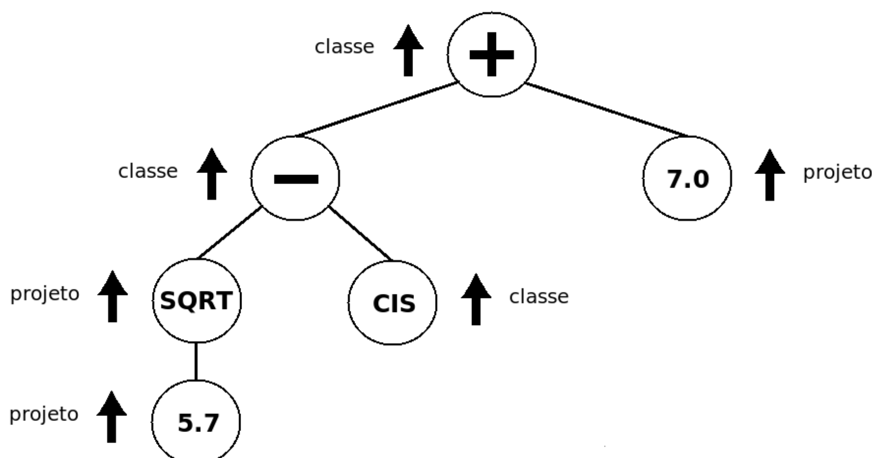

Figura 14. Exemplo de funcionamento da estratégia recursiva, de obter o *extractsFrom* da métrica derivada.

A classe *Metric* de uma métrica derivada possui duas diferenças. A primeira é o valor booleano do atributo *derived*, que é *true*. A outra é o preenchimento da *String expression* (nas classes de métricas básicas essa *String* possui o valor nulo). Nesta *String* está armazenada a expressão matemática da métrica derivada. A expressão matemática possui como componentes: (i) Os operadores matemáticos +, -, *, / e ^, representando respectivamente soma, subtração, multiplicação, divisão e exponenciação; (ii) parênteses "(", ")", utilizados para indicar a precedência das operações matemáticas; (iii) a função *sqrt()*, que calcula a raiz quadrada da expressão matemática localizada dentro dos parênteses; (iv) números de ponto flutuante; (v) as siglas das métricas pré-existentes.

Como exemplos de expressões matemáticas de métricas derivadas é possível citar:

- *-((MLOC*2.5)+NORM)*
- *sqrt(CIS)*(CAM/NORM)*
- *NOM – CIS*

Uma vez que uma métrica derivada esteja armazenada no banco de dados, pode-se criar um objeto *DerivedMetric* dinamicamente após extrair do banco de dados, um objeto da classe *Metric*. Para que tal ação ocorra é necessário o auxílio da classe *DerivedMetricService*.

Dentro da classe abstrata *MetricManager* existe o método *public void setup(Metric metric)*, que inicializa os objetos *MetricManager* das métricas. No caso de *DeriveMetric*, esse método é implementado de forma a obter a expressão da *String expression* da classe *Metric* e transforma-la em um uma árvore de objetos *MetricExpression*, formando assim a árvore de expressão matemática correspondente. A implementação deste método é descrita no código da Figura 15.

O método *buildExpression* da classe *DerivedMetricService* recebe como parâmetro uma *String* (representando uma expressão) e retorna um objeto *MetricExpression* que representa o nó raiz da árvore de expressão. Esse método trata-se da implementação de um padrão de projeto conhecido como *Builder* [15], padrão esse que permite a construção de diferentes representações de objetos complexos através de um mesmo procedimento. Dentro desse método chama-se o método *String eliminateSpaces(String expression)*, que elimina todos os espaços contidos na *String*.

```
public void setup(Metric metric) {
    if (metric == null) {
        throw new InstantiationError("The metric for derived metric can't
be null");
    }
    this.metric = metric;

     try {
        DerivedMetricService dms = new DerivedMetricService();
        this.metricExpression= dms.buildExpression(metric.getExpression());
    } catch (ServiceException ex) {
        throw new InstantiationError(ex.getMessage());
    }
}
```

Figura 15. Implementação do método *setup(Metric metric)* na classe *DeriveMetric*.

A partir dessa *String* previamente tratada, chama-se o método *buildTokens(String expression)* que devolve um objeto do tipo *Token*. A classe *Token* possui os atributos: (i) *Token prox*; (ii) *Token left*; (iii) *Token right*; (iv) *int type* representando o tipo daquele *Token*. O tipo do *Token* pode ser cada um dos tipos de nós de uma árvore de expressão, ou seja parênteses, soma, subtração, número de ponto flutuante, uma métrica, etc; (v) a *String metricAcronym*, utilizada apenas caso o tipo do *Token* seja o tipo métrica, representando a sigla da respectiva métrica; (vi) um atributo *double* de nome *doubleValue*, utilizado apenas quando o tipo de *Token* é do tipo número de ponto flutuante, guardando dessa maneira seu respectivo valor.

Esses *Tokens* podem formar uma lista, ou, como será mostrado mais adiante, uma árvore binária. O objeto *Token* retornado por *buildTokens* representa a cabeça da lista de *Tokens* ligados pelo atributo *Token prox*. A Figura 16 mostra como fica a lista de *Tokens* passando à *buildTokens* a expressão -((NORM*5)+CIS^5).

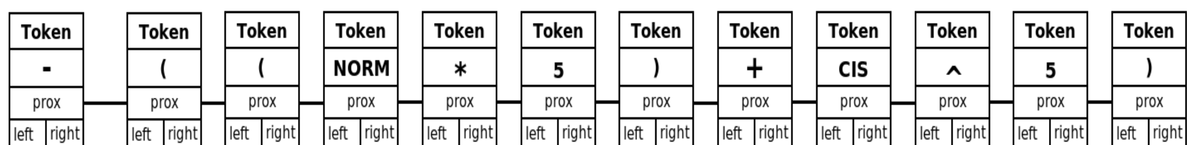

Figura 16. Lista de objetos *Token*.

Uma vez obtida uma lista encadeada de objetos *Token*, então, uma classe do tipo *PushDownAutomaton* é criada. Essa classe representa um automato de pilha [26], que reconhece se a expressão está na linguagem que foi determinada e caso não esteja devolve o erro da expressão. O método chamado é expressionReckonigze(Token token), que além de reconhecer a linguagem, também diferencia os símbolos “-” em *Sub* ou *UnarySub*. Todos os símbolos “-” são marcados inicialmente como *Sub*. A diferenciação do símbolo de subtração ocorre conforme a localização deste símbolo na expressão matemática e pode ser obtido facilmente no próprio processo de reconhecimento da linguagem. O processo que foi utilizado para reconhecimento da expressão é representado na Figura 17. Na figura, *to_U-* significa a transformação de um *Sub* em *UnarySub* (aqui será representado por *U-*). *push(1)* e *pop()* são ações sobre a pilha do autômato. As ações são separadas por ponto e vírgula (;). A linguagem é reconhecida caso o autômato termine no estado final (nó 1) e com a pilha vazia.

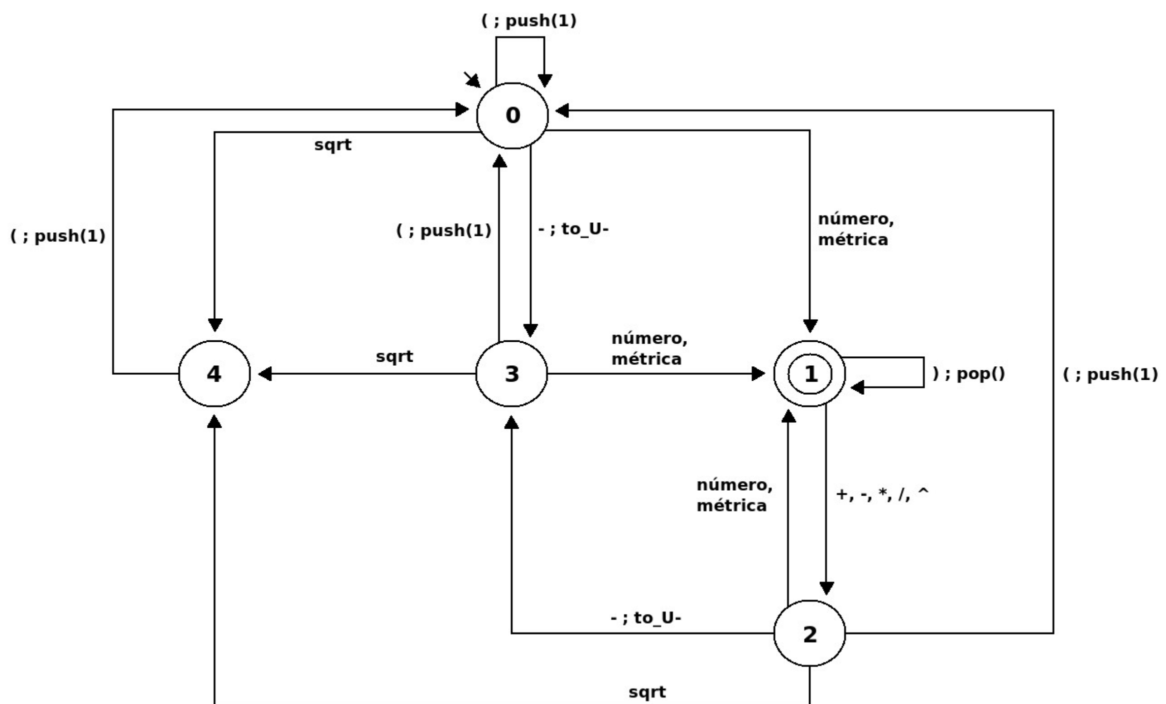

Figura 17. Funcionamento do método *expressionReckonigze*.

Como exemplo analisa-se, através do autômato, a expressão (NOM*5)*(-CAM). Começando no estado 0, após ler o *Token (* continua-se no estado 0 e adiciona-se 1 ao topo da pilha. É então lido o *Token* que representa a métrica NOM. Passa-se então ao estado 1. É lido o *Token* que representa o operador * e em seguida o *Token* que representa o número 5, o que faz com que, respectivamente, o autômato passe ao estado 2 e depois retorne ao estado 1. É então lido o *Token)*, o que faz com que o autômato continue no estado 1 e que remova-se o

topo da pilha, deixando a vazia. É então lido o *Token* que representa o operador *, fazendo o autômato passar ao estado 2. É lido o *Token (* , levando ao estado 0 e adicionando-se 1 ao topo da pilha. É lido o *Token* que representa o operador binário - e esta operação faz com que esse *Token* seja substituído por outro que representa a operação *UnarySub*. O autômato passa para o estado 3. É lido o *Token* que representa a métrica CAM e passa-se ao estado 1. Por último é lido o *Token),* o que faz com que o autômato continue no estado 1 e que seja removido o topo da pilha, deixando novamente a pilha vazia. Terminada a leitura da expressão, como o estado se encontra em 1 (estado final) e a pilha se encontra vazia verifica-se então que a expressão é válida.

Após o término do método e, consequentemente, o reconhecimento da linguagem e diferenciação dos símbolos de subtração, é chamado o método *createTreeExpression(Token head), que, a partir de uma lista de Tokens* ligadas pelo atributo *prox*, gera uma árvore da expressão matemática.

Esse método basicamente obtém uma lista de *Tokens* e então os processa de acordo com sua prioridade. A prioridade na resolução de uma fórmula matemática são os parênteses. Primeiramente removem-se esses parênteses e devolve-se sua expressão matemática de acordo com uma regra definida. Portanto, o método primeiramente encontra um conjunto de parênteses. Esse conjunto de parênteses são transformados em uma única árvore. O nó raiz dessa árvore é colocado no lugar de onde estava esse conjunto de parênteses. Quando o sistema encontra um parêntese aberto, percorre a expressão até encontrar o fechamento do parênteses para daí, chamar a função recursivamente.

Como exemplo, tem-se a expressão 6+(COM + (CAM-2))*3. O método encontra uma abertura de parênteses, e então percorre a expressão até encontrar seu respectivo fechamento. Portanto nesse exemplo, COM + (CAM-2) irá ser processada recursivamente e gerará uma árvore que será colocada onde estava esse conjunto de parênteses. É importante notar que tanto a abertura quanto o fechamento de parênteses são eliminados, já que suas funções eram puramente a de descrever uma precedência das operações matemáticas. Uma vez a precedência resolvida, após a geração de uma árvore, esses parênteses não são mais necessários. Adicionalmente, a própria remoção de parênteses é necessária para o funcionamento correto da recursão do método *createTreeExpression*.

Após a resolução dos parênteses, é necessário resolver o *sqrt* e o *UnarySub*. Como ambos são operadores unários, e uma vez que os parênteses já estão resolvidos, basta colocar o seu *prox* em *right* e em seu novo *prox* colocar o *prox* do antigo *prox*.

Após esse passo, processam-se as operações binárias. A resolução das operações

binárias consiste em colocar em seu *left* e *right* respectivamente os nós anteriores e posteriores a ela. As operações serão processadas na ordem de precedência matemática dos operadores e em caso de dois operadores terem mesma prioridade, o processamento ocorre da esquerda para a direita. Primeiro são processadas todas as operações de exponenciação. Em seguida todas as operações de multiplicação e divisão (essas últimas duas tem a mesma precedência, sendo então realizada primeiro a operação que surgir). Por último as operações de adição e subtração são processadas.

As Figuras 18, 19, 20, 21, 22, 23, 24, 25 e 26 mostram um exemplo, passo a passo, dessas transformações utilizando como entrada a expressão -((NORM*5)+CIS^5), a mesma da Figura 16. Na Figura 18, os *Tokens* acabam de passar pelo método *expressionReckonigze(Token token)*, e portanto além de terem sido reconhecidos como expressão válida, tiveram seus "-" diferenciados. Na Figura 19 é mostrada a configuração dos Tokens após a primeira chamada recursiva de *expressionReckonigze*. Na Figura 20 mais uma recursão é chamada. Na Figura 21 é feita o processamento de um dos operadores, o que termina por formar o trecho da lista de Tokens em um trecho da árvore que será formada ao termino desse método. Na Figura 22 é mostrado a configuração da lista após o término da segunda recursão. A Figura 23 mostra o processamento do operador ^ e a Figura 24 do operador +. A Figura 25 mostra o retorno à instância original do método *expressionReckonigze*. A Figura 26 mostra o processamento do *UnarySub* e o fim do método, mostrando então a configuração final da árvore de expressão matemática.

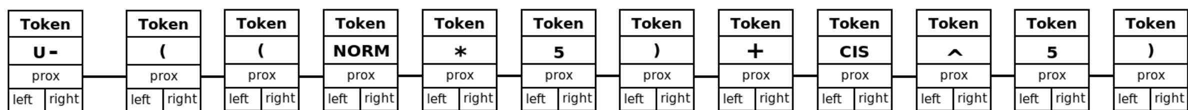

Figura 18. Conjunto de *Tokens* ligados em uma lista.

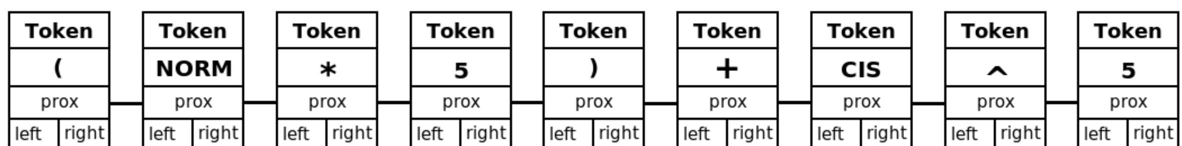

Figura 19. Primeira recursão do método *expressionReckonigze*.

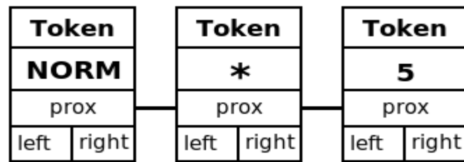

Figura 20. Segunda recursão do método *expressionReckonigze*.

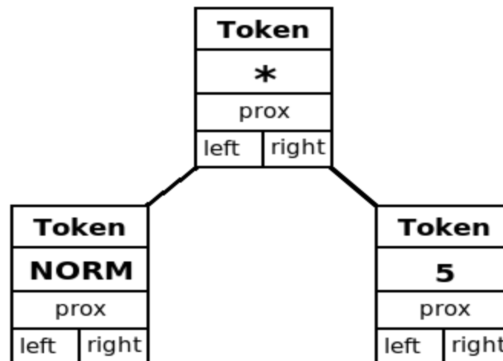

Figura 21. Resolução do operador * na segunda recursão.

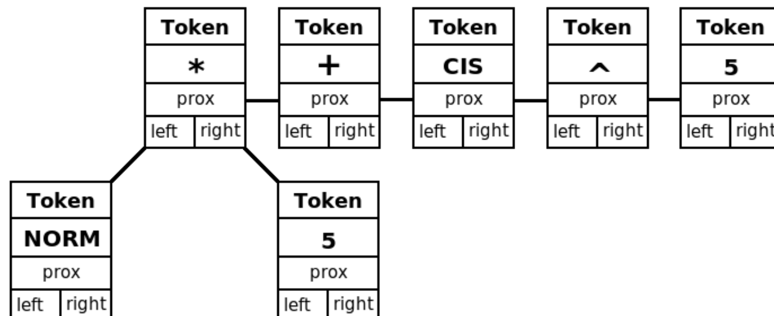

Figura 22. Retorno a primeira recursão.

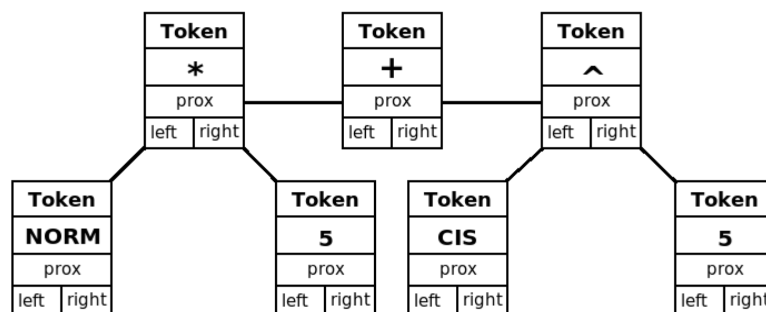

Figura 23. Resolução do operador ^ na primeira recursão.

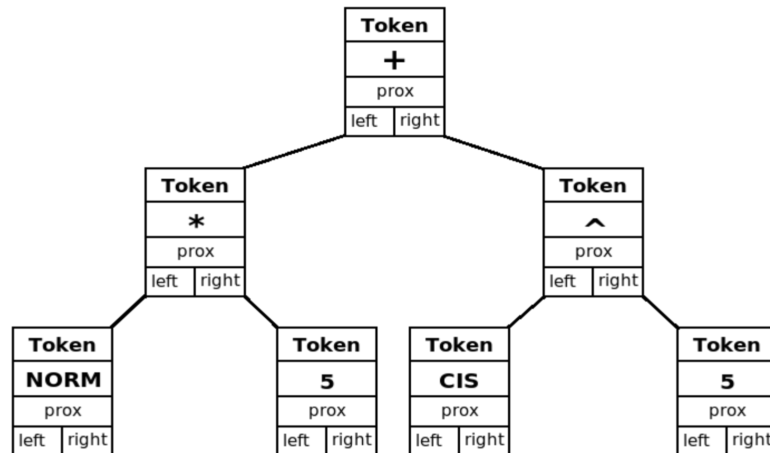

Figura 24. Resolução do operador + na primeira recursão.

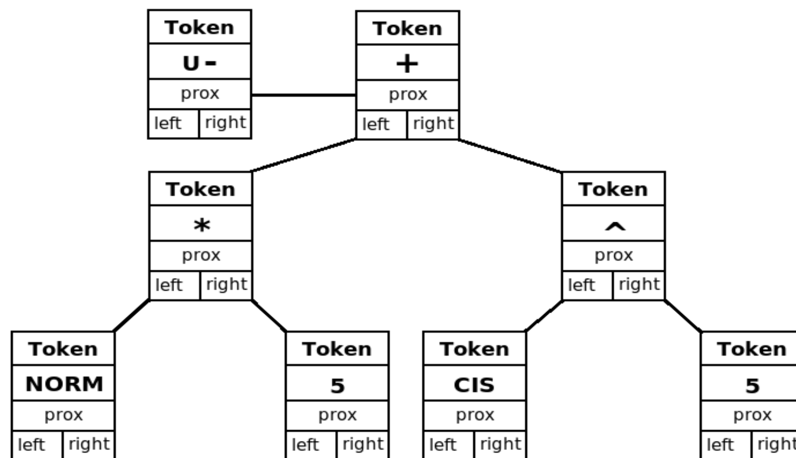

Figura 25. Retorno à instância original do método expressionReckonigze.

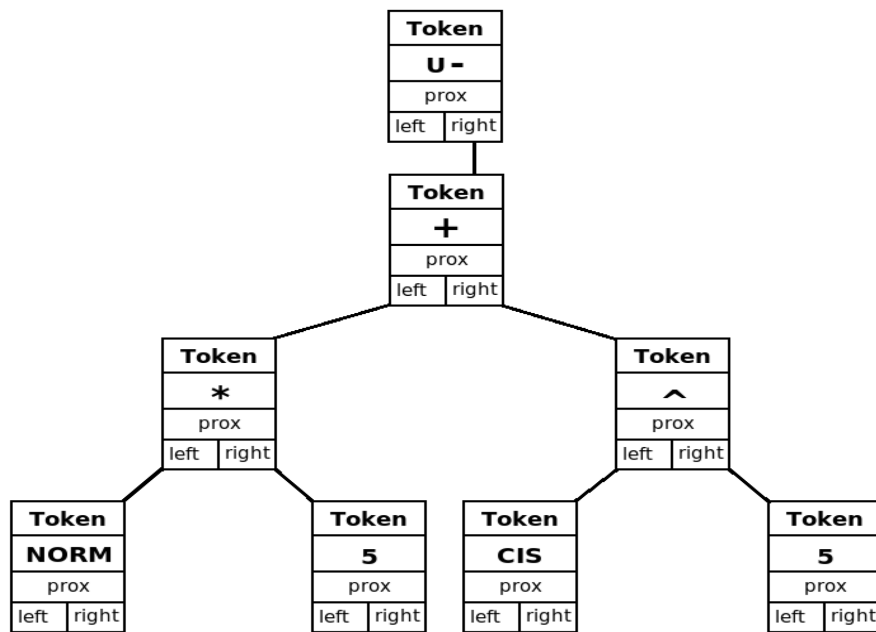

Figura 26. Resolução do *UnarySub*.

Uma vez processado o conjunto de *Tokens* é obtido um *Token* que representa o nó raiz da árvore da expressão matemática. Essa árvore é então transformada em um objeto *MetricExpression* através do método *createExpressionFromTree(Token head)*. Esse método percorre recursivamente a árvore de objetos *Token* e cria uma árvore de objetos *MetricExpression*.

4.3 CONSTRUINDO NOVAS MÉTRICAS INTERATIVAMENTE

Uma vez que toda a estrutura e mecanismo de criação de métricas derivadas foi explicada, agora será explicado como funciona a interface que permite ao usuário criar as métricas compostas. A métrica é criada na parte web do Oceano através de uma página HTML [27]. Seguindo a filosofia MVC [15] existe uma classe de nome CreateMetricBean. Esta classe controla a página *DetailMetric.xhtml*. Esta página possui um campo para escrever o nome da métrica (*Metric Name)*, sua sigla (*Metric Acronym*), sua expressão matemática (*Expression*) e sua descrição (*Description*). Existe ainda uma lista de métricas já criadas anteriormente, que possui, para cada métrica, uma breve descrição das mesmas. Em cada um desses campos existe um botão que ao se pressionar adiciona essas siglas ao campo

Expression. A Figura 27 mostra a aparência dessa pagina.

Create Metrics

Metric Name: SQRT of CAM
Metric Acronym: SQCAM
Expression: sqrt(CAM)
Description: Square root of CAM

Create Metric

Expression Examples

Metric Name	Acronym	Description
Abstractness	RMA	This metric extract the Abstractness of a given configuration.
Average Number Of Ancestors	ANA	Indicates the average number of classes that each project inherits information.
Class Interface Size	CIS	Class Interface Size
Cyclomatic Complexity	VG	This metric returns the Cyclomatic Complexity Number.
Data Access	DAM	Indicates the ratio of private attributes (protected) and the total number of attributes.
Design Size In Classes	DSC	Design Size In Classes
Direct Class Coupling	DCC	Indicates the number of differents classes with a class that relates.
Lack Of Cohesion Of Methods	LCOM	This metric returns the Lack Of Cohesion Of Methods.
Lines Of Code	LOC	This metric returns the Lines Of Code.
Measure Of Aggregation	MOA	Indicates the number of data declarations, wich a user-definied data.
Measure Of Functional Abstraction	MFA	Indicates the ratio of the inherited methods and all accessible methods of a class.
Method Lines Of Code	MLOC	This metric returns the Method Lines Of Code.
Number Of Attributes	NOA	This metric returns the Number of Attributes.

Figura 27. Página de criação de métricas derivadas.

Após preenchidos os campos, pressiona-se o botão *Create Metric*. O método *buildExpression(String expression)* da classe *DerivedMetric* tenta então criar uma métrica *MetricExpression*. Caso esse processo falhe é retornado o erro, que será mostrado na página ao usuário. Caso contrário é criado com sucesso um objeto da classe *MetricExpression*. É criado então um novo objeto *Metric*, o qual será preenchido com várias informações como a expressão, nome, sigla, etc. Como houve sucesso na criação do objeto da classe *MetricExpression* sabe-se que a expressão que o usuário digitou é uma expressão válida. O objeto *Metric* recém-criado é então persistido no banco de dados através da classe *MetricService*. Usa-se para tal o método *save(Metric m)*.

4.4 EXEMPLOS

Alguns exemplos podem melhor mostrar o funcionamento desse módulo do sistema. Primeiro, será criado um exemplo com uma expressão malformada e por consequência será obtido o erro correspondente. Depois será criado outro exemplo com uma expressão válida, tendo como resposta o armazenamento da métrica no banco de dados.

Primeiro coloca-se a expressão (DAM*5.0)+2) no campo *expression*. Como é possível observar, existe um parêntese a mais na expressão e, portanto, o sistema deve reportar o erro ao usuário. A Figura 28 mostra os dados sendo preenchidos e a Figura 29 mostra o erro sendo reportado.

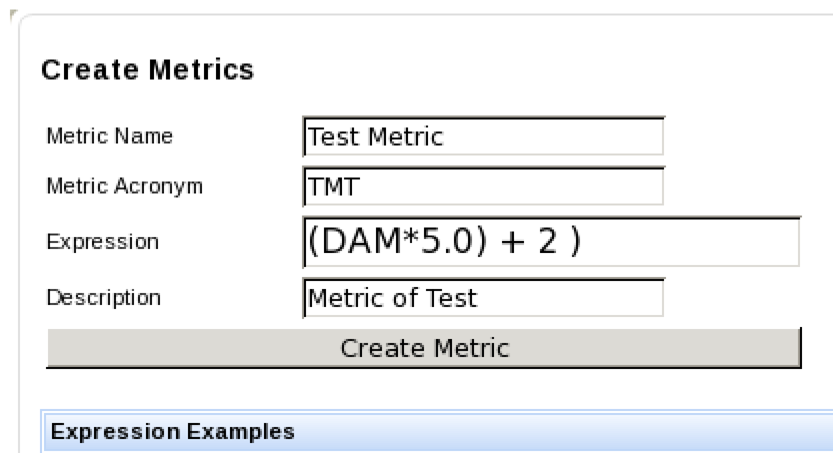

Figura 28. Preenchimento dos dados para a criação de métrica derivada.

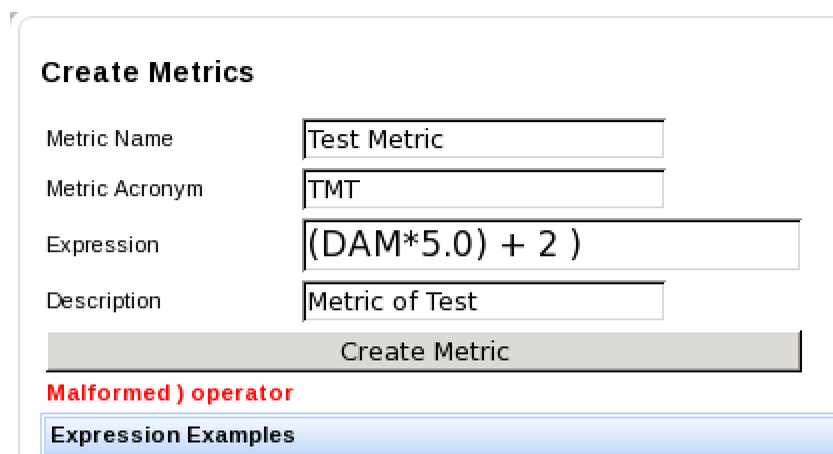

Figura 29. Erro sendo retornado.

Como é possível observar, o sistema retornou o erro *Malformed) operator*. Esse erro ocorre em decorrência da criação de expressões com números de fecha parênteses maior que o número de abre parênteses. Uma vez mostrado o erro, o usuário pode então corrigir sua expressão e tentar novamente criar a métrica.

Como outro exemplo, tenta-se criar uma métrica através da expressão NOA/NOM. Essa métrica vai se chamar *Attributes Per Methods*. Na figura 30 é mostrado como são preenchidos os campos de criação. Na figura 31 já é mostrada a métrica criada, aparecendo na lista de métrica pré-criadas, sendo a segunda métrica da lista (logo abaixo de *Abstractness*).

Com esses dois exemplos é possível observar como é o processo de criação das métrica. Neste módulo o objetivo foi o de criar uma interface simples, limpa, porém intuitiva ao usuário. Os botões, além de mostrar as métricas disponíveis ao usuário, também facilitam na hora de escrever a expressão.

Metric Name	Acronym	Description
Abstractness	RMA	This metric extract the Abstractness of a given configuration.
Average Number Of Ancestors	ANA	Indicates the average number of classes that each project inherits information.
Class Interface Size	CIS	Class Interface Size
Cyclomatic Complexity	VG	This metric returns the Cyclomatic Complexity Number.
Data Access	DAM	Indicates the ratio of private attributes (protected) and the total number of attributes.

Figura 30. Preenchimento dos campos.

Create Metrics

Metric Name: Attributes Per Methods
Metric Acronym: APM
Expression: NOA/NOM
Description: Indicates the ratio betweer
Create Metric

Expression Examples

Metric Name	Acronym	Description
Abstractness	RMA	This metric extract the Abstractness of a given configuration.
Attributes Per Methods	APM	Indicates the ratio between the number of attributes and methods.
Average Number Of Ancestors	ANA	Indicates the average number of classes that each project inherits information.
Class Interface Size	CIS	Class Interface Size
Cyclomatic Complexity	VG	This metric returns the Cyclomatic Complexity Number.
Data Access	DAM	Indicates the ratio of private attributes (protected) and the total number of attributes.

Figura 31. Métrica criada e mostrada na lista de métricas.

4.5 CONSIDERAÇÕES FINAIS

Este capítulo tratou sobre o subsistema de criação de métricas derivadas. Como foi apresentado, este módulo, para cumprir suas funções, utiliza-se de diversas áreas da computação, desde a interpretação de uma gramática, passando pelo uso de algumas estruturas de dados como listas, pilhas e árvores, até o uso das hierarquias presentes nas linguagens orientadas a objeto. A interação dessas diversas áreas da computação é traduzida na criação de um módulo que consegue estender a forma como são coletadas as informações sobre projetos de software.

5 MONITORAMENTO GRÁFICO

Um monitor gráfico é uma ferramenta de grande importância para o estudo das métricas. Sabendo-se que serão coletadas muitas métricas e que um projeto de software pode ter em torno de algumas centenas de revisões, o número de informação pode ser grande demais para que se possa tirar alguma conclusão desses dados com o auxílio de apenas tabelas de números. Gráficos podem condensar o grande volume de informação e mostra-los de uma forma mais intuitiva.

Por esse motivo criou-se um conjunto de gráficos que pudesse facilitar o exame e estudo das métricas. Foram escolhidos para tal gráficos de controle e um histograma. Um gráfico de controle é um gráfico que possui uma linha de limite superior, uma linha de limite inferior e uma linha central que geralmente mostra a média dos valores do conjunto de amostras. Já um histograma é um gráfico, normalmente de barras verticais, que representa a distribuição de frequências do conjunto de amostras.

Neste projeto o conjunto de amostras são os valores das métricas das revisões de um software. Cada gráfico utiliza como base os valores de uma determinada métrica de um determinado projeto em suas diversas revisões. O gráfico de controle mostra uma linha central representando a média desses valores, duas linhas (uma superior e outra inferior) representando o limite de um desvio padrão para mais e para menos, e duas linhas (uma superior e outra inferior) representando o limite de três desvios padrão para mais e para menos. Gráficos de controle costumam também possuir duas linhas representando o limite de dois desvios padrão (para mais e para menos), porém neste trabalho considerou-se não adicionar estas linhas nos gráficos por medida de simplificação dos mesmos. Já o histograma mostra a distribuição das frequências destes valores.

Na Seção 5.1 são mostrados os gráficos de controle e o histograma. Na Seção 5.2 é discutido o funcionamento interno destes gráficos. Na Seção 5.3 são exibidos alguns exemplos de utilização. Na Seção 5.4 são apresentados algumas considerações finais.

5.1 MONITORAMENTO VIA GRÁFICOS DE CONTROLE E HISTOGRAMA

Na entrada do módulo de monitoramento gráfico das métricas é exibido uma página com várias opções de projetos de software. Após a escolha de um dos projetos, entra-se em outra página que descreve esse projeto em detalhes. É mostrado informações como a URL do repositório, o seu sistema de controle de versão, etc. A Figura 32 mostra a aparência dessa página. Nesta figura é exibido o projeto GWT Maven Plugin [28] e é mostrado seu nome, se é um projeto Maven, o endereço de seu repositório, o sistema de controle de versão utilizado, o número de revisões medidas e o número de revisões nas quais não foi possível a compilação das classes do projeto.

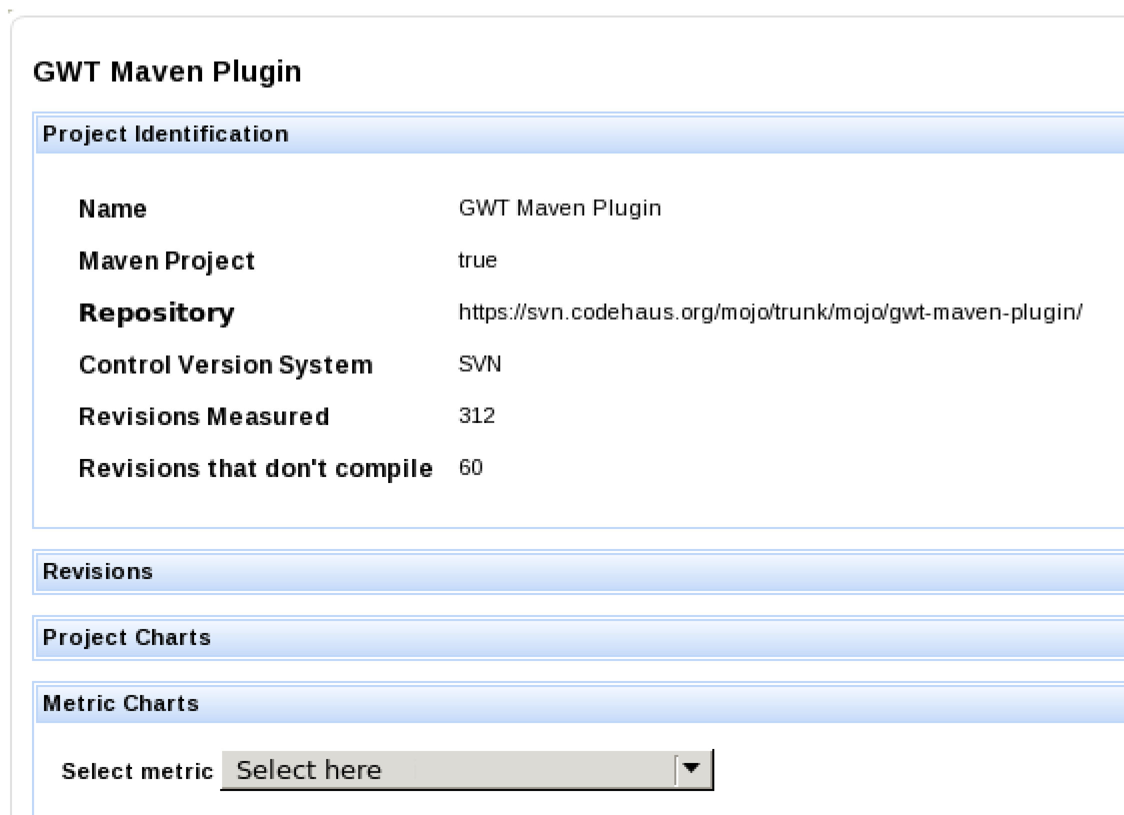

Figura 32. Página de monitoramento gráfico de um projeto.

Como se pode observar pela imagem, dentro dessa pagina existe um *combo box*, que permite a escolha de qual métrica se deseja examinar. Ao escolher uma métrica, é automaticamente gerado pelo sistema um conjunto de gráficos.

O primeiro gráfico a ser mostrado é um gráfico de controle com os valores absolutos da métrica. Esse gráfico exibe os valores das métricas em relação ao passar das revisões do

projeto. Também exibe uma linha verde representando o valor médio que a métrica exibe em todas as revisões, duas linhas amarelas representando o valor de um desvio padrão para mais e para menos, e duas linhas vermelhas representando três desvios padrões para mais e para menos.

Esse gráfico, nas métricas de classe e de pacote, indica o valor absoluto daquela métrica em cada revisão específica, mais especificamente a soma dos valores daquela métrica em todos os arquivos ou pacotes do projeto daquela determinada revisão. Pegue como exemplo a métrica *Lines of Code.* Essa é uma métrica de classe e representa o número de linhas de cada classe. No gráfico é mostrado o número total de linhas no projeto, somando o total de linhas de cada classe que compõe esse projeto. A Figura 33 mostra o gráfico absoluto da métrica *Lines of Code* no projeto GWT Maven Plugin.

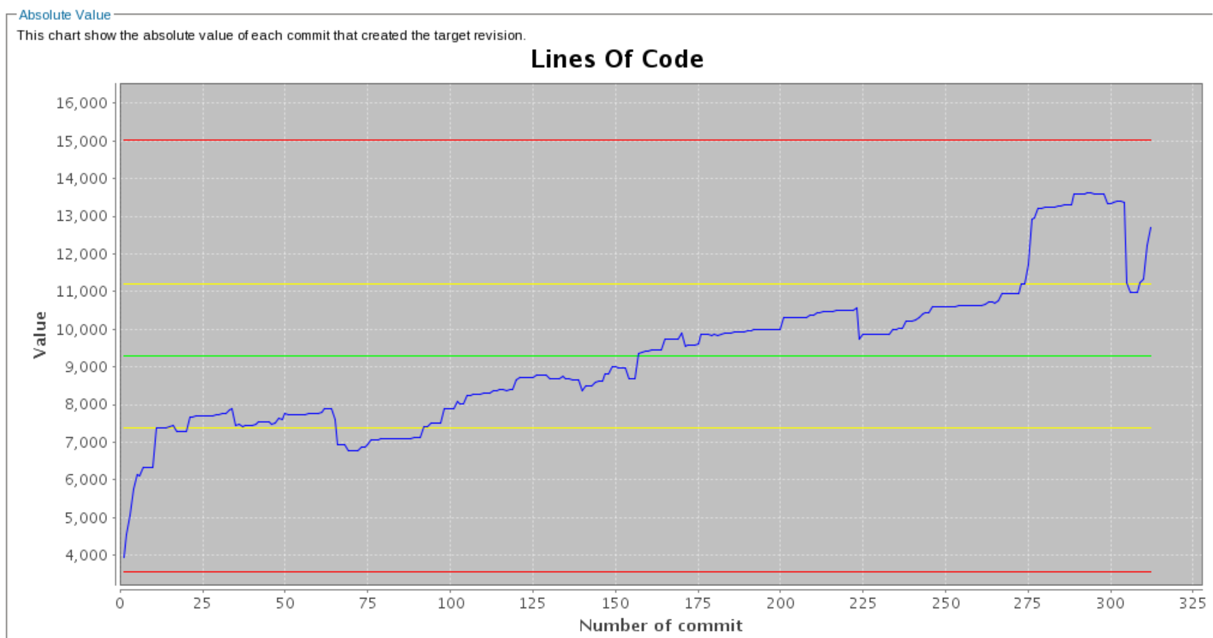

Figura 33. Gráfico de controle de valores absolutos da métrica *Lines of Code*.

No caso de métricas de projeto, o gráfico de valor absoluto demonstra o valor daquela métrica sobre o projeto em si, em cada revisão. Como toda métrica de projeto possui somente um valor por projeto, o seu cálculo depende de todas as classes do projeto e, portanto, o valor da métrica é relativo ao projeto como um todo. A Figura 34 exibe a métrica de projeto *Design Size in Classes* no projeto GWT Maven Plugin.

Figura 34. Gráfico de valores absolutos de métrica *Design Size in Classes.*

O segundo gráfico, o gráfico delta, apresenta a variação do valor daquela métrica em relação à revisão anterior. Ou seja, ela indica o quanto determinada revisão aumentou ou diminuiu os valores das métricas nos arquivos que foram modificados. Se os arquivos modificados em uma determinada revisão tiveram um aumento médio no valor de determinada métrica, o delta será positivo e o seu valor será o de quanto aumentou. Um exemplo de gráfico delta é mostrado na Seção 5.3.

É importante salientar que os arquivos modificados em uma determinada revisão são comparados com os valores antigos dos mesmos arquivos. Por exemplo em uma revisão de número 100 modificou-se os arquivos A, B e C. Na revisão de número 101 modificou-se os arquivos A, D e E e na revisão 102 modificou-se os arquivos A, B, C e D. Na revisão 102, para fins de cálculo de delta, os arquivos A e D são comparados com os arquivos A e D da revisão 101, enquanto que os arquivos B e C são comparados com o arquivos B e C da revisão 100.

Os dois gráficos exibem, toda vez que o mouse está sobre um ponto que representa uma revisão, informações adicionais sobre a revisão, como quem à criou e a sua data de criação. Essas informações são exibidas em uma caixa de texto, chamada de *tooltip.*

Por último, existe um histograma que exibe em um dos seus eixos, várias faixas de valores que a métrica pode assumir, e em outro a quantidade de revisões. A principal motivação para esse histograma é poder ver graficamente em qual faixa de valor está concentrada a maior parte das revisões e também em qual faixa há menor concentração. Cada

histograma tem por padrão cinco faixas de larguras idênticas, entretanto ele pode ser modificado tanto em relação à quantidade de faixas, quanto em relação à largura de cada faixa. A Figura 35 exibe o histograma a métrica *Lines of Code* no projeto GWT Maven Plugin.

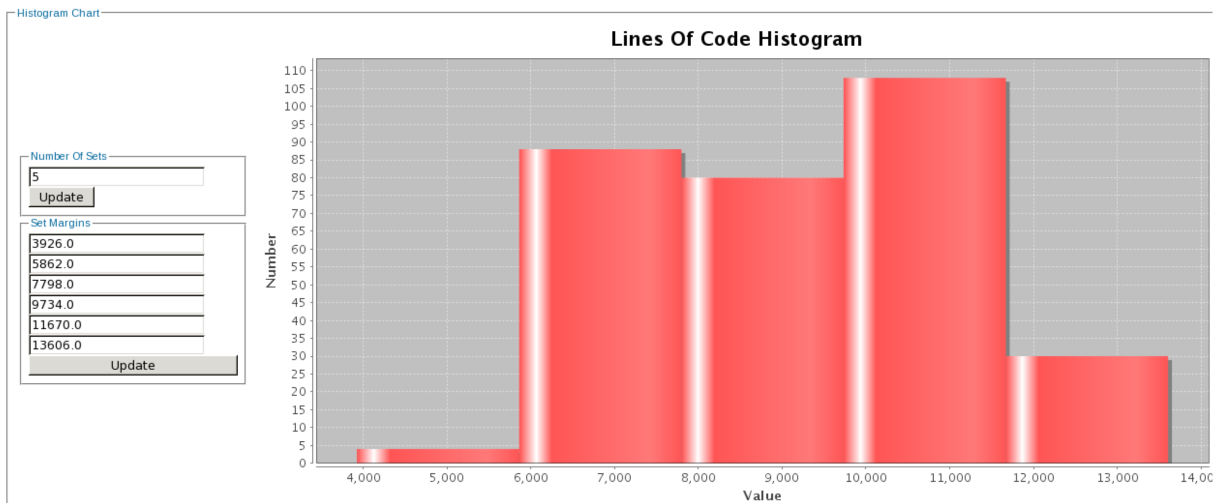

Figura 35. Histograma da métrica *Lines of Code.*

Através desses gráficos é possível observar características, que provavelmente uma tabela não conseguiria exibir de forma intuitiva. Com essa ferramenta o usuário consegue observar centenas de valores de métricas de um projeto, tirar conclusões, perceber valores que se mostram comuns ou atípicos, enfim, ter uma visão geral do comportamento do projeto com o decorrer do tempo. A observação visual do comportamento das métricas proporcionada pelos gráficos, permite ao usuário enxergar a variação das métricas como um todo e se necessário, focar apenas em momentos específicos do projeto.

5.2 FUNCIONAMENTO DO MONITOR

Os gráficos são exibidos em uma página HTML [27] através das funcionalidades do *framework* JSF [9]. Cada gráfico é criado na classe *MetricChartService*, localizado no pacote *br.uff.ic.oceano.ostra.service* do Oceano-Web. Para criar os gráficos foi utilizada o pacote JFreeChart [29], uma biblioteca que automatiza a criação de vários tipos de gráficos.

Cada gráfico criado é armazenado em uma imagem PNG [30], que fica armazenada no servidor onde o programa está executando. Para utilizá-la é necessário então saber a

localização interna de cada imagem, para que elas possam ser exibidas na pagina HTML ao usuário. Para que isso ocorra existe uma classe de nome *ChartValue* que armazena informações relativas à localização das imagens dos gráficos.

Através da pagina HTML o usuário escolhe o projeto e a métrica, a qual ele deseja observar. Tendo a métrica e o projeto, utiliza-se o método *public ChartValue getChartValue(SoftwareProject softwareProject, Metric metric, int x, int y, boolean isDelta, ServletContext sc)* da classe *MetricChartService* para a criação dos gráficos de controle. Os parâmetros *softwareProject* e *metric* representam respectivamente o projeto e a métrica a qual deseja-se observar.

Dentro deste método, através do método *getAbsoluteValuesByProjectAndMetric (SoftwareProject softwareProject, Metric metric)* da classe *MetricValueService*, obtêm-se uma lista de valores de métrica. Esses valores de métrica são armazenados em uma lista de objetos da classe *RevisionMetricValueDto*, a qual armazena informações de cada revisão, como o número da revisão, quem criou a revisão e a sua data de criação. Através do método *getDeltaValuesByProjectAndMetric(SoftwareProject softwareProject, Metric metric)* da classe *MetricValueService*, são obtidos os valores delta da métrica. Da mesma forma, esses valores deltas são armazenados em uma lista de objetos da classe *RevisionMetricValueDto*.

Com uma lista de objetos *RevisionMetricValueDto*, pode se criar a classe *XYSeriesCollection* através do método *createDataSet(List<RevisionMetricValueDto> dataValues, boolean isDelta)*. Essa classe cria uma estrutura que guarda pontos em suas coordenadas x e y os quais servem para a geração de um gráfico de controle.

Através do método *createChart(int x, int y, XYSeriesCollection dataset, String metricName, String chartPath, ServletContext sc)* é então criado um gráfico de tamanho *x* por *y*. O gráfico fica armazenado em um arquivo cujo caminho já foi previamente criado, representado pela *String chartPath*. Como saída o método devolve um objeto *ChartRenderingInfo* que possui informações a cerca do gráfico, como posição dos pontos, linhas, etc.

Nessa etapa é chamado o método *createImageMap(ChartRenderingInfo info, List<RevisionMetricValueDto> dataValues, boolean isDelta, String nameMap)*. Esse método, através das informações dos componentes do gráfico armazenadas na variável *info* e dos valores das métricas armazenadas na lista *dataValues*, cria uma *String* que representa uma *tag image map* em HTML. Com essa *tag* e a imagem do gráfico em uma pagina web, cada vez que o *mouse* do computador estiver sobre o ponto que representa uma revisão, é mostrado um *tooltip* com informações adicionais sobre aquela determinada revisão.

Tanto o caminho da imagem quanto a *tag image map* são armazenadas em um objeto *ChartValue*. Esse objeto é então enviado à classe *MonitoringChartBean*, que por sua vez monta e envia essas informações à página HTML para que, desta maneira, os gráficos de controle possam ser exibidos para o usuário. A Figura 36 exibe o diagrama de sequência da criação destes gráficos.

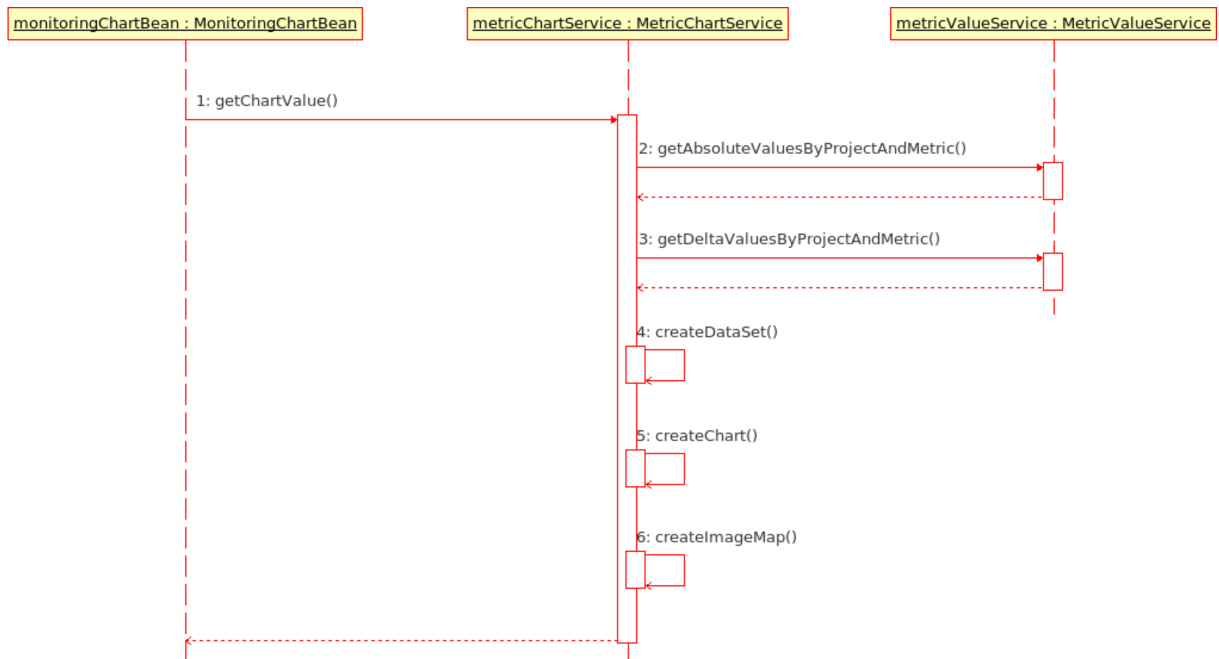

Figura 36. Diagrama de sequência da criação dos gráficos de controle.

Na primeira vez que o histograma é criado (quando são exibidos os valores padrão, ou seja, quando o usuário ainda não definiu uma quantidade de faixas nem suas respectivas larguras) são chamados os métodos *getSoftwareProjectMetricMaxValue(SoftwareProject softwareProject, Metric metric)* e *getSoftwareProjectMetricMinValue(SoftwareProject softwareProject, Metric metric)*. Esses métodos retornam respectivamente o valor máximo e mínimo de uma determinada métrica em um determinado projeto. O objetivo é encontrar uma grande faixa que vai do maior ao menor valor dessa métrica e então dividi-la em 5 (cinco) faixas de tamanhos iguais. Os valores de inicio e final de cada faixa são então armazenados em um vetor do tipo *double* de tamanho seis.

Para a criação do histograma é chamado o método *getHistogramValue (SoftwareProject softwareProject, Metric metric, int numberOfSets, double doubleHistogram[], int x, int y, ServletContext sc)*, onde *numberOfSets* representa o número de faixas que o histograma possui e *doubleHistogram[]* representa o vetor que guarda onde

começa e termina cada faixa do histograma. Assim, como no método de criação dos gráficos de controle, também é necessário obter uma lista de objetos *RevisionMetricValueDto* que armazenam os valores das métricas em suas várias revisões.

Análogo ao método *createDataSet* dos gráficos de controle é chamado o método *createHistogramDataSet(List<RevisionMetricValueDto> dataValues, int numberOfSets, double doubleHistogram[], boolean isDelta)*, o qual retorna a classe *XIntervalSeriesCollection*. Essa classe cria uma estrutura que guarda a quantidade de revisões presente em cada uma das faixas de valores do histograma, sendo que essas faixas de valores são representadas pelo vetor *doubleHistogram[]*. Essa classe é utilizada para a criação do histograma.

De maneira semelhante ao que ocorre no gráfico de controle é então criado o histograma através do método *createHistogram(int x, int y, XIntervalSeriesCollection dataset, String metricName, String chartPath, ServletContext sc)* e o *image map* para o histograma através do método *createHistogramImageMap(ChartRenderingInfo info)*. Ambos são armazenadas numa classe *ChartValue*. A Figura 37 exibe o diagrama de sequência da criação do histograma.

Figura 37. Diagrama de sequência da criação do histograma.

O histograma é criado por padrão com 5 faixas de valores, porém tanto o número de faixas quanto as suas larguras podem ser modificados através de caixas de textos geradas com JSF. Após o pressionamento do botão *Update*, dentro da classe *MonitoringChartBean* o vetor que representa o inicio e fim de cada faixa é atualizado, é iniciada uma nova chamada do método *getHistogramValue* e criado um novo histograma com esses novos dados de entrada.

5.3 EXEMPLOS DE UTILIZAÇÃO

Como exemplo mostra-se o projeto Publico Core, um projeto de software auxiliar ao *IdUFF* [4]. Neste projeto foi escolhida a métrica *Number Of Static Attributes*. Uma vez escolhidos o projeto e a métrica, o sistema gera o conjunto de dois gráficos de controle e um histograma. A Figura 38 exibe o gráfico dos valores absolutos, a Figura 39 exibe o gráfico dos valores delta e a Figura 40 exibe o histograma.

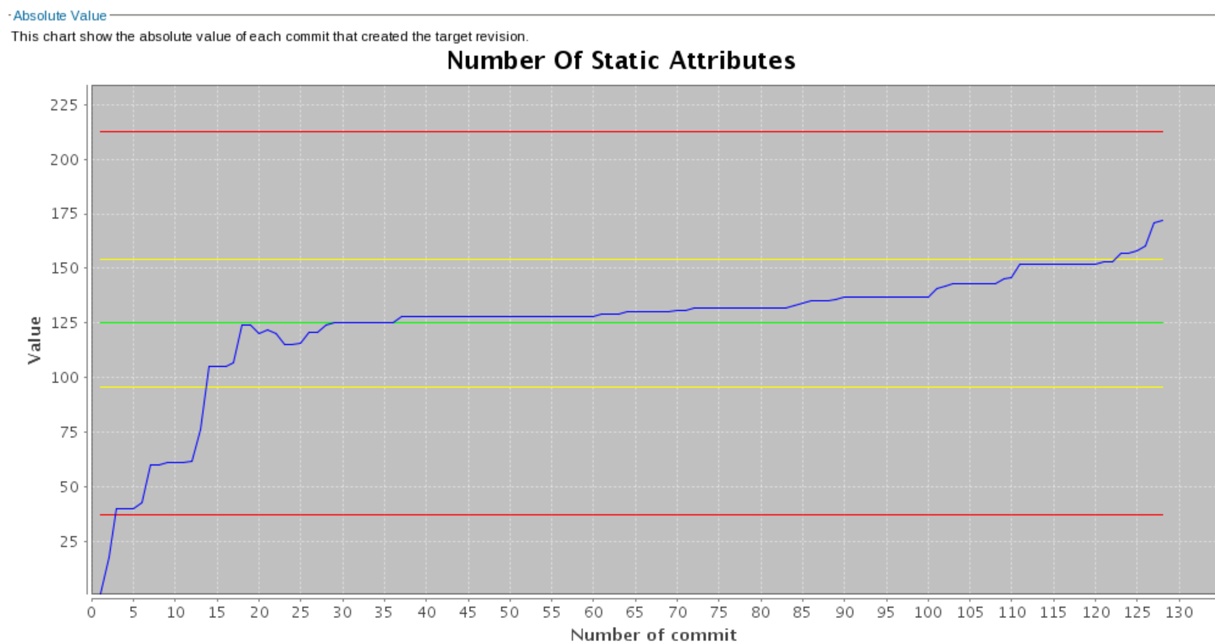

Figura 38. Gráfico de valores absolutos da métrica *Number Of Static Attributes*.

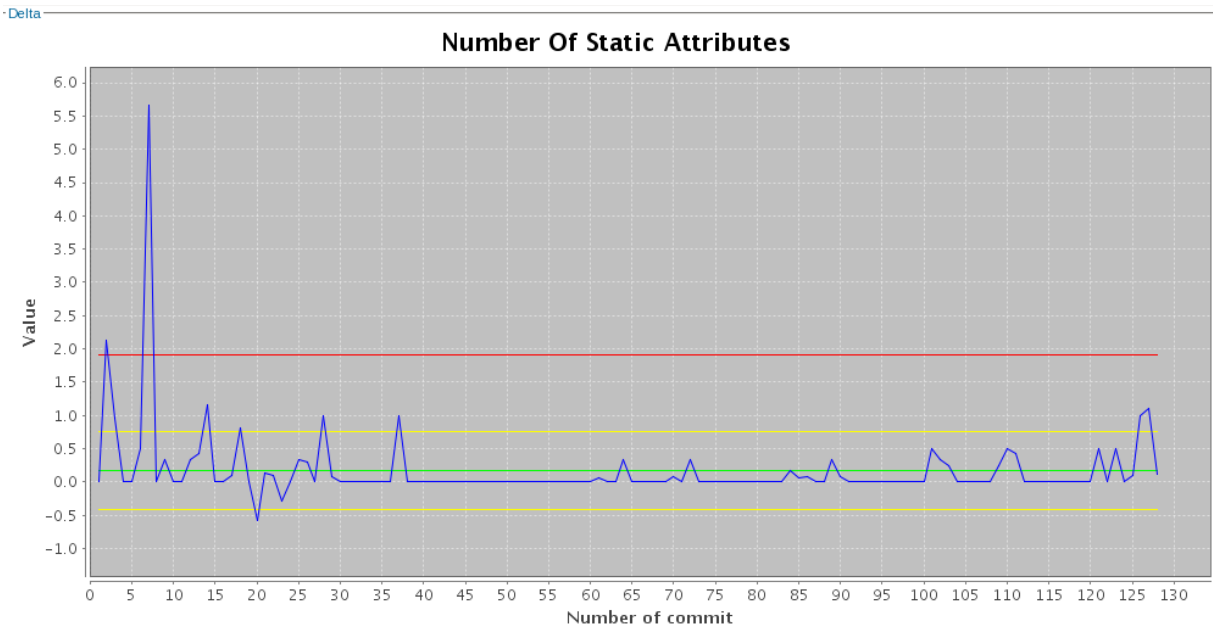

Figura 39. Gráfico delta da métrica *Number Of Static Attributes*.

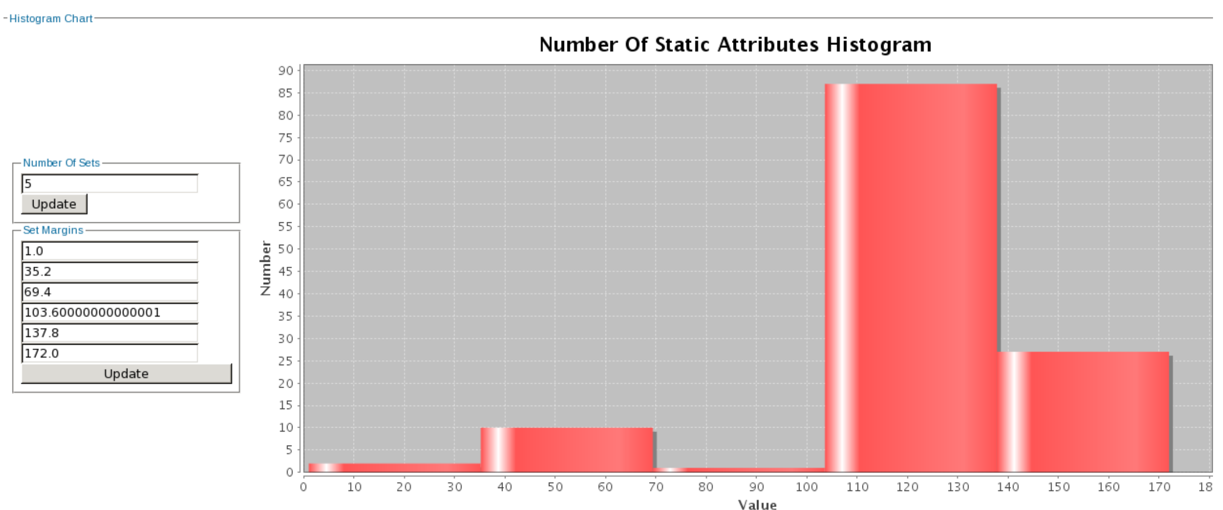

Figura 40. Histograma da métrica *Number Of Static Attributes*.

O histograma mostra que os valores dessa métrica estão quase todos acima de 103. Portanto, modificou-se o histograma para estudar como estão distribuídos os valores a partir de 103. Os seis campos que mostram os limites de cada faixa do histograma foram então alterados para os valores 103; 116,8; 130,6; 144,4; 158,2 e 172. Então ao pressionar o botão *Update* é gerado um novo histograma representado na Figura 41. Foram geradas faixas de tamanhos iguais apenas para fins de exemplificação. Os limites de cada faixa podem ser alterados para quaisquer valores, podendo gerar tamanhos de faixas distintos.

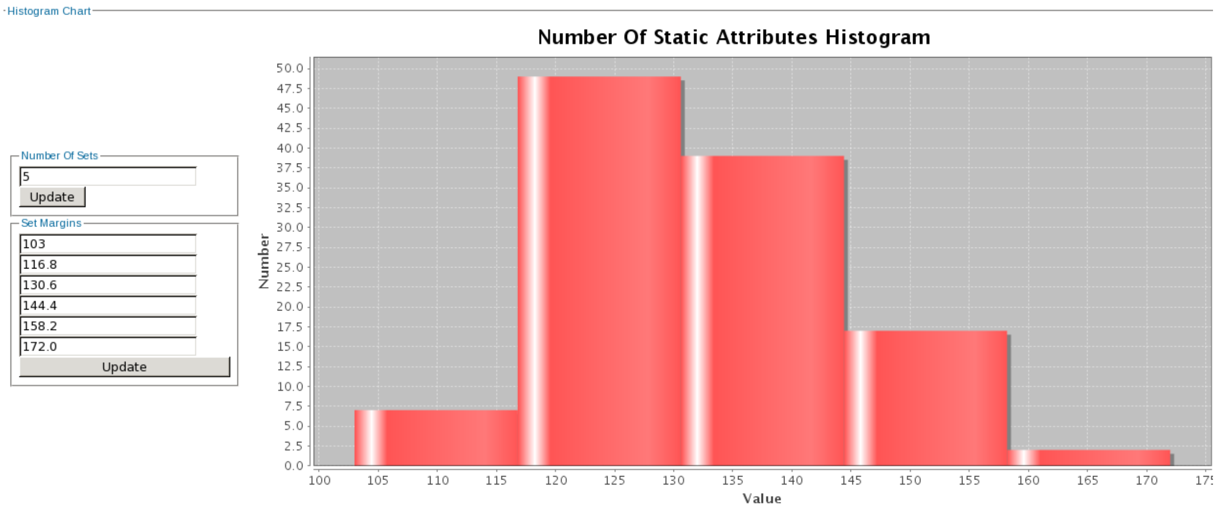

Figura 41. Histograma com os valores de limite modificados.

Como agora notado no novo histograma, a maior parte dos valores da métrica se encontram entre 116,8 e 144,4. Decidiu-se então modificar novamente o histograma, com o intuito de estudar a distribuição de valores nesta faixa. Os limite foram então alterados para 116,8; 122,32; 127,84; 133,36; 138,88 e 144,4. O resultado esta na Figura 42.

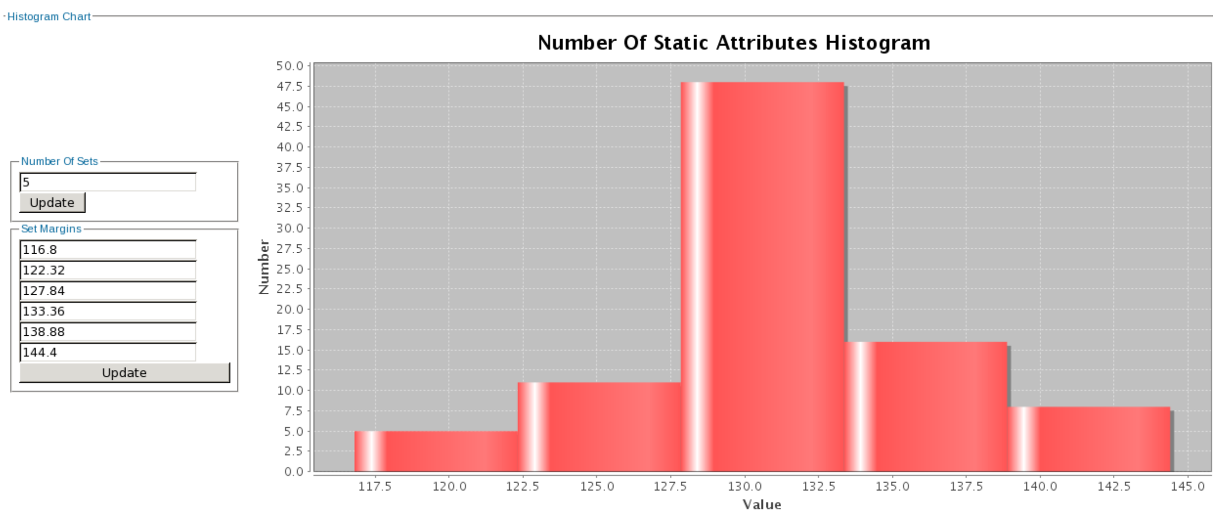

Figura 42. Histograma com os valores de limite modificados novamente.

Como pode ser observado pela imagem a maior parte dos valores está entre de 127,84 e 133,36. Observa-se que com o uso deste gráfico pode se mudar os limites das faixas indefinidamente e ter uma interpretação cada vez mais fina sobre a distribuição dos valores das métricas.

5.4 CONSIDERAÇÕES FINAIS

Este capítulo tratou sobre o módulo que apresenta os gráficos das métricas aos usuários. O trabalho realizado neste módulo foi o de organizar o conjunto de métricas dos diversos projetos de forma a que elas pudessem ser utilizadas para geração de gráficos. A parte de geração de gráficos fica por conta da ferramenta JFreeChart, que uma vez tendo um conjunto de dados organizados, consegue criar gráficos de forma bastante simples para o programador. Uma vez tendo os gráficos, o outro trabalho realizado foi exibi-los para o usuário utilizando para isso as diversas ferramentas do JSF. Essas ferramentas foram utilizadas tanto para a exibição, quanto para a criação de alguns componentes na página, que pudessem ser utilizados como interface de entrada ao usuário. Os campos de atualização das faixas do histograma são um exemplo desses componentes. No Capítulo 6 estes gráficos são utilizados para se obter conclusões a cerca de um projeto de software real.

6 AVALIAÇÃO EXPERIMENTAL

6.1 AVALIAÇÕES REALIZADAS

Para avaliar as funcionalidades deste projeto foi escolhido um projeto de software, extraídas algumas de suas métricas e estudados alguns de seus gráficos. O projeto escolhido foi o Publico Core, um projeto que auxilia o IdUFF [4]. O idUFF é o sistema de gestão acadêmica da Universidade Federal Fluminense. O Publico Core é o sistema que auxilia o IdUFF no acesso a dados compartilhados por outros sistemas, como dados de alunos e funcionários. Escolheu-se o Publico Core por ser um sistema relativamente simples, porém que possui grande importância para vários sistemas da UFF. As métricas extraídas foram *Number Of Methods*, *Total Cyclomatic Complexity* e *Lines Of Code*. O objetivo foi estudar o comportamento do projeto selecionado, com relação às métricas escolhidas, observar as variações de cada métrica e tentar inferir algumas implicações para tal comportamento. Essas métricas foram escolhidas, pois neste estudo tenta-se encontrar as relações entre a quantidade de código escrito e complexidade total do sistema.

6.2 RESULTADO DAS AVALIAÇÕES

No estudo do gráfico da métrica *Number Of Methods* verifica-se que o número de métodos do projeto tende a crescer de forma contínua e pouco acentuada em grande parte das revisões. Entretanto, o primeiro ponto que chama a atenção é o que ocorre entre a décima e décima segunda revisão. A décima revisão possui 399 métodos, enquanto que a décima segunda revisão possui 653 métodos, significando um aumento de 254 métodos. Também, observa-se outro trecho com aumento relativamente significativo, entre a décima quinta (653 métodos) e décima sexta (752 métodos) revisão. Outro ponto interessante é que o único trecho

onde diminui o número de métodos é entre a vigésima e vigésima primeira revisão, tendo respectivamente, 774 e 735 métodos. Após estes pontos o gráfico tende a ter um aumento pequeno e contínuo de métodos. A Figura 43 exibe o gráfico da métrica *Number Of Methods*.

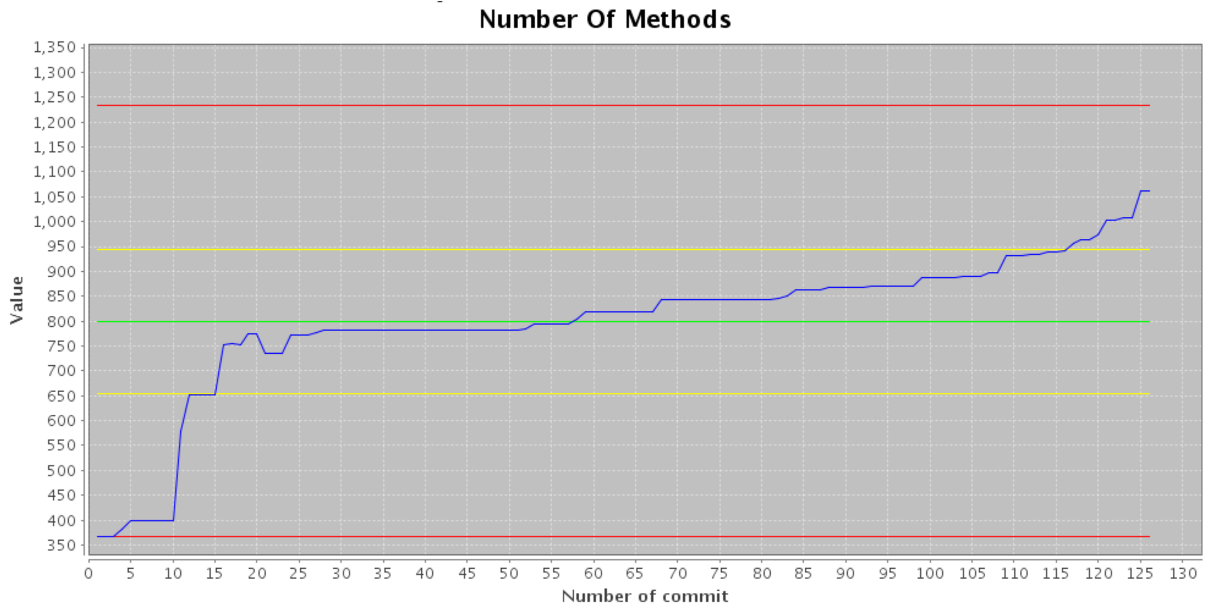

Figura 43. Gráfico do valor absoluto da métrica *Number Of Methods*.

Também é analisado o histograma dessa métrica. O que pode ser observado é que grande parte das revisões possui valores entre 645 e 923. Há um predomínio de valores entre 784 e 923. Pode-se deduzir que em algumas poucas revisões o projeto possui um número mais baixo de métodos (o início do projeto) e depois que atinge determinado valor o número de métodos tem um crescimento relativamente constante. A Figura 44 exibe o histograma de *Number Of Methods*.

Investiga-se então se há alguma relação entre os valores da métrica *Number Of Methods* e os valores das métricas *Total Cyclomatic Complexity* e *Lines Of Code*, estudando principalmente as revisões que chamaram atenção na métrica *Number Of Methods*. Primeiro verifica-se o gráfico da métrica *Lines Of Code*. Nesta métrica a décima revisão possui valor 7.199, enquanto que a décima segunda revisão possui valor 7.205, ou seja, houve um aumento de 6 linhas de código entre as revisões. Também é verificado que a décima quinta revisão possui 7.209 linhas de código, e a décima sexta revisão possui 7.693 linhas, um aumento de 484. Observa-se que na vigésima revisão o projeto possui 7.879 linhas e na vigésima primeira revisão possui 7.579 linhas, uma diminuição de 300 linhas. A Figura 45 exibe o gráfico dessa métrica.

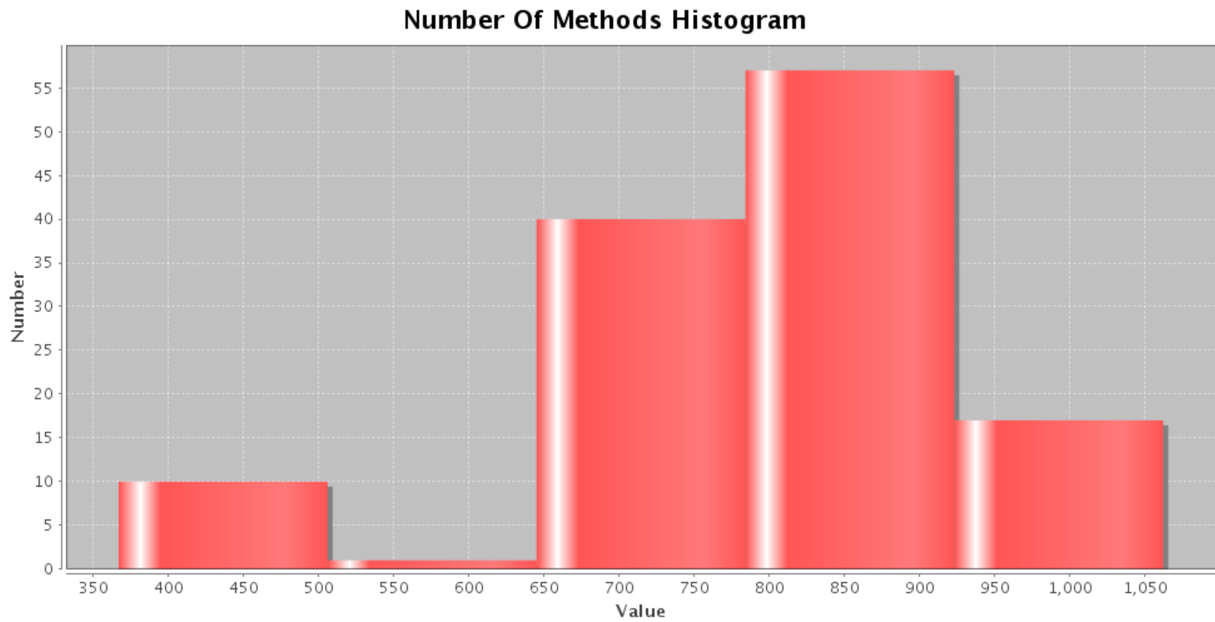

Figura 44. Histograma da métrica *Number Of Methods*.

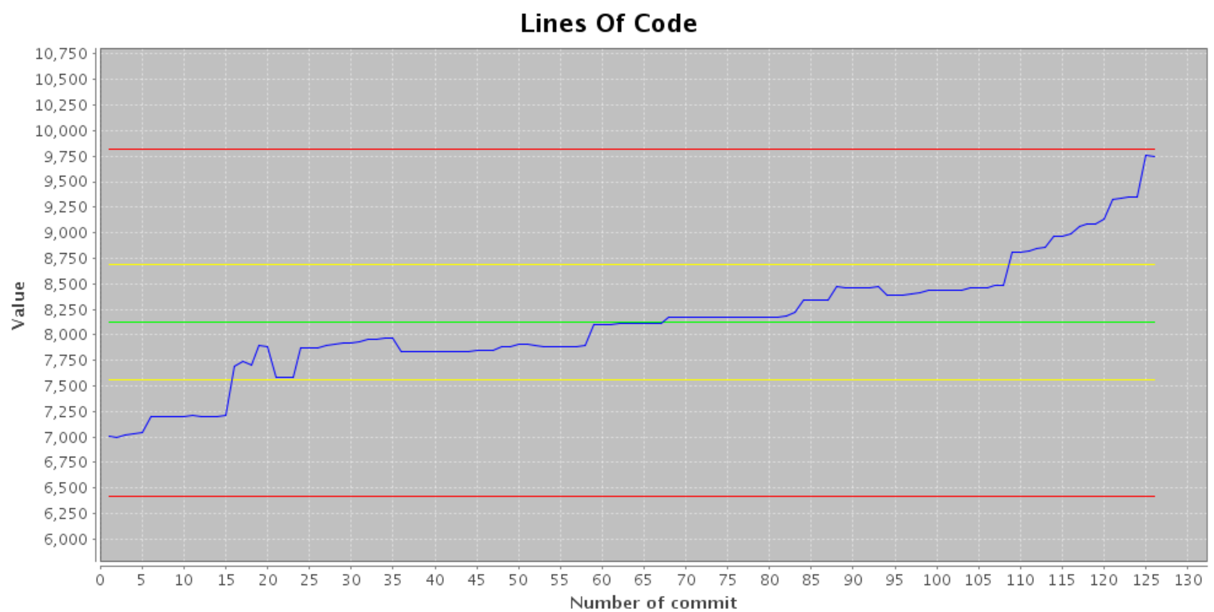

Figura 45. Gráfico do valor absoluto da métrica *Lines Of Code*.

Verifica-se também seu histograma. Esse histograma possui uma concentração de revisões entre os valores 7551 e 8653, como uma pequena predominância entre os valores 8102 e 8653. De forma semelhante à métrica *Number Of Methods*, o número de linhas de código parece possuir poucas linhas em certo número de revisões (no início do projeto) e então, uma vez atingindo certo patamar, o número de linhas de código mantém um aumento relativamente constante por um certo período de tempo até que há uma acentuação desse

aumento. A Figura 46 exibe este histograma.

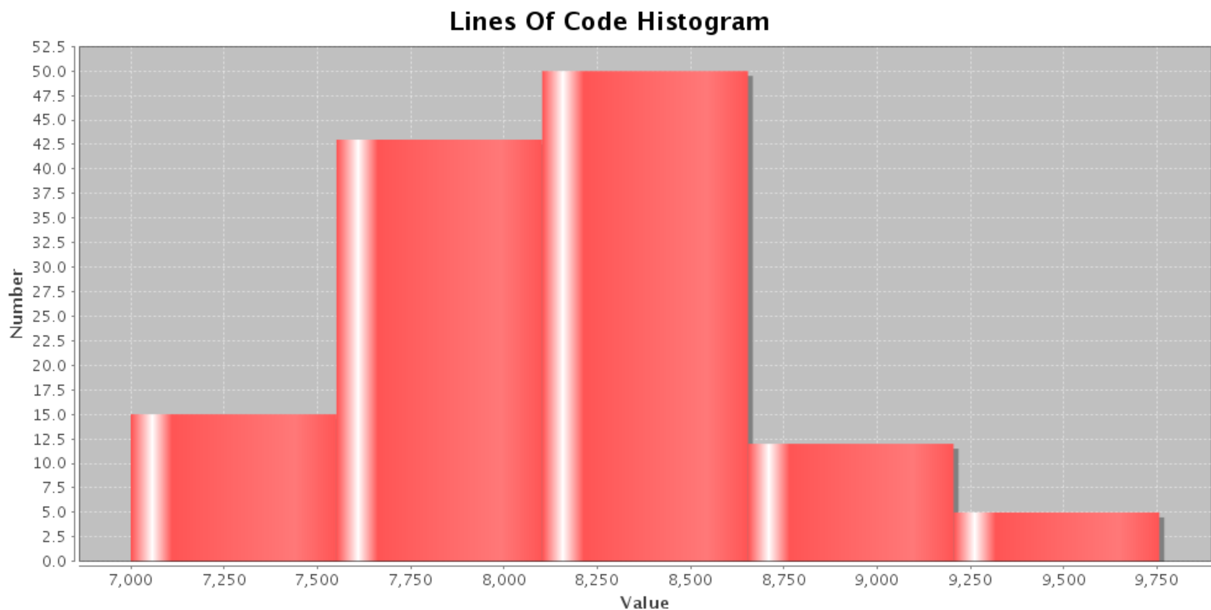

Figura 46. Histograma da métrica *Lines Of Code*.

Por último estuda-se o comportamento da métrica *Total Cyclomatic Complexity*. Observa-se que essa métrica possui um aumento de seus valores constante e pouco acentuado. Descobre-se que a décima revisão possui complexidade total de 1.184 e que a décima segunda revisão de 1.173, uma diminuição de 11 no valor da métrica. A décima quinta revisão possui 1.173 e a décima sexta revisão possui 1.221. Na vigésima revisão o projeto possui complexidade ciclomática total de 1.245, já na vigésima primeira revisão possui complexidade de 1.203. A Figura 47 exibe o gráfico desta métrica.

Observando-se o histograma de *Total Cyclomatic Complexity*, verifica-se que essa métrica possui uma faixa de valores com grande quantidade de revisões. Essa faixa está entre os valores 1248 e 1331. As outras faixas possuem valores bastante parecidos. Tal histograma pode ser observado na Figura 48.

Comparando as três métricas, alguns fatos interessantes podem ser observados. O primeiro deles é entre a décima e décima segunda revisão. Neste trecho há um aumento de 254 métodos, 6 linhas de código e diminuição de 11 de complexidade ciclomática total. Com uma diferença tão pequena no número de linhas de código, pode-se inferir que o que houve na verdade foi uma refatoração, que pretendia separar funcionalidades em mais métodos. Neste caso, a complexidade ciclomática total não deveria aumentar muito, já que não houve adição de funcionalidades, apenas o espalhamento destas funcionalidades em mais métodos. O que se

observou foi que, na verdade, houve uma diminuição da complexidade ciclomática total. Ou seja, separar funcionalidades em mais métodos pode simplificar a complexidade dos algoritmos que as implementam.

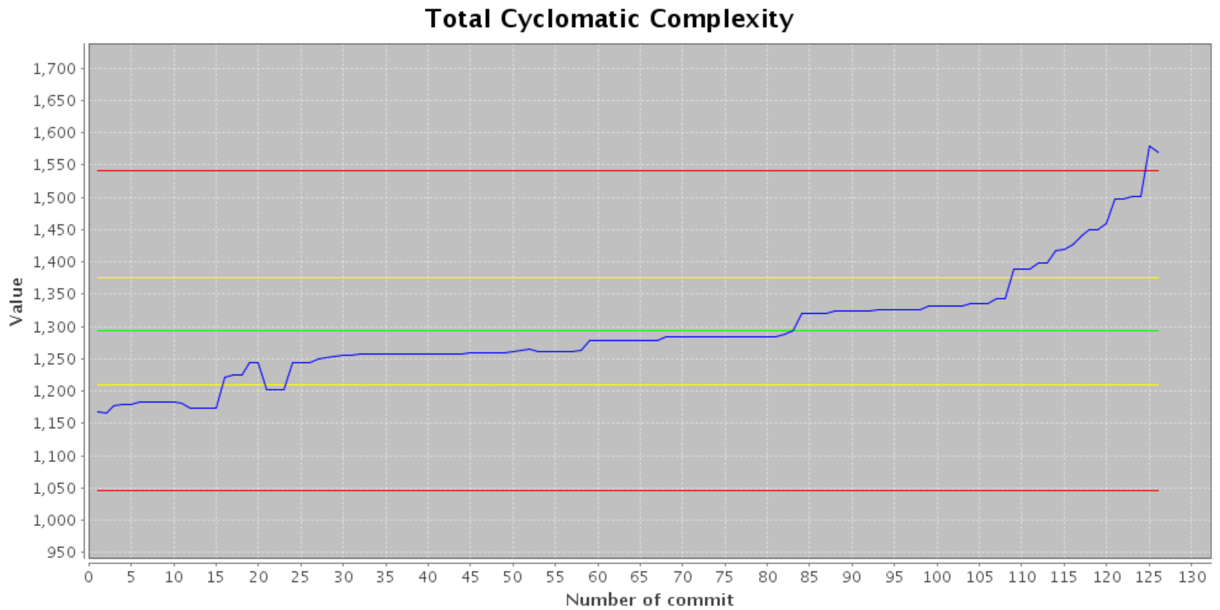

Figura 47. Gráfico do valor absoluto da métrica *Total Cyclomatic Complexity*.

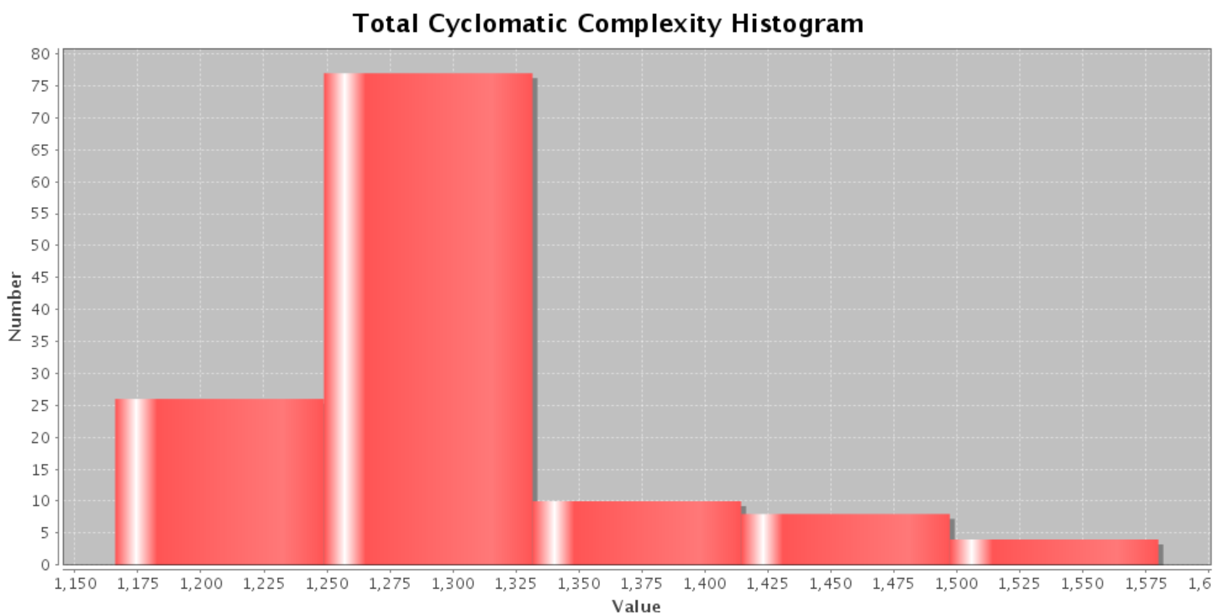

Figura 48. Histograma da métrica Total *Cyclomatic Complexity*.

Entre a décima quinta e décima sexta revisão há um aumento de 99 métodos, 484 linhas e 48 na complexidade ciclomática. Esse trecho chama atenção inicialmente pelo grande

aumento no número de métodos em relação aos outros trechos do gráfico. Entretanto, esse trecho se comporta dentro da normalidade de um período de criação de funcionalidades, onde o número de métodos aumenta, o número de linhas aumenta e a complexidade total aumenta. Pode ocorrer que neste período houve uma boa definição de como fazer o projeto funcionar e por isso houve um intenso fluxo de trabalho, que depois normaliza-se.

Ainda é possível observar que entre a vigésima e vigésima primeira revisão há uma diminuição de 39 métodos, 300 linhas de código e 42 de complexidade ciclomática. Talvez nesse caso tenha havido a retirada de alguma funcionalidade ou substituição por uma mais simples. Existem casos onde ao final da implementação de um grande conjunto de funcionalidades num fluxo intenso de trabalho, observa-se que algumas funcionalidades não são necessárias, ou que se pode simplificar o sistema seguindo outras abordagens, como substituição de funcionalidades por outras mais simples que utilizem menos métodos, menos linhas de código e sejam menos complexas.

Após estes trechos, os gráficos tendem a ter um comportamento mais estável, o que pode significar que o sistema encontra-se em um ritmo estável de desenvolvimento. Entretanto, pode-se observar no gráfico da métrica Total *Cyclomatic Complexity* que entre a penúltima (revisão 21511) e última (revisão 21633) revisão existe um decréscimo de valor. Na revisão 21511 o valor da métrica é 1.580 e na revisão 21633 o valor é 1.570, uma diminuição de 10.

Observando o gráfico de *Number of Metohods*, a penúltima revisão possui o valor de 1.061 e a última revisão possui o valor de 1.062, ou seja, houve o aumento de 1 método. Também há um decréscimo de 6 linhas de código, sendo 9.756 linhas na penúltima revisão e 9.750 linhas na última revisão. Um aumento do número de métodos seguido pelo não aumento do número de linhas pode ser um forte indício de uma refatoração. Neste caso ainda houve um decréscimo do número de linhas, o que quer dizer que além de dividir a funcionalidade em mais métodos, cada método se tornou suficientemente pequeno para que a soma de suas linhas pudesse ser menor que a do método original. Consequentemente, essa diminuição de linhas pode gerar uma diminuição de complexidade e foi o que ocorreu. Pode ter ocorrido que neste momento o programador tivesse tido o único objetivo nesta revisão: tornar 1 método mais simples dividindo-o em 2. Isto pôde acarretar, no final das contas, uma diminuição do número de linhas totais. No final, o programador conseguiu seu objetivo, que era tornar o sistema mais simples, já que conseguiu diminuir a complexidade ciclomática total.

6.3 CONSIDERAÇÕES FINAIS

Neste capítulo, foram apresentados alguns resultados sobre o uso do sistema criado neste projeto de conclusão de curso. Com a extração de três métricas foi possível vislumbrar algumas implicações dos seus valores. Por não conhecer informações adicionais sobre este projeto, como, por exemplo, o número de participantes, o seu histórico e as diversas situações ocorridas nele, talvez não se possa encontrar o real motivo de muitas características apresentadas nestas avaliações. Espera-se que esse sistema possa ser utilizado em projetos cujo usuário tenha contato. Dessa maneira, espera-se que as diversas características encontradas nos projetos, com o auxílio desta ferramenta, possam encontrar explicações e que essas mesmas características possam auxiliar o usuário a melhorar seus próprios projetos.

7 CONCLUSÃO

Este projeto reúne uma gama de métricas que podem ser utilizadas tanto no contexto deste projeto, quanto adaptadas para serem utilizadas em outros sistemas. Muitas métricas estavam espalhadas em diversos projetos de software livre e muitas outras ainda não tinham se quer sido implementadas. Além disso, cada projeto seguia uma metodologia quanto à coleta das métricas. Em alguns, passava-se como parâmetro uma classe e em outros o caminho da raiz do projeto. Esse projeto unificou a extração das métricas, fazendo com que, dessa maneira, a coleta das métricas pudesse ser totalmente automatizada. Uma vez possuindo um projeto escrito em Java [8], gerenciado pelo Maven [14] e hospedado em um repositório Subversion [10], o sistema cuida de todo o resto e extrai as métricas de uma forma transparente ao usuário.

Outra contribuição foi a de permitir a criação de métricas através de outras métricas. Esse ponto tem significante importância principalmente para usuários que desejam testar e estudar seus próprios projetos em profundidade. Dentro desse contexto, alguns usuários podem acreditar que certas métricas não são devidamente calibradas para seus próprios projetos. Alguns outros podem crer que uma determinada métrica faz sentido apenas se multiplicada por uma determinada constante ou somada a alguma variável. Por fim, um usuário pode desejar analisar a correlação entre duas métricas, criando uma métrica composta para tal fim. Nesse sentido, é ampla a liberdade que as métricas compostas dão aos seus usuários.

Espera-se que as vinte e duas métricas originais deste projeto se transformem em muito mais, no decorrer da utilização do sistema, de forma que os próprios usuários estudem e validem suas próprias métricas. A exploração do poder desse software pode gerar, cada vez mais, a compreensão do que realmente conseguem indicar as métricas de software. Quanto mais for utilizado, melhor será para o estudo e validação das diversas métricas existentes, ou até mesmo para a criação de novas métricas.

Por último, foram criados os gráficos que fornecem uma forma de interação com os valores coletados pelas métricas. Cada projeto pode possuir centenas ou milhares de revisões. Um numero tão grande de revisões gera um número enorme de dados, cujo estudo poderia ser extremamente demorado e contraproducente. Uma forma de resolver o problema da grande quantidade de dados constituiu em criar um gráfico de controle que pudesse mostrar, em uma figura, a variação dos valores das métricas em relação ao tempo percorrido. Esse gráfico mostra os valores das métricas como um todo e, caso haja dúvidas sobre alguma determinada revisão, ele pode buscar informações complementares sobre essa revisão. Dessa forma, o usuário pode se concentrar em pontos que realmente chamam a atenção no decorrer do projeto ou descobrir, analisando o gráfico, se o seu projeto está seguindo como o esperado.

Outra informação que se supôs necessária ao usuário foi em relação à distribuição dos valores das métricas. Para tal, foi criado o histograma (cuja função é mostrar quantas revisões estão em determinada faixa de valor). A principal motivação nesse caso é mostrar ao usuário se os valores das métricas são constantes em determinados valores, ou se os valores das métricas estão mais espalhados, podendo ocorrer frequências bastante parecidas em diversos valores. Algumas métricas podem ter seus valores espalhados, tendo uma grande variação, sem que isso signifique necessariamente uma situação anômala. Outras métricas podem ter sempre seus valores concentrados em alguns pontos. Cada métrica tem um comportamento e o histograma é, portanto, uma ferramenta de grande utilidade ao usuário, tanto para o estudo de como os valores das métricas se comportam como também para indicar alguma situação preocupante no software.

Este projeto foi criado com o intuito de oferecer uma ferramenta genérica para os que desejam estudar o comportamento do desenvolvimento do seu software por meio de métricas. Entretanto, a extração de métricas é altamente dependente da linguagem de programação utilizada no projeto. Optou-se então por extrair métricas apenas de projetos na linguagem Java. A escolha do Java se deu por ser uma das principais linguagens utilizadas nas empresas e universidades atualmente e por ter um grande ecossistema de bibliotecas, o que facilita a extensão do projeto.

Outro ponto a ser considerado é que varias métricas são extraídas de artefatos compilados. Portanto, é de suma importância que os projetos sejam compilados. Como não há padrões a serem seguidos para se compilar um programa em Java (as bibliotecas externas podem estar em locais não padronizados), ficaria muito difícil compilar um projeto genérico para se obter suas métricas. Então optou-se por aceitar somente projetos que utilizem o Maven e que, portanto, podem ser compilados automaticamente. Projetos Maven têm a vantagem de

manter os arquivos fontes, arquivos compilados e as bibliotecas externas localizados em diretórios padronizados.

Por ultimo, em função dos projetos estarem armazenados *online*, o processo de *download* das revisões do projeto, as informações relativas a cada projeto e a localização dos seus arquivos em um repositório devem ser padronizados. Portanto, optou-se por tratar somente projetos armazenados em repositórios gerenciados pelo sistema de controle de versão Subversion.

Essas são as três principais limitações, sendo que as duas últimas são limitações pela escolha do padrão a ser utilizado e a primeira é uma limitação intrínseca à própria natureza da coleta de métricas, que depende quase que totalmente da própria linguagem na qual o projeto foi escrito.

Por fim, melhorias que podem ser realizadas sobre este projeto são o aumento de tipos de sistema de controle de versão suportados (Git [31] e Mercurial [32], por exemplo) e de sistema de construção de software (Ant [33], Make [34] e Rake [35], por exemplo). Essas extensões aumentariam o número de projetos que poderiam utilizar o sistema proposto neste trabalho.

Outra melhoria consistiria em dar ao usuário o poder de limitar o número de revisões mostradas nos gráficos de controle. A quantidade de informação mostrada ao usuário pode ser grande mesmo para ser mostrado em um gráfico. Ele poderia não estar interessado em revisões anteriores a alguma data específica, querendo se concentrar apenas no sistema a partir de uma revisão determinada. Dessa maneira, o sistema mostraria ao usuário apenas o momento do projeto interessante a ele.

Outra melhoria pode ser a criação de novos tipos de gráficos que auxiliem de outras formas o usuário. Um estudo para saber qual tipo de gráfico seria de melhor ajuda aos usuários poderia ser realizado.

Uma dúvida que pode surgir é em que momento de desenvolvimento estava o projeto em determinada revisão. Poderia então criar-se um sistema de marcações, o qual serviria para indicar a qual fase do projeto cada revisão pertence. Adicionalmente este sistema de marcação poderia guardar outras informações. Cada usuário do sistema poderia, por exemplo, após realizar uma análise sobre o comportamento das métricas em um determinado projeto, registrar seu diagnóstico e considerações nesse sistema. Dessa forma todas as análises realizadas poderiam ser consultadas posteriormente por outros usuários.

Por final poderia se expandir a forma de medição das métricas em outro sentido. Neste projeto as métricas são coletadas através do código fonte armazenado em repositórios.

Poderia-se estender esse sistema de coleta de forma que as métricas pudessem ser extraídas a partir de um projeto em execução.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] R. Pressman, *Engenharia de software*, 6º ed. São Paulo: McGraw-Hill, 2006.

[2] T. Mikkelsen, *Practical software configuration management : the latenight developer's handbook*. Upper Saddle River NJ: Prentice Hall PTR, 1997.

[3] "Oceano". [Online]. Available: http://gems.ic.uff.br/trac/oceano. [Accessed: 21-set-2011].

[4] "IdUFF - Sistema de identificação única da Universidade Federal Fluminense". [Online]. Available: https://sistemas.uff.br/iduff/sid137avUfd98/. [Accessed: 15-nov-2011].

[5] Jagdish Bansiya e Carl G. Davis, "A Hierarchical Model for Object-Oriented Design Quality Assessment", 2002.

[6] Jim A. McCall, Paul K. Richards, e Gene F. Walters, "Factors in Software Quality", 1977.

[7] ISO 9126, "ISO/IEC TR 9126-Software engineering, product quality, quality model", International Organization for Standardization, 2001.

[8] J. Gosling, B. Joy, G. L. Steele, e G. Bracha, *The Java Language Specification*, 3º ed. Addison-Wesley, 2005.

[9] "JavaServer Faces Technology". [Online]. Available: http://www.oracle.com/technetwork/java/javaee/javaserverfaces-139869.html. [Accessed: 15-nov-2011].

[10] B. Collins-Sussman, *Version control with Subversion*, 1º ed. Sebastopol CA: O'Reilly Media, 2004.

[11] "SVNKit :: Subversion for Java". [Online]. Available: http://svnkit.com/. [Accessed: 03-out-2011].

[12] "The Java Persistence API - A Simpler Programming Model for Entity Persistence". [Online]. Available: http://www.oracle.com/technetwork/articles/javaee/jpa-137156.html. [Accessed: 24-nov-2011].

[13] "Hibernate - JBoss Community". [Online]. Available: http://www.hibernate.org/. [Accessed: 03-out-2011].

[14] J. Van Zyl, *Maven: Definitive Guide*, First. [[O'Reilly Media]], 2008.

[15] E. Gamma, *Padrões de projeto : soluções reutilizáveis de software orientado a objetos*. Porto Alegre: Bookman, 1995.

[16] J. Han, Data mining : concepts and techniques, 3o ed. Burlington MA: Elsevier, 2011.

[17] "Software Metrics". [Online]. Available: http://agile.csc.ncsu.edu/SEMaterials/tutorials/metrics/. [Accessed: 08-set-2011].

[18] H. Fujita, *New trends in software methodologies, tools and techniques*. Amsterdam ;;Oxford :: IOS Press,, 2007.

[19] “Metric Descriptions”. [Online]. Available: http://gromit.iiar.pwr.wroc.pl/p_inf/ckjm/metric.html. [Accessed: 08-set-2011].
[20] “JDepend”. [Online]. Available: http://clarkware.com/software/JDepend.html. [Accessed: 08-set-2011].
[21] “Dependency Finder”. [Online]. Available: http://depfind.sourceforge.net/. [Accessed: 08-set-2011].
[22] “Apache Commons BCEL™ -”. [Online]. Available: http://commons.apache.org/bcel/. [Accessed: 08-set-2011].
[23] “ckjm — Chidamber and Kemerer Java Metrics”. [Online]. Available: http://www.spinellis.gr/sw/ckjm/. [Accessed: 08-set-2011].
[24] “JavaNCSS - A Source Measurement Suite for Java”. [Online]. Available: http://javancss.codehaus.org/. [Accessed: 08-set-2011].
[25] “LOCC — Collaborative Software Development Laboratory”. [Online]. Available: http://csdl.ics.hawaii.edu/Plone/research/locc/. [Accessed: 08-set-2011].
[26] M. Sipser, *Introdução à teoria da computação*. São Paulo: Thomson Learning, 2007.
[27] “W3C XHTML2 Working Group Home Page”. [Online]. Available: http://www.w3.org/MarkUp/. [Accessed: 15-nov-2011].
[28] “Maven GWT Plugin - Introduction”. [Online]. Available: http://mojo.codehaus.org/gwt-maven-plugin/. [Accessed: 06-dez-2011].
[29] “JFreeChart”. [Online]. Available: http://www.jfree.org/jfreechart/. [Accessed: 08-set-2011].
[30] “PNG (Portable Network Graphics) Home Site”. [Online]. Available: http://www.libpng.org/pub/png/. [Accessed: 15-nov-2011].
[31] “Git - Fast Version Control System”. [Online]. Available: http://git-scm.com/. [Accessed: 09-dez-2011].
[32] “Mercurial SCM”. [Online]. Available: http://mercurial.selenic.com/. [Accessed: 09-dez-2011].
[33] “Apache Ant - Welcome”. [Online]. Available: http://ant.apache.org/. [Accessed: 15-nov-2011].
[34] “GNU Make Manual - GNU Project - Free Software Foundation (FSF)”. [Online]. Available: http://www.gnu.org/software/make/manual/. [Accessed: 09-dez-2011].
[35] “Rake -- Ruby Make”. [Online]. Available: http://rake.rubyforge.org/. [Accessed: 09-dez-2011].