UNIVERSIDADE FEDERAL FLUMINENSE
CURSO DE CIÊNCIA DA COMPUTAÇÃO

Marcus José Pereira Loureiro e Rafael Fernandes Muniz

Ferramenta Gráfica para Gerar Organogramas e Grafos Dinamicamente em aplicações Java para Web

NITERÓI

2011

Marcus José Pereira Loureiro e Rafael Fernandes Muniz

FERRAMENTA GRÁFICA PARA GERAR ORGANOGRAMAS E GRAFOS DINAMICAMENTE EM APLICAÇÕES JAVA PARA WEB

Monografia apresentada ao Departamento de Ciência da Computação da Universidade Federal Fluminense como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Orientador: José Raphael Bokehi

NITERÓI

2011

Marcus José Pereira Loureiro e Rafael Fernandes Muniz

FERRAMENTA GRÁFICA PARA GERAR ORGANOGRAMAS E GRAFOS DINAMICAMENTE PARA JAVA WEB

Monografia apresentada ao Departamento de Ciência da Computação da Universidade Federal Fluminense como parte dos requisitos para obtenção do grau de Bacharel em Ciência da Computação.

Aprovado em Agosto de 2011

BANCA EXAMINADORA

José Raphael Bokehi

Orientador

UFF

Cristina Nader Vasconcelos

UFF

Leonardo Cruz da Costa

UFF

NITERÓI

2011

RESUMO

Atualmente, o aluno da UFF apenas consegue visualizar sua grade curricular através de imagens estáticas. A carência de uma ferramenta disponível que auxilie o universitário nesta tarefa motivou a realização deste projeto, que tem como objetivo a criação de uma ferramenta para web que crie dinamicamente este fluxograma. Do ingresso ao término da faculdade, o aluno poderá acompanhar seu rendimento no curso através de uma imagem com as disciplinas cursadas e a cursar. Com o *JgraphX*, uma biblioteca gráfica para *Java-Swing*, foi modelado um sistema Web que com o acesso a um banco de dados como o IdUFF, permite que o aluno tenha à sua disposição a situação acadêmica atualizada da sua grade curricular, de uma forma dinâmica e de acordo com seu curso e período. Não através de tabelas, mas sim por formas gráficas, gerando seu fluxograma completo, com cada matéria relacionada aos seus pré-requisitos. É capaz de promover ainda a análise de todas as matérias já concluídas, inscritas e ainda não realizadas. A criação desta ferramenta facilitará o aluno em várias ações durante a sua vida universitária, ajudando-o a fazer um planejamento mais eficiente nas suas inscrições em disciplinas que já são feitas via Internet por grande parte dos cursos da Universidade Federal Fluminense.

Palavras-chave: fluxograma, currículo, gráfico, serviço, internet.

ABSTRACT

Nowadays, the UFF`s student can only see his curricular grid trough statics images. The need for a tool that helps the student in this task is what motivated this project, that proposes the creation of a web tool which creates a dynamically flowchart. From the beginning to the end of college, students can monitor their performance through an image with the subjects taken and attend to take. With a graphics library for Java-Swing, JgraphX, a system was modeled with Web access to a database, as the IdUFF. This system allows the student to have, in a dynamic way, the status of his current academic curriculum according to his course and period. Not through tables, but through graphic shapes, creating a complete flowchart, with every subject related to the prerequisites. It can also promote the analysis of all the subjects already completed, and the ones matriculated but not done yet. The creation of this tool will facilitate the university life in a lot of ways, like helping to plan more efficient the application in disciplines, which are already made by Internet for most of courses of the Universidade Federal Fluminense.

Tags: flowchart, curriculum, graphic, service, internet.

LISTA DE FIGURAS

Figura 1 - Organograma gerado pelo JSF/AJAX Facelets[01]. 11
Figura 2 - Estrutura do JavaServer Faces. 12
Figura 3 – Componente calendar do Rich-Faces[03]. 15
Figura 4 – Componente suggestionbox do Rich-Faces [03]. 15
Figura 5 - “Hello World” em Graphiz[04]. 17
Figura 6 - PSV em graphiz[04]. 17
Figura 7 – Padrões de Projeto MVC [05]. 18
Figura 8 - Ambiente de desenvolvimento Java típico [06]. 21
Figura 9 - Representaçao gráfica utilizando o Jgrah e API Swing. 23
Figura 11 – Representação de organograma com *Jgraph*. 24
Figura 12 - Representação de Workflow com Jgraph. 24
Figura 13 - Exemplo de diagrama de banco de dados usando *Jgraph*. 25
Figura 14 – Exemplo de *Vertex* e *Edge* [10]. 27
Figura 15 - Exemplo usando método *getDefaultVertexStyle*. 28
Figura 16 - Exemplo usando método *getDefaultEdgeStyle*. 28
Figura 17 - Exemplo usando algumas chamadas de *MxConstants*. 29
Figura 18 - exemplo usando *mxHierarchicalLayout* e *mxOrthogonalLayout*. 30
Figura 19 - Leitura de 2.000.000 por índice [11]. 31
Figura 20 - Inserção de 350.768 linhas [11]. 31
Figura 21 - Um típico sistema cliente/servidor [12]. 32
Figura 22 - A solução *lock-modify-unlock* [12]. 33
Figura 23 - A solução *copy-modify-merge* [12]. 34
Figura 24 - A solução *copy-modify-merge* 2 [12]. 34
Figura 25 - Arquitetura básica do MOR com Hibernate [14]. 36
Figura 26 - Classe Aluno[15]. 37
Figura 27 - Representação da tabela Aluno em um arquivo XML [15]. 37
Figura 28 - Representação da tabela Aluno com Annotations [15]. 38
Figura 29 - consulta em HQL [16]. 39
Figura 30 - Camada persistente [17]. 41
Figura 31 - AWT x Swing [20]. 43
Figura 32 - Diagrama de classe ref proprio projeto 44
Figura 33 - Exemplo da classe Matéria usando Hibernate. 45
Figura 34 – Exemplo de *NamedQuerys* da classe Materia. 46
Figura 35 - Tela de cadastro de curso usando JSF. 48
Figura 36 – Exemplo de PDF gerado pelo JgraphX. 50
Figura 37 – *Applet* usando o framework *JgraphX*. 51
Figura 38 - *Applet* exibindo imagem gerada pelo *JgraphX*. 52

Figura 39 - Exemplo de *view* fazendo chamada de *applet*. 53
Figura 40 - Exemplo de *view* fazendo chamada de *applet* com passagem de parâmetro. 53
Figura 41 - Exemplo de aplicação *applet* recebendo parâmetros. 54
Figura 42 - Configurando rich-faces. 55
Figura 43 - Principais métodos da aplicação que se comunica com a *View*. 56
Figura 44 - *View* usando o componente *paint2D* e se comunicando com controlador GraficoMB. 57
Figura 45 - Fluxograma parcialmente completo gerado pelo *JgraphX*. 57
Figura 46 - Tela para cadastro de curso. 58
Figura 47 - Tela para administrar disciplinas de um curso. 59
Figura 48 - Tela para visualizar graficamente o fluxograma de um curso. 59
Figura 49 - Tela para administrar alunos fictícios. 60
Figura 50 - Tela para controlar as matérias do aluno. 60
Figura 51 - Tela para exibir fluxograma de um aluno específico. 61
Figura 52 - Fluxograma usando estilo ortogonal para cada seta 'Edge'. 61
Figura 53 - Fluxograma usando estilo mais suave para cada seta (*Edge*). 62
Figura 54 - Desenhando o fluxograma sem nenhum estilo para as Edge's. 63
Figura 55 - Desenhando as ligações a partir de pontos específicos. 64
Figura 56 - Desenhando os Vertex's usando opacidade. 64
Figura 57 - Exemplo do fluxograma completo de um aluno. 65

SUMÁRIO

RESUMO **4**

ABSTRACT **5**

1 INTRODUÇÃO **9**

1.1 OBJETIVO GERAL 9
1.2 OBJETIVO ESPECÍFICO 9
1.3 ESTRUTURA DO TRABALHO 10

2 FUNDAMENTAÇÃO TEÓRICA **11**

2.1 FERRAMENTAS GRÁFICAS 11
2.1.1 Java Server Faces/AJAX Facelets *11*
2.1.2 Applets *13*
2.1.3 RichFaces *14*
2.1.4 Graphviz - Graph Visualization *16*
2.2 PADRÕES DE PROJETO – MVC 18
2.3 A LINGUAGEM JAVA 20
2.4 O FRAMEWORK JGRAPH 22
2.4.1 Introdução *22*
2.4.2 Exemplos e aplicações *23*
2.4.3 JgraphX *25*
2.5 DATABASE MYSQL 30
2.6 SUBVERSION 32
2.7 HIBERNATE 35
2.7.1 Mapeamento Objeto Relacional *36*
2.7.2 HQL *38*
2.8 JPA 40
2.9 JAVA SWING E AWT 42

3 IMPLEMENTAÇÃO **44**

3.1 ESTRUTURA DO PROJETO 44
3.1.1 Padrão MVC *44*
3.1.2 Diagrama de Classe *44*
3.1.3 Mapeamento do Projeto *45*
3.1.4 Representação dos dados *46*
3.2 APLICAÇÃO DO JAVASERVER FACES 48
3.3 EXPORTANDO GRAFICOS DO JGRAPHX 49
3.3.1 Exportando para um objeto BufferedImage *49*
3.3.2 Exportando para um PDF *50*
3.4 USANDO O APPLET 51
3.4.1 Configurando o Applet *51*
3.4.2 Parâmetros via Applet *53*
3.5 USANDO O RICH FACES 55
3.5.1 Configurando o RICH FACES *55*
3.5.2 O Componente PAINT2D *56*

4 RESULTADOS **58**

4.1 SIMULADOR 58
4.2 PROBLEMAS ENCONTRADOS 61
4.3 SOLUÇOES E RESULTADOS FINAIS 63

5 CONCLUSÕES **66**

5.1 PERSPECTIVAS FUTURAS 66

6 REFERÊNCIAS BIBLIOGRÁFICAS **67**

6.1 OBRAS CONSULTADAS 69

1 INTRODUÇÃO

É importante que ao longo de sua vida universitária o aluno consiga acompanhar seu desempenho acadêmico através das grades curriculares disponibilizadas pelas coordenações dos cursos, para assim poder planejar a melhor forma de alocar as matérias que faltam ser cursadas nos próximos períodos de seu curso.

Após pesquisas por *frameworks* capazes de desenhar um organograma para um sistema via *web*, o *JGraph* foi eleito como o principal framework que atendesse as necessidades e pudesse ajudar para atingir a conclusão do projeto, porém até o presente momento não existe um caso de sucesso na tentativa de reproduzir um fluxograma dinamicamente a partir de uma base de dados, apenas a exibição estática do fluxograma.

Neste projeto é apresentada como solução o uso do *Java swing* com o framework *JGraph* para resolver o problema da geração do fluxograma dinâmico. Esta solução permite a exibição de maneira satisfatória de uma prévia de um modelo simples de um fluxograma universitário.

Com a preocupação de auxiliar o aluno e também a universidade na visualização do estado atual da grade curricular do aluno, este trabalho apresenta como protótipo uma nova ferramenta construída com base em estudos e pesquisas de tecnologias e padrões existentes que gere graficamente, via *web*, um fluxograma contendo as informações de disciplinas cursadas e a cursar, indicando os pré-requisitos de cada uma delas.

1.1 OBJETIVO GERAL

A pesquisa, estudo e aprendizagem de uso de bibliotecas para desenhos gráficos analisando qual seriam capazes de atender a necessidade de criar um desenho em forma de fluxograma, considerando questões de customização e possibilidade de alterar parte do código caso fosse necessário, além de avaliar a ideia da construção de uma própria biblioteca formam parte do objetivo deste projeto.

1.2 OBJETIVO ESPECÍFICO

O protótipo desenvolvido no projeto será capaz de simular a geração do fluxograma de

um curso da graduação e também o histórico de um aluno em um sistema Web como o IdUFF(atual o sistema acadêmico da graduação, responsável pelo módulo de inscrição online da Universidade Federal Fluminense) a partir de uma base de dados, possibilitando a criação de um curso, suas matérias e alunos fictícios. Além disso, também deverá simular o histórico de um aluno, selecionando as matérias concluídas ou não.

1.3 ESTRUTURA DO TRABALHO

Este trabalho está organizado da seguinte forma. No capitulo 2 é apresentada todas as ferramentas utilizadas no projeto com uma explicação sobre cada uma delas. No capitulo 3 é mostrado toda metodologia utilizada na construção do protótipo. Os resultados obtidos são apresentados e discutidos no Capítulo 4. Por fim, no Capítulo 5, encontram-se as conclusões.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capitulo serão apresentadas as principais ferramentas pesquisadas e utilizadas para o desenvolvimento deste projeto.

2.1 FERRAMENTAS GRÁFICAS

Durante o estudo do projeto foi necessário o uso de um *framework* que funcionasse via *web* e fosse capaz desenhar graficamente organogramas, a seguir apresentaremos os frameworks estudados.

2.1.1 Java Server Faces/AJAX Facelets

O *Facelets* é um subprojeto do *Java Server Faces* (JSF) mantido pela *Sun*, que é facilmente integrado ao próprio JSF.

O JSF/AJAX *Facelets*, desenvolvido pela empresa *Apprisant* Technologies é um framework capaz de desenhar dinamicamente gráficos utilizando *Facelets*. Ele é capaz de gerar diversos tipos de diagramas em JSF.

Essa ferramenta utiliza componentes capazes de gerar um organograma similar ao que seria a grade curricular do aluno, como demonstrado na figura 1, porém o framework é pago.

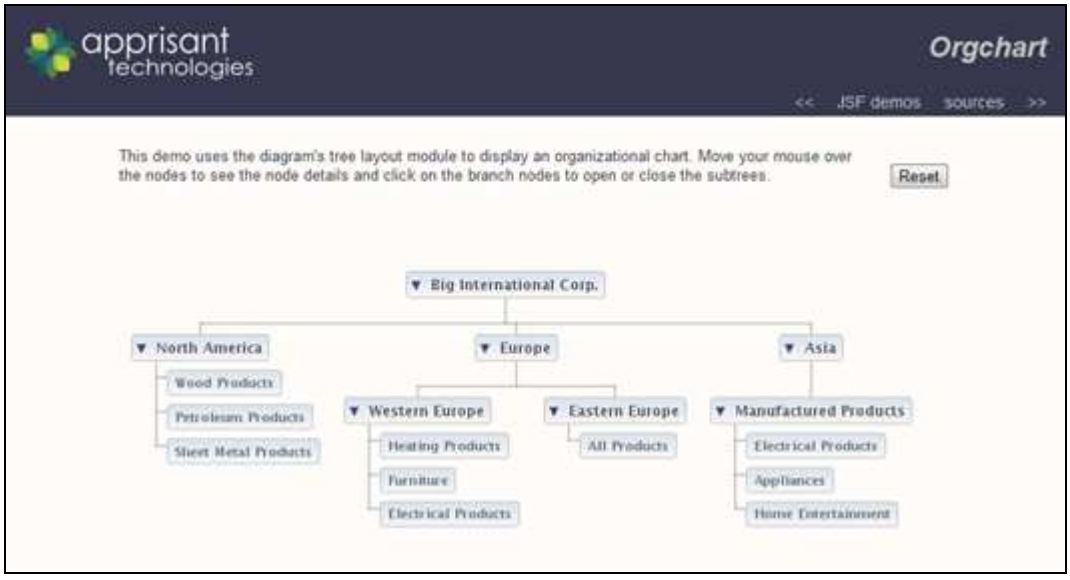

Figura 1 - Organograma gerado pelo JSF/AJAX Facelets[01].

O JSF incorpora características de um *framework* MVC (*model, view e controller*) de um modelo de interfaces gráficas baseado em eventos. Por se basear no padrão de projeto MVC, uma de suas melhores vantagens é a clara separação entre a visualização e regras de negócio.

Outras características que marcam o *Java Server Faces* são que permite que o desenvolvedor crie UIs (classes que representam os componentes de interface) através de um conjunto de componentes UIs predefinidos. Fornece um conjunto de *tags Java Server Page* (JSP) para acessar os componentes, e da reusabilidade de componentes da página, possibilitando criar *includes*, que são pequenos módulos que formam a *view*.

A figura 2 representa o diagrama de classes da estrutura do JSF.

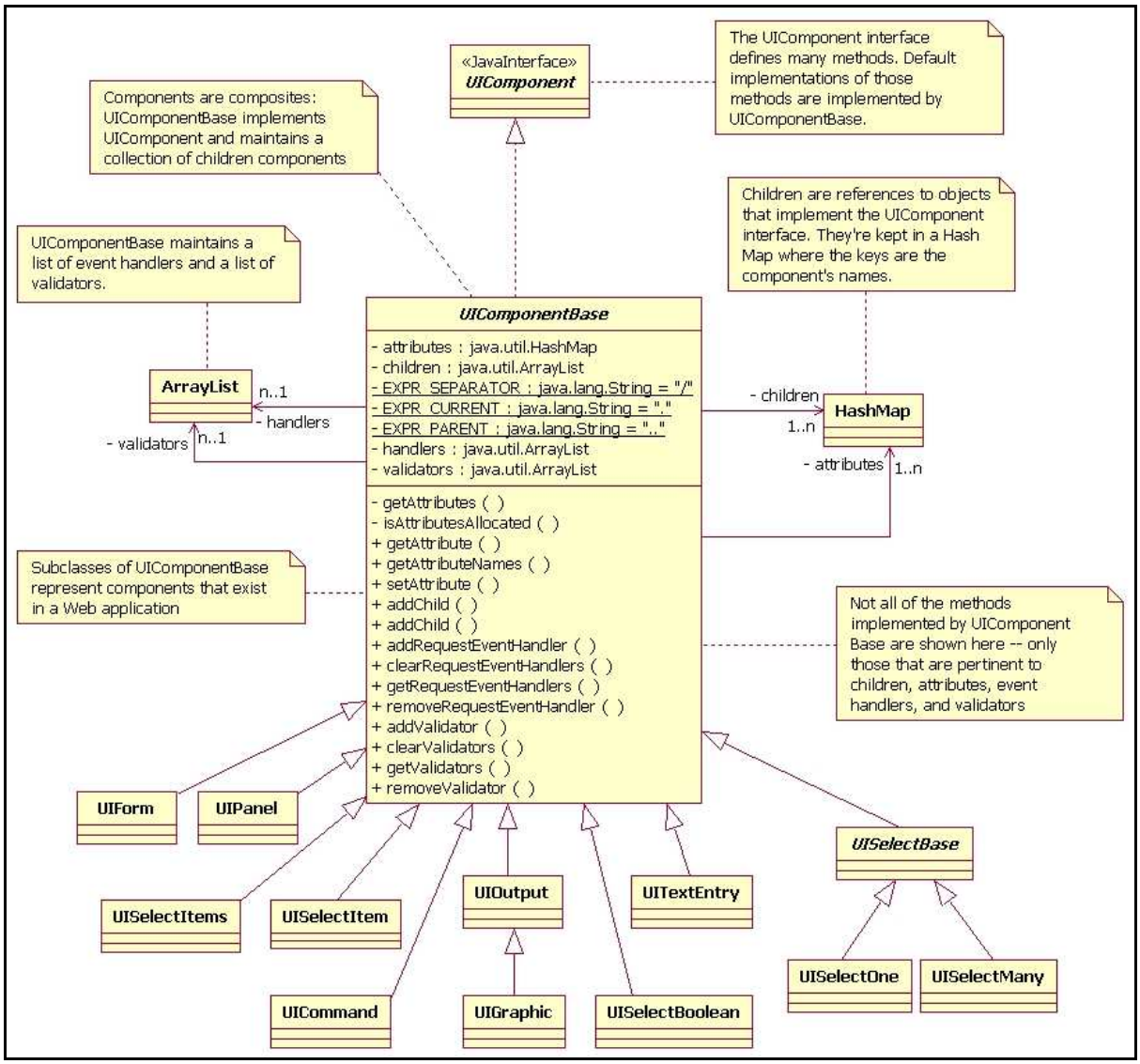

Figura 2 - Estrutura do JavaServer Faces.

2.1.2 Applets

Applets são programas em Java que podem ser incorporados em documentos *Hypertext Markup Language* (HTML), ou seja, páginas na Internet. Quando um navegador carrega uma página da Web contendo um *applet*, o *applet* é baixado no navegador e executado [2].

O Navegador que executa um *applet* é genericamente conhecido como contêiner de *applets*. O *Java Development Kit* (JDK) inclui o contêiner de *applets appletviewer* para testar *applets* à medida que eles são desenvolvidos e antes de ser incorporados a páginas da Web. Para execução de *applets*, é necessário a instalação do *J2SE RunTime Environment* (JRE), hoje em dia a maioria dos navegadores suportam a execução de *applets*.

Durante a execução de um *JApplet* são chamados métodos de seu ciclo de vida, esses métodos são chamados por um contêiner de *applets*, são eles *o init(), start(), paint(), stop()* ou *destroy().*

O comando *init*:

Chamado uma vez pelo contêiner de *applets* quando um *applet* é carregado para execução. Esse método inicializa um *applet*. Ações típicas realizadas aqui são inicializar campos, criar componentes GUI (*Gaphical User Interface*), carregar sons, carregar e exibir imagens e criar threads.

O comando *start*:

Chamado pelo container de *applets* depois de o método *init* completar a execução. Além disso, se um usuário navegar para outro site e retornar novamente para a página do *applet*, o método start é chamado novamente. O método realiza todas as tarefas que devem ser completadas quando o *applet* é carregado pela primeira vez, e isso deve ser realizado todas as vezes que a página HTML do *applet* é revisitada. As ações realizadas aqui incluiriam iniciar uma animação ou iniciar outra thread de execução.

O comando *paint*:

Chamado pelo contêiner de *applets* depois dos métodos *init* e start. O método *paint* também é chamado quando o *applet* precisa ser repintado. Por exemplo, se o usuário abrir o *applet* sob outra janela aberta na tela e mais tarde exibir o *applet*, o método *paint* será chamado. Ações típicas realizadas aqui envolvem desenhar com o objeto g de *Graphics* que é passado para o método *paint* pelo contêiner de *applets*.

O comando *stop*:

Chamado quando o usuário sai da página em que se encontra o *applet*, ou seja, vai para outra página da Web. Como é possível que o usuário retorne à página contendo esse *applet*, o método *stop* realiza as tarefas que seriam necessárias para suspender a execução do *applet*, assim *applet* não utiliza o tempo de processamento do computador quando não é exibido na tela. Ações típicas realizadas aqui interromperiam a execução de animações e threads.

O comando *destroy*:

O método *destroy* é chamado quando o *applet* é removido da memória. Isso ocorre quando o usuário encerra a sessão de navegação, fechando todas as janelas do navegador corrente e também pode ocorrer sem que o navegador saiba quando o usuário vai para outra página na Internet. O método realiza qualquer tarefa necessária para limpar recursos alocados pelo *applet*.

2.1.3 RichFaces

O *RichFaces* é uma biblioteca de componentes UI (classes que representam os componentes de interface) para aplicações web que utilizam o framework JSF. Os componentes desta biblioteca possuem um incrível suporte *Asynchronous Java script and XML* (AJAX), e ela, pode ser considerada uma extensão do Ajax4jsf com inúmeros componentes com Ajax “embutido” e com um suporte a *Skins* que podem deixar as interfaces da aplicação com um visual padronizado.

Essa biblioteca disponibilizava um componente chamado <rich:paint2D/>, que possibilita a criação e manipulação de imagens.

O *RichFaces* possui uma vasta coleção de componentes que ajudam a suavizar telas e melhorar a interação do sistema com o usuário.

<rich:calendar>: Com esse componente o usuário pode selecionar uma data com interface de uma forma bem prática (Figura 3).

Figura 3 – Componente calendar do Rich-Faces[03].

<rich:suggestionbox>: Com esse componente o usuário pode visualizar sugestões conforme for digitando o campo, o exemplo a seguir seria um campo para selecionar umas das capitais dos Estados Unidos (Figura 4).

Figura 4 – Componente suggestionbox do Rich-Faces [03].

<rich:datascroller>: *Tag DataScrooller* em conjunto com *DataTable* permite paginar vários registros em uma única página do sistema.

<rich:panelBar>: Esse componente permite o usuário manusear painéis independentes, a partir de *clicks* é possível visualizar todo seu conteúdo de uma lista por exemplo.

Existem diversas *Tags* de *RichFaces* que poderiam ser citadas inclusive componentes que possuem habilidade de *Drag and Drop* onde é possível clicar e arrastar um elemento na tela, mas para o desenvolvimento do fluxograma foi estudado o componente <rich:paint2D>

<rich:paint2D>: O componente Paint2D permite criar uma imagem gráfica usando os recursos graphic2D do JDK(*Java Development Kit*). O atributo '*paint'* deve apontar para um método do Controlador (*Bean*) que aceita parâmetros. O primeiro parâmetro tem um tipo de Graphic2D, ou seja, é uma tela gráfica que você pode gerar. O segundo parâmetro é o de dados personalizado que você pode passar para um método que desenhe usando um atributo de dados do componente paint2D.

2.1.4 Graphviz - Graph Visualization

Graphviz é um software open source de visualização gráfica iniciado pela *AT&T Research Labs* para desenho de gráficos especificados com vários tipos diferentes de *layouts*. Possui interface *web* e interativa com bibliotecas e ferramentas auxiliares.

Com as principais vantagens de ser uma ferramenta de fácil aprendizado e com grande numero de usuários. Além disso, permite criar um grafo simples com poucas linhas de código.

Por outro lado foi desenvolvido para linguagem *DOT*. Apesar de possuir biblioteca para Java (*Grappa*), mas que contem pouca documentação e poucas demonstrações de uso,

anulando as vantagens e tornando o aprendizado mais difícil.

Como exemplo é apresentado na figura 5 o resultado do seguinte trecho de código:

echo "digraph G {Hello->World}" | dot -Tpng >hello.png

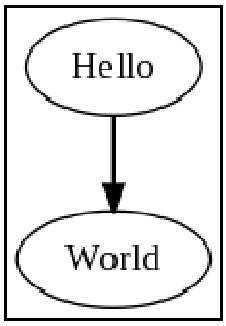

Figura 5 - "Hello World" em Graphiz[04].

Graphviz é capaz de desenhar gráficos complexos e mais próximos do que seria um organograma de uma grade curricular, como na figura 6.

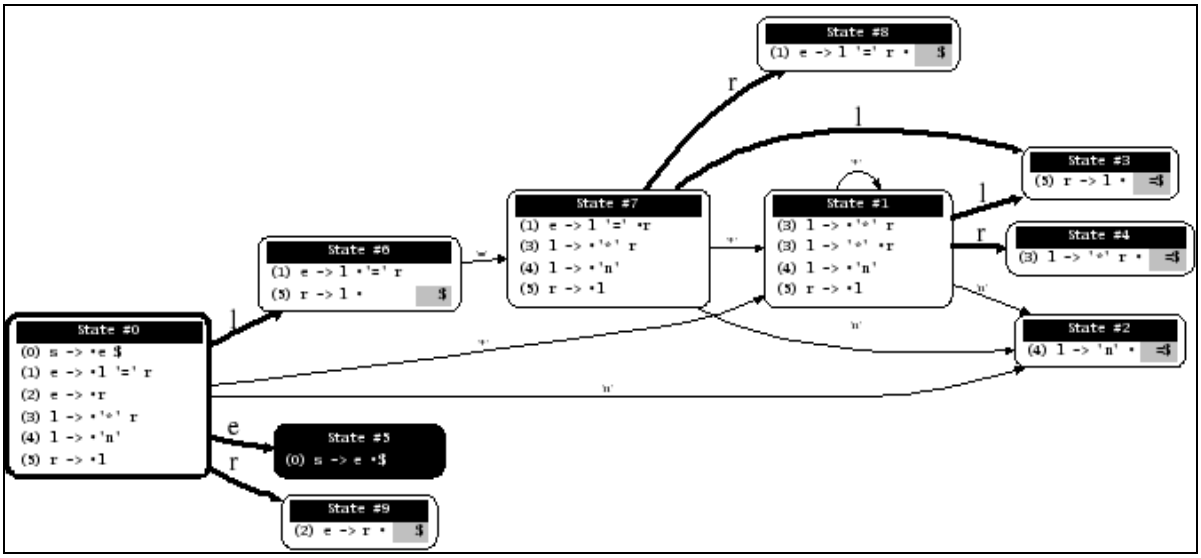

Figura 6 - PSV em graphiz[04].

2.2 PADRÕES DE PROJETO – MVC

A abordagem MVC é composta por três tipos de objetos, o Modelo é objeto de aplicação, a Visão é a apresentação na tela e o Controlador é o que define a maneira como a interface do usuário reage as entradas do mesmo. Antes da MVC, os projetos de interface para o usuário tendiam a agrupar esses objetos. A MVC separa esses objetos para aumentar a flexibilidade e a reutilização.

A abordagem MVC separa visão e modelos pelo estabelecimento de um protocolo do tipo inserção/notificação (*subscribe*/*notify*) entre eles, como mostra a figura 7. Uma visão deve garantir que a sua aparência reflita o estado do modelo. Sempre que os dados do modelo mudam, o modelo notifica as visões que dependem dele. Em resposta, cada visão tem a oportunidade de atualizar-se. Esta abordagem permite ligar múltiplas visões a um modelo para fornecer diferentes apresentações. Da mesma forma, você também pode criar novas visões para um modelo sem ter de reescrevê-lo.

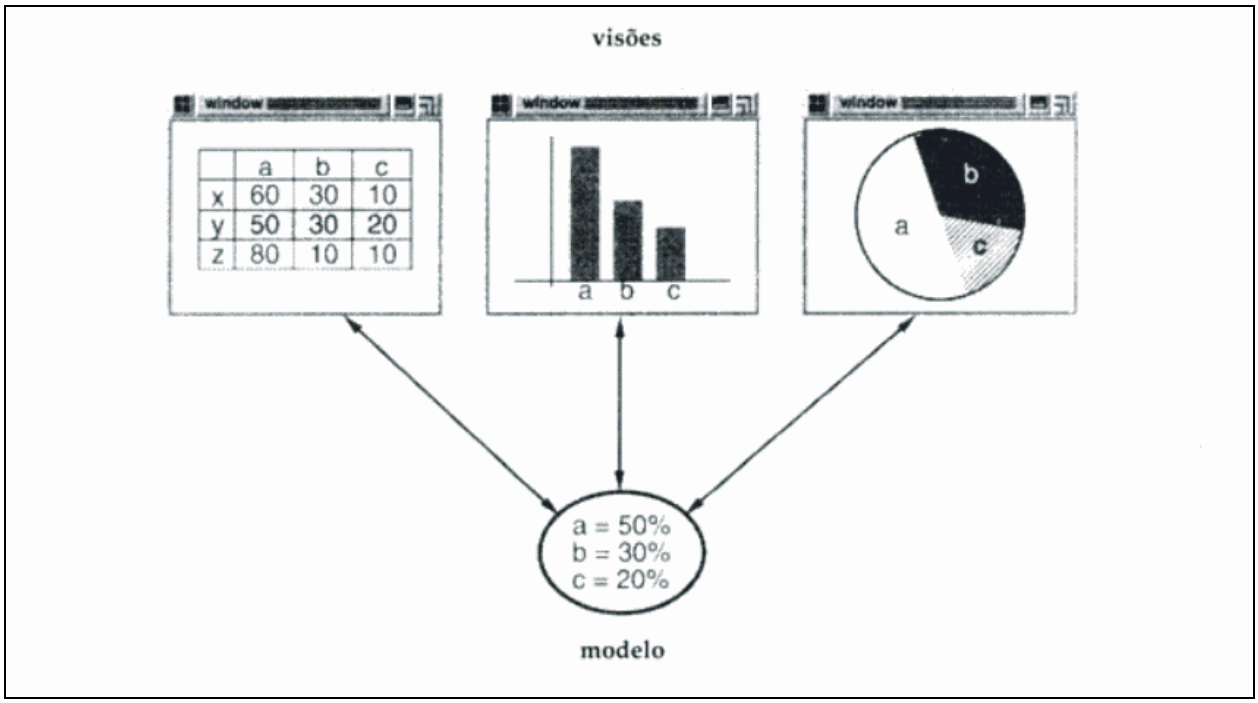

Figura 7 – Padrões de Projeto MVC [05].

O seguinte diagrama mostra um modelo de três visões. O modelo contém alguns valores de dados, e as visões que definem uma planilha, um histograma, e um gráfico de

pizza, apresentam esses dados de varias maneiras. Um modelo se comunica com suas visões quando seus valores mudam, e as visões se comunicam com o modelo para acessar esses valores.

Tomado no seu direito, esses exemplo reflete um projeto que separa visões e modelos. No entanto, o projeto é aplicável a um problema mais geral: separar objetos de maneira que mudanças ocorridas em um possam afetar um numero qualquer de objetos, sem exigir que o objeto alterado conheça detalhes dos outros. Esse projeto mais geral é descrito pelo padrão de projeto *Observer*.

Outra característica da MVC é que visões podem ser encaixadas. Por exemplo, um painel de controle de botões pode ser implementado como uma visão complexa contendo visões encaixadas compostas de botões. A interface do usuário para um objeto "inspetor" pode consistir em visões encaixadas, que podem ser reutilizadas em um depurador (*debugger*). A abordagem MVC suporta visões encaixadas com a classe *CompositeView*, uma subclasse de *View*; uma visão composta pode ser usada em qualquer lugar que uma visão possa ser usada, mas ela também contem e administra visões encaixadas.

Novamente, pode-se pensar nesse projeto como sendo um projeto que é permitido tratar uma visão composta tal como é tratado um dos seus componentes. Mas o projeto é aplicável a um problema mais geral, que ocorre sempre que se quer agrupar objetos e tratos o grupo como um objeto individual. Este projeto mais geral é descrito pelo padrão *Composite*. Ele permite criar uma hierarquia de classes na qual algumas subclasses definem objetos primitivos e outras classes definem objetos compostos, que agrupam os primitivos em objetos complexos.

A abordagem MVC também permite mudar a maneira como uma visão responde as entradas do usuário sem mudar sua apresentação visual. Por exemplo, você pode querer mudar a forma de como ela responde ao teclado ou fazer com que use um menu *pop-up* em lugar de teclas de comando. A MVC encapsula o mecanismo de resposta em um objeto Controlador. Existe uma hierarquia de classes de *Controllers*, tornando fácil criar um novo controlador como uma variante de um existente.

Uma visão usa uma instância de uma subclasse de *controller* para implementar uma estratégia particular de respostas; para implementar uma estratégia diferente, simplesmente substitua a instância por um tipo diferente de controlador. É possível mudar o controlador de uma visão em tempo de execução, para mudar a maneira como responde às entradas do

usuário. Por exemplo, uma visão pode ser desabilitada de maneira que simplesmente não aceite entradas, fornecendo um controlador que ignora os eventos de entrada.

O relacionamento *View-Controller* é um exemplo do padrão *Strategy*. Um *Strategy* é um objeto que represente um algoritmo. Ele é útil quando você quer substituir o algoritmo tanto estática como dinamicamente, quando há muitas variantes do algoritmo, ou quando o algoritmo tem estruturas de dados complexas que você deseja encapsular.

A abordagem MVC usa outros padrões de projeto, tais como *Factory Method*, para especificar por falta a classe controladora para uma visão Decorador para acrescentar capacidade de rolagem a uma visão. Mas os principais relacionamentos na MVC são fornecidos pelos padrões *Observer*, *Composite* e *Strategy* [05].

2.3 A LINGUAGEM JAVA

A linguagem Java foi desenvolvida por um grupo de engenheiros da *Sun Microsystems*, liderados por James Gosling, Patrick Naughton e Sun Fellow, com o objetivo de ser uma linguagem orientada a objetos que pudesse fornecer uma ligação com a linguagem C++, mas com maior segurança e com portabilidade do código (com um interpretador que especifica a forma do nível do sistema operacional) além de ser uma linguagem simples e prática, gerando um código bem limitado em tamanho.

Foi formalmente anunciada em uma conferência em maio de 1995 e hoje são utilizada para desenvolver grandes aplicativos corporativos e diversos outros propósitos [06].

A Maquina Virtual Java (JVM - *Java Virtual Machine*) é responsável por executar os programas em Java, assim um programa em Java poderá rodar em qualquer Sistema Operacional que tenha JVM instalado. O compilador Java (*javac*) traduz um programa com extensão .java para *bytecodes* desse programa. *Bytecode* é uma espécie de codificação que traduz tudo o que foi escrito no programa para um formato que a JVM entenda e seja capaz de executar.

Quando um programa é compilado, é gerado um arquivo com a extensão .class. Este tipo de arquivo é o *bytecode* do programa e que serão executados pela JVM, como mostrado na figura 8.

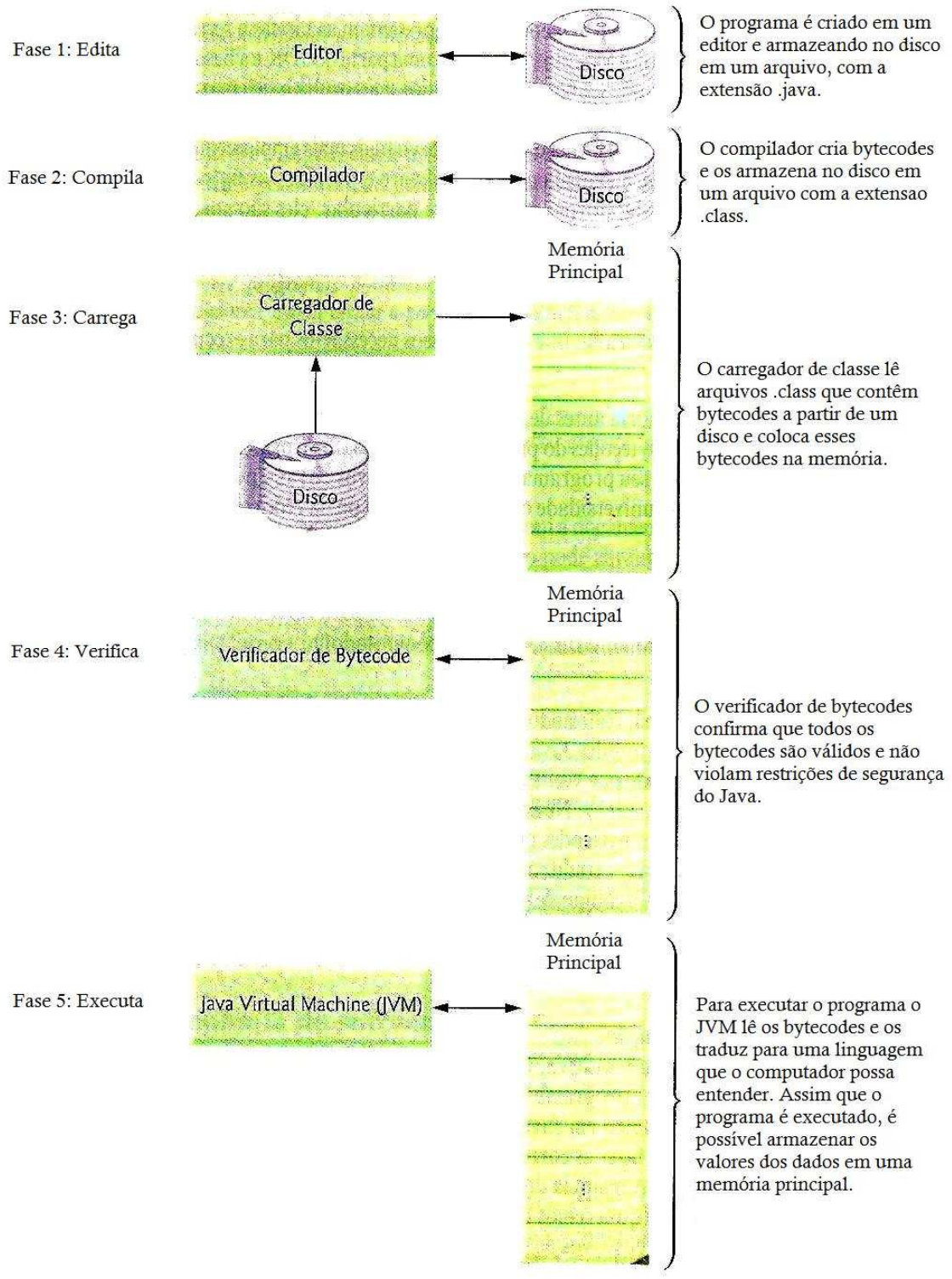

Figura 8 - Ambiente de desenvolvimento Java típico [06].

2.4 O FRAMEWORK JGRAPH

2.4.1 Introdução

JGraph é uma poderosa biblioteca *open-source*, projetada em 2000 por Gaudenz Alder na sua Universidade [07], que fornece as mais diversas funcionalidades requeridas para construção de um grafo.

O framework vem com uma coleção de layouts que posicionam automaticamente os diagramas, alguns exemplos são gráficos de fluxograma, *workflow* e o layout em árvore que é o esquema clássico para organogramas.

Ao contrário de outros frameworks de código aberto, o *Jgraph* vem sendo desenvolvido ao longo dos anos, com constantes correções de *bugs* e atualizações, além de apresentar mais exemplos a cada versão. Atualmente a ferramenta encontra - se na versão 1.7.1.5, publicada dia 1 de agosto de 2011.

Exemplos de aplicações em que o uso da ferramenta já obteve sucesso incluem desenhos de diagramas, representações de tráfegos, de banco de dados, diagramas *Unified Modeling Language* (UML), redes de computadores e telecomunicações, circuitos eletrônicos, *Computer-aided design* (CAD), *Very-large-scale integration* (*VLSI*), redes sociais e financeiras, *data mining*, ciclos biológicos, entre outros.

O *Jgraph* está disponível nas plataformas Java e JavaScript, com forte interação com *Application Programming Interface* (API), AWT e *Swing*, o desenvolvimento do projeto foi focado na linguagem Java, pois é a mesma do sistema acadêmico IdUFF.

A figura 9 mostra um exemplo do *Jgraph* usando *Swing*.

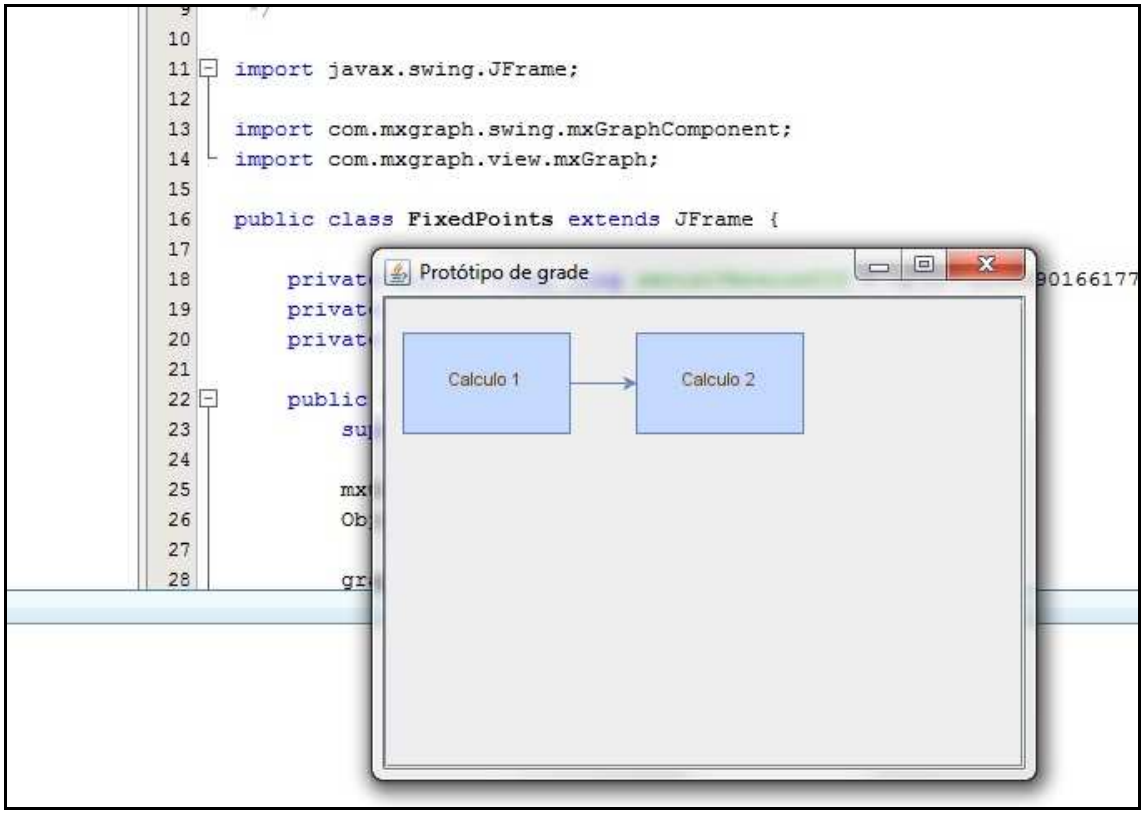

Figura 9 - Representaçao gráfica utilizando o Jgrah e API Swing.

Algumas aplicações gráficas em JavaScript usando o *Jgraph* estão disponíveis na Web no próprio site do framework, com elas é possível desfrutar de todo o potencial da ferramenta e desenvolver diversas formas de diagramas.

2.4.2 Exemplos e aplicações

2.4.2.1 Organogramas

Organogramas, são gráficos que representam a estrutura formal de uma organização. Eles mostram como estão dispostas unidades funcionais, a hierarquia e as relações de comunicação existentes entre estes [08].

Alguns exemplos são os órgãos ou departamentos que são unidades administrativas com funções bem definidas como gerência administrativa, diretoria e etc. Os órgãos possuem um responsável, cujo cargo pode ser chefe, supervisor, gerente, coordenador, diretor,

secretário, governador ou presidente por exemplo.

A figura 10 representa um organograma gerado com *Jgraph*.

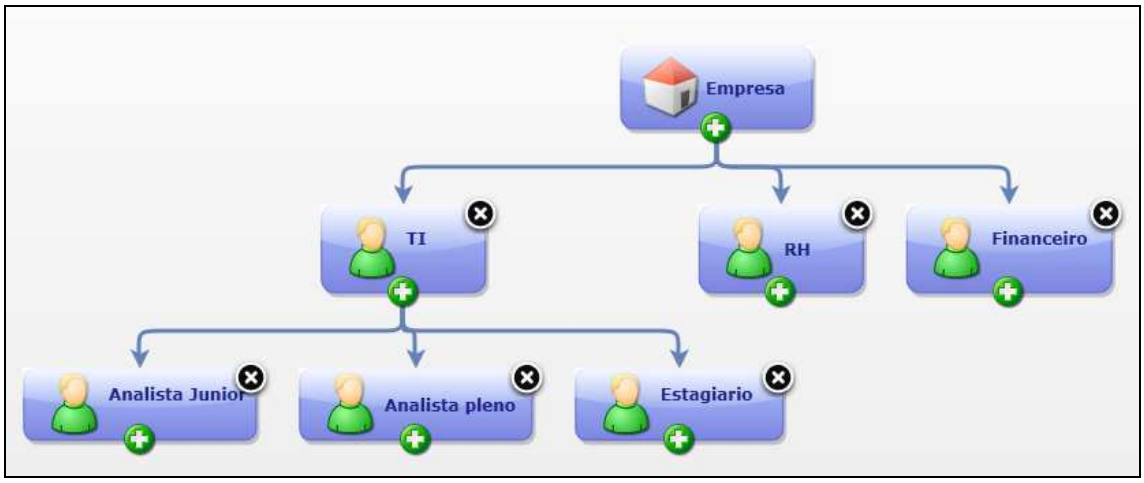

Figura 10 – Representação de organograma com *Jgraph*.

2.4.2.2 Workflow

Workflow é "um conjunto coordenado de atividades (sequenciais ou paralelas) que são interligadas com o objetivo de alcançar uma meta comum", sendo atividade conceituada como "uma descrição de um fragmento de trabalho que contribui para o cumprimento de um processo" Assim, pode-se assumir que *workflow* é a divisão de um grande trabalho em várias tarefas menores, com pré-requisitos entre elas, que devem ser respeitados para o avanço da atividade. [09]. Na figura 11 é representado um *workflow* simples gerado no *Jgraph*.

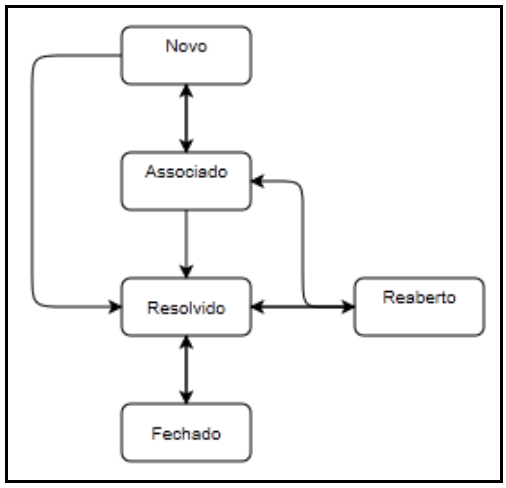

Figura 11 - Representação de Workflow com Jgraph.

2.4.2.3 Diagrama de Banco de Dados

Com o *Jgraph* também é possível gerar representações gráficas do modelo de uma base de dados, referenciando suas entidades e chaves estrangeiras como mostrado na figura X. Grandes empresas como a Oracle, por exemplo, usam uma tecnologia similar para construir projetos de banco de dados a partir de ferramentas gráficas, gerando automaticamente comandos em SQL. Um exemplo onde existe essa ferramenta é o *OracleExpress* 10g, uma versão de administrador de banco de dados completamente desenvolvido para uso em navegadores. Com ela é possível visualizar e criar todas as tabelas e relacionamentos. A figura 12 representa um diagrama do banco de dados do projeto gerado no *Jgraph*.

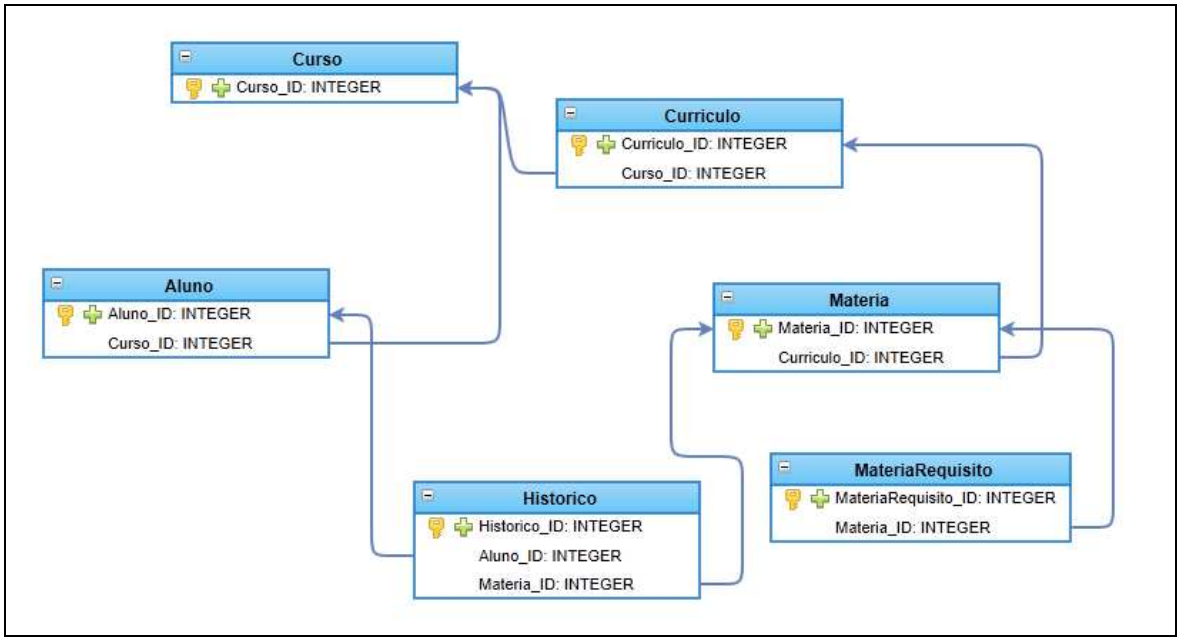

Figura 12 - Exemplo de diagrama de banco de dados usando *Jgraph*.

2.4.3 JgraphX

JgraphX ou *mxGraph* é a versão da biblioteca para *Java-Swing* e permite produzir aplicações Java Swing que possuem habilidades de manipular diagramas interativamente. *JGraphX* é projetado principalmente para esse ambiente de trabalho, como Java é uma linguagem portável, é permitido então utilizar o *JGraphX* em ambiente Web [10].

As versões completas de *JGraphX* são disponíveis em arquivos zip. Ao se descompactar, uma pasta chamada *JGraphX* será criada, essa pasta é a pasta raiz da biblioteca [10].

Estrutura de arquivos da biblioteca está representada na tabela 1

Tabela 1: Estrutura de diretórios do Jgraph [10]

/doc	Documentation root
/src	Source of the library
/lib	Contains pre-built jar of the library.
/examples	Examples demonstrating the use of JGraphX
license.txt	The licensing terms

2.4.3.1 Principais métodos

O *JGraphX* possui uma classe principal chamada *mxGraphModel*, projetada para armazenar toda a estrutura do gráfico. A biblioteca possui também uma estrutura de transação, para poder lidar com eventos. Seus comandos principais são:

- *beginUpdate,* que inicia uma nova transação.
- *endUpdate*, que finalize a transação.
- *addVertex,* esse cria um novo '*Vertex'* ou nó no gráfico.
- *addEdge*, que cria uma '*Edge'* uma aresta entre dois *Vertex*.

O comando *insertVertex* recebe sua configuração por parâmetros:

Exemplo: *mxGraph.insertVertex* (*parent*, id, valor, x, y, largura, altura, estilo)

Onde em x e y são definidos a posição onde será desenhado no *mxGraph*, assim como também suas dimensões altura e largura, o valor do conteúdo do 'nó' e um estilo.

O comando *insertEdge* não é muito diferente do *insertVertex*:

Exemplo*: mxGraph.insertEdge* (*parent*, id, valor, origem, destino, estilo)

Como representa uma aresta, não possui parâmetros para definir dimensões, apenas seu nó de origem-destino, valor e um estilo.

A figura 13 esclarece o efeito desses comandos visualmente.

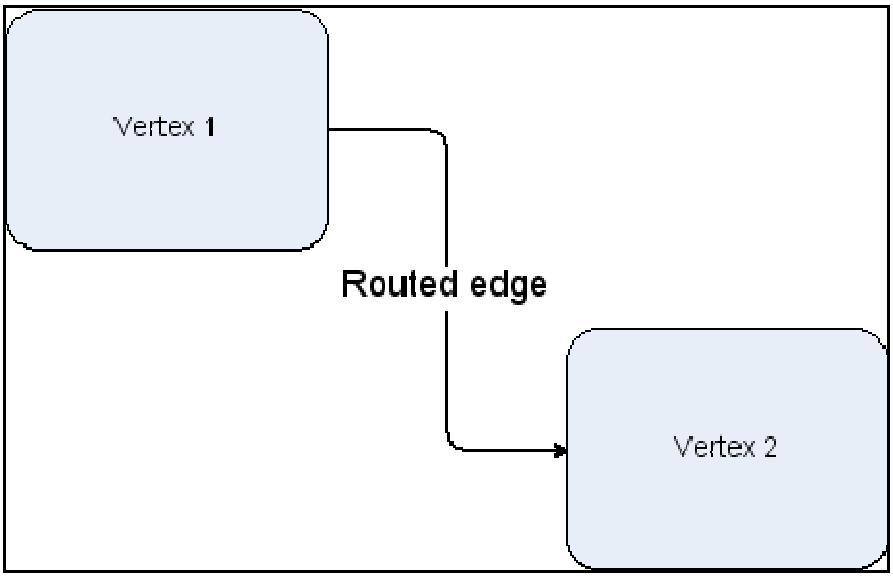

Figura 13 – Exemplo de *Vertex* e *Edge* [10].

1.1.1.2 Estilos

O *mxGraph* possibilita também definir estilos de modo geral ou específicos dentro de um diagrama. O método *getStylesheet* no *graph* permite definir estilos suavizando as

ligações entre os vértices ou seja as *Edges*, ou mesmo os *vértex's*, desenhando-os em degrade, cores diferentes ou suavizando suas bordas.

Os métodos padrões para definição de estilo de um *graph* são *getDefaultVertexStyle* e *getDefaultEdgeStyle*. Um exemplo demonstrando a aplicação de cada método estão nas figuras 14 e 15.

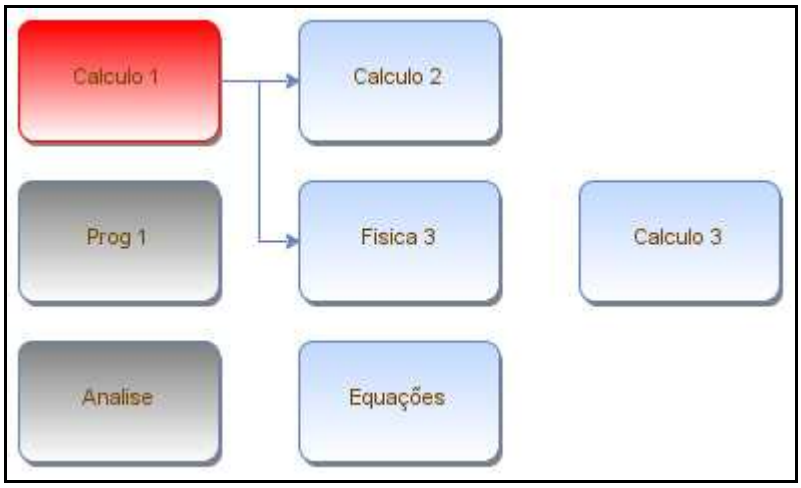

Figura 14 - Exemplo usando método *getDefaultVertexStyle*.

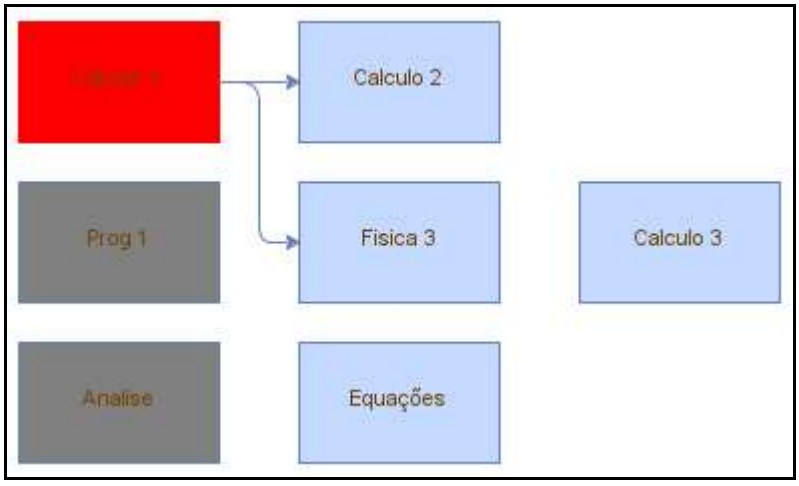

Figura 15 - Exemplo usando método *getDefaultEdgeStyle*.

Além desses métodos padrões o *JgraphX* possui uma classe chamada *mxConstants*. Nessa classe é possível customizar totalmente um diagrama bastando definir no código toda a configuração desejada. A figura 16 mostra um exemplo usando as seguintes constantes:

STYLE_GRADIENTCOLOR, STYLE_OPACITY, STYLE_FONTCOLOR, STYLE_SHAPE e *STYLE_SHADOW*.

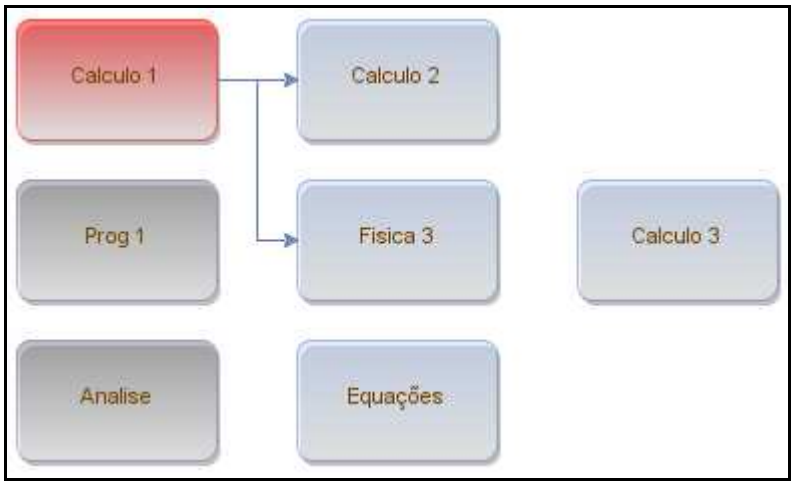

Figura 16 - Exemplo usando algumas chamadas de *MxConstants*.

1.1.1.3 Geometria

A composição do gráfico ao ser desenhado na tela se baseia nos eixos cartesiano X e Y. Ao inserir um *Vértex* o referencial é configurado de acordo com o formato do gráfico escolhido pela classe *MxIGraphLayout*. Essa classe possui alguns estilos organizacionais de diagramas, podendo ser gerado de forma hierárquica como na figura 17, ou definido fixamente ao passar por parâmetros esses dados para cada Vértice da figura.

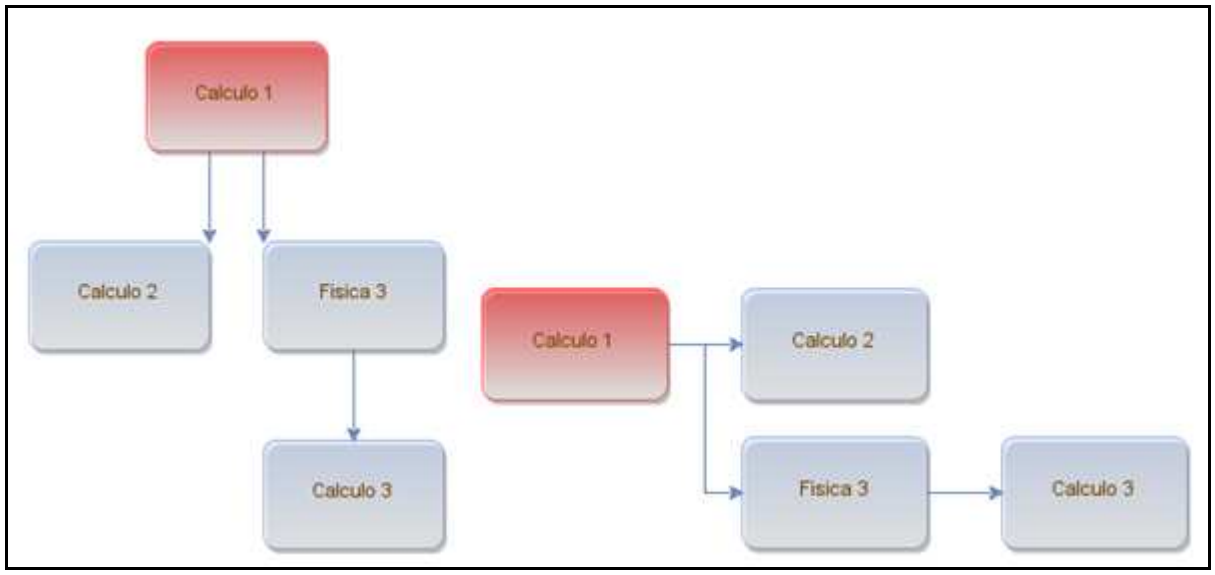

Figura 17 - exemplo usando *mxHierarchicalLayout* e *mxOrthogonalLayout*

Outros formatos já pré-definidos pelo JgraphX na classe *mxHierarchicalLayout* são: *mxCircleLayout*, *mxFastOrganicLayout*, *mxFastOrganicLayout*, *mxOrthogonalLayout e mxEdgeLabelLayout.*

2.5 DATABASE MYSQL

O *MySQL* teve origem quando os desenvolvedores David Axmark, Allan Larsson e Michael "Monty" Widenius, na década de 90, precisaram de uma interface SQL compatível com as rotinas ISAM que utilizavam em suas aplicações e tabelas. Em um primeiro momento, tentaram utilizar a API mSQL, contudo a API não era tão rápida quanto eles precisavam, pois utilizavam rotinas de baixo nível (mais rápidas que rotinas normais). Utilizando a API do mSQL, escreveram em C e C++ uma nova API que deu origem ao *MySQL*.

O programa *MySQL* é um servidor robusto de bancos de dados SQL (*Structured Query Language* - Linguagem Estruturada para Pesquisas) muito rápido, multitarefa e multiusuário. O Servidor pode ser usado em sistemas de produção com alta carga e missão crítica bem como pode ser embutido em programa de uso em massa.

MySQL é de Licença Dupla. Os usuários podem escolher entre usar o programa como um produto *Open Source* sob os termos da GNU (*General Public License*) ou podem comprar uma licença comercial padrão.

Desenvolvido utilizando as linguagens de programação C e C++, unido com o uso de GNU *Automake*, *Autoconf* e *Libtool*, torna o *MySQL* uma aplicação altamente portável entre diferentes sistemas, plataformas e compiladores. Além disso, fornece sua API para várias outras linguagens, como Java, Python, PHP, Perl, C, C++, entre outras.

O *MySQL* conta com uma série de recursos para análise do tempo de execução de operações. Isto possibilitou que seus desenvolvedores realizassem alguns testes comparativos com outros bancos de dados.

Esses testes foram realizados baseados em um computador com Windows NT4 e acesso via ODBC. Dentre os bancos de dados mais conhecidos, os resultados mostrados nas figuras 18 e 19 foram obtidos [11].

Banco de dados	Segundos
mysql_odbc	464
db2_odbc	1.206
ms-sql_odbc	1.634
oracle_odbc	20.800
sybase_odbc	17.614

Figura 18 - Leitura de 2.000.000 por índice [11].

Banco de dados	Segundos
mysql_odbc	619
db2_odbc	3.460
ms-sql_odbc	4.012
oracle_odbc	11.291
sybase_odbc	4.802

Figura 19 - Inserção de 350.768 linhas [11].

2.6 SUBVERSION

Subversion é um sistema de controle de versão livre/open-source projetado pela *CollabNet*, *Inc* no inicio do ano 2000 para substituir o CVS(*Concurrent Version System*) que havia se firmado como um padrão de *fact* no mundo open source, mas que continha muitos bugs e inconveniências.

Em seu núcleo está um repositório, que é uma central de armazenamento de dados. O repositório armazena informação em forma de uma *árvore de arquivos* - uma hierarquia típica de arquivos e diretórios. Qualquer número de *clientes* se conecta ao repositório, e então lê ou escreve nestes arquivos. Ao gravar dados, um cliente torna a informação disponível para outros; ao ler os dados, o cliente recebe informação de outros. Figura 20 ilustra isso.

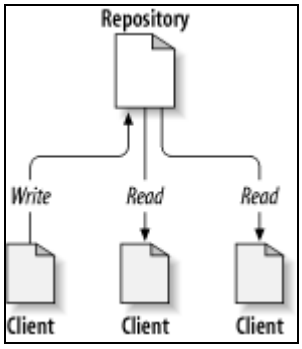

Figura 20 - Um típico sistema cliente/servidor [12].

Para evitar problemas quando dois ou mais usuários compartilham um mesmo arquivo e acaba se tornando muito comum esses usuários acidentalmente sobrescreverem as mudanças feitas pelos outros no repositório, o Subversion tem duas soluções:

A Solução *Lock-Modify-Unlock* que permite que apenas uma pessoa modifique o arquivo, quando um usuário "travar" (*lock*) um arquivo os demais usuários apenas terão acesso de leitura nesse mesmo arquivo até que o usuário responsável por travar o arquivo destrave (*unlock*) o mesmo. Essa situação é demonstrada na figura 21:

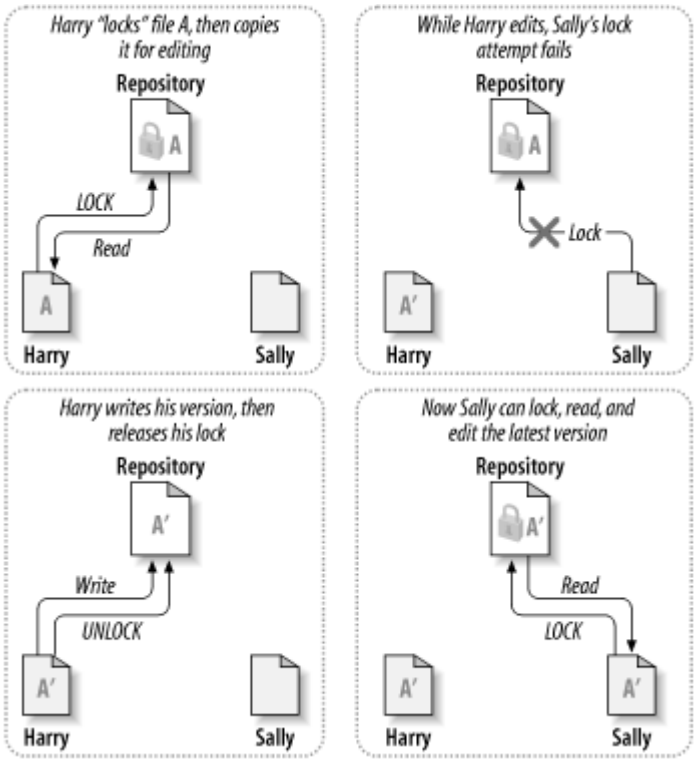

Figura 21 - A solução *lock-modify-unlock* [12].

Essa solução apresenta alguns problemas, caso um usuário queira editar apenas o inicio do arquivo outro usuário não poderá editar o final do mesmo arquivo, causando uma serialização desnecessária. Outro tipo de problema é uma falsa sensação de segurança, se um usuário travar um arquivo, outro usuário poderá editar sem problemas um arquivo que dependa totalmente do primeiro arquivo, podendo ocorrer mudanças semanticamente incompatíveis e os dois arquivos não funcionarem mais juntos.

Uma solução mais inteligente adotada pelo subversion é o *Copy-Modify-Merge* Nesse modelo, cada usuário se conecta ao repositório do projeto e cria uma cópia de trabalho pessoal (*personal working copy*, ou cópia local) - um espelho local dos arquivos e diretórios do repositório. Os usuários então trabalham simultaneamente e independentemente, modificando suas cópias privadas. Finalmente, as cópias privadas são fundidas (*merged*) numa nova versão final. O sistema de controle de versão frequentemente ajuda com a fusão, mas, no final, a intervenção humana é a única capaz de garantir que as mudanças foram realizadas de forma correta. As figuras 22 e 23 mostram este processo [12].

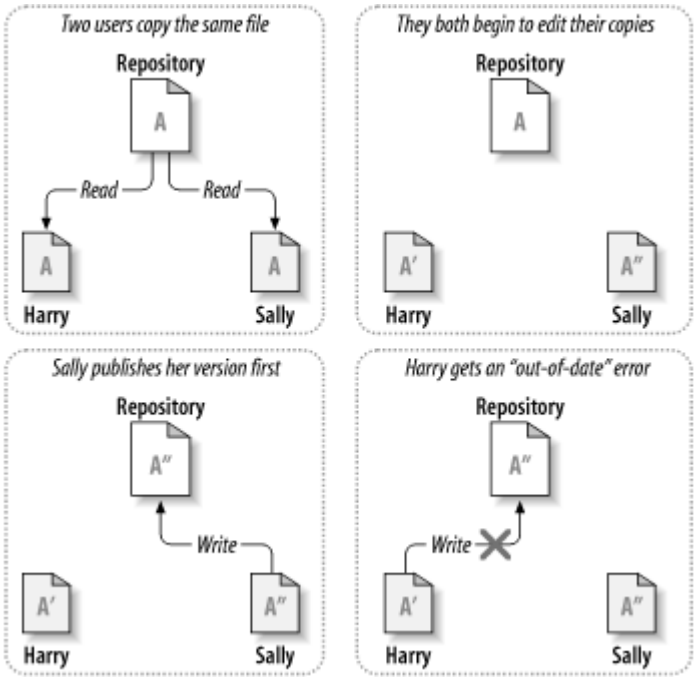

Figura 22 - A solução *copy-modify-merge* [12].

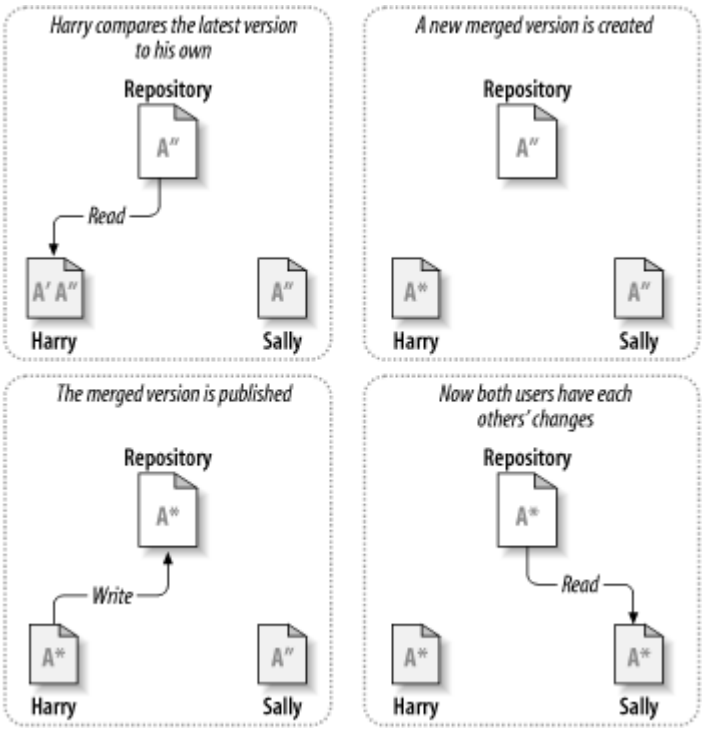

Figura 23 - A solução *copy-modify-merge* 2 [12].

2.7 HIBERNATE

O Hibernate vem com uma poderosa linguagem de consulta HQL (*Hibernate Query Language)* que é muito parecida com o SQL (*Structured Query Language*). No entanto, comparado com o SQL o HQL é totalmente orientado à objetos, e compreende noções de herança, polimorfismo e associações.

O Hibernate é uma ferramenta de mapeamento objeto/relacional para Java, mas também pode fazer parte de projetos desenvolvidos na plataforma .Net da Microsoft graças ao NHibernate.

O framework Hibernate, assim como a maioria dos frameworks MOR(*Mapeamento Objeto-Relacional*), é um software livre de código aberto distribuído com a licença LGPL (*Lesser General Public License*). Ele transforma os dados tabulares de um banco de dados em um grafo de objetos definido pelo desenvolvedor, fazendo o mapeamento entre o modelo de objetos e o modelo de dados relacional mediante a utilização de arquivos XML para estabelecer a relação.O termo MOR refere-se à técnica de mapeamento de uma representação de dados em um modelo de objetos para um modelo de dados relacional baseado em um esquema E/R.

O framework Hibernate foi desenvolvido por uma equipe de programadores Java liderada por Gavin King e teve sua primeira versão divulgada em 2004. Segundo King, um dos objetivos ao criar o projeto era resolver seus problemas referentes à persistência causados pelo EJB (*Enterprise Java Beans*) 2.0, o qual considerava muito complexo. Após isso a empresa JBoss Inc contratou os principais desenvolvedores do projeto para fazer seu suporte [13].

A figura 24 represente a arquitetura do Hibernate

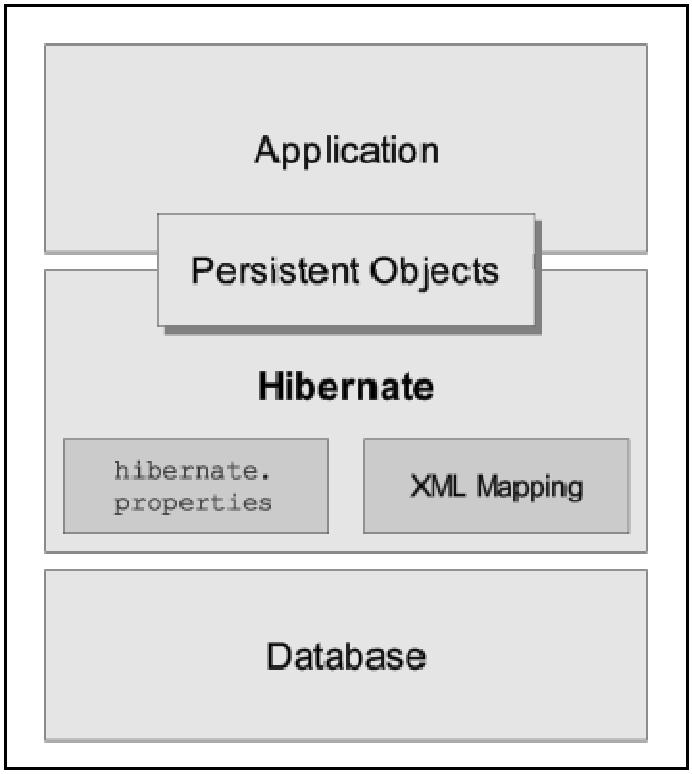

Figura 24 - Arquitetura básica do MOR com Hibernate [14].

O Hibernate persiste objetos Java comuns que podem ser *JavaBeans* ou *POJOs*, onde a única coisa especial sobre eles é que são associados a (exatamente uma) *Session*. Quando a *Session* é fechada, eles são separados e liberados para serem usados dentro de qualquer camada da aplicação. As classes persistentes são definidas em documentos de mapeamento. [14]

2.7.1 Mapeamento Objeto Relacional

O Framework Hibernate requer uma forma de mapeamento de metadados para determinar como a comunicação entre classes e tabelas do banco relacional serão realizadas. Os dois principais meios de realizar essa comunicação é através de um arquivo XML ou através de um recurso chamado *Annotations.*

O mapeamento objeto-relacional definido através de um arquivo XML permite que se trabalhe com dados persistentes quase da mesma maneira como você trabalha com *POJOs* persistentes. Uma árvore XML analisada, pode ser considerada como apenas uma maneira de representar os dados relacionais como objetos, ao invés dos *POJOs* [14].

A figura 26 e representa um exemplo de mapeamento com arquivo XML onde criamos a tabela aluno(classe representada na figura 25)

Figura 25 - Classe Aluno[15].

```
<meta attribute="class-description">Classe Aluno.</meta>
<id name="studentId" type="int" column="id">
<generator class="native" />
</id>
<property name="studentMatric" type="int" length="100" not-null="true" column="matricula" />
<property name="studentName" type="String" length="100" not-null="true" column="nome" />
<property name="studentcpf" type="long" length="100" not-null="true" column="cpf" />

<key column="id" />
</set>
</class>
</hibernate-mapping>
```

Figura 26 - Representação da tabela Aluno em um arquivo XML [15].

O recurso *Annotations* podem ser definidos como metadados que aparecem no código fonte e são ignorados pelo compilador. Qualquer símbolo em um código Java que comece com uma @ (arroba) é um *Annotations*. Este recurso foi introduzido na linguagem Java a partir da versão Java SE 5.0.

Dois tipos de anotações são suportados pelo Java 5.0: anotações simples e meta anotações. As anotações simples são usadas apenas para agregar significado especial ao código fonte, mas não para criar algum tipo de anotação. Já as meta anotações são aquelas utilizadas para definir tipos de anotações.

A classe Aluno (figura 25) pode ser mapeada com anotações como é possível observar na figura 27

```
//Anotação que informa que a classe mapeada é persistente
@Entity
//Informando nome e esquema da tabela mapeada
@Table(name="aluno", schema="anotacoes")
public class Aluno {
//Definição da chave primária
@Id
//Definição do mecanismo de definição da chave primária
@GeneratedValue(strategy = GenerationType.SEQUENCE)
//Informa o nome da coluna mapeada para o atributo
@Column(name="id_aluno")
private int id;
private int matricula;
private String nome;
private long cpf;
public void Aluno(){}
//Métodos getters e setters
//...
}
```

Figura 27 - Representação da tabela Aluno com Annotations [15].

2.7.2 HQL

O HQL(*Hibernate Query Language*) é uma linguagem de consulta do Hibernate, muito semelhante com o SQL(*Structured Query Language*), mas ao contrário da SQL, a HQL é totalmente orientada a objetos, não é uma linguagem de manipulação de dados como SQL, mas é usado para obter objetos, nela não se referencia tabelas, e sim os objetos do modelo que foram mapeados para as tabelas do banco de dados, como mostra a figura 28.

```
Session sessao = HibernateUtility.getSession();
Transaction tx = sessao.beginTransaction();
Query select = sessao.createQuery("from Turma as turma where turma.nome = :nome");
select.setString("nome", "Jornalismo");
List objetos = select.list();
System.out.println(objetos);
tx.commit();
sessao.close();
```

Figura 28 - consulta em HQL [16].

A figura X retornaria todos os objetos da classe turma que tivessem o nome “Jornalismo”.

O Hibernate permite escolher entre quarto estratégias diferentes de captura para qualquer associação:

***Immediate fetching* - Captura Imediata**

Captura associativa imediata ocorre para recuperar uma entidade do banco de dados e então recuperar imediatamente outra entidade associada ou entidades em uma requisição futura do banco de dados ou do cachê. não é uma estratégia captura eficiente a não ser que espere que as entidades associadas estejam quase sempre no cachê.

***Lazy fetching* - Captura preguiçosa**

O objeto associado ou coleção é capturado de "forma preguiçosa", quando é acessado. Quando um cliente requisita uma entidade e seu grafo de objetos associado do banco de dados, não é usualmente necessário recuperar todo o grafo de todos os objetos associados (indiretamente), por exemplo, carregar um pedido simples, não deve disparar uma carga de todos os itens do pedido. O *Lazy fetching* permite ao programador decidir quantos grafos de objetos serão carregados no primeiro acesso ao banco de dados e quais associações devem ser carregadas somente quando forem acessadas pela primeira vez. Este é um conceito fundamental em persistência de objetos e o primeiro passo para obter um desempenho aceitável, pois pode ajudar a reduzir a carga no banco de dados e é geralmente uma boa estratégia padrão.

***Batch fetching* - Captura em Lote**

Captura em lote não é estritamente uma estratégia de carga associada; é uma técnica

que pode ajudar a melhorar a performance da carga preguiçosa ou carga imediata. Geralmente, quando se carrega um objeto ou uma coleção, a cláusula *WHERE* do SQL especifica o identificador do objeto ou os objetos que possuem a coleção, se a carga em lote estiver habilitada, Hibernate procura ver que outras instâncias ou coleções não inicializadas são referenciadas na seção corrente e tenta carregá-los ao mesmo tempo especificando múltiplos valores identificadores na cláusula *WHERE*.

2.8 JPA

Após o sucesso do Hibernate, a especificação *Java Persistence* API (JPA) foi criada com o objetivo de padronizar as ferramentas ORM para aplicações Java e consequentemente diminuir a complexidade do desenvolvimento.

Ela específica um conjunto de classes e métodos que as ferramentas de ORM devem implementar. Veja que a JPA é apenas uma especificação, ela não implementa nenhum código.

Java Persistence API, mais conhecido como JPA é a especificação padrão para o gerenciamento e mapeamento objeto relacional o *Object-relational mapping* (ORM) , JPA surgiu na plataforma JEE 5.0 na versão 3.0 do Java Beans, foi criada com o objetivo de padronizar as ferramentas ORM para aplicações Java e consequentemente diminuir a complexidade do desenvolvimento.

A API possui uma especificação completa para realizar o ORM, também dá suporte a uma linguagem de consulta semelhante a linguagem do SQL conhecida como EJB: *Query Language – EJB-QL*. Portanto JPA pode ser considerado um framework utilizado na camada de persistência, veja figura 29, para o desenvolvedor ser mais produtivo, nota-se que não é necessário implementar códigos no JPA.

Atualmente existem algumas implementações do JPA que são mais populares, como por exemplo, Hibernate, no entanto existem mais algumas como *TopLink*, *EclipseLink* e *OpenJPA* . A API JPA permite persistir objetos em banco de dados relacionais, permite debug e auditora das operações entre a aplicação e o banco de dados, permite também consultas complexas.

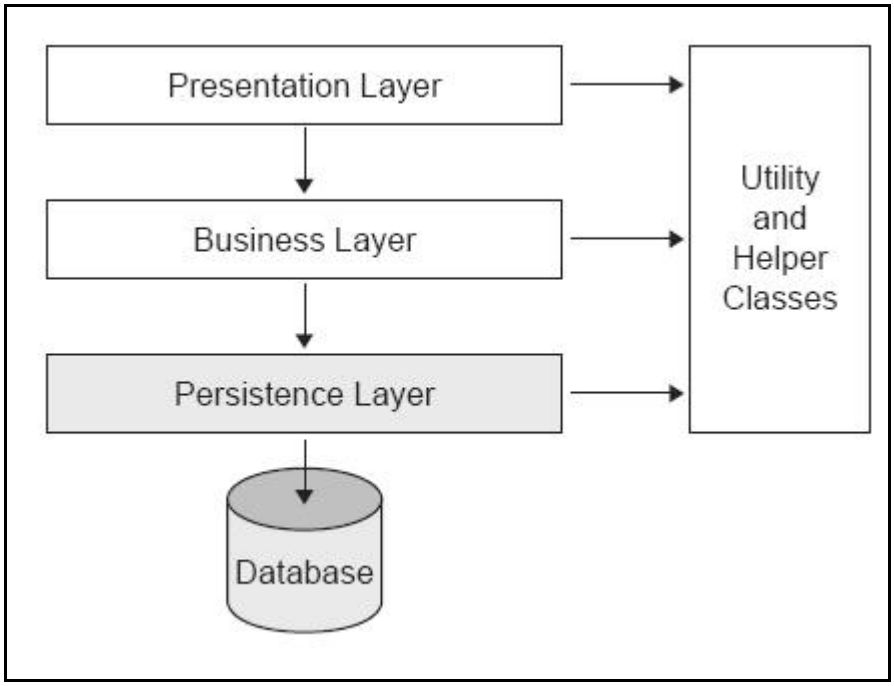

Figura 29 - Camada persistente [17].

Conceitos de Entidades no JPA

Toda entidade precisa ter uma chave primária, uma chave simples ou composta, entidades implementarão o conceito de Serialização se o objeto necessitar ser enviado via rede, essas entidades não são associadas a um contexto de persistência. A entidade divide-se em alguns ciclos são eles o *New*, onde cria-se uma nova entidade (não tem id *persistente*, pode ser criada usando a palavra new), *Managed* (tem id persistente associado a um contexto de persistência), *Removed* (tem um id persistente, está escalonada para ser excluída do banco) , *Detached* (tem um id persistente, mas não está associada a um contexto de persistência).

Esse id será o identificador de persistência, ou seja, a chave primária do banco, @Id é um exemplo de como pode ser nomeado na classe JPA.

Entity Manager

Característica do Hibernate o *Entity Manager* controla o ciclo das entidades, são eles o *new*() – cria nova entidade, *persist*() – persiste uma entidade, *refresh*() - atualiza o estado de uma entidade, *remove*() – marca para remoção e *merge*( ) - sincroniza o estado das desacopladas.

Relacionamento

Como as tabelas no banco de dados relacional, a API JPA tem como característica a opção Relacionamento, como por exemplo o Um para um (*@OneToOne*), Um para N(*@oneToMany*), N para Um (*@ManyToOne*) e N para N (*@ManyToMany*) [18].

2.9 JAVA SWING e AWT

A biblioteca AWT (*Abstract Windowing Tollkit*) foi umas das primeiras bibliotecas criadas para a construção de GUI's (*Graphical User Interfaces*). O AWT tem estado presente em todas as versões do Java. Os objetos AWT são construídos sobre objetos de código nativo, o que fornece um *look-and-feel* nativo, isso porque esta biblioteca faz uso dos recursos do sistema operacional que roda a maquina Java. Com isto as janelas e os componentes visuais do AWT variam de acordo com o Sistema Operacional aonde roda a maquina Java e limita a um numero pequeno os componentes comuns a todos os Sistemas Operacionais.

Para melhorar os componentes AWT existentes e proporcionar avançados controles tal como tables e árvores hierárquicas foi criada a biblioteca Swing, que fica no pacote javax.swing .

Swing tem três melhoramentos em relação ao AWT. Primeiro, não usa componentes nativos. Foi escrito inteiramente em Java, e cria seus próprios componentes. Esta característica resolve problemas de portabilidade. Segunda, porque Swing é tem controle completo dos componentes. Controla como os componentes aparecem na tela, e controla o comportamento de sua aplicação. Pode-se escolher entre vários "*look-and-fell*" prontos, ou poderá construir o seu próprio. Esta característica é chamada de "*Pluggable Look-and-Feel*" ou PLAF, e pode ser configurada em execução sem ter que fechar a aplicação. Isso permite o usuário ter um *feedback* do L&F configurado decidindo pelo mais confortável. Terceiro, Swing faz uma clara distinção entre os dados do componente (*model*) e a atual visão deles (*view*). Esta separação significa que componentes são extremamente flexíveis. Pode-se facilmente adaptar componentes para mostrar novos tipos de dados ou alterar a aparência dos componentes [19].

O seguinte esquema visto na figura 30 mostra a maioria das classes que compõem o

Java Swing e mostra também a relação entre as classes AWT (a amarelo) e as classes *Swing* (a azul):

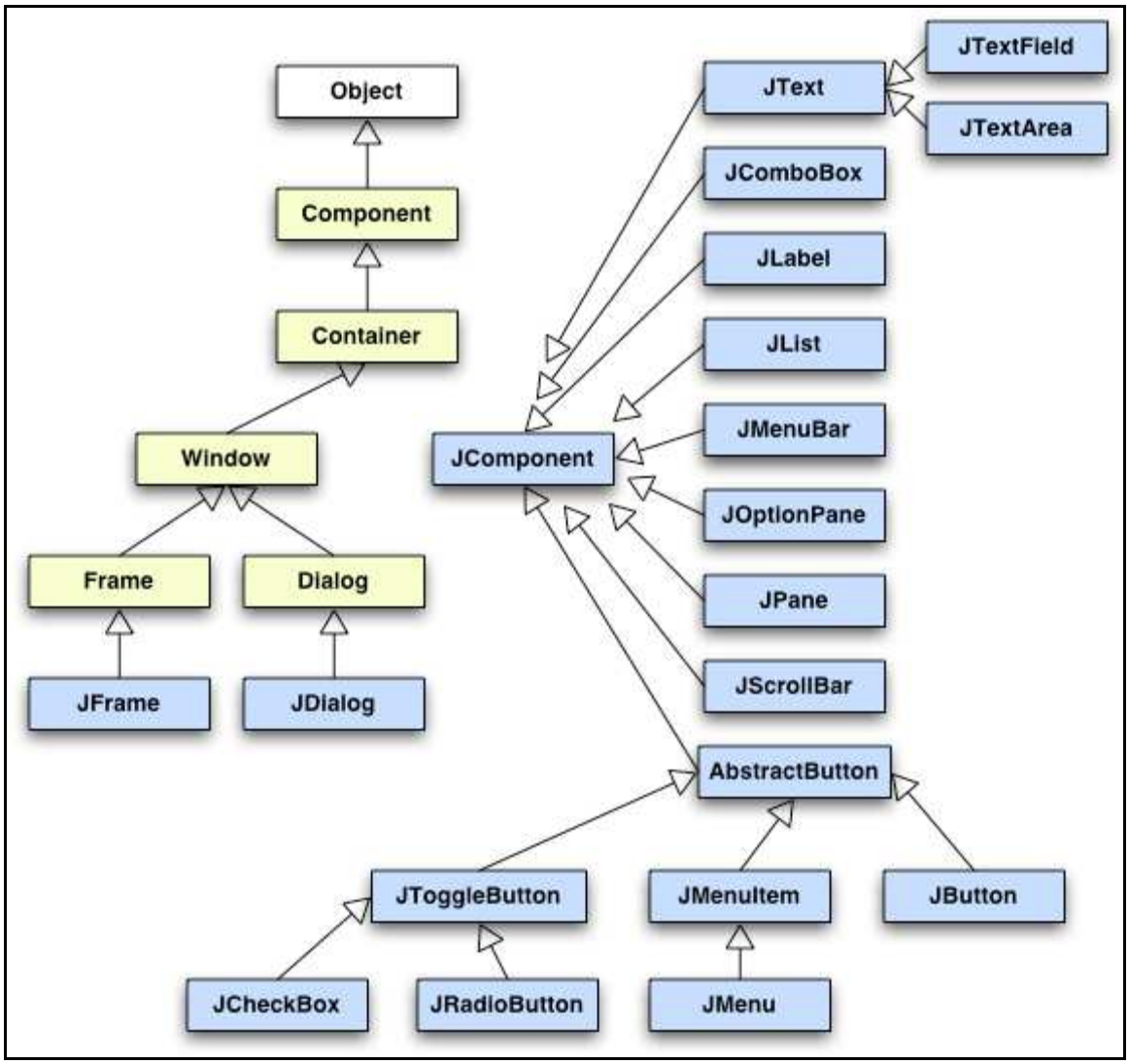

Figura 30 - AWT x Swing [20].

3 IMPLEMENTAÇÃO

3.1 ESTRUTURA DO PROJETO

3.1.1 Padrão MVC

Como o projeto está no formato MVC. A *View*, onde encontra-se as páginas em xhtml e os códigos JSF, Controladores e *Services* que representam a camada de controle e regras de negócio, e Modelo, onde se encontram todos os modelos usados no projeto, são as entidades Aluno, Curriculo, Curso, Histórico, Matéria e MateriaRequesito, cada um representa uma Classe com seus atributos em Java, referenciando as tabelas no banco de dados que foi criado em *MySQL*.

O objetivo desse banco foi simular de forma simples o uso das informações de uma base de dados e preencher o estado da grade curricular de um aluno de um Curso.

3.1.2 Diagrama de Classe

O diagrama de classe da figura 31 representa o mapeamento do projeto.

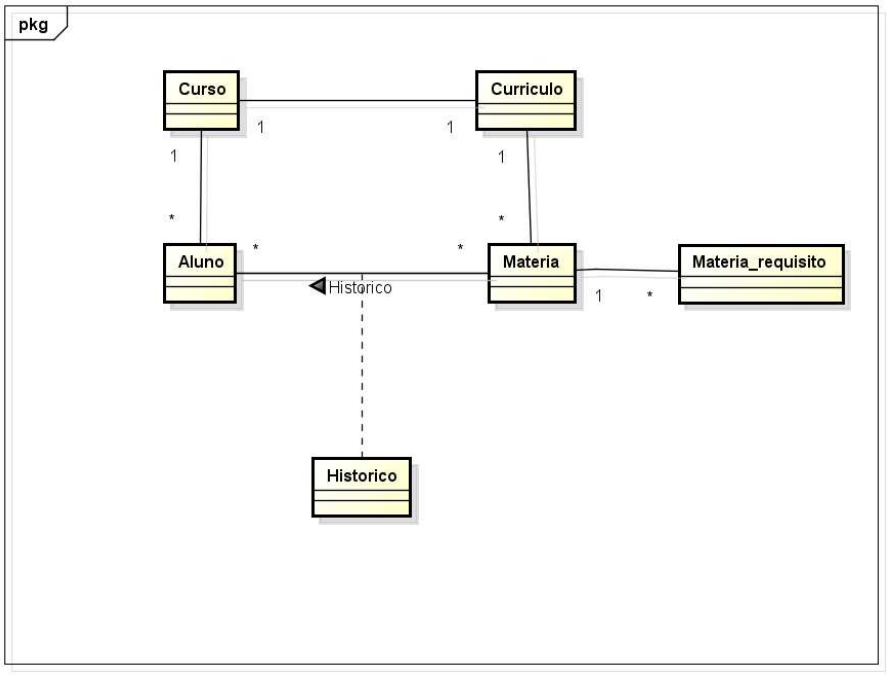

Figura 31 - Diagrama de classe ref proprio projeto

3.1.3 Mapeamento do Projeto

Como é possível notar, um Curso possui apenas um currículo no projeto, algo que não representa a realidade dos cursos oferecidos em uma Universidade. O foco do projeto foi de manusear a informação e exibir em tela graficamente para o usuário, portanto o projeto não foi baseado nas tabelas do sistema acadêmico (IdUFF), apesar da oportunidade oferecida por um dos gerentes do sistema. A repercussão do projeto na equipe do sistema acadêmico foi positiva, muitos se interassaram, inclusive funcionários do Departamento de Administração Escolar (DAE).

Para o mapeamento e consultas ao banco de dados foi usado no projeto o Hibernate, uma ferramenta de mapeamento objeto/relacional feita para Java. Ela transforma os dados tabulares de um banco de dados em um grafo de objetos definido pelo desenvolvedor. Usando o Hibernate, o desenvolvedor ganha tempo não escrevendo consultas de acesso a banco de dados em SQL puramente, acelerando a velocidade do desenvolvimento.

A figura 32 mostra como fica o mapeamento da Classe Matéria do Projeto.

```
@Entity
@SequenceGenerator(name = "Sequencia", sequenceName = "seq_materia")
@Table(name = "materia")
public class Materia implements Serializable {

    @Id
    @GeneratedValue(generator = "Sequencia", strategy = GenerationType.AUTO)
    private Long id;
    @ManyToOne
    private Curriculo curriculo;
    private String nome;
    private int periodo;
    private int posicao;
    @OneToMany(mappedBy = "materia")
    private List<MateriaRequisito> prerequisito;
```

Figura 32 - Exemplo da classe Matéria usando Hibernate.

Note que o Hibernate oferece as *annotations* *@id*, *@ManyToOne*, *@OneToMany* e entre outras facilitando todo o relacionamento das classes no projeto, e o processamento das '*sequences'* das tabelas do banco de dados.

No projeto essas consultas ao banco de dados podem ficar armazenadas em qualquer camada do MVC, portanto, o local foi escolhido foi no modelo, facilitando a visualização dos seus atributos, ganhando assim tempo ao realizar consultas.

Na figura 33, foram feitas 4 consultas ao banco para retornar informações relacionadas com a entidade 'Materia'.

```
@NamedQueries({
    @NamedQuery(name = "Materia.recuperaTodasAsMateriasOrdenadasPorPeriodoPosicao",
    query = " select m from Materia m "
            + " where m.curriculo = ?"
            + " order by periodo,posicao"),
    @NamedQuery(name = "Materia.recuperaMateriasPorCurso",
    query = " select m from Materia m "
            + " where m.curriculo = ? "
            + " order by periodo,posicao"),
    @NamedQuery(name = "Materia.recuperaPorPosicaoPeriodoECurriculo",
    query = " select m from Materia m "
            + " where m.periodo = ? and m.posicao = ? and m.curriculo = ? "),
    @NamedQuery(name = "Materia.recuperaTodosPorPeriodoCurriculo",
    query = " select m from Materia m "
            + " where m.periodo <= ? and m.curriculo = ? and m != ? ")
})
```

Figura 33 – Exemplo de *NamedQuerys* da classe Materia.

3.1.4 Representação dos dados

A estrutura de dados abordada no projeto, apesar de representada em forma de Matriz de *Vertex's* (Matérias), na realidade fora usada vetores, e esses ordenados de acordo com período da disciplina. O preenchimento do vetor, que no caso é um *ArrayList* do Java, foi realizado conforme a informação retornava na consulta em HQL, um simples '*select*' ordenando as matérias, e cada matéria desse vetor possui outro vetor de pré-requisitos. Com todas essas informações foi possível criar um *graph*, ou seja, o objeto contendo um conjunto

de *vertex's* e de *edges* que são componentes do *framework*, JgraphX.

Na implementação são dois os métodos que geram o objeto *graph*, eles se encontram no '*GraficoService'*, uma camada estruturada para conter todos os dados referêntes ao gráfico, todas as constantes, definições de estilos e onde é feita a entrada do *ArrayList* contendo todos os dados necessários. Esses dois métodos que recebem essa lista por parâmetro são chamados de "gerarFluxograma" e "gerarFluxogramaDoAluno", o primeiro gera somente o fluxograma de um determinado curso, o segundo o do aluno, exibindo matérias já concluídas com cores diferentes.

Foram criadas algumas constantes na classe *GraficoService*, melhorando as formas de adaptar o código e claro a sua visualização, foi definido padrões de estilos para as arestas, as setas que ligam as matérias entre si e do próprio *vertex*, seu tamanho, dimensão e posicionamento, como foi visto na parte a respeito do *framework*, o objeto *graph* chama seus métodos e define todas as suas configurações.

Um dos métodos do objeto *graph* do tipo *MxGraph* de longe o mais utilizado no projeto é o *graph.insertVertex*() e o *graph.insertEdge*() que são os métodos fundamentais. O *insertVertex* cria o nó que representa nossa matéria no fluxograma, e o *insertEdge* liga um vertex ao outro através de seus parâmetros, assim como a definição de seu estilo.

Exemplos inserindo *vertex* e *edges*:

mxGraph.insertVertex(pai, id, valor, x, y, largura, altura, estilo)

mxGraph.insertEdge(pai, id, valor, origem, destino, estilo)

Exemplo de *insertVertex* na implementação:

graph.insertVertex(*parent*, *null*, *materia.getNome*(), calculaX(*materia.getPeriodo*()), calculaY(*materia.getPosicao*()), largura, altura);

O *insertVertex* possui seu 'id' identificador, seu conteúdo 'valor', seus valores x e y para posicionar o *vertex* no lugar correspondente e suas dimensões de largura e altura, esses valores são definidos respectivamente pelo período, posição no *ArrayList* e pelas constantes de tamanho definidas no código.

Exemplo de *insertEdge* na implementação:

graph.insertEdge(parent, null, "", requisitoCorrente, materiaCorrente, "estiloCorrente");

O *insertEdge* possui o 'id' identificador assim como *insertVertex*, um valor de conteúdo que também pode ser escrito na linha, seus parâmetros de direção e sentido da seta, e um parâmetro para se definir seu estilo.

Com todo esse processo usando *graph*, é possível finalmente reproduzir um simples fluxograma usando o framework.

3.2 Aplicação do JavaServer Faces

Para implementação do projeto, foi desenvolvido um pequeno sistema Web para exibir os resultados obtidos da pesquisa. Essa aplicação aplica na *view* componentes em JSF.

O exemplo mostrado da figura 34 se refere a uma tela simples de cadastro de um Curso, a *tag* <ui: composition> está definindo a parte do corpo no *Template* do sistema, o qual está dividido em *menu*, título e corpo.

```
10  <ui:composition template="/WEB-INF/template/templateExterno.xhtml">
11
12      <ui:define name="tituloPagina">
13          Cursos
14      </ui:define>
15
16      <ui:define name="conteudo">
17
18          <h:form>
19              <h:panelGrid columns="2">
20                  <h:outputLabel value="Nome:" />
21                  <h:inputText value="#{CursoMB.cursoCorrente.nome}"/>
22                  <h:outputLabel value="Codigo:" />
23                  <h:inputText value="#{CursoMB.cursoCorrente.codigo}"/>
24                  <h:outputLabel value="Curriculo:" />
25                  <h:inputText value="#{CursoMB.curriculoCorrente.codigo}"/>
26              </h:panelGrid>
27
28              <h:commandButton value="Salvar" action="#{CursoMB.salvar}"/>
29              <h:commandLink value="Voltar" action="#{UsuarioMB.irParaPaginaDeAdministrarCursos}"/>
30          </h:form>
31
32
33      </ui:define>
34  </ui:composition>
35 </html>
```

Figura 34 - Tela de cadastro de curso usando JSF.

A *Tag* <ui: define> se refere ao título desse corpo e seu conteúdo. O <h:form> representa o escopo do formulário, definindo as entradas de dados e seus botões que emitem uma ação.

O <h:panelGrade> está modelando a forma o qual os componentes aparecerão na tela, nesse acaso alinhado em 2 colunas.

O <h:outputLabel> representa somente uma String antes de cada <h:inputText>, onde ocorre a entrada dos dados.

E para finalizar os componentes de botão e link <h:commandButon> e <h:CommandLink> que realizam ações ou métodos de algum controlador.

Esses foram os principais componentes JSF usados para desenvolver a aplicação e divulgar os resultados obtidos do projeto.

3.3 EXPORTANDO GRAFICOS DO JGRAPHX

3.3.1 Exportando para um objeto BufferedImage

Estudando exemplos de aplicações usando *JgraphX*, foi feita uma análise de formas como representar o diagrama reproduzido em forma de imagem.

A partir de exemplos oferecidos do *JgraphX* e fóruns da biblioteca, comprovou-se que o framework disponibiliza formas de exportar seu gráfico, tanto para *Portable Document Format* (PDF), como para um Objeto *BufferedImage*, que pertence a Classe *Image* do Java.

Como muitos frameworks de imagem, o *JgraphX* não ficou para trás, disponibilizou uma forma de construir o gráfico gerado mesmo em modo swing para um objeto através da classe "*com.mxgraph.util.mxCellRenderer*" da biblioteca, onde se encontra o método "*createBufferedImage*" que executa a operação.

3.3.2 Exportando para um PDF

Já para exportar para um PDF, o tratamento é feito de outra forma, sendo necessário importar no projeto as classes *java.awt.Graphics2D* para criar o desenho e "*java.io.FileOutputStream*" para a criação do arquivo, uma operação de entrada e saída.

Também é necessário usar o *iText*, uma biblioteca que permite criar e manipular documentos em PDF.

A figura 35 representa o resultado obtido graficamente ao exportar para PDF.

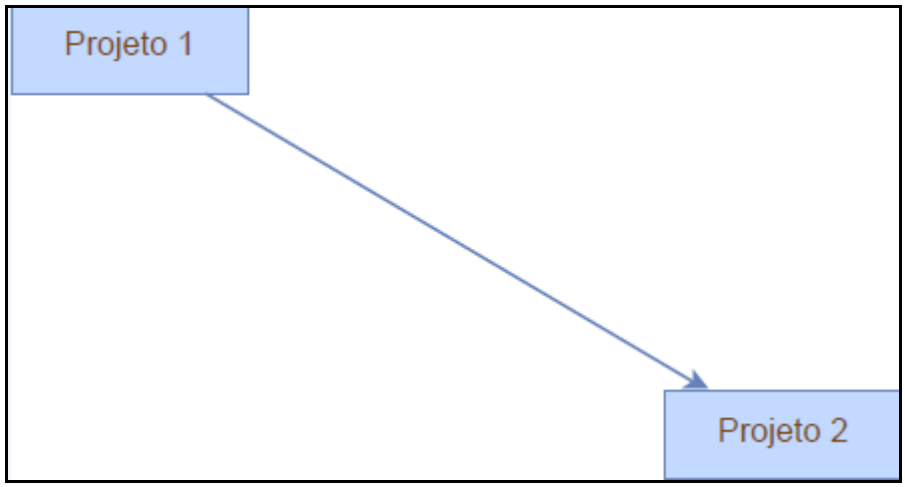

Figura 35 – Exemplo de PDF gerado pelo JgraphX.

Apesar de ser simples a forma de exportar para PDF, o foco da implementação é exibir o fluxograma na própria página do sistema em um formato de imagem e estática. Para isso foi feito uma nova pesquisa buscando formas de usar esse objeto *BufferedImage*, e resultou primeiramente em um estudo sobre applets e o seus recursos.

3.4 USANDO O APPLET

3.4.1 Configurando o Applet

Como visto, o JgraphX disponibiliza uma forma de exportar para um objeto o conteúdo gerado no *graph*, logo quando relacionado a applets foi utilizado o método paint(), que permite o uso do objeto g de Graphics que disponibiliza diversas funções para desenhar figuras geométricas como quadrados, retângulos, círculos, linhas, polígonos, além de Strings e figuras já construídas por outra aplicação ou *framework*.

Dessa forma foi possível reproduzir em um *Applet* o conteúdo do *Bufferedimage* previamente gerado, o objeto g possui um método chamado *g.drawImage*(imagem, x, y, *this*), onde é possível controlar a área de visualização inserindo os valores nos parâmetros x e y, e o objeto *BufferedImage* no primeiro parâmetro nomeado de 'imagem' no código da figura X.

A figura 36 é um exemplo que foi aplicado de um simples *Applet* para um teste, note que é necessário estender a classe para *Applet*. Nessa pequena demonstração foi usado o método criar() que não gera nada mais que um fluxograma estático contendo duas matérias. Esse método retorna no objeto 'imagem' um *Buffered* da imagem formada, usando o objeto g, esse objeto é passado por parâmetro, juntamente com as especificações de altura e largura.

```
1   package teste;
2
3   import ...
12
13  public class TesteApplet1 extends Applet {
14
15      private static final long serialVersionUID = -2707712944901661771L;
16      private static final int altura = 60;
17      private static final int largura = 100;
18
19      @Override
20      public void paint(Graphics g) {
21          BufferedImage imagem = criar();
22          g.drawImage(imagem, 20, 20, this);
23      }
24
25      public BufferedImage criar() {...}
54  }
55
```

Figura 36 – *Applet* usando o framework *JgraphX*.

Executando a classe teremos como resultado a imagem gerada dentro de um *Applet*, como na tela da figura 37.

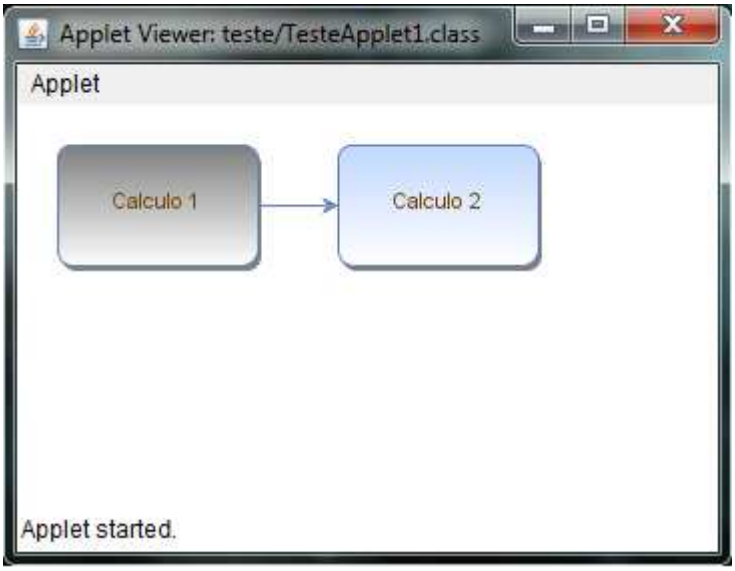

Figura 37 - *Applet* exibindo imagem gerada pelo *JgraphX*.

Agora que a imagem do *Buffered* está sendo exibida na forma de *Applet*, é possível gerar um fluxograma em página de Web, através do contêiner de *applets* controlado pelo navegador.

Para fazer a chamada do *Applet*, é necessário primeiramente executá-lo ou seja compila-lo. Compilando-se o *Applet*, é gerado um arquivo com extensão .jar, esse arquivo deve ser inserido em alguma pasta dentro da aplicação Web junto também com as suas bibliotecas, de preferência uma pasta nova, para organizar os *applets* dentro da aplicação.

Para fazer a chamada do *applet* na *View*, ou seja, na tela em JSF, é necessário inserir o trecho em HTML da figura 38.

```
<f:view>
    <f:verbatim>
        <applet codebase="applet/"
                width="100%"
                height="100%"
                code="teste/TesteApplet1.class"
                archive="mxgraph-all.jar,mxgraph-core.jar,mxgraph-swing.jar">
        </applet>
    </f:verbatim>
</f:view>
```

Figura 38 - Exemplo de *view* fazendo chamada de *applet*.

Note que é feito a chamada do *Applet* dentro da pasta "teste", como mostra o parâmetro '*code'*, e de suas bibliotecas pelo parâmetro '*archive'*, são elas, *mxgraph-all.jar*, *mxgraph-core.jar* e *mxgraph-swing.jar*.

Com isso monta-se a imagem do pequeno fluxograma na aplicação Web. Porém isso não é tudo, será necessário executar o *applet*, passando os parâmetros necessários com as informações recolhidas do banco de dados do aluno.

3.4.2 Parâmetros via *Applet*

No escopo do *applet*, na *view*, é possível inserir parâmetros em formato de String como no exemplo da figura 39.

```
<applet code=TesteApplet1.class width=360 height=40 ="Applet de mensagem">
    <param name=frase value="Teste de String!">
</applet>
```

Figura 39 - Exemplo de *view* fazendo chamada de *applet* com passagem de parâmetro.

Usando essa função é feita a passagem do valor "Teste de String!" para dentro do *Applet* através do parâmetro '*name'*. No caso do Projeto, pode-se passar as informações a

respeito das máterias do aluno, seus requesitos, seus respectivos períodos, posição no fluxograma e seu estado curricular atual, ou seja, se a matéria já foi concluída ou não.

Dentro da aplicação *Applet*, esses parâmetros são recebidos como segue o trecho de código da figura 40.

```
import java.applet.*;
import java.awt.*;
public class TesteApplet1 extends Applet {
    String msg;
    public void paint(Graphics g) {
        msg = getParam("frase");
        g.drawString(msg, 100, 20);
    }
}
```

Figura 40 - Exemplo de aplicação *applet* recebendo parâmetros.

Nesse código possui a String ‘msg’ que recebe o determinado valor chamando o método *getParam*("frase"), após isso essa informação pode ser modelada da forma que o programador desejar.

No caso do Fluxograma do aluno, seria necessário um grande esforço e organização para lidar com esse tipo de procedimento, uma quantidade muito grande de informação teria que ser modelada em pura String, comprometendo a qualidade e o desempenho do código.

Outra solução seria criar toda a configuração que existe do sistema Web no *applet*, ou seja, todo o padrão MVC, o controle de transação com banco, e o mapeamento do sistema com o Hibernate. Seria necessário fugir do padrão de um *Applet*. A funcionalidade de um *Applet* deve ser mais simples, seria um erro lidar com esse tipo de projeto, o custo de manutenção seria muito exaustivo.

A solução real foi descartar o uso do *Applet* e pesquisar outra forma de se lidar com a imagem gerada pelo *JgraphX*.

3.5 USANDO O RICH FACES

3.5.1 Configurando o RICH FACES

Primeiramente é necessário fazer o download do *RichFaces* e adicionar as bibliotecas "*richfaces-3.0.x.jar*", "*ajax4jsf-1.1.0*" e "*oscache2.3.jar*" na pasta WEB-INF/lib da aplicação.

Algumas configurações são necessárias no arquivo web.xml do projeto como mostrado na figura 41:

```
<context-param>
   <param-name>org.ajax4jsf.SKIN</param-name>
   <param-value>blueSky</param-value>
</context-param>
<filter>
   <display-name>Ajax4jsf Filter</display-name>
   <filter-name>ajax4jsf</filter-name>
   <filter-class>org.ajax4jsf.Filter</filter-class>
</filter>
<filter-mapping>
   <filter-name>ajax4jsf</filter-name>
    <servlet-name>Faces Servlet</servlet-name>
    <dispatcher>REQUEST</dispatcher>
    <dispatcher>FORWARD</dispatcher>
    <dispatcher>INCLUDE</dispatcher>
</filter-mapping>
```

Figura 41 - Configurando rich-faces.

No início das páginas em xhtml do projeto foram adicionadas *tags*, como uma simples importação para o uso da biblioteca <xmlns:rich="http://richfaces.ajax4jsf.org/rich">

3.5.2 O Componente PAINT2D

No projeto foi implementado no Controlador de Gráficos (GraficoMB) o método para reproduzir a imagem referenciada pelo componente do *RichFaces*, *Paint2D*.

A imagem 42 mostra dois métodos, o primeiro foi criado para gerar somente o fluxograma de um determinado curso. Já o segundo além de gerar o fluxograma do curso do aluno, mostra também as matérias já cursadas, mas isso não vem ao caso no momento. Ao fazer a chamada desses métodos, notam-se as variáveis 'imagem' e 'imagemDoAluno', essas já foram preenchidas pelo *framework*, sendo assim o objeto em questão é um objeto *BufferedImage* já preenchido no momento em que é chamado na implementação.

```
55
56      /* Imprime na tela o buffer de imagem corrente */
57      public void desenharFluxograma(Graphics2D g2d, Object obj) {
58          g2d.drawImage(imagem, null, 0, 0);
59      }
60
61      /* Imprime na tela o buffer de imagem do aluno corrente */
62      public void desenharFluxogramaDoAluno(Graphics2D g2d, Object obj) {
63          g2d.drawImage(imagemDoAluno, null, 0, 0);
64      }
```

Figura 42 - Principais métodos da aplicação que se comunica com a *View*.

O *g2d* do tipo *Graphics2D* oferece alguns métodos que passam o *BufferedImage* por parâmetro para ser exibido em tela, junto com outros parâmetros de altura e largura, que nesse caso estão setados em 'zero', ignorando a restrição, ou seja, o tamanho com que a imagem será tratada é configurado na própria *view*. Por questões de praticidade foi definido adotar esse padrão ao projeto.

Enfim é feita a chamada na *view*, para a exibição da imagem na tela como prossegue na figura 43.

```
<s:div  rendered="#{GraficoMB.imagem ne null}">
    <fieldset><legend>Fluxograma</legend>
        <rich:paint2D id="painter"
                      width="935"
                      height="600"
                      format="png"
                      paint="#{GraficoMB.desenharFluxograma}" />
    </fieldset>
</s:div>
```

Figura 43 - *View* usando o componente *paint2D* e se comunicando com controlador GraficoMB.

O conteúdo dentro do componente <*s:div*> só será exibida se o valor da variável 'imagem' dentro do controlador GraficoMB não estiver nula. A *tag* <rich:paint2D> é chamada com seus parâmetros *id*, *width* e *height* definindo sua largura e altura, e o formato da imagem *format*, este definido em '*png'*. Por final o atributo *paint* que chama o método do controlador *GraficoMB*. Esse controlador coletará a imagem no Objeto *BufferedImage* 'imagem' já preenchido e assim obtendo o resultado final.

A figura 44 representa um fluxograma ainda incompleto gerado pelo projeto**.**

Figura 44 - Fluxograma parcialmente completo gerado pelo *JgraphX*.

4 RESULTADOS

4.1 SIMULADOR

O sistema desenvolvido no projeto simula a geração do fluxograma do aluno a partir de uma base de dados, possibilitando a criação de um curso, suas matérias e alunos fictícios. Além disso, pode-se simular também o histórico de um aluno, alterando o estado das matérias em concluídas ou não.

O objetivo principal desse sistema foi acompanhar os resultados da implementação do projeto.

A figura 45 representa a tela de cadastro do curso do link “Cursos”. Contendo apenas nome, código e número do currículo.

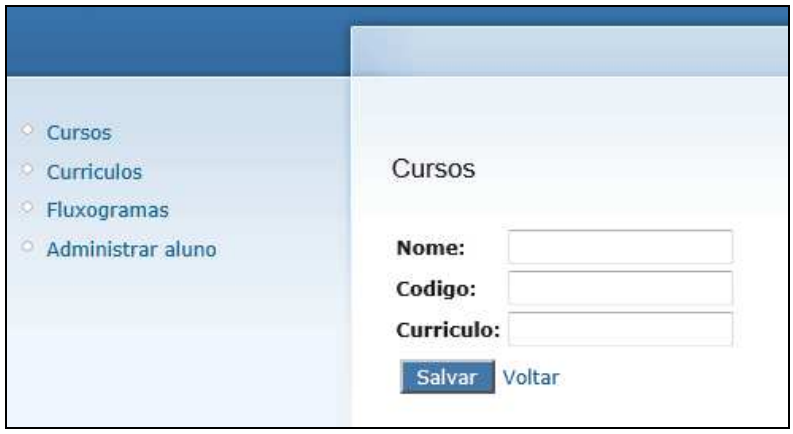

Figura 45 - Tela para cadastro de curso.

Na figura 46 representa o link “Currículos” o usuário do sistema pode cadastrar novas disciplinas para um determinado curso, além de associar seus requisitos, informar seu período e posição no fluxograma.

Figura 46 - Tela para administrar disciplinas de um curso.

No link "Fluxogramas" pode-se visualizar o fluxograma de cada Curso cadastrado como exibido na imagem 47.

Figura 47 - Tela para visualizar graficamente o fluxograma de um curso.

No link “Administrar aluno” é possível cadastrar novos alunos além de poder administrar suas matérias e exibir o fluxograma corrente como nas figuras 48, 49 e 50.

A figura 50 nota-se a diferença de cores das matérias já ‘realizadas’ ou seja, as disciplinas selecionadas da tela da figura 49. Simulando assim o fluxograma do aluno de forma dinâmica.

Figura 48 - Tela para administrar alunos fictícios.

Figura 49 - Tela para controlar as matérias do aluno.

Figura 50 - Tela para exibir fluxograma de um aluno específico.

4.2 PROBLEMAS ENCONTRADOS

Com todo o fluxograma completo e usando o estilo ortogonal é possível notar a dificuldade de compreender as ligações entre as matérias. As *Edge's* (ligações) passam uma por cima da outra ou por trás de uma matéria como demonstrado na figura 51.

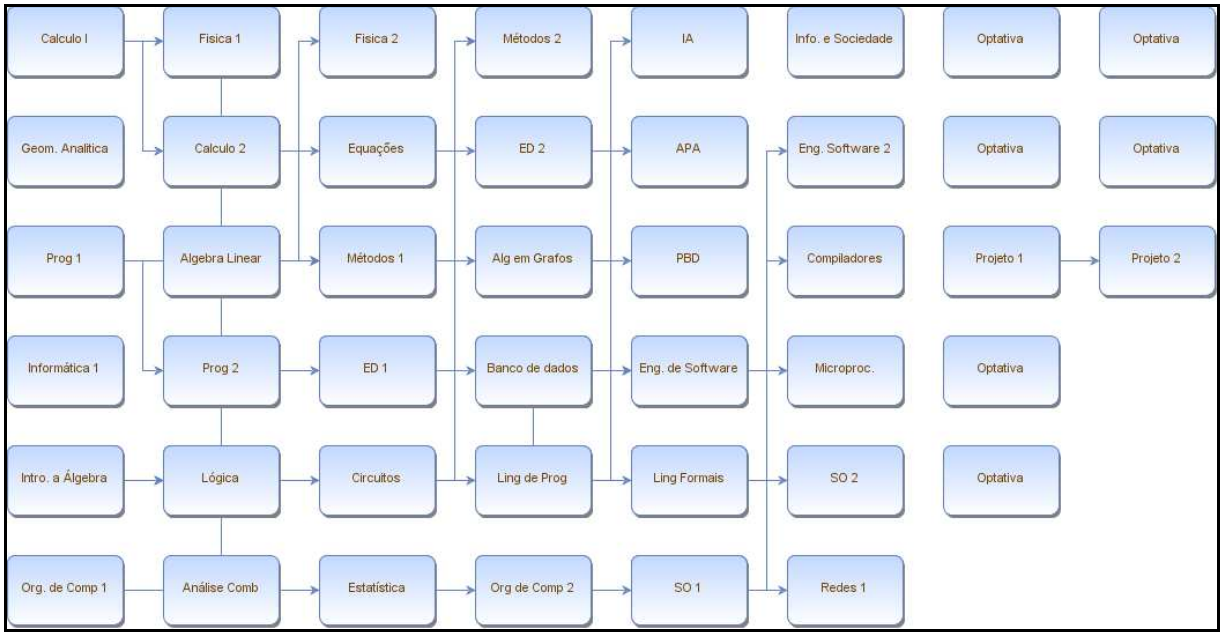

Figura 51 - Fluxograma usando estilo ortogonal para cada seta 'Edge'.

Uma forma de suavizar o formato ortogonal das *Edge's* foi o uso do estilo padrão delas, *getStylesheet().getDefaultEdgeStyle()*. Porém ainda assim não soluciona a colisão entre as *Edge's* e nem a passagem delas por baixo dos *Vertex's*. A figura 52 representa as esse padrão de *layout* para as ligações.

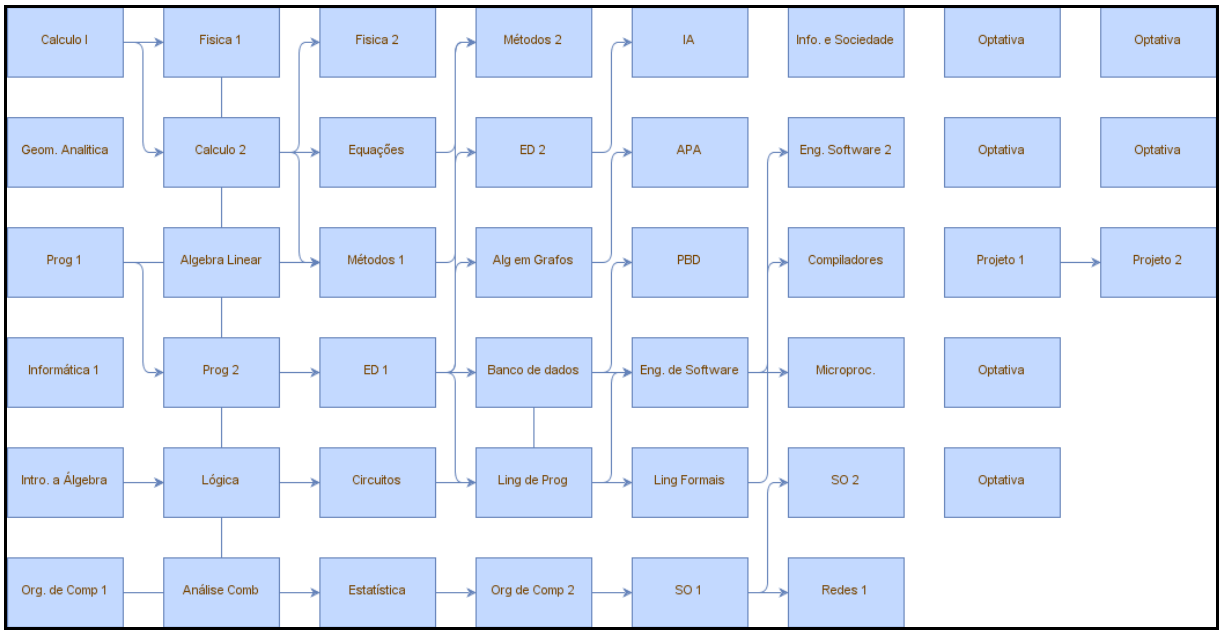

Figura 52 - Fluxograma usando estilo mais suave para cada seta (*Edge*).

Abandonando o estilo das *edge's* é possível solucionar o problema delas se colidirem, ou seja, passarem uma por cima da outra. A figura 53 mostra esse resultado, como é possível ver, ficou mais legível compreender quais os requisitos de cada matérias, mas não foi solucionado ainda o problema quanto a passagem das ligações por trás de cada Nó.

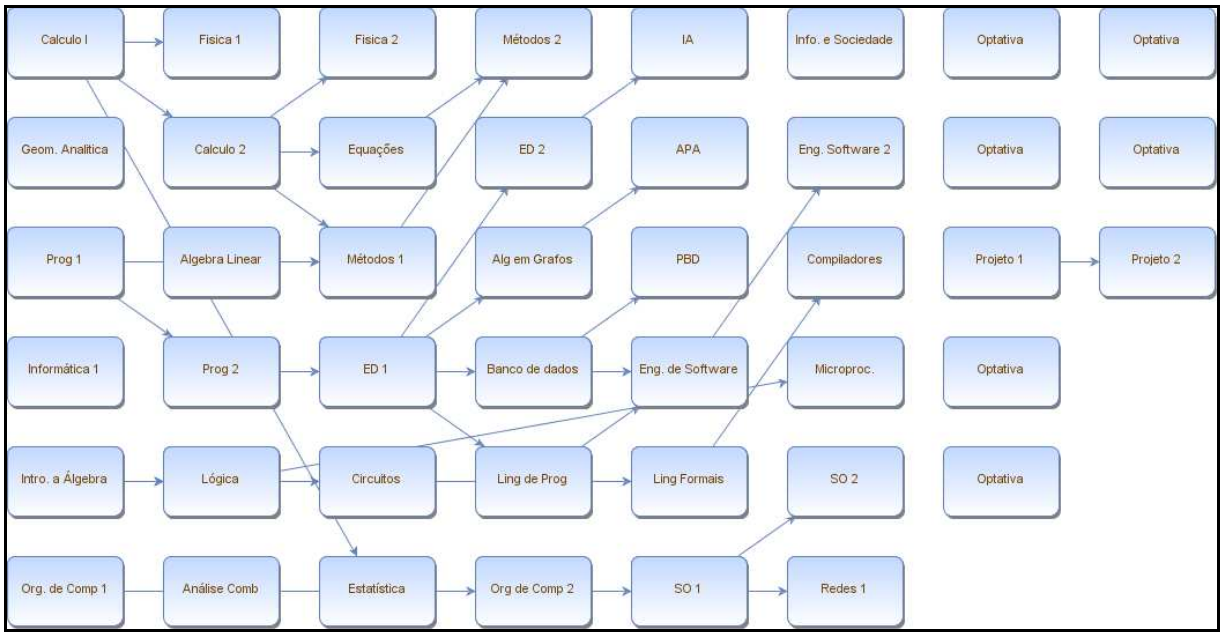

Figura 53 - Desenhando o fluxograma sem nenhum estilo para as Edge's.

4.3 SOLUÇOES E RESULTADOS FINAIS

A solução foi desenvolver separadamente o desenho das arestas separadamente e em função das dimensões dos *vertex's*. Manipulando nós nulos nas posições finais de cada bloco. A figura 54 representa essa primeira etapa do processo. Com isso resolve-se os problemas com os caminhos das *Edge's* que passavam por cima uma da outra e soluciona a falta de organização com que elas saiam de suas origens conforme na figura 54.

Na segunda etapa da construção do fluxograma adiciona-se cada *vertex*, representando as matérias do curso. O diferencial aqui é a opacidade dada à eles. A opacidade é usada de forma funcional, com ela é possível solucionar a confusão que ficava quando as setas passavam por baixo das matérias ao longo de sua trajetória. Exemplo na figura 55.

A figura 56 e 57 representam respectivamente os resultados finais obtidos do fluxograma de um curso e do estado curricular de um aluno do curso de Ciência da Computação, com todos os nomes sem abreviações.

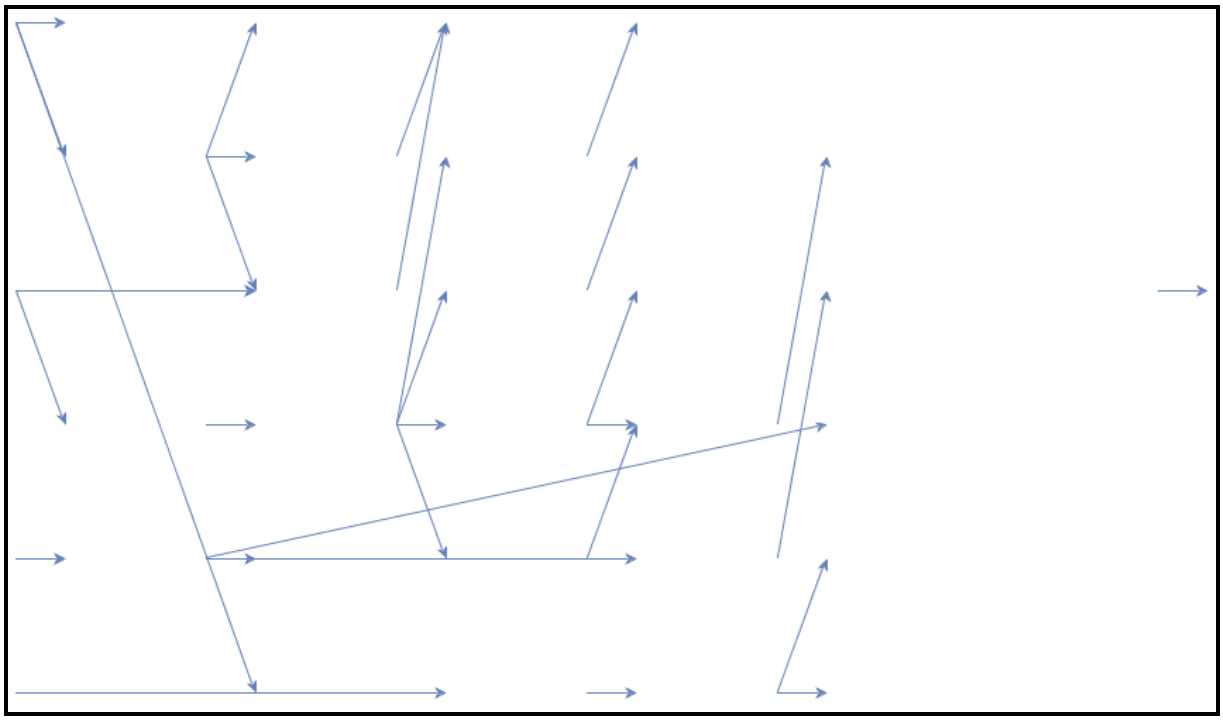

Figura 54 - Desenhando as ligações a partir de pontos específicos.

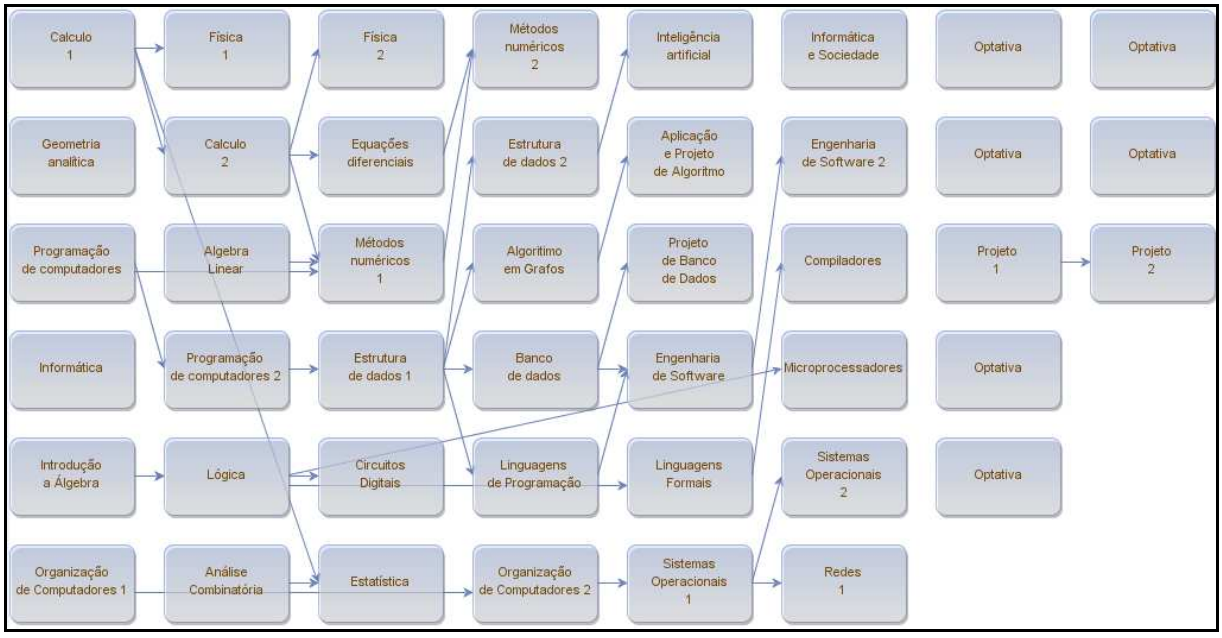

Figura 55 - Desenhando os Vertex's usando opacidade.

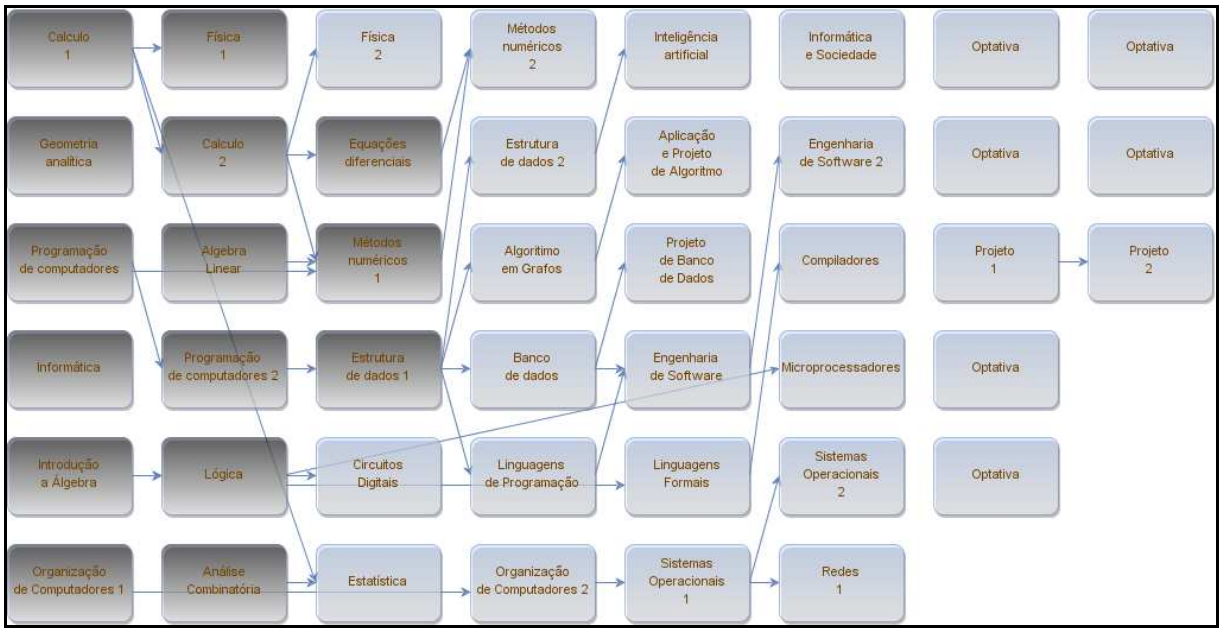

Figura 56 - Exemplo do fluxograma completo de um aluno.

5 CONCLUSÕES

Neste projeto foi desenvolvida uma forma de representar visualmente o fluxograma de um curso da graduação e também o histórico de um aluno em um sistema Web como o IdUFF. O IdUFF é atualmente o sistema acadêmico da graduação, responsável pelo módulo de inscrição online, da Universidade Federal Fluminense.

A implementação do projeto foi conduzida pelo estudo do *JgraphX*, uma biblioteca gráfica para *Java-Swing* capaz de reproduzir diversos formatos de grafos organizacionais e que, ao longo do desenvolvimento, foi adaptada para executar em uma aplicação Web.

Algumas limitações do *framework JgraphX* foram analisadas durante o desenvolvimento. A representação dos caminhos das setas entre as matérias confundiam o usuário questionando o potêncial da biblioteca. Para contornar esses problemas, o código foi desenvolvido de forma que cada problema encontrado na imagem possa ser tratado de forma genérica. Outros meios estudados foram os tipos de *layouts* do *framework*, a possibilidade de gerar a imagem com opacidade contribuiu para o atual estado do projeto.

Com um grande potencial de contribuição ao módulo de inscrição do sistema UFF, este projeto foi implementado com base nas mesmas ferramentas, linguagens e frameworks, de forma que a adaptação ao sistema real possa ser feita de forma ágil e também customizada.

5.1 PERSPECTIVAS FUTURAS

Em uma pequena apresentação feita ao grupo de desenvolvimento da Superintendência de Tecnologia da Informação (STI), foi comentado que a implementação deste projeto fará um diferencial tanto para o órgão responsável pelo módulo de currículo, quanto para as consultas de alunos da graduação que poderão planejar e organizar seu plano de estudo ao longo da faculdade, dado seu grande impacto visual.

Por fim, a implementação do projeto deverá abrir um grande conjunto de oportunidades de análise e estudo. Ainda é possível desenvolver algoritmos mais eficientes, capazes de identificar requisitos de matérias e traçar caminhos mais diversificados. Também pode-se desenvolver novos *layout*, definir novas formas de representação do fluxograma, representar matérias-candidatas durante a inscrição do aluno ou mesmo o número de vezes em que a matéria foi cursada.

6 REFERÊNCIAS BIBLIOGRÁFICAS

[01] JSF/AJAX DEMOS WITH FACELETS VIEWS. Disponível em: <http://www.apprisant.com/jsfdemos/>. Acesso em: 25 de outubro de 2010

[02] HORSTMANN, C. **Padrões e Projeto Orientados a Objetos**. 2 Edição. , Editora Bookman, 2008. ISBN8560031510.

[03] JAVASCRIPT DIAGRAM COMPONENT. Disponível em <http://www.jgraph.com/mxgraph.html>. Acesso em: 25 de outubro de 2010.

[04] GRAPH VISUALIZATION SOFTWARE. Disponível em <http://www.graphviz.org>. Acesso em: 23 de outubro de 2010.

[05] GAMMA, E. et al. **Padroes de Projeto**. 3 edição. Editora Bookman, 2007. ISBN 9788573076103.

[06] DEITEL, H. M.; DEITEL, P. J. **Java como Programar**. 6 edição. São Paulo: Editora Pearson Prentice Hall. 2005. ISBN 8576050196.

[07] DESIGN AND IMPLEMENTION OF THE JGRAPH SWING COMPONENT. Disponível em <http://www.jgraph.com/downloads/jgraph/legacy/jgraph-paper.pdf>. Acesso em: 14 de janeiro de 2011.

[08] ORGANOGRAMA. Disponível em <http://pt.wikipedia.org/wiki/Organograma>. Acesso em: 04 de julho de 2011.

[09] TELECKEN, T. **Definição de Processos de Workflow.** Disponível em <http://www.arquivar.com.br/espaco_profissional/sala_leitura/artigos/Definicao_de_processos_de_workflow.pdf>. Acesso em: 04 de julho de 2011.

[10] BENSON, D.; ALDER, G. **JGraphx (JGraph 6) User Manual**. Disponível em <http://www.jgraph.com/doc/mxgraph/index_javavis.html>.2011. Acesso em: 08 de agosto de 2011.

[11] MILANI, A. **MySQL - Guia do Programador**. 1 edição. Editora Novatec. 2007. ISBN 8575221035.

[12] PILATO, C. M.; COLLINS-SUSSMAN, B. C.; FITZPATRICK, B. W. **Version Control with Subversion.** 1 edição. Editora O'Reilly. Disponível em <http://svnbook.red-bean.com/en/1.4/svn-book.pdf>. Acesso em: 11 de agosto de 2011.

[13] PRIMO, I.; SANTOS, S. Desenvolvendo com Hibernate. **Java Magazine.** Numero 73. 2010. ISSN: 16768361. Disponível em <http://www.devmedia.com.br/articles/viewcomp_forprint.asp?comp=14756>. Acesso em: 06 de agosto de 2011.

[14] KING, G. et al. **Documentação de Referência Hibernate**. Disponível em <http://docs.jboss.org/hibernate/core/3.6/reference/pt-BR/html/index.html>. Acesso em: 06 de agosto de 2011.

[15] HIBERNATE MAPPING Mapping Many-to-Many. Disponível em <http://www.vaannila.com/hibernate/hibernate-example/hibernate-mapping-many-to-many-1.html>. Acesso em: 08 agosto de 2011.

[16] LINHARES, M. **Introdução ao Hibernate 3.** Disponível em <http://www.guj.com.br/content/articles/hibernate/intruducao_hibernate3_guj.pdf>. Acesso em: 10 agosto de 2011.

[17]SACRAMENTO, W. M. **Introdução à Java Persistence API – JPA.** Disponível em <http://www.devmedia.com.br/articles/viewcomp.asp?comp=4590>. Acesso em: 10 de agosto de 2011.

[18]BELLA, R. Uma Aplicação Java EE Completa. **Java Magazine**. Numero 44. Mensal. 2007 ISSN: 16768361.

[19] VISÃO GERAL DO JAVA SWING. Disponível em <http://wiki.sintectus.com/bin/view/GrupoJava/LicaoVisaoGeralDoSwing>. Acesso em: 10 de agosto de 2011.

[20] CAMPOS, P. **Introdução ao Java Swing e AWT. D**isponível em <http://cee.uma.pt/people/faculty/pedrocampos/docs/guia-IHM.pdf>. Acesso em: 10 de agosto de 2011.

6.1 OBRAS CONSULTADAS

JGRAPH The leading Java Graph Drawing Component. Disponível em: <http://www.jgraph.com/>. Acesso em: 25 de outubro de 2010.

SAUVÉ, J. P. **Java Server Faces. D**isponível em <http://www.dsc.ufcg.edu.br/~jacques/cursos/daca/html/jsf/jsf.htm>. Acesso em: 08 agosto de 2011.

FERNANDES, R. G.; LIMA, G. F. **Hibernate com Anotações.** Disponível em <http://wiki.futurepages.org/lib/exe/fetch.php?media=quickstart:hibernate_anotacoes.pdf>.

Acesso em: 08 agosto de 2011.

NAKASHIMA, C. **Hibernate**. Disponível em <http://www.dainf.ct.utfpr.edu.br/~caio/hibernate/index.html>. Acesso em: 10 de agosto de 2011.

MONOGRAFIA regras da ABNT. Disponível em <http://www.monografia.net/index.html>. Acesso em: 02 de dezembro de 2010.

TRABALHOS ACEDÊMICOS: Normas da ABNT. Disponível em <http://www.firb.br/abntmonograf.htm>. Acesso em: 22 agosto de 2011.