Universidade Federal de Minas Gerais
Instituto de Ciências Exatas
Departamento de Ciência da Computação

Roberta Coeli Neves Moreira
Orientadora: Profa. Dra. Mariza Andrade da Silva Bigonha
Coorientadora: Profa. Dra. Kecia Aline Marques Ferreira

Relatório Técnico-Científico de Iniciação Científica

Projeto: K3B – Experimentos em Larga Escala usando CONNECTA

Belo Horizonte – MG
2011 / 1º semestre

Sumário

1. Introdução

O presente relatório apresenta a descrição das atividades realizadas pela autora, aluna de graduação do curso de Engenharia de Computação do Centro Federal de Educação Tecnológica de Minas Gerais (CEFET-MG), em seus trabalhos de Iniciação Científica no período de março de 2011 a agosto de 2011, sob orientação da professora Dra. Mariza Andrade da Silva Bigonha e coorientação da professora Dra. Kecia Aline Marques Ferreira.

O trabalho realizado consistiu em duas etapas: na primeira, foi feito um estudo mais aprofundado do modelo *Little House*, o qual é a base principal utilizada por Ferreira (1) para descrever a estrutura macroscópica genérica das redes de *software*; na segunda etapa, por sua vez, foram realizadas modificações e aprimoramentos na ferramenta *Connecta*, desenvolvida por Ferreira em seu Mestrado e Doutorado com a finalidade de avaliar *softwares* em Java. *Connecta* analisa arquivos de *bytecodes*, coletando métricas para avaliação da coesão, estabilidade e conectividade dos programas, a fim de permitir que os desenvolvedores possam construir *softwares* com maior grau de manutenibilidade.

O presente relatório visa a apresentar, em cada uma das próximas seções, os conceitos relacionados aos trabalhos desenvolvidos, os procedimentos realizados, os resultados obtidos e a importância dos mesmos no estudo.

Seção 2 apresenta as ferramentas disponíveis para cálculo de métricas de *software*, cuja análise permitiu realizar um estudo comparativo entre as métricas de coesão LCOM, TCC e COR. Estas métricas são de grande importância no estudo de *softwares* orientados a objetos, auxiliando na detecção de problemas estruturais nas classes que constituem um programa.

Seção 3 contempla o estudo do modelo *Little House* por intermédio da realização de testes em *softwares* Java, nos quais se classificou cada uma das classes presentes quanto a sua funcionalidade geral no programa. Este estudo teve como finalidade verificar a existência de um padrão de classes em cada componente da estrutura proposta, analisando-se como cada classe se agrupa na estrutura de acordo com sua função.

Seção 4 apresenta a implementação de uma tela de configurações para *Connecta*, a fim de possibilitar ao usuário a escolha pelo cálculo de determinadas métricas, proporcionando resultados mais exatos e a redução no tempo de execução do programa.

Seção 5 descreve a reformulação da interface gráfica do *Connecta*, apresentando os principais recursos utilizados para obter uma aparência mais agradável e familiar ao usuário, de modo que o mesmo possa explorar todas as funcionalidades da ferramenta.

Seção 6, por sua vez, sumariza as atividades realizadas neste período de Iniciação Científica e propõe novos trabalhos que podem contribuir para a melhoria da ferramenta.

2. Ferramentas de Avaliação de Coesão

Métricas de *Software* são padrões de medição relacionados a determinado *software* que permitem avaliar suas características e comportamento. As métricas são de grande utilidade no planejamento do *software*, em que servem de base para estimar o custo e a quantidade de recursos necessários para o desenvolvimento do programa, bem como são extremamente relevantes no processo de manutenção, sendo utilizadas para identificar as partes a serem reparadas e os efeitos dos reparos realizados.

Algumas métricas de avaliação de *softwares* também devem ser levar em consideração o paradigma de programação utilizado. Na orientação a objetos, por exemplo, devem possibilitar avaliar aspectos inerentes a este paradigma, tais como a coesão interna das classes, o acoplamento entre as classes e herança.

As métricas de coesão, tema principal do presente estudo, permitem analisar como os métodos de uma classe estão relacionados entre si. Uma classe é considerada coesa quando a mesma desempenha uma única função. Classes que desempenham uma ou mais funções diferentes apresentam uma coesão muito baixa e devem ser reestruturadas em duas ou mais classes menores. Uma maior coesão é desejável em um *software* OO, uma vez que melhora o encapsulamento, ocultação da informação. Por outro lado, classes pouco coesas indicam estruturação inadequada e alta complexidade do *software*, assim como uma maior propensão a erros.

Algumas das principais métricas de coesão são:

1. LCOM (*Lack of Cohesion of Methods*): é uma medida do número de pares de métodos não conectados em uma classe. Esta métrica representa a diferença entre o número de pares de métodos que não possuem variáveis de instância em comum e o número de pares de métodos que possuem variáveis de instância em comum.
2. TCC (*Tight Class Cohesion*): é uma medida da coesão entre os métodos públicos de uma classe. Esta métrica mede a razão entre o número atual de métodos públicos diretamente conectados dividido pelo número máximo de conexões entre os métodos públicos de uma classe. Dois métodos visíveis são diretamente conectados se eles acessam as mesmas variáveis de instância de uma classe.
3. LCC (*Loose Class Cohesion*): é uma medida relativa do número de métodos públicos diretamente ou indiretamente conectados. Esta métrica mede a razão entre o número de conexões diretas ou indiretas entre os métodos públicos e o número máximo de conexões entre os métodos públicos da classe.

Outra métrica de coesão, proposta nos estudos de Ferreira (1), é a métrica de Coesão de Responsabilidade (COR). A COR indica o número de responsabilidades que determinada classe implementa.

Seu valor é dado por $\frac{1}{r}$, sendo **r** a quantidade de conjuntos disjuntos formados por métodos com relacionamento entre si na classe. Maiores informações sobre esta métrica são encontradas na seção 4 do presente relatório.

Com a finalidade de se realizar um estudo comparativo entre as métricas de coesão, foi necessário pesquisar por outras ferramentas que coletassem as principais métricas deste tipo: LCOM e TCC. A pesquisa realizada nesta atividade contribuiu para o artigo "*Métrica de Coesão de Responsabilidade – A Utilidade da Métrica de Coesão na Identificação de Classes com Problemas Estruturais*"[1], publicado nos anais do X Simpósio Brasileiro de Qualidade de Software (SBQS 2011).

2.1. Metodologia

Para a realização do presente trabalho, primeiramente foram pesquisados *softwares* para coleta das métricas TCC e LCOM, buscando-se uma ferramenta para análise de programas em Java que fosse de fácil uso e instalação. Foi conduzida uma busca tradicional na Internet, utilizando-se como palavra-chave "*software metrics tool*" (do inglês, ferramenta de métricas de *software*), a fim de se encontrar uma lista de programas que atendessem ao requisito básico buscado. Como segundo critério, foi feita uma pesquisa nos *websites* dos *softwares* e artigos relacionados por ferramentas que calculassem LCOM ou TCC em programas Java, apontando-se o nome do programa e o *site* da ferramenta.

2.2. Ferramentas de Coleta de Métricas de Coesão

Aplicando os critérios especificados, foram encontrados 5 *softwares* potenciais para uso:

1. **Metrics 1.3.6**: *plugin* para o IDE Eclipse, de código aberto. Mede várias métricas e detecta ciclos, bem como dependências de tipo.
2. **JHawk**: programa proprietário para coleta de métricas de programas Java. Coleta métricas em 4 diferentes níveis: relativas a métodos, a classes, a pacotes e ao sistema como um todo.
3. **Analyst4j**: disponível tanto como um *plugin* para o IDE Eclipse quanto uma Aplicação RIA (*Internet Rich Application*). Realiza pesquisas, calcula as métricas básicas e gera avaliações para programas em Java.
4. **Chidamber & Kemerer Java Metrics**: é uma ferramenta *open-source* de linha de comando. Permite calcular as métricas C&K para *softwares* OO por meio do processamento de arquivos de

1 FERREIRA, K. A. M. ; BIGONHA, M. A. ; BIGONHA, R. S. ; ALMEIDA, H. C. ; MOREIRA, R. C. N. *Métrica de Coesão de Responsabilidade - A Utilidade de Métrica de Coesão na Identificação de Classes com Problemas Estruturais*. In: X Simpósio Brasileiro de Qualidade de Software, 2011, Curitiba-PR. Proceedings of X Simpósio Brasileiro de Qualidade de Software, 2011

bytecode de Java.

5. **VizzAnalyzer**: é uma ferramenta proprietária de análise de qualidade de *software*. Possibilita ler o código-fonte do *software* analisado e outras especificações, desempenhando as análises necessárias. Possui também um *plugin* gratuito para o IDE Eclipse, VizzMaintenance 2.0, o qual apresenta informações detalhadas a respeito da manutenibilidade das classes, realizando o cálculo das 17 métricas mais frequentemente abordadas na literatura.

Para se escolher a ferramenta a ser empregada na pesquisa, estabeleceu-se outros critérios, de modo a utilizar o *software* gratuito mais completo dentre os listados acima, que calcula todas as métricas de coesão necessárias, é de fácil manipulação e instalação. Assim sendo, foi escolhido o VizzMaintenance 2.0, uma vez que este *plugin,* além de ser leve, gratuito e de fácil uso, é o único que permite calcular as métricas necessárias, TCC e LCOM, bem como LCOM4, uma das variantes da métrica LCOM[2].

2.3. Conclusão

A fim de se obter resultados confiáveis, várias pesquisas estão sendo conduzidas na área de qualidade de *software*. O cálculo das métricas de avaliação de *software* é de grande relevância para os desenvolvedores, uma vez que permite quantificar a qualidade do produto, provendo informações a cerca das características do programa. Dessa forma, os valores obtidos possibilitam o planejamento e organização adequada dos programas, aumentando o grau de manutenibilidade dos mesmos. Mais especificamente, a análise de métricas de coesão pode auxiliar na detecção de problemas estruturais em *softwares* orientados a objetos, possibilitando a reestruturação de suas classes de maneira que cada uma delas apresente apenas uma função específica.

2 A métrica LCOM possui variantes: LCOM1, LCOM2, LCOM3 e LCOM4. Embora realizem os cálculos de modo distinto, todas geram uma medida relacionada à quantidade de métodos não conectados de uma classe.

3. Análise de *Softwares* em Java segundo o Modelo *Little House*

Como mencionado na seção 1 do presente relatório, estudos de Ferreira (1) indicaram que o modelo *Little House* se mostra uma boa representação para a estrutura macroscópica genérica das redes de *software*. O modelo pode ser verificado na Figura 3.1.

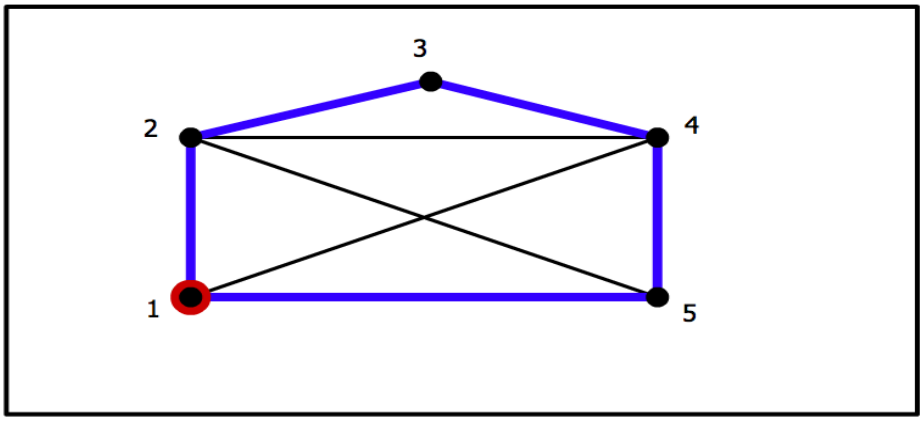

Figura 3.1: Representação do grafo *Little House*, um modelo para a organização da estrutura macroscópica genérica das redes de *software*.

Fonte: http://webspace.ship.edu/deensley/DiscreteMath/flash/ch7/sec7_1/euler.html

Little House consiste em um grafo que se assemelha a uma "casinha", contendo 5 vértices e 8 arestas. Sua estrutura foi verificada pela manipulação do *bow-tie*, denominação dada ao modelo para a estrutura macroscópica da Web, o qual indica que as páginas da Web podem ser divididas em 5 grupos: *LSSC, In, Out, Tendrils, Tubes* e *Disconnected*.

Adaptando-se o modelo para a análise das redes de *softwares*, *LSSC* consiste na parte central do *software*, ao qual todas as demais classes estariam conectadas, direta ou indiretamente. *In* representa o conjunto das classes a partir do qual se estabelecem conexões com os componentes *LSSC*, *Tendrils*, *Tubes* e *Out,* porém nenhuma conexão é direcionada a este componente. Ao contrário de *In, Out* consiste no conjunto de classes para o qual as conexões são direcionadas, embora deste componente não partam conexões. *Tendrils* e *Tubes*, por sua vez, representam grupos de classes que atuam como "intermediários": o componente *Tendril* permite conectar uma classe em *In* a uma classe em *Out* e o componente *Tubes* que estabelece a conexão *Tendril – In – Tendril – Out*. O conjunto de classes *Disconnected*, por sua vez, constitui as classes que não apresentam conexão com nenhuma das demais classes do *software*.

Embora Ferreira (1) tenha explicado o modelo *Little House*, sua análise não contemplou os tipos de classes apresentados em cada conjunto. Assim sendo, esta seção apresenta os estudos experimentais realizados em 5 *softwares* a fim de se verificar os tipos de classes apresentados por cada um dos componentes do modelo.

3.1. Metodologia

Para a realização do presente trabalho, foram utilizados 5 *softwares* em Java: JHotDraw (versão 1.0), JUnit (versão 4.8.1), DBUnit (versão 2.0), um *software* desenvolvido por um aluno iniciante em programação orientada a objetos e o próprio *Connecta*.

Primeiramente, utilizou-se o aplicativo *Pajek*© para desenhar o grafo que corresponde ao modelo *Little House*, utilizando, para isso, os arquivos de extensão .net gerados pelo *Connecta* a partir da análise dos *bytecodes* de cada um dos *softwares*. Além de elaborar o grafo, este aplicativo também permitiu verificar, para cada uma das classes do software, a qual componente cada uma delas pertencia.

Logo em seguida, as classes foram organizadas em uma tabela, a qual continha o nome da classe, tipo da classe e o componente a que tal classe pertencia. Para cada uma das classes, o código-fonte foi estudado, verificando-se qual(is) a(s) funcionalidade(s) a mesma apresentava: interface gráfica, interface para organização de informações, classe abstrata, armazenamento de informações, controle de informações, manipulação de informações, consulta a banco de dados, entre outros.

Por fim, fez-se uma análise comparativa de cada um dos componentes para cada um dos *softwares*, com a finalidade de se verificar se existia um padrão do tipo de classes para cada um dos 5 componentes do *Little House*.

3.2. Classificação das Funcionalidades das Classes

Conforme a análise feita do código-fonte dos *softwares* utilizados no estudo, pode-se classificar as funcionalidades das classes nos seguintes grupos principais:

1. **Interface Gráfica**: consiste na classe que emprega métodos, classes e outras configurações pré-estabelecidas dos pacotes *java.awt* e *javax.swing* para constituir a GUI (*Graphical User Interface*) de um programa Java.
2. **Interface para Organização de Informações:** é uma estrutura utilizada em Java para estabelecer o que as classes devem implementar. É chamada, no estudo, de "interfaces para organização de informações", porque "organizam" os atributos e métodos que as classes que as implementam devem apresentar.
3. **Classe Abstrata**: de modo similar à "interface para organização de informações", esta classe consiste em uma estrutura que serve de modelo para uma classe concreta, não sendo possível instanciar objetos a partir dela. Esta classe permite explorar o recurso de herança em Java, constituindo superclasses das quais as subclasses herdam a implementação.
4. **Classe de Exceção**: a denominação "classe de exceção" foi dada a toda classe que permite gerar

uma exceção no programa. Em Java, esta classe é classe herdeira da superclasse *Exception* e permite detectar possíveis eventos inesperados que podem ocorrer durante a execução do programa.

5. **Classe para Armazenamento de Informações**: consiste na classe que permite armazenar valores numéricos ou caracteres em seus atributos.
6. **Classe para Manipulação de Informações**: classes que manipulam informações são aquelas que recebem os dados armazenados por outras classes e permitem realizar determinadas alterações sobre os mesmos.
7. **Classe para Controle de Informações**: é toda classe que permite avaliar se determinada operação deverá ser executada no programa, de acordo com as configurações selecionadas pelo usuário.
8. **Classe para Consulta a Banco de Dados**: classes deste tipo permitem utilizar recursos do pacote *java.sql* para inserir dados ou realizar consultas em banco de dados no padrão SQL (*Structured Query Language*).
9. **Classe para Entrada e Saída de Dados**: esse tipo de classe possibilita empregar recursos do pacote java.io, controlando as informações que são veiculadas pelos dispositivos de entrada e direcionando os dados apropriados aos periféricos de saída.
10. **Classe de Serviços**: consiste na classe que apresenta métodos estáticos e públicos para utilização por várias classes no programa.
11. **Classe Depositária:** a denominação "classe depositária" foi dada a toda classe que permite armazenar atributos que atuam como constantes no programa.

3.3. Resultados

Ao se analisar as classes presentes em cada componente para todos os 5 *softwares* usados no estudo, verificou-se que os resultados divergiram bastante. No geral, nenhum dos *softwares* testados apresentou classes específicas relacionadas a cada componente, bem como as funcionalidades das classes englobadas em cada grupo não possuíam correspondência. Detalhes sobre cada um dos *softwares* analisados são abordados no restante dessa seção.

3.3.1. JHotDraw

O componente *Disconnected* do *JHotDraw* apresentou predominantemente classes de interface, embora algumas classes distintas, para armazenamento e controle de informações, também estivessem presentes. Uma vez que interfaces são estruturas que um programador Java utiliza como o principal meio de organizar as informações de suas classes e não apresentam implementação, o resultado obtido foi condizente às características do componente no qual estas interfaces estavam inseridas.

A única classe presente no componente *LSSC* consiste em uma classe de interface gráfica. De acordo com suas funcionalidades básicas, aparentemente esta classe é condizente com as características do

componente a qual pertence, uma vez que direciona as principais funções dentro do *JHotDraw*, permitindo a construção de janelas de desenho, a criação de menus-padrão, criação da paleta de ferramentas, criação de novos quadros para desenho, etc.

No componente *In,* todas as classes presentes eram classes de interface gráfica, as quais, diferentemente dos casos anteriores, não apresentavam funcionalidades diretamente relacionadas às funções deste componente.

O componente *Out* agrupou a maioria das classes do software e, assim sendo, várias foram as funcionalidades apresentadas por este componente: armazenamento, controle, transferência e manipulação de informações, classes abstratas e até mesmo classes de interface gráfica.

Assim como o *Out,* os componentes *Tubes* e *Tendrils* também não apresentaram uma funcionalidade específica, possuindo desde classes abstratas e auxiliares a classes de interface gráfica e armazenamento de informações.

3.3.2. *JUnit*

O componente *Disconnected* neste software apresentou, em sua maioria, classes de interface para organização de informações, embora também tenha apresentado algumas classes para armazenamento de informações e classes de exceção. Como estas classes não incluem implementação, atuando somente como "instrumentos" para facilitar a implementação de outras classes no programa, o agrupamento no componente referenciado foi condizente com o esperado.

No componente *LSSC*, a classe presente tem como função o controle de informações no *JUnit*, reportando o programa sobre o progresso dos testes realizados. Como *LSSC* poderia envolver o controle geral do programa, uma vez que é o componente ao qual se conectam todos os demais, a função da classe, embora consista no controle de uma parte específica do programa, não é suficientemente "abrangente" para que a mesma seja englobada neste componente.

No componente *In*, por sua vez, foram agrupadas classes com diferentes funcionalidades, podendo ser: controle, armazenamento e manipulação de informações, classes auxiliares, abstratas e de exceção. Do mesmo modo, os componentes *Out, Tubes* e *Tendrils* não apresentaram uma funcionalidade específica, sendo que muitas classes presentes em cada um deles apresentavam funções coincidentes, por exemplo, classes para armazenamento de informações, estavam presentes em todos os componentes.

3.3.3. *DBUnit*

No componente *Disconnected* havia predominantemente classes de interface para organização de informações, embora também tenha apresentado muitas classes com outras funcionalidades, tais como manipulação de banco de dados, armazenamento de informações, classes de exceção e classes de entrada e

saída de dados.

Tanto em *LSSC* quanto em *In* predominaram as classes de armazenamento de informações e manipulação de banco de dados (SQL), havendo também classes abstratas. O resultado não foi o esperado, uma vez que tal funcionalidade não condiz com a principal característica desse componente, a qual se refere, respectivamente, ao controle do programa e à entrada de dados no programa.

Em *Out,* as poucas classes agrupadas eram classes de exceção e auxiliares, de modo que estas não correspondem à função do componente o qual as abrange. Do mesmo modo, em *Tubes*, não se esperava a ocorrência de classes de exceção e armazenamento de informações, bem como classes abstratas, de manipulação de banco de dados e de armazenamento de informações não condiziam com as características de *Tendrils.*

3.3.4. *Software* desenvolvido por aluno iniciante em programação orientada a objetos

O software desenvolvido pelo aluno não se enquadrou no modelo *Little house*, uma vez que apresentou apenas 3 dos componentes: *LSSC*, *In* e *Tendrils*, o que não permite configurar o grafo esperado para as conexões entre as classes do programa.

No componente *LSSC*, a única classe presente permite o armazenamento de informações, no caso, os dados do objeto "Cliente", não sendo uma classe central a qual se conecta todo o programa, como esperado para o componente em que se encontra.

Em *In*, estavam presentes classes de armazenamento de informações, classes de serviços e de controle, bem como a classe principal (*Main*), a qual esperava-se que estivesse no componente *LSSC*. Em *Tendrils*, da mesma forma, encontraram-se, além das classes com mesma funcionalidade do *In,* classes abstratas e de apresentação de menus.

3.3.5. *Connecta*

O software *Connecta* apresentou, em seu componente *Disconnected*, apenas uma classe, a qual foi classificada como uma classe depositária. Este resultado condiz com o esperado, uma vez que classes depositárias não se comunicam com as demais classes, sendo utilizadas apenas para organizar as constantes a serem empregadas no programa.

Em *LSSC*, por sua vez, a única classe presente possui como funcionalidade realizar a apresentação em tela (interface gráfica), o que não corresponde ao esperado para o componente em que se encontra, posto que não realiza a comunicação entre as classes do programa (função esta realizada pela classe *Main*). Assim como *LSSC,* no componente *In* as duas classes presentes também são classes de interface gráfica, sendo que, dentre elas, encontra-se a *Main*.

No *Out*, além das classes de interface gráfica, encontraram-se também classes de manipulação de banco de dados, armazenamento de informações e de serviços. Em *Tubes*, encontraram-se classes de interface gráfica, manipulação e coleta de informações. Em *Tendrils*, encontram-se classes de interface gráfica e de serviços.

3.4. Análise dos Resultados

De acordo com os resultados, para nenhum dos 5 *softwares* analisados no trabalho houve um padrão para o tipo de classe presente em cada componente. No entanto, em alguns casos, algumas classes ficaram restritas a determinado conjunto, como, por exemplo, interfaces e classes auxiliares que se concentraram, em sua maioria, no componente *Disconnected*.

Nos demais, as classes agrupadas não apresentaram funcionalidades semelhantes, bem como suas funções não possuíam correspondência dentro de um mesmo grupo. Esta característica foi exibida por todos os 5 *softwares* usados no estudo, sendo que muitos apresentaram, por exemplo, classes de armazenamento, manipulação, controle de informações e até mesmo classes abstratas em um mesmo componente, sendo que as classes de armazenamento de informações foram encontradas em vários componentes.

3.5. Conclusão

Esta seção tratou da realização de um estudo sobre as classes presentes em cada um dos componentes que fazem parte do modelo *Little House*, buscando-se verificar se cada um dos grupos apresentavam classes com funcionalidades similares. De acordo com os testes executados, no entanto, concluiu-se que não existe um padrão para as funções das classes englobadas por cada componente, isto é, as classes agrupadas em um mesmo componente não possuem funcionalidades semelhantes.

4. Implementação da Tela de Configurações do *Software Connecta*

Os cálculos realizados para coleta das métricas do *software Connecta* foram baseados em estudos e adaptações de Ferreira (1) para análise de *bytecodes* de *softwares* desenvolvidos em Java. *Connecta* possibilita o cálculo das seguintes métricas: Estabilidade, COF, CBO, DIT, LCOM, K3B e COR. Tais métricas permitem identificar as possíveis modificações e melhorias para um determinado sistema analisado, sendo uma ferramenta de grande valia no processo de manutenção de *softwares*.

Dentre as métricas avaliadas para o presente estudo, a métrica de estabilidade fornece a probabilidade de um módulo ser modificado após se realizar alteração em outro. Para calcular esta probabilidade, o algoritmo primeiramente proposto por Myers[3] e adaptado por Ferreira (1) é bastante complexo, levando em consideração o grau de coesão dos módulos envolvidos, o grau de acoplamento direto e indireto entre estes módulos.

Em virtude da utilização de um algoritmo complexo e que emprega recursividade, a execução do cálculo da métrica de estabilidade tem um custo computacional elevado, podendo despender uma quantidade significativa de tempo de acordo com a quantidade de classes a serem analisadas no *Connecta*. Testes anteriores feitos no programa mostraram que o tempo de execução pode alcançar a ordem de horas, dependendo do *software* empregado.

A Coesão de Responsabilidade (COR), também analisada neste estudo, é uma métrica que permite avaliar a coesão interna de uma classe, ou seja, o "grau de interrelacionamento entre os elementos de um módulo" [Ferreira, 2011]. Esta métrica indica o número de responsabilidades[4] que determinada classe implementa. Os valores da COR variam de 0 a 1, sendo que quanto mais próximo de 1, maior é a coesão da classe testada.

Ao se avaliar a COR, no entanto, a presença de métodos construtores pode aumentar o valor obtido, devido ao fato destes métodos referenciarem uma grande quantidade de atributos da classe. Assim sendo, o ideal é desconsiderá-los do cálculo, a fim de se obter um valor mais exato.

Tendo em vista a necessidade de tais adaptações, a fim de proporcionar ao usuário do *software* a maior acurácia possível, foi implementada uma tela de configurações em *Connecta*. Esta tela de configurações deve permitir que o usuário opte por calcular ou não a métrica de estabilidade, uma vez que o cálculo da mesma pode despender uma quantidade considerável de tempo, mesmo que seu valor não seja de grande interesse ao usuário. Além disso, deve possibilitar optar pelo cálculo da métrica COR considerando ou não os métodos construtores, deixando a encargo do usuário avaliar se a inclusão dos mesmos é válida

3 *Myers, G. J. (1975).* Reliable software through composite design. *Petrocelli/Charter. Nova York, 2 edição.*
4 Segundo Ferreira (1), a responsabilidade de uma classe é dada por um conjunto de métodos os quais estabelecem um relacionamento entre si na classe.

para a análise a ser feita. Esta seção apresenta as modificações feitas em *Connecta*, por meio da descrição do modo de implementação da tela de configurações e os testes realizados para a verificação do funcionamento correto da mesma.

4.1. Metodologia

Para a realização deste trabalho, utilizou-se o ambiente de desenvolvimento Netbeans 6.9.1®, o qual permitiu criar a interface gráfica da tela de configurações e também manipular o código-fonte para a adaptação ao *Connecta*.

Para a tela de configurações, criou-se, dentre as classes do *Connecta*, um novo formulário *JFrame*, adicionando-se os botões e rótulos necessários para a funcionalidade desejada. Posteriormente, inseriu-se dois atributos do tipo *boolean* na classe de cálculo das métricas do *Connecta*, permitindo habilitar ou desabilitar o cálculo da métrica de estabilidade e utilizar ou não os métodos construtores para a métrica COR.

Para descobrir a forma como os métodos construtores se apresentam, solicitou-se, por meio de comandos específicos, a impressão do nome de cada um dos métodos em que se realizava o cálculo da métrica COR. Na realização deste processo, verificou-se, por meio de testes (vide seção 4.2.2) que aparentemente os métodos construtores são apresentados sob o nome "<init>" e, em alguns casos – como para as classes auxiliares (somente com métodos estáticos) - são gerados construtores com o nome "<clinit>".

Após detectar a forma como o método se apresenta, acrescentou-se ao código-fonte do *Connecta* o seguinte algoritmo genérico implementado em Java:

```
se (calcular_com_construtor = falso)
        se (nome_metodo = "<init>" ou nome_metodo = "<clinit>")
                calcular_metrica_COR_com_construtor = falso
```

Neste caso, o *boolean* "calcular_com_construtor" é definido como "verdadeiro" ou "falso" de acordo com a marcação ou não da caixa de seleção "Considerar métodos construtores para cálculo da métrica COR" pelo usuário na tela de configurações.

Para a métrica de estabilidade, buscou-se pelo método que realizava o cálculo de seu valor, adicionando-se, ao código-fonte do *Connecta*, o seguinte algoritmo genérico implementado em Java:

```
se (calcular_estabilidade)
        calcular_metrica_estabilidade ( )
```

Neste caso, o *boolean* “calcular_estabilidade” é definido como “verdadeiro” ou “falso” em função da marcação ou não da caixa de seleção efetuada pelo usuário na tela de configurações. O método calcular_metrica_estabilidade representa o método responsável pelo cálculo da métrica referenciada na classe.

Após as implementações necessárias, foram realizados testes, analisando-se os valores antes e depois das métricas com a modificação nas caixas de seleção (*checkboxes*) da tela de configurações.

4.2. Resultados e Testes

4.2.1. Tela de Configurações

Tendo feito o esboço da tela e os ajustes necessários no programa, a interface gráfica da tela de configurações se apresentou como na Figura 4.1.

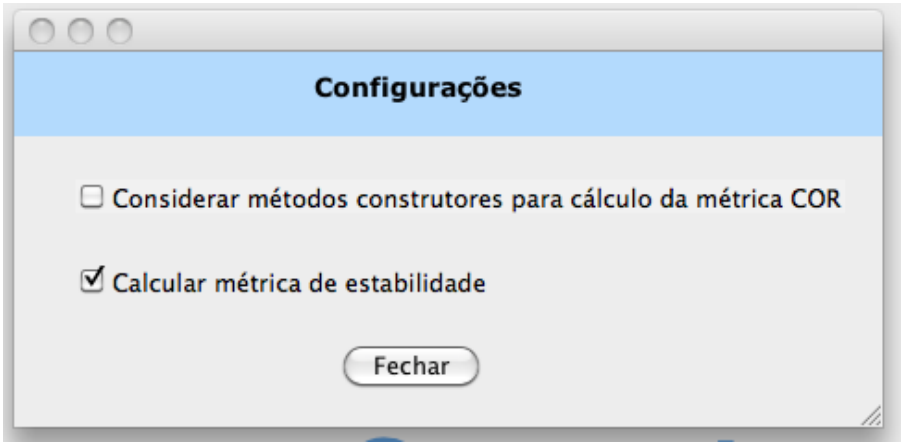

Figura 4.1: Tela de Configurações do Connecta.

Como padrão para cálculo no programa, definiu-se que a métrica de estabilidade sempre seria calculada, bem como os métodos construtores não seriam considerados para cálculo da métrica COR, uma vez que o ideal, para obter um valor mais exato, é desconsiderá-lo.

4.2.2. Testes

Para a realização de testes após a modificação feita, utilizou-se 3 trabalhos práticos desenvolvidos por um aluno, bem como um projeto básico feito para verificação da correção da implementação. Os resultados para os testes referidos estão nas Tabelas 4.1, 4.2, 4.3 e 4.4.

Tabela 4.1: Teste feito para Trabalho Prático 1

Nome da Classe	COR com Construtor	COR sem Construtor	Nome da Classe	COR com Construtor	COR sem Construtor
Gente	1,000	1,000	Main	0,500	1,000
Estagiario	0,500	0,500	ControleGeral	0,111	0,125
Listagens	0,500	1,000	Funcionario	1,000	1,000
VetorGente	1,000	1,000	Professor	1,000	1,000
ParametrosSalario	0,500	1,000	InclusaoGente	0,500	1,000
TempoTrabalhado	0,500	1,000	HorarioTrabalho	0,500	1,000

O Trabalho Prático 1 consiste em um *software* que permite o cadastro, alteração e exclusão de professores, estagiários e funcionários de um ambiente acadêmico, podendo realizar o controle de pagamentos, parâmetros de salário, número de horas trabalhadas e horário de trabalho.

Tabela 4.2: Teste feito para Trabalho Prático 2

Nome da Classe	COR com Construtor	COR sem Construtor	Nome da Classe	COR com Construtor	COR sem Construtor
Rastreamento	1,000	1,000	Console	0,500	1,000
ExpVariavel	1,000	1,000	Auxiliar	0,250	0,333
ArquivoFonte	1,000	1,000	Main	0,500	1,000
ExpBinaria	1,000	1,000	Rotulos	1,000	1,000
Expressao	0,500	1,000	Vaiso	0,500	1,000
ExpConstante	1,000	1,000	Repete	0,500	1,000
Trem	1,000	1,000	Interpretador	1,000	1,000

O Trabalho Prático 2 consistiu na implementação de um interpretador de linguagem, o qual permite ler um arquivo em determinada linguagem de programação criada e interpretar cada linha do código-fonte.

Tabela 4.3: Teste feito para Trabalho Prático 3

Nome da Classe	COR com Construtor	COR sem Construtor	Nome da Classe	COR com Construtor	COR sem Construtor
ControlePeca	0,333	0,500	ControleObra	0,333	0,500
Teatro	1,000	1,000	ControleAlocacao	0,333	0,500
ControleSala	0,500	1,000	Main	0,500	1,000
Data	1,000	1,000	Sala	1,000	1,000
Filme	1,000	1,000	Obra	0,333	0,333
Relatorios	0,500	1,000	AlocacaoSala	1,000	1,000
ControleFilme	0,333	0,500	ControleGeral	0,250	0,333

O Trabalho Prático 3 consiste em um sistema para um centro cultural, o qual permite incluir, alterar e

excluir salas e obras (podendo a obra ser de 2 tipos: filme ou peça teatral), bem como realizar a alocação de salas para determinada obra e gerar relatórios.

Tabela 4.4: Teste feito para Projeto Básico

Nome da Classe	COR com Construtor	COR sem Construtor
Main	1,000	1,000
Teste1	1,000	0,250
Teste2	1,000	0,333

Neste projeto básico, cada uma das classes apresentava apenas atributos, o método construtor e métodos *get* e *set*[5], sendo utilizado, portanto, apenas para os testes iniciais da tela de configurações.

Para todos os 4 testes feitos, a métrica de estabilidade foi igual a 0 quando se optou por não calculá-la e gerou o resultado correto quando foi marcada a opção para obter seu valor na tela de configurações.

4.3. Análise dos Resultados

Como se observou pelos valores obtidos para as métricas de estabilidade e COR nos testes realizados, pode-se inferir que a tela de configurações foi implementada com sucesso, exercendo a funcionalidade almejada no *Connecta*. A métrica de estabilidade, como esperado, gerou um valor igual a 0 quando o usuário optou por não calculá-la.

A métrica COR, por sua vez, apresentou diferentes valores para cada uma das classes dos projetos, tendo considerado ou não os métodos construtores em seu cálculo. No geral, a COR teve 3 comportamentos distintos:

I) ***Valor aumentou ao se considerar os métodos construtores***: tal resultado ocorreu apenas para as classes em que, além dos métodos construtores, somente havia métodos *get* e *set*: mais especificamente as classes Teste1 e Teste2 do projeto básico de teste. Dessa forma, considerando-se os métodos construtores para estas classes, a coesão interna será maior, uma vez que estes referenciam todos os atributos presentes na classe, enquanto os *get* e *set* são restritos a apenas um atributo cada, isoladamente.

II) ***Valor diminuiu ao se considerar os métodos construtores***: tal resultado ocorreu para todas as classes auxiliares e de controle, em que não havia atributos ou, caso existentes, eram poucos, no máximo 2. Isto pode ser justificado pelo fato de que os métodos construtores destas classes são geralmente métodos-padrão (vazios), não havendo referência para nenhum método ou atributo da classe, o que reduz o valor da coesão da classe.

5 Os métodos *get* e *set* são utilizados para, respectivamente, obter o valor de um atributo e modificá-lo.

III) ***Valor permaneceu inalterado ao se considerar os métodos construtores***: tal resultado ocorreu para classes que apresentavam métodos em que a referência aos demais métodos e atributos é "similar", ou seja, o acréscimo ou não do método construtor não alterou a coesão da classe.

4.4. Conclusão

Esta seção tratou da implementação de uma tela de configurações para o *software Connecta.* Esta tela permite ao usuário optar por calcular ou não a métrica de estabilidade, uma vez que o algoritmo para esta métrica envolve cálculos complexos e até mesmo recursividade, o que pode prolongar muito o tempo de execução do programa, dependendo da quantidade de classes que o usuário selecione.

A tela também possibilita que o usuário opte por calcular a métrica COR (Coesão de Responsabilidade) com ou sem o uso de métodos construtores. O ideal para se obter um valor mais exato e confiável desta métrica é desconsiderar os métodos construtores, posto que, como observado pelos testes realizados, estes métodos podem alterar o valor real para mais ou para menos, de acordo com a funcionalidade da classe no sistema.

5. Modificação da Interface Gráfica do *Software Connecta*

A interface gráfica consiste em um instrumento para facilitar a interação do usuário com determinado programa por meio de uma tela ou representação gráfica, visual, com desenhos, imagens, etc. Também chamada GUI (do inglês *Graphical User Interface*), ela confere a aparência e a apresentação do aplicativo, permitindo que o utilizador do serviço possa se familiarizar e adaptar facilmente às diversas funcionalidades que o programa pode oferecer.

Na linguagem de programação Java, utilizada no presente estudo, a GUI é constituída por um conjunto de componentes inseridos dentro de *containers*. Os *containers* mais usados são as janelas (*JFrame*) e painéis (*JPanel*), nos quais podem ser inseridos botões (*JButton*), menus, rótulos (*Label*), outros painéis menores, dentre outros.

Nas primeiras versões da linguagem, a GUI original provinha das classes do pacote java.awt (do inglês *Abstract Window Toolkit*). Os elementos providos pela AWT são implementados utilizando o conjunto de ferramentas gráficas de cada plataforma (Windows, MacOS, Linux, etc), preservando, assim, a aparência de cada um dos sistemas operacionais em que o programa é usado. A desvantagem disto está no fato de que a interface gráfica designada em uma plataforma poderia apresentar problemas quando utilizada em outra.

A partir do Java 2, foi introduzido o pacote javax.swing, o qual veio também a preencher a lacuna da interface dependente da plataforma usada. A Swing é compatível com a AWT, no entanto, diferentemente da última, renderiza todos os componentes por conta própria, ao invés de repassar este encargo ao sistema operacional. Sua desvantagem é que, por ser uma API de mais alto nível, ela apresenta um desempenho mais baixo e consome mais memória RAM em comparação às outras APIs gráficas. Todavia, a Swing possui uma vasta quantidade de funcionalidades e a interface de seus programas é independente do Sistema Operacional. A diferença entre os pacotes java.awt e javax.swing pode ser vista no diagrama da Figura 5.1.

O JFC (do inglês *Java Foundation Classes*) abrange um conjunto de recursos para criar GUIs e adicionar efeitos interessantes para aplicativos Java. Ele é definido por conter os seguintes recursos:

- **Componentes Swing**: como dito anteriormente, o pacote javax.swing inclui tudo desde painéis e botões até tabelas elaboradas. Seus componentes permitem classificar, imprimir, arrastar e selecionar, além de várias outras funções.
- ***Look and Feel***: permite escolher a aparência geral de um programa. Java suporta o *look and feel* GTK+[6], disponibilizando, assim, centenas de visuais gráficos (*look and feel's*) para programas que

6 GTK+ é um conjunto de ferramentas (*toolkit*) multi-plataforma para criar interfaces gráficas. O toolkit pode ser configurado pelo usuário e oferece muitas funcionalidades, podendo-se emular, por exemplo, a aparência de aplicativos em Windows ou Qt.

utilizam Swing.

- **API de acessibilidade**: permite o uso de tecnologias assistivas, tais como leitores de tela ou *displays* de Braille.
- **API Java 2D:** permite que os desenvolvedores possam inserir gráficos, textos e imagens de alta qualidade em 2D em seus aplicativos e *applets*. O Java 2D apresenta diversas APIs para geração e envio de conteúdo gráfico para os dispositivos de saída apropriados.
- **Internacionalização**: permite a construção de aplicações que podem interagir com usuários em todo o mundo em seus idiomas nativos, possibilitando até mesmo a criação de textos em línguas que usam milhares de diferentes caracteres tais como japonês, chinês ou coreano.

Tendo em vista a distribuição do *software*, surgiu a necessidade de reformulação da interface gráfica do *Connecta*, de forma que a mesma apresentasse um visual similar àquele presente em *softwares* proprietários, facilitando a manipulação do mesmo pelo usuário. Assim sendo, esta seção contempla as modificações feitas na GUI do *Connecta*, apresentando as classes e componentes principais empregados, bem como recursos auxiliares utilizados para se alcançar a aparência almejada.

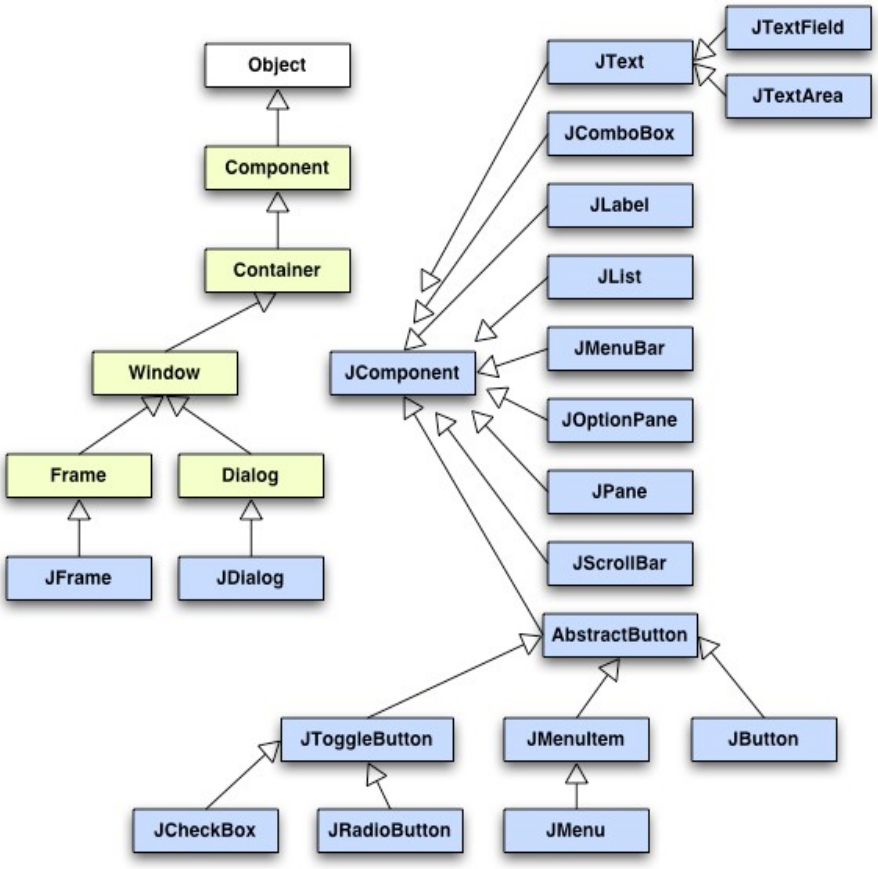

Figura 5.1: Diagrama resumido das classes AWT e Swing.

Fonte: http://cee.uma.pt/people/faculty/pedro.campos/docs/guia-IHM.pdf

5.1. Metodologia

Para a realização deste trabalho, utilizou-se o IDE (ambiente de desenvolvimento integrado) Netbeans 6.9.1®, o qual permitiu manipular a interface gráfica e também modificar o código-fonte para a adaptação ao *Connecta*.

A reformulação dos componentes gráficos teve como base a GUI do próprio IDE, uma vez que este apresenta uma aparência "limpa" e agradável, permitindo que o usuário se familiarize mais facilmente com todos os recursos que o programa oferece.

Além de adaptar a interface gráfica de cada uma das classes relacionadas, foram retirados os componentes referentes à utilização de Banco de Dados (SQL), uma vez que se optou por não utilizar este recurso no programa, recorrendo somente ao armazenamento de dados no modo texto (arquivos .txt).

Ademais, utilizando-se o programa GIMP (do inglês *GNU Image Manipulation Program*), tentou-se elaborar uma imagem diferenciada com o título do programa, de forma a personalizar e tornar mais característico seu visual para o usuário.

Por fim, o *software*, originalmente constituído em português, foi traduzido para a língua inglesa, tendo em vista a necessidade de se distribuí-lo para uma maior quantidade de usuários.

5.2. O Java *Look and Feel*

Embora o termo *look and feel* faça referência à aparência e comportamento dos componentes da interface gráfica, este recurso abrange uma série de aspectos adicionais aos componentes com os quais trabalha, como, por exemplo, a terminologia, o *layout* e o comportamento geral do programa.

As janelas, os menus e outros componentes da GUI diferem bastante em plataformas distintas, apresentando tanto diferenças na disposição e tamanho dos componentes, quanto nas cores e nos menus. Em outras palavras, o *look and feel* pode apresentar alterações quando se compara diferentes sistemas operacionais, devendo ser escolhido cuidadosamente para o tipo de usuário a que se destina, já que cada um partilha do que se denomina "experiência do usuário".

A experiência do usuário está ligada à rotina que ele desenvolve na máquina, tornando-se habituado com as fontes, cores e disposição da GUI em seu computador, o que permite que seu trabalho seja feito de forma mais rápida. Assim, um programa que se utiliza de uma interface gráfica diferente daquela a que ele já está familiarizado pode se tornar um tanto confuso e "estressante" ao usuário, o qual pode vir até mesmo a perder o interesse pelo *software*.

Uma vez que Java se baseia no princípio "uma vez codificado, é executado em qualquer máquina" (*"write once, run anywhere"*), encontrar uma interface agradável a todos os usuários, em diferentes

plataformas, é uma tarefa de grande preocupação de seus desenvolvedores. Para solucionar esta questão, a *Sun*© adicionou a classe *LookAndFeel* ao pacote Swing, a qual possibilitou o emprego de GUI's de diferentes tipos aos programas em Java.

5.2.1. Classe *UIManager*

A classe *UIManager* é responsável por configurar e modificar o *look and feel* (*l&f*). Para alterar o *l&f*, invoca-se o método estático UIManager.setLookAndFeel, passando, como argumento, uma *string* ou uma instância da classe *LookAndFeel*. Se for utilizada uma *string* como argumento, esta deverá apresentar o nome completo da classe que possui o visual desejado. Assim, caso o desenvolvedor queira utilizar o visual "Metal", o mesmo pode passar tanto a *string* "javax.swing.plaf.metal.MetalLookAndFeel" como new javax.swing.plaf.metal.MetalLookAndFeel(). Para realizar a alteração, além de invocar o método *set*, deve-se também utilizar o método updateComponentTreeUI(Component c) da classe javax.swing.SwingUtilities, o qual permite atualizar a classe UI para cada um dos componentes da GUI.

Para encontrar todos os *look and feels* disponíveis para uso, pode-se também usar o método estático getInstalledLookAndFeels, o qual retorna um vetor com os *l&f's* presentes na plataforma. Para permitir que a aparência do programa possua o visual do sistema operacional em uso e assim evitar eventuais problemas, utiliza-se o método getSystemLookAndFeelClassName.

5.2.2. Classe *LookAndFeel*

A classe *LookAndFeel* é responsável por armazenar toda a informação sobre a aparência empregada no programa. Ela possui 5 métodos abstratos que devem ser implementados: getDescription(), getID(), getName(), isNativeLookAndFeel() e isSupportedLookAndFeel(). O método getDescription retorna a descrição do *look and feel*. O getID retorna a ID que identifica o *l&f* para as aplicações e serviços. O getName retorna o nome do *l&f*. O método isNativeLookAndFeel retorna *true* (verdadeiro) se o *l&f* é nativo para o sistema operacional. Por fim, isSupportedLookAndFeel retorna *true* (verdadeiro) se o sistema operacional suporta o *look and feel* utilizado, sendo que somente retorna *false* (falso) se houver motivos legais relacionados à utilização de determinado *l&f* para certas plataformas.

5.2.3. Implementação do *LookAndFeel* em *Connecta*

Uma vez que é desejado que o *software Connecta* seja multi-plataforma e apresente uma GUI agradável e prática, foi necessário implementar um *look and feel* que se adequasse à aparência do sistema utilizado pelo seu usuário. Assim sendo, o seguinte código foi acrescentado:

```
UIManager.setLookAndFeel( ).setLookAndFeel( UIManager.getSystemLookAndFeelClassName());

SwingUtilities.updateComponentTreeUI(this);
```

5.3. Ajustes Gerais em *Connecta*

Visando padronizar a interface gráfica do programa e deixá-la mais amigável ao usuário, além da configuração do *look and feel,* também foram feitas mudanças na fonte utilizada para o texto, disposição das janelas, títulos e botões.

Além disso, utilizando-se o *software* de tratamento e edição de imagens The Gimp®, elaborou-se uma imagem que poderá ser utilizada no *Connecta,* contendo o título personalizado do programa. Para permitir a distribuição do mesmo para uma maior quantidade de usuários, também se optou por traduzir o programa para a língua inglesa.

5.4. Conclusão

Muitos desenvolvedores costumam focar sua atenção para produzir um algoritmo de qualidade, robusto e o mais exato possível. Apesar da importância indiscutível de um bom código, o que se esquece, na maior parte das vezes, é que a interatividade do programa com o usuário também é fundamental. Em outras palavras, a interface gráfica é um aspecto que certamente deve ser levado em conta durante o desenvolvimento do *software*, uma vez que ela é o meio de comunicação do usuário com as funcionalidades disponibilizadas pelo programa.

Esta seção tratou da modificação da interface gráfica do *software Connecta*, de forma a tornar a aparência do programa mais agradável e amigável ao usuário. Para isto, foi necessário buscar pelo visual padrão utilizado pelo usuário em sua plataforma, empregando o *look and feel* adequado para cada tipo de sistema operacional. Além disso, é importante ajustar a posição de janelas, botões e textos, padronizando também a disposição dos componentes na tela, uma vez que recursos que estão fora da ordem habitual ao usuário podem ser sinônimo de dificuldade.

6. Conclusão

De acordo com os estudos e trabalhos descritos no presente relatório, pode-se resumir as principais atividades realizadas durante o período de Iniciação Científica do seguinte modo:

1. Execução de testes em *softwares* Java para aprofundamento na descrição do modelo *little house*, os quais permitiram a realização de estudos para verificar a existência de um padrão de classes para cada componente que constitui o modelo.
2. Implementação de uma tela de configurações para o *software Connecta*, a fim de otimizar o tempo de execução do programa, bem como a exatidão e correção dos resultados obtidos para determinadas métricas.
3. Reformulação da interface gráfica de *Connecta*, de modo a proporcionar ao usuário um programa com um visual mais familiar e prático.

Para aprimoramento futuro da ferramenta, alguns trabalhos que podem ser realizados são:

1. O desenvolvimento de um site para divulgação do *software Connecta.*

2. A inserção de recursos gráficos para visualização das conexões descritas pelas métricas.

3. A expansão de *Connecta*, de modo a permitir avaliar *softwares* em outras linguagens de programação.

7. Referências

1. FERREIRA, K. A. M. *Avaliação de Conectividade em Sistemas Orientados por Objetos*. Dissertação (Mestrado) – Departamento de Ciência da Computação / Universidade Federal de Minas Gerais, Belo Horizonte, 2006.

2. FERREIRA, K. A. M. *Um Modelo de Predição de Amplitude da Propagação de Modificações Contratuais em Software Orientado por Objetos*. Dissertação (Doutorado) – Departamento de Ciência da Computação / Universidade Federal de Minas Gerais, Belo Horizonte, 2011.

3. FERREIRA, K. A. M. *et al*. *CONNECTA*. Departamento de Ciência da Computação / Universidade Federal de Minas Gerais, Belo Horizonte, 2006.

4. DEITEL, P. J. *Java: como programar.* Tradução de Edson Furmankiewicz. 8ª ed. São Paulo: Pearson Prentice Hall, 2010.

5. PAJEK (2010). *Networks / Pajek Program for Large Network Analysis - for Windows*. http://vlado.fmf.uni-lj.si/pub/networks/pajek/.

6. Universidade Federal do Rio de Janeiro. *Introdução à Interface Gráfica com o Usuário (GUI) e ao pacote javax.swing*. Disponível em: <http://www.dcc.ufrj.br/~comp2/TextosJava/Eventos%20e%20GUI%201.htm>. Acesso em agosto de 2011.

7. RICARTE, I. L. M. *Desenvolvimento de Aplicações Gráficas*. Disponível em: <http://www.dca.fee.unicamp.br/cursos/PooJava/aplicgraf.html>. Acesso em agosto de 2011.

8. SILVA, C. B. *Aparências de Interface: Look and Feel*. Disponível em: <http://javafree.uol.com.br/artigo/3229/Aparencias-de-interface-Look-and-Feel.html>. Acesso em agosto de 2011.

9. KENNETH, Thomas. *Using Swing's Pluggable Look and Feel*. Disponível em: <http://today.java.net/pub/a/today/2004/02/27/laf.html>. Acesso em agosto de 2011.