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

Dissertação de Mestrado

Kecia Aline Marques Ferreira

Avaliação de Conectividade em Sistemas Orientados por Objetos

Belo Horizonte

Junho de 2006

KECIA ALINE MARQUES FERREIRA

Avaliação de Conectividade em Sistemas Orientados por Objetos

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientadora: Prof.ª Mariza Andrade da Silva Bigonha
Co-orientador: Prof. Roberto da Silva Bigonha

Belo Horizonte

Junho de 2006

Agradecimentos

Aos meus pais, pela minha vida e pelo incentivo aos estudos.
Aos meus queridos irmãos, pela amizade e apoio.
A todos os meus amigos, pela torcida.
Aos colegas de mestrado.
Aos professores Lucila Ishitani, José Wilson e Lúcio Mauro.
Aos meus orientadores, Professora Mariza e Professor Bigonha, pela dedicação.
Ao Synergia, ao CNPq, à FUNCESI e à PUC-MG, pelas oportunidades nos momentos necessários.
E, acima de tudo, a Deus, por todos os momentos de minha vida, por todas as bênçãos e por ter colocado todos vocês em meu caminho.

Resumo

Dentre os fatores de avaliação da qualidade de um software, destaca-se a manutenibilidade, a medida da facilidade de realizar sua manutenção. A manutenção de software é responsável pela maior parte do custo total de um sistema. Desta forma, faz-se importante a obtenção de recursos que contribuam para a criação de software cuja manutenção seja mais fácil. Dentre outros aspectos, a obtenção de software com esta característica é proporcionada principalmente pelo projeto de software que vise estrutura flexível e estável, na qual o grau de interdependência entre as partes constituintes do software, seus módulos, seja o menor possível. Quanto maior o grau de conectividade de um software, mais rígida a sua estrutura, menor a manutenibilidade e maior o custo do sistema. A orientação por objetos é um paradigma de construção de software caracterizado por potencializar a obtenção de software de alta qualidade, favorecendo aspectos como modularidade, manutenibilidade e reusabilidade.

Este trabalho tem como tese que a conectividade é o fator preponderante na avaliação da qualidade estrutural de um software e, conseqüentemente, deve ser tida como fator de grande importância na manutenção e no custo de um sistema. O objetivo principal desta dissertação é a proposta de um modelo de avaliação de conectividade em software orientado por objetos, bem como a construção de uma ferramenta de coleta de métricas que viabiliza a aplicação do modelo proposto para softwares implementados na linguagem Java.

Palavras-chave: qualidade de software, manutenibilidade, conectividade, coesão, acoplamento, orientação por objetos, métricas, apoio à decisão.

Abstract

Amongst the factors of software quality evaluation , maintainability, the measure of the easiness to carry through its maintenance, distinguishes. Software maintenance is mostly responsible for the total system cost. In such a way, the attainment of resources that contribute for the software creation whose maintenance is easier becomes important. Amongst other aspects, the development of software with this characteristic is achieved mainly by means of software project that aims at flexible and stable structure, in which the interdependence degree between the constituent parts of software, their modules, is minimized. How much bigger the degree of connectivity in a software, more rigid is its structure, minor maintainability and greater the cost of the system.

This work has as thesis that the connectivity is the main factor in the evaluation of the structural quality of a software and, consequently, may be taken as factor of great importance in the maintenance and the cost of a system. The aim of this work is the proposal of a model of evaluation of connectivity in object-oriented software, as well as the construction of an automatic tool for metric collection that it makes possible the application of the aforesaid model for software implemented in the Java language.

Keywords: software quality, maintainability, connectivity, cohesion, coupling, object-orientation, metric, support to decision.

Lista de Figuras

Lista de Tabelas

Lista de Siglas

AHF	Fator Ocultação de Atributo
AIF	Fator Herança de Atributo
CBO	*Coupling between Object Class*
CK	conjunto de métricas proposto por Chidambere Kemerer
CLF	Fator Agrupamento
COF	Fator Acoplamento
DIT	*Depth of Inheritance Tree*
ERSw	Especificação de Requisitos de Software
LCOM	*Lack of Cohesion in Methods*
MACSOO	Modelo de Avaliação de Conectividade em Sistemas Orientados por Objetos
MOOD	Metrics for Object Oriented Design
MHF	Fator Ocultação de Método
MIF	Fator Herança de Método
NOC	*Number of Children*
OO	Orientação por Objetos
PF	Fator Polimorfismo
RF	Fator Reúso
RFC	*Response for a Classe*
WMC	*Weighted Methods per Class*

Sumário

Capítulo 1

Introdução

Desenvolver software de alta qualidade e baixo custo não é tarefa simples. Nas últimas décadas, muito se estudou e investigou sobre o processo de desenvolvimento de software [36] [37] [41] [54] [14]. Os sistemas de informação tornaram-se mais complexos e mais presentes na vida das pessoas. Eles estão presentes na medicina, na educação, nas telecomunicações, nos processos e serviços prestados pelo governo à população, nas relações comerciais, enfim, em uma série de situações no dia a dia das pessoas. Algumas destas situações envolvem risco para a vida humana, como sistemas que controlam aviões, lançamentos de foguetes e linhas ferroviárias; outras envolvem manipulação de grande quantidade de valor monetário, como a automação bancária, etc. Pressman [44] ressalta que à medida que a importância dos softwares na sociedade cresce, aumenta também a preocupação da comunidade produtora de software em obter recursos para desenvolver software de forma mais fácil, mais rápida e com custos menores. A difusão de modelos de capacitação em processos de software hoje confirma isso. Segundo Paula [42], este tipo de modelo visa avaliar a maturidade de uma organização para produzir software de boa qualidade, com custos razoáveis e cumprir prazos. Dentre eles, destaca-se o CMM (Capability Maturity Model) [14] , modelo que o Departamento de Defesa Americano utiliza para avaliar as organizações que lhe fornecem softwares e que tem sido mundialmente reconhecido [42].

Embora exista hoje à disposição dos desenvolvedores de software uma gama de conceitos, tecnologias, metodologias e linguagens de programação poderosas, desenvolver software é ainda muito difícil e caro, um campo fértil a ser explorado e melhorado. Um bom exemplo disso é a Orientação por Objetos, uma das grandes contribuições dos últimos tempos para a produção de software de alta qualidade e baixo custo [36].

Este paradigma de projeto e programação representa a possibilidade de transposição do contexto caótico muitas vezes vivenciado por grande parte da comunidade desenvolvedora de software para um ambiente mais harmonioso no qual softwares possam ser produzidos com alta qualidade e com custo consideravelmente amenizado. Isso porque uma das características principais da orientação por objetos é a modularidade, a independência entre os componentes de software, o que viabiliza a obtenção de reusabilidade, de sistemas mais flexíveis, mais fáceis de testar e manter. Apesar de todos os recursos da orientação por objetos, ainda é difícil desenvolver software que seja reutilizável e flexível, como apontam Gamma et al [23].

A qualidade de software é um dos fatores de maior importância na produção de software. Meyer [36] sintetiza este fato definindo a engenharia de software como a produção de software de qualidade. Paula [42] define qualidade como a conformidade do software com os seus requisitos e ressalta que ela depende do processo utilizado para produzir o software. De acordo com Meyer [36], a qualidade de um software pode ser observada sob dois pontos de vista: a forma como ele foi construído - fatores internos - e a forma como ele se apresenta ao usuário - fatores externos. Por meio dos fatores externos é possível avaliar se o software está adequado aos requisitos do sistema. Os fatores internos são relacionados à estrutura do software. Estes são determinantes para se alcançar os fatores externos.

Outro fator crítico na produção de software é o custo. Como Paula [42] destaca, o custo de um software é um dos fatores que define a sua viabilidade. Segundo Pressman [44], a manutenção é a fase que mais demanda esforço no ciclo de vida de um sistema. Confirmado isso, Meyer [36] aponta que mais de 70% do custo total de um sistema é referente a custo de manutenção. Segundo Pfleeger [43], este percentual ultrapassa 80%. De uma forma geral, sistemas têm vida longa e não é possível dizer que um sistema está livre de erros, ou que não sofrerá alterações ou que não necessitará de novas funcionalidades. O estudo de Lientz e Swanson [1] (1980 apud Meyer [36]) revelou que 41.8% do custo de manutenção é decorrente de alterações em requisitos de usuário. Meyer [36] conclui, então, que a manutenção de um software é penalizada pela dificuldade de realizar alterações no mesmo. Esses dados mostram que reduzir os custos da manutenção de software é imperativo e que tal redução pode ser obtida principalmente quando for possível realizar alterações no software de forma mais fácil. Myers [38] aponta a modularidade como a resposta para tal problema, visto que a independência entre os módulos de um sistema permite que modificações em um

[1]LIENTZ, Bennet P. e SWANSON, E. Burton. em Software Maintenance Management: a Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations. Addison-Wesley, 1980.

módulo afetem um número pequeno de módulos. A modularidade contribui para a redução da complexidade do software e para a sua manutenibilidade [44].

Segundo Myers [38], a modularidade de um sistema é determinada basicamente por duas práticas fundamentais: a minimização do relacionamento entre módulos e a maximização do relacionamento entre elementos de um mesmo módulo. A medida do grau de relacionamento entre módulos é chamada acoplamento; a medida do grau de relacionamento entre os elementos internos de um módulo é chamada coesão. Uma situação ideal seria aquela na qual uma modificação realizada em determinado módulo afetasse o menor número possível de outros módulos. Um sistema com esta característica possui alto grau de *estabilidade*, que é a capacidade de um software se manter inalterado diante de uma alteração em seu ambiente [25]. Para ilustrar suas idéias, Myers [38] modela um sistema como um conjunto de lâmpadas conectadas entre si, no qual o fato de uma lâmpada estar acesa afeta o estado das demais lâmpadas conectadas a ela. Cada lâmpada representa um módulo. A mudança de estado de uma lâmpada representa atividade de manutenção do módulo correspondente. Neste modelo, descrito em detalhes no Capítulo 6, fica evidente o impacto da conectividade no custo de manutenção.

Este modelo de Myers é a inspiração para a tese defendida nesta dissertação: o grau de *conectividade* de um software, dada pela existência de dependência entre os seus módulos, tem importância preponderante na sua estabilidade e, conseqüentemente na sua manutenibilidade. Um olhar sobre o grau de conectividade de um software deve ser a primeira análise a ser realizada a fim de se obter uma idéia do grau de dificuldade de realizar sua manutenção. Diante de um alto grau de conectividade no sistema, deve-se procurar sanar as causas, por exemplo, a partir de uma análise e melhoria de fatores como o grau de acoplamento existente nas conexões e grau de coesão interna dos módulos.

Essa idéia de que a conectividade entre módulos de um sistema é a variável de ordem superior na determinação da manutenibilidade de um software encontra apoio na literatura sobre projeto e construção de software. Em um artigo clássico da área, Parnas [41] defende que um dos benefícios da modularização é a flexibilidade do software, o que torna possível realizar alterações drásticas em um módulo sem afetar os demais. Pfleeger [43] ressalta a importância da qualidade do projeto do software para a manutenibilidade do software, apontando que o software estruturado com componentes coesos e independentes é mais inteligível e sua manutenção é facilitada. Já Pressman [43] alerta que sistemas nos quais existe forte dependência entre seus módulos podem se tornar pesadelos no caso de depuração. Os sistemas mais fáceis de alterar são considerados por Xavier [60] como aqueles constituídos por módulos fáceis

de entender e o mais independente possível uns dos outros. Uma das mais importantes referências sobre construção de software orientado por objetos [36] define duas regras, dentre outras, para a obtenção de software modular: todo módulo deve se comunicar com um menor número de módulos possível; se dois módulos se comunicam, eles devem trocar o mínimo de informação possível. Estas duas regras apóiam a tese defendida nesta dissertação.

Conforme Pfleeger [43], outros fatores têm impacto na manutenibilidade de um software, dentre eles destacam-se: qualidade da documentção, complexidade do tipo de aplicação e a rotatividade de pessoal nas equipes. De acordo com isso, não se pode tomar a avaliação da conectividade de um software como a solução de todos os problemas de manutenção de software, pois as demais questões não devem ser desprezadas. Todavia, a redução da conectividade deve receber atenção especial na construção de software, visto que tem impacto determinante na qualidade e no custo do sistema. Ainda que os demais aspectos favoreçam a manutenibilidade de um sistema, se o aspecto conectividade não for adequado, a qualidade e o esforço de manutenção do software são seriamente comprometidos.

Para saber se a conectividade em um determinado software é satisfatória, é preciso conhecê-la e, para isso, faz-se necessário uma forma de medi-la. A partir do momento em que se toma conhecimento que a conectividade de um software não é satisfatória, deve-se, então, analisar os fatores adjacentes que contribuem para tal situação. Neste caso também é imprescindível uma forma de medir tais fatores. Assim, surgem as seguintes necessidades: identificar os fatores que têm impacto na conectividade de um software; identificar uma métrica que permita a avaliação da conectividade de um software; e identificar as métricas que permitam que os fatores determinantes da conectividade sejam avaliados.

A área de métricas de software é um assunto que tem sido estudado há cerca de três décadas [44]. A literatura sobre métricas de software é ampla, por exemplo, [2] [7] [12] [13] [18] [21] [61]. Esta dissertação tem interesse específico pelas métricas que possam auxiliar no processo de avaliação da conectividade em sistemas orientados por objetos. O interesse em avaliar softwares orientados por objetos deve-se ao fato de que se pretende, aqui, contribuir para a produção de software de alta qualidade e custo reduzido e, conforme, citado anteriormente, este é um paradigma dotado de características que potencializam este objetivo. Desta forma, pretendemos que a avaliação de conectividade em sistemas orientados por objetos seja um instrumento de grande valia na construção de software neste paradigma.

Muitos pesquisadores e empresas têm oferecido contribuições valiosas para a medição

de softwares orientados por objetos [1] [2] [13] [18] [32] . Destacam-se, dentre esses trabalhos, o conjunto de métricas proposto por Chidamber e Kemerer, conhecido na literatura como métricas CK [13], e o conjunto de métricas proposto por Abreu e Carapuça, conhecido como MOOD [2]. Encontram-se na literatura propostas de ferramentas de coleta dessas métricas para várias linguagens, como exemplo, a proposta do INESC/Portugal denominada MOODKIT [3], que permite a coleta das métricas MOOD [2] em códigos como C++, Java, Smalltalk ou Eiffel. Contudo, conforme Fenton e Neil [18], há uma carência de formas efetivas de aplicação de métricas de software como suporte à decisão na produção de softwares. Faz-se necessário a existência de instrumentos de apoio à decisão gerencial no ciclo de vida do sistema, que auxiliem gerentes e técnicos a realizarem acompanhamento e atuação na produção de software, no sentido de decidirem em que ponto agir para corrigir, evitar ou reverter situações indesejadas.

Como consideramos a conectividade o fator preponderante na manutenibilidade de software, e temos interesse específico em sistemas orientados por objetos, pela razão apresentada anteriormente, apresentamos, nesta dissertação, um Modelo de Avaliação de Conectividade de Sistemas Orientados por Objetos - MACSOO. Este modelo destina-se a auxiliar a tomada de decisão no processo de software. O modelo proposto indica quais métricas devem ser avaliadas bem como quais aspectos devem ser melhorados, caso obtenha-se um valor não apropriado para o aspecto principal do software avaliado: a conectividade.

Esta dissertação tem como objetivos:

- apresentar as principais métricas de software OO propostas na literatura;
- identificar os principais fatores que têm impacto na conectividade de software;
- identificar um conjunto de métricas que possibilite avaliar a conectividade, bem como os fatores relaciondos a ela;
- propor um Modelo de Avaliação de Conectividade de Sistemas Orientados por Objetos;
- elaborar uma ferramenta que viabilize a implementação do modelo proposto para sistemas desenvolvidos na linguagem Java.

1.1 Proposta de Solução

O objetivo principal deste trabalho é a proposta de um instrumento de apoio à decisão no processo de desenvolvimento de software que se baseia na avaliação da conectividade de um sistema orientado por objetos. Conforme descrito anteriormente, acreditamos que a conectividade seja o indicador principal da estabilidade de um software e, conseqüentemente, da sua manutenibilidade. Desta forma, propomos como solução para o problema investigado que a conectividade seja adotada como indicador primário na predição da dificuldade de manutenção de software.

A nossa proposta de solução não se restringe apenas à avaliação da conectividade. A idéia básica é avaliar a conectividade do sistema e, diante de um indicador de conectividade considerado insatisfatório, avaliar, então, os demais aspectos que possam ter influenciado esta situação negativa, para que se possa conhecer as causas e intervir para saná-las. Como o foco principal são os sistemas orientados por objetos, realizamos uma análise sobre as características deste paradigma a fim de, dentre elas, identificar aquelas que têm impacto importante sobre o aspecto conectividade em um sistema OO. Propomos, assim, o Modelo de Avaliação de Conectividade de Sistemas Orientados por Objetos – MACSOO – e uma ferramenta que viabiliza a implementação do modelo proposto na avaliação de softwares desenvolvidos em Java.

1.2 Contribuições da Dissertação

As seguintes contribuições são resultados deste trabalho:

1. Avaliação da relação entre conectividade e complexidade de sistemas.

   Uma avaliação é apresentada sobre conectividade de sistemas e a forma como esta característica influi na complexidade de sistemas.

2. Adaptação dos conceitos de acoplamento e coesão à luz da orientação por objetos.

3. Identificação das métricas de software orientado por objetos impactantes no aspecto conectividade de sistema.

   Tendo em vista que a conectividade é fator de peso na complexidade de sistemas e, conseqüentemente, na qualidade e custo de sistemas, uma atenção especial

deve ser dada às métricas relacionadas ao aspecto conectividade. Assim, é identificado um conjunto de métricas dentre as propostas na literatura que se relacionam com este aspecto.

4. Proposta de duas métricas auxiliares na avaliação da conectividade.

5. Proposta de MACSOO - Modelo de Conectividade em Sistemas Orientados por Objetos.

   Um modelo de avaliação de conectividade é proposto aqui, utilizando-se as métricas identificadas, bem como métricas adicionais necessárias.

6. A construção de *Connecta*, uma ferramenta de coleta de métricas de software orientado por objetos relacionadas a conectividade de sistema.

   Uma ferramenta para coleta de métricas foi construída baseada no modelo de avaliação proposto neste trabalho. A ferramenta destina-se a avaliar sistemas desenvolvidos na linguagem Java.

1.3 Estrutura da Dissertação

Esta dissertação está estruturada da seguinte forma.

Capítulo 2 apresenta e analisa os principais conceitos relacionados à qualidade de software, tais como modularidade, coesão, acoplamento, manutenibilidade, extensibilidade e reusabilidade. O objetivo deste capítulo é apresentar o resultado do estudo sobre os princípios para a construção de software modular, identificando aqueles que contribuem para a diminuição de conectividade em software.

Capítulo 3 trata sobre métricas de software, abordando sua importância e as principais métricas propostas na literatura, em particular aquelas destinadas a medição de software orientado por objetos. Neste capítulo, analisamos tais métricas, com o objetivo de identificar aquelas que auxiliam na avaliação da conectividade de softwares OO.

Capítulo 4 apresenta trabalhos relacionados ao assunto investigado nesta dissertação, em particular, um modelo de avaliação de software e ferramentas de coleta de métricas.

Capítulo 5 relata o resultado do estudo dos aspectos coesão e acoplamento à luz da Orientação por Objetos.

Capítulo 6 apresenta a tese de que a conectividade é o fator preponderante na manutenibilidade de software e descreve o Modelo de Avaliação de Conectividade de Sistemas Orientados por Objetos – MACSOO – proposto nesta dissertação.

Capítulo 7 descreve *Connecta*, a ferramenta criada com o objetivo de viabilizar a implementação de MACSOO.

Capítulo 8 relata e analisa experimentos realizados neste trabalho.

Capítulo 9 apresenta a conclusão deste trabalho e as indicações de trabalhos futuros.

Capítulo 2

Qualidade de Software

Antes de se falar em avaliação de qualidade de software, há de se definir o que é qualidade. O dicionário Aurélio da língua portuguesa [19] define qualidade como:

1. Propriedade, atributo ou condição das coisas ou das pessoas que as distingue das outras e lhes determina a natureza.
2. Superioridade, excelência de alguém ou algo.

Qualidade é palavra de ordem em quase todos os segmentos da produção humana nos tempos atuais, talvez pelo caráter competitivo que a maior parte das relações possui. A qualidade do trabalho pode significar o sucesso de um profissional, assim como a qualidade dos produtos e serviços pode significar o sucesso de uma organização. Não é diferente no caso da produção de software. Em um contexto de mercado competitivo e forte demanda por sistemas cada vez mais complexos, a qualidade de um software é o fator determinante para o seu sucesso. Assim, é importante a existência de formas de se avaliar a qualidade de software.

Pressman [44] define qualidade de software como "conformidade a requisitos funcionais e de desempenho explicitamente declarados, padrões de desenvolvimento explicitamente documentados e características implícitas, que são esperadas em todo software desenvolvido profissionalmente".

A abordagem de Bertand Meyer [36] sobre qualidade de software considera dois aspectos: as características internas e externas de um software. Quando se avalia a

qualidade de um produto qualquer, pode-se considerar basicamente dois pontos de vista: o de quem utiliza o produto e o de quem o produz. Quem utiliza o produto avalia a sua qualidade baseando-se em aspectos externos, tais como boas condições de uso e atendimento às necessidades do usuário. Os aspectos externos constituem o objetivo fim de quem se propõe a construir um produto, pois são determinantes na aceitação ou não por parte de quem irá utilizá-lo. Para atingir o objetivo de construir um produto que seja aceito, investe-se na qualidade da forma utilizada para se construir tal produto, por exemplo: utilização de matéria-prima de qualidade e boas técnicas de produção. Pode-se dizer, então, que os aspectos internos, aqueles relativos à produção, ajudam a atingir os aspectos externos.

Software é um produto e, como todo produto, tem também a sua qualidade avaliada sob dois aspectos: fatores internos e fatores externos. Os requisitos do software são referenciais para se avaliar um software do ponto de vista externo; a estrutura do software corresponde aos seus aspectos internos de qualidade. A estrutura de um software é determinada pelo seu projeto, também denominado desenho, tradução do termo inglês *design*. O projeto de software define uma estrutura que possa ser implementada e que atenda aos requisitos especificados para determinado software [42]. A forma como o software é estruturado é a chave para a obtenção de software de alta qualidade.

Projetar software é tarefa árdua e tem demandado atenção de pesquisadores e profissionais desde os primórdios da produção de software. Glenford Myers, em 1975 [38], escreveu que talvez o maior problema enfrentado naquela época fosse a extrema dificuldade e custo de criar e manter sistemas de programação de grande porte. Hoje, trinta anos depois, a situação não é diferente. É claro que muito se evoluiu nesta área: ferramentas, métodos, técnicas, linguagens, etc., estão disponíveis para quem se propõe a criar um software. Mas, a produção de software ainda conta com o mesmo problema: criar e manter softwares complexos a um custo que seja ao menos razoável.

Do ponto de vista do projeto de softwares, um dos fatores mais importantes é a modularidade. A modularidade é um mecanismo para melhorar a flexibilidade e a compreensão de um sistema, além de possibilitar a diminuição do tempo de seu desenvolvimento. Um programa modular possui partes que podem ser entendidas, implementadas e alteradas de forma independente, o que torna o software flexível, mais fácil de manter, propício à reutilização, mais fácil de testar e, conseqüentemente, impacta na redução de custo de produção de software.

O caminho para obter software com baixo grau de conectividade, construído de forma que os seus módulos sejam o mais independentes possível, é investir em aspectos de

sua construção que potencializam este objetivo. Este capítulo apresenta uma revisão da literatura sobre qualidade de software, identificando tais aspectos.

Este capítulo está estruturado da seguinte forma:

Seção 2.1 descreve os principais fatores externos de qualidade de software, identificando a influência que a conectividade tem sobre eles.

Seção 2.2 descreve o papel da modularidade na construção de software, apresenta as definições dos conceitos *coesão* e *acoplamento*, apresenta os critérios, regras e princípios para construção de software modular abordados por Meyer [36], destacando a importância desses aspectos para a conectividade.

Seção 2.3 discorre sobre a manutenibilidade de sistemas e os fatores que a determinam, enfatizando o papel da conectividade sobre este aspecto.

Seção 2.4 aborda o código como artefato principal na produção de software, destacando a importância de se investir na sua qualidade.

Seção 2.5 aponta o paradigma OO como detentor de recursos úteis na obtenção de software de qualidade, justificando o porquê da escolha deste paradigma como objeto de estudo desta dissertação.

Seção 2.6 apresenta as conclusões deste capítulo.

2.1 Fatores Externos de Qualidade de Software

Fatores externos de qualidade de software são aquelas características que podem ser avaliadas por quem utiliza o software. Bertrand Meyer [36], destaca como fatores externos de qualidade de software:

- ***Correção***: é a capacidade do software realizar as tarefas como foram definidas em sua especificação de requisitos. Certamente, este é o primeiro aspecto a ser observado em um software.
- ***Robustez***: um software é robusto se realiza as suas tarefas de forma correta mesmo quando submetido a condições anormais.

- ***Extensibilidade***: é a característica de um software poder ser facilmente adaptado a inclusões e alterações de requisitos. A importância desta característica dá-se pelo fato de que, em geral, a vida de um software é longa, e, ao longo de sua existência, alterações ou novos requisitos são inevitáveis.

- ***Reusabilidade***: é a característica de um software que pode ser reutilizado ao todo ou em parte por outros softwares. Na produção de software a possibilidade de se reutilizar elementos já construídos facilita o projeto e o seu desenvolvimento, torna o produto mais confiável e impacta principalmente no seu custo. Do ponto de vista de projeto e desenvolvimento de software, a reutilização é melhorada em função da independência dos constituintes do software. Quanto menor o nível de dependência de um módulo ou de um conjunto de módulos, maior a facilidade de reutilizá-lo, por exemplo, na construção de um outro software.

- ***Compatibilidade***: é a facilidade de se combinar o software com outros softwares. Essa característica é importante porque raramente um software é construído sem interação com outros softwares.

- ***Eficiência***: refere-se ao bom uso que o software faz dos recursos de hardware, tais como memória e processadores.

- ***Portabilidade***: é a facilidade de se utilizar o software em diferentes ambientes de hardware e software.

- ***Verificabilidade***: é a facilidade de se preparar rotinas para se verificar a conformidade do software com os seus requisitos.

- ***Integridade***: é uma característica relacionada à segurança de dados, programas e documentos. Integridade é a habilidade de proteger tais componentes contra acessos não autorizados.

- ***Facilidade de uso***: também denominada usabilidade, é a facilidade com que o software pode ser aprendido e utilizado.

O objetivo principal na produção de software é satisfazer esses fatores, pois eles determinam o sucesso do produto. A conectividade é um aspecto da estrutura de um software que permite atingir os fatores externos de qualidade de software, sobretudo a extensibilidade e a reusabilidade, que são fatores fortemente influenciados pela independência dos módulos do software.

O caminho para obter software com baixa conectividade é construi-lo de forma que suas partes constituintes sejam o mais independentes possível. Esta característica, denominada *modularidade*, é detalhada na próxima seção.

2.2 Modularidade

Modularidade é a característica de um software construído a partir de unidades básicas, denominadas módulos. É o aspecto chave para a construção de software de arquitetura flexível e, conseqüentemente, para a obtenção de softwares mais fáceis de se manter. Um bom projeto de software deve definir módulos o mais independente possível, que possam ser entendidos, implementados e alterados sem a necessidade de conhecer o conteúdo dos demais módulos e com o menor impacto possível sobre eles.

Muitas definições são dadas para módulo na literatura. Myers [38] define um módulo como um conjunto de instruções de programas com as seguintes características: fisicamente juntas na listagem do programa, limitadas por fronteiras identificáveis, coletivamente referenciadas por um nome, podem ser referenciadas pelo nome do módulo por qualquer outra parte do programa.

Para Staa [54], um módulo é um conjunto de um ou mais arquivos que podem ser compilados com sucesso. Para Bertand Meyer [37], um módulo é a menor unidade na qual o sistema pode ser decomposto. Todas essas definições convergem para a idéia de que um módulo é a unidade básica de estruturação de um sistema, são peças que se articulam para dar forma ao software. A forma como esta articulação é realizada determina a flexibilidade do sistema.

Segundo Myers [38], um projeto modular ótimo é aquele no qual o relacionamento entre elementos que não estejam no mesmo módulo são minimizados. Ghezzi e Jazayeri [24] consideram um bom módulo aquele que representa uma abstração utilizável, que interage com outros módulos de uma maneira bem definida e regular.

Myers [38] define dois caminhos principais para se obter modularidade:

1. Minimizar o relacionamento entre módulos, o que ele denominou acoplamento (*coupling*).

2. Maximizar o relacionamento entre elementos no mesmo módulo, o que ele denominou coesão (*strength*).

A seguir, são descritos estes dois conceitos importantes, tal como foram definidos por Myers [38], para a obtenção de software modular: coesão interna de módulos e acoplamento entre módulos.

2.2.1 Coesão

Coesão é a medida do relacionamento entre elementos internos de um módulo. Um bom projeto visa à maximização da coesão de módulos. Glenford Myers [38] define a seguinte classificação para a coesão interna de módulos, considerando do pior para o melhor nível:

1. **Coesão Coincidental**

   Um módulo tem coesão coincidental se não há relacionamento com significado relevante entre os seus elementos. Os elementos parecem ter sido reunidos ao acaso, portanto não é possível descrever a função do módulo. É o pior tipo de coesão interna de módulo porque módulos com essa característica são prejudiciais a todos os fatores internos de qualidade de software.
   Como exemplo, seja um programa no qual a seguinte seqüência de instruções sem relacionamento entre si apareça várias vezes no código.

   ```
   x = 3;
   scanf ("%d",y);
   if (a > 5) then b = 100;
   printf("%d", k);
   ```

   O desenvolvedor, então, para economizar linhas de código resolve criar um módulo M, que contenha tais instruções, cuja chamada substitui o conjunto de instruções em questão. O módulo M resultante possui coesão coincidental.

2. **Coesão Lógica**

   Um módulo com este tipo de coesão contém elementos que apresentam uma relação lógica entre si. É um nível de coesão melhor do que a coincidental porque aqui os elementos internos do módulo apresentam alguma relação entre si. Porém, módulos com coesão lógica executam mais de uma função, mas com uma única interface com os demais módulos e com passagem de parâmetros desnecessários.
   Um exemplo de módulo com esse tipo de coesão é aquele que se responsabiliza por abrir todos os arquivos em um programa e recebe como parâmetro um indicador que determina qual conjunto de arquivos deve ser aberto a cada chamada do módulo.
   Este tipo de coesão está associado ao acoplamento de controle e é indesejável, pois cria forte dependência com os demais módulos já que o seu conteúdo precisa ser conhecido externamente para que o mesmo possa ser utilizado.

3. **Coesão Clássica**

   Tal como no caso da coesão lógica, um módulo com este tipo de coesão possui uma série de funções logicamente relacionadas, porém são temporalmente dependentes, isto é, precisam ser executadas em conjunto e em determinada seqüência. Como não há parâmetros para determinar o fluxo de execução do módulo, este tipo de coesão introduz uma melhoria em relação à coesão lógica.
   Exemplos típicos desta coesão são módulos de inicialização e finalização em um programa. Por exemplo, seja um módulo que possua uma seqüência de instruções que realizam as seguintes ações:

```
Abre conexão com servidor
...
Captura data e hora do sistema
...
Inicializa variáveis
```

   Um módulo com esta característica tem coesão clássica porque as funções estão logicamente relacionadas, visto que se destinam a iniciar o programa, e estão relacionadas no tempo, pois devem ser executadas juntas em um momento distinto no programa.
   Myers aponta que a coesão clássica pode ser inevitável em algumas situações, por exemplo, naqueles módulos que se destinam à recuperação de

erros em sistema cujas seqüências de passos sejam do tipo: identificar o erro ocorrido, recuperar da falha e continuar a execução.

4. **Coesão Procedural**

   É similar à coesão clássica, porém, além de as funções executadas pelo módulo precisarem ser executadas em determinada seqüência, elas pertencem ao domínio do problema a ser solucionado. Ainda não é um nível ideal de coesão porque o módulo não executa uma única função.
   Como exemplo, seja um software que se destina a reproduzir músicas a partir de um CD de áudio. Suponhamos que seja um requisito funcional do software: antes de executar uma música, deve ser exibida uma imagem selecionada aleatoriamente no fundo da janela da aplicação. Um módulo deste software possui a seguinte seqüência de instruções que implementam tal requisito:

```
Exiba uma imagem de fundo selecionada aleatoriamente.
...
Reproduza a música.
...
```

   Este módulo tem coesão procedural porque o relacionamento temporal do conjunto de instruções se dá em função da aplicação, do problema a ser solucionado e não em função dos procedimentos do programa.

5. **Coesão Comunicacional**

   Um módulo com este tipo de coesão é um módulo com coesão procedural com uma característica a mais: os elementos do módulo comunicam-se entre si por meio de dados. Os elementos do módulo são dependentes de dados comuns quando referenciam um mesmo conjunto de dados ou quando trocam dados entre si. Neste caso, isso pode ser feito quando um elemento do módulo tem como entrada a saída gerada pelo elemento anterior.
   Como exemplo, seja um programa que se destina a gerenciar a matrícula de alunos de uma escola. Suponhamos que seja requisito deste software um relatório que exiba, para determinada turma, a lista de alunos que a compõe. O módulo para atender tal requisito foi implementado com a seguinte seqüência de passos.

```
Obter alunos da turma
...
Exibir lista de alunos
...
```

Este módulo tem coesão comunicacional, porque os dados obtidos no primeiro passo são entrada para o segundo passo.

6. **Coesão Funcional**

   É o mais alto grau de coesão interna de módulo. Um módulo com este tipo de coesão é caracterizado por desempenhar uma única função bem definida.
   Um módulo cujo objetivo seja calcular a raiz quadrada de um número é um exemplo de módulo com coesão funcional.

7. **Coesão Informacional**

   Um módulo com coesão informacional implementa um tipo abstrato de dados; ele armazena informações e operações, sendo que cada operação realiza um função específica e manipula as informações.
   Um módulo que implemente o tipo abstrato de dados Fila possui coesão informacional. Seja, por exemplo, um módulo que implemente uma Fila de Tarefas. A *tarefa* constitui o elemento de informação armazenado; as operações *enfileira*, *desenfileira*, *vazia* e *fazFilaVazia* realizam, cada uma delas, um função específica sobre a fila.

Deve-se buscar obter módulos com alto grau de coesão, pois módulos fortemente coesos tendem a ser mais independentes [38], o que contribui para a construção de software com baixa conectividade, mais reutilizáveis, estáveis e mais fáceis de manter.

2.2.2 Acoplamento

Acoplamento é uma medida do nível de relacionamento existente entre dois módulos. Dois módulos estão acoplados se existe algum tipo de comunicação entre eles. A forma de comunicação entre dois módulos é importante porque determina quão dependentes os módulos são entre si. Quanto mais forte for o elo entre dois módulos, mais dependentes eles são. Myers [38] define uma escala para nível de acoplamento de módulos.

Esta escala é apresentada a seguir, do pior para o melhor nível de acoplamento entre módulos.

1. **Acoplamento por Conteúdo**

   É o tipo mais forte de acoplamento entre dois módulos. Este tipo de acoplamento existe quando um módulo faz referência direta ao conteúdo do outro módulo. O acesso é feito a elementos não exportados do módulo, ou seja, o módulo detentor do conteúdo acessado não autoriza tal acesso, porém, de forma sub-reptícia, outro módulo obtém o acesso. O exemplo típico do acoplamento por conteúdo se dá devido ao uso de desvios, em linguagens mais antigas, a partir de um módulo para outro módulo com o comando GO TO.

2. **Acoplamento por Dado Comum**

   Existe este tipo de acoplamento entre um grupo de módulos quando eles compartilham uma área de estrutura de dados comum e é facultado a cada um deles uma interpretação específica da área de dados.
   Como exemplo, seja um software que mantenha o cadastro de alunos. Neste software é definido um vetor de caracteres que representa a estrutura global para os dados do aluno. Os caracteres no vetor são organizado como mostra a Figura 2.1.

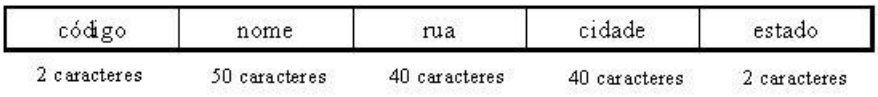


Figura 2.1: Vetor de caracteres representando dados dos alunos

   O software possui quatro módulos que usam esta estrutura, com as funcionalidades de incluir, excluir, alterar e listar alunos. Neste caso, se, por exemplo, o tamanho do campo *código* for alterado, todos os módulos sofrerão impacto desta alteração e necessitarão ser alterados também.

3. **Acoplamento Externo**

   Existe este tipo de acoplamento em grupo de módulos que referenciam um mesmo termo declarado externamente. Este tipo de acoplamento é muito próximo do acoplamento por dado comum. A diferença básica está na unidade compartilhada entre os módulos: no acoplamento externo os

módulos usam itens de dados individuais declarados em uma área de dados comum e não a área comum completa, como ocorre no acoplamento por dado comum.
Como exemplo, seja o mesmo software descrito no item anterior, porém define-se a seguinte estrutura global para os dados do aluno:

```
struct {
  char codigo[2];
  char nome[50];
  char rua[40];
  char cidade[40];
  char estado[2];
}Aluno;
```

Neste caso, se, por exemplo, o tamanho do campo *código* for alterado, somente os módulos que utilizam este elemento da estrutura sofrerão impacto desta alteração.
O dano causado por este tipo de acoplamento é menor do que aquele causado pelo acoplamento por dado comum porque uma alteração na área de dados comum ou em algum dos módulos que a utilizam, não afeta necessariamente todos os módulos envolvidos.

4. **Acoplamento de Controle**

O acoplamento entre dois módulos é desse tipo quando um módulo passa um parâmetro que determina diretamente o fluxo de execução do outro módulo.
Por exemplo, seja a seguinte função escrita em C:

```
void ExibeMensagem (int controle){
   switch (controle){
    case 1:  printf ("Erro na leitura do arquivo. \n";)
             break;
    case 2:  printf ("Erro na gravação do arquivo. \n";)
             break;
    case 3:  printf ("Dado inválido. \n";)
             break;
    default: printf ("Erro. \n";)
             break;
   }
}
```

Uma função que chame a função ExibeMensagem acima está acoplada a ela por controle, pois passa-lhe um parâmetro que define a sua execução.

5. **Acoplamento de Referência ou *Stamp***

   Ocorre quando dois módulos compartilham uma área de dados não declarada globalmente. Isso se dá quando a comunicação entre dois módulos é feita por meio de chamada de rotinas com passagem de parâmetro por referência. Este tipo de acoplamento é menos grave do que os acoplamentos por dado comum e externo, porém ainda apresenta o problema de efeito colateral de alterações sobre os dados compartilhados.

6. **Acoplamento por Informação**

   Dois módulos estão acoplados por informação se a comunicação entre eles é feita por chamada de rotina com passagem de parâmetros por valor, desde que tais parâmetros não sejam elementos de controle.
   Se dois módulos precisam se comunicar, este é o melhor nível de acoplamento que pode existir entre eles, pois reflete uma situação ideal em que:

   - um módulo não conhece detalhes do outro;
   - um módulo recebe apenas valores enviados pelo outro e pode manipular tais informações na sua execução sem efeito colateral em outros módulos.

7. **Desacoplado**

   Dois módulos estão desacoplados se não existe tipo algum de comunicação entre eles.

Myers propôs as classificações para acoplamento e coesão em uma época em que o paradigma estruturado era o mais difundido. Assim, os conceitos têm base nas possibilidades de projeto e implementação decorrentes dos recursos disponíveis pelas linguagens de programação utilizadas na época, como Fortran e PL/I. Embora estas escalas tenham sido propostas há muito tempo, as observações a respeito dos impactos de cada nível de acoplamento entre módulos e coesão interna de módulos no custo e qualidade na produção de software são válidas para o paradigma da orientação por objetos. Porém, um novo olhar sobre essa classificação à luz da orientação por objetos se faz necessário, para que se possa detalhar como esses tipos de acoplamento e coesão são personificados pelos conceitos e recursos desse paradigma.

O Capítulo 5 desta dissertação apresenta uma análise sobre as formas de acoplamento entre módulos e coesão interna de módulos no contexto da orientação por objetos.

2.2.3 Construção de Software Modular

A procura por técnicas que auxiliem a construção de software modular tem sido motivação para vários trabalhos na área de produção de software [31, 38, 39, 40, 21]. Tamanha importância dada ao assunto não é casual. Como Pressman aponta [34], a modularidade é a chave para um bom projeto, que por sua vez é chave para a qualidade de software.

O projeto de software deve ser orientado a obter o maior grau de modularidade possível. Para isso, o projetista deve utilizar um método que viabilize atingir tal objetivo. Bertrand Meyer [36] define critérios, regras e princípios para se alcançar modularidade. Os critérios são os requisitos que o método utilizado para criar o software deve atender para ser dito modular. Eles são independentes entre si, isto é, um método pode atender um critério e ferir outro. As regras sucedem os critérios, e são normas a serem seguidas para se atingi-los. Os princípios decorrem das regras, e são características necessárias para que as regras sejam aplicadas corretamente.

Os critérios para que um método seja modular são:

- Decomposibilidade: um método atende a esse critério se permite a construção de software a partir da decomposição do problema a ser resolvido, dividindo-se o problema em subproblemas.
- Composibilidade: um método atende a esse critério se permite a construção de um software a partir de elementos já existentes. Esse critério está relacionado à reusabilidade.
- Inteligibilidade: em um método que segue esse critério, é possível entender um módulo sem a necessidade de recorrer aos demais módulos.
- Continuidade: o método que segue esse critério gera sistemas com estrutura tal que uma alteração em um módulo resulta em poucas alterações nos demais módulos.
- Proteção: esse critério determina que a ocorrência de um erro dentro de um módulo deve gerar pouca propagação de erros para os demais módulos.

Verifica-se em todos os critérios a preocupação com a minimização de dependência entre os módulos do sistema, visando uma estrutura mais flexível.

As regras a serem seguidas para se alcançar os critérios expostos são:

- Mapeamento direto: essa regra determina que deve ser possível definir um modelo para o domínio do problema a ser resolvido. Então, deve haver uma correspondência direta entre tal modelo e a solução dada para o problema.
- Poucas interfaces: a estrutura do sistema deve conter poucas interfaces entre os módulos que o compõe. Essa regra relaciona-se diretamente com o conceito de conectividade e é a chave para atingir os critérios de modularidade.
- Interface pequena: essa regra está relacionada ao conceito de acoplamento de módulos. Ela determina que se dois módulos se comunicam, eles devem trocar o menor número de informação possível.
- Interface explícita: se dois módulos se comunicam, tal comunicação deve estar clara no texto de ambos.
- Ocultação de informação: as informações que são de conhecimento relevante para os demais módulos devem ser publicadas, as demais devem estar ocultas, sendo de conhecimento somente do módulo proprietário.

Os princípios decorrentes dessas regras são os seguintes:

- Unidade modular lingüística: um módulo deve corresponder a uma unidade sintática na linguagem utilizada para representar o sistema, seja uma linguagem de especificação, de projeto ou de implementação.
- Auto-documentação: um módulo deve ser auto-documentado. A documentação de um módulo deve estar fisicamente próxima dele, para garantir a sua legibilidade.
- Acesso uniforme: esse princípio relaciona-se com a regra de ocultação de informação. Os serviços oferecidos por um módulo devem ser disponíveis por meio de uma notação definida e conhecida externamente, não sendo necessário que os clientes de tais serviços tenham conhecimento de seus detalhes.
- Aberto-fechado: este princípio determina que um módulo deve estar aberto para extensão e fechado para modificação. Isto significa que deve ser possível estender o comportamento do módulo, contudo sem modificar o seu código fonte. Módulos que seguem este princípio propiciam a reusabilidade e a manutenibilidade.

- Escolha única: se um sistema possui um conjunto de alternativas de execução, somente um módulo deve conhecer essa lista de opções.

A maior parte dessas orientações de Meyer refletem o cuidado que se deve tomar em relação às comunicações entre os módulos constituintes de um sistema, e leva a uma direção em que se obtém um software constituído por módulos que possuem o mínimo de conexões possíveis entre si. Isso é fundamental para a manutenibilidade, questão abordada na próxima seção.

2.3 Manutenibilidade

Manter um software significa garantir a sua existência. Quando a implementação de um software é finalizada, não se pode dizer que ele está pronto e acabado. Provavelmente, ao longo de sua existência, alguma modificação ele terá que sofrer, seja por uma alteração de requisito, pela identificação de um requisito novo ou para a correção de um erro. A manutenibilidade é a medida da facilidade de se manter um software.

A manutenção de software é ponto crítico para a Engenharia de Software devido ao peso que ela tem no custo total de um sistema. Estatísticas relatadas na literatura mostram que os custos de manutenção de um software são os maiores de todo o seu ciclo de vida. Pfleeger [43] alerta que atualmente o custo de manutenção de software é acima de 80% do custo do sistema. Para Pfleeger, estão entre os fatores que contribuem para o aumento do custo de manutenção de software:

- complexidade do tipo de aplicação: a dificuldade de manutenção é um problema inerente a aplicações de caráter complexo;
- remanejamento de pessoal (*turnover*): a rotatividade de pessoal nas equipes dos sistemas impacta no tempo e no custo de manutenção porque um novo membro na equipe demandará tempo de aprendizado para estar apto a manter o software;
- qualidade da documentação: a inexistência de documentação de especificação de requisitos, de projeto e implementação torna a atividade de manutenção

impossível. As informações contidas na documentação devem estar corretas e atualizadas, caso contrário a manutenção também será inviabilizada;

- qualidade do projeto: a qualidade do projeto é definida pela existência de componentes coesos e independentes. Quando tais características não estão presentes no projeto, o seu entendimento é prejudicado e a facilidade de manutenção é seriamente comprometida;
- qualidade do código: se o código não segue as determinações do projeto, a dificuldade de dar manutenção é muito grande.

A chave para minimizar o custo de manutenção está na arquitetura do sistema. Uma arquitetura de sistema flexível, que atenda aos preceitos de um bom projeto modular, contribui de forma determinante para amenizar os problemas acima citados.

Dentre esses fatores, destacamos, na próxima seção, a importância da *qualidade do código* como ponto crítico na manutenibilidade, tendo em vista que a manutenção de fato se dá sobre o produto. Se o produto for mal construído, a sua manutenção é gravemente prejudicada, ainda que problemas nos demais fatores, como remanejamento de pessoal e qualidade da documentação, sejam superados.

2.4 A Importância da Qualidade do Código de um Software

A gestão de qualidade deve ser feita durante todo o processo de desenvolvimento do software [14]. No processo de software, entende-se como artefato qualquer resultado tangível que seja fruto de alguma etapa no processo ou sirva como insumo para alguma delas [42]. Por exemplo: a proposta de software, o documento de especificação de requisitos, o modelo de dados e o código do sistema. É importante a verificação da qualidade dos artefatos durante todo o processo, pois quanto mais cedo se identificar um problema, menor será o custo de seu ajuste [44] [42].

Um ponto importante na obtenção de software de qualidade é a especificação dos requisitos. Uma especificação de requisitos deve ser completa, correta, precisa e verificável [42], pois nela deve estar definido o objetivo do software, os requisitos que ele deve atender. Contudo, a fase determinante da qualidade do software é o

projeto, pois nesta fase define-se como o software será construído. O projeto deve ser criterioso, atender aos requisitos especificados e modelar o sistema de forma a obter uma estrutura flexível. Mas, de nada adianta uma boa especificação de requisitos, um projeto cuidadoso se o produto final - o código - não refletir todos os cuidados que precederam sua implementação. O código é o software propriamente dito, é o artefato mais importante de todo o processo, o objeto de maior valor, pois é com ele que o usuário interage diretamente e é ele quem deve estar preparado para sofrer possíveis alterações ao longo de sua vida. A especificação de requisitos, o projeto, os testes, a manutenção, tudo tem como objeto central o código.

Não se pode correr o risco de confiar demasiadamente nas avaliações realizadas nos artefatos que precedem a implementação do código e dispensar a sua própria avaliação. Questões como legibilidade, eficiência das rotinas utilizadas, tratamento de exceções etc, [42] são relevantes na avaliação do código de um software. Porém, assim como no caso do projeto do software, a estrutura do software implementado é determinante para a sua manutenção, reutilização e para o seu custo. A forma utilizada para implementar deve guardar com rigor os cuidados para se obter software modular. Assim, é necessária a existência de modelos de avaliação de qualidade de código de software do ponto de vista de sua estrutura. E mais ainda, é necessária a existência de ferramentas que viabilizem tal avaliação, tendo em vista o porte que os sistemas reais em geral assumem.

2.5 A Orientação por Objetos e a Qualidade de Software

A orientação por objetos é um paradigma amplamente difundido [36] [2]. Tamanha aceitação desta forma de se conceber, projetar e implementar um software deve-se ao fato de que a orientação por objetos possui características que possibilitam a obtenção de software de alta qualidade, mais fáceis de manter e de reutilizar.

A base da orientação por objetos é a diminuição da distância semântica entre o mundo real e a modelagem do sistema [36]. O software é construído a partir dos objetos que o sistema manipula e da troca de mensagens que ocorre entre os objetos. Classes de objetos são abstrações utilizadas para representar um grupo de objetos com as mesmas características e comportamentos, isto é, representa objetos que possuem em comum um conjunto de atributos e métodos. O recurso de herança possibilita que se modelem eventos do mundo real nos quais uma classe de objetos herda as

características e o comportamento de uma ou mais classes, formando o que se chama hierarquia de classes.

As características e os recursos da orientação por objetos - classes, herança, ligação dinâmica e polimorfismo - permitem atingir fatores de qualidade de software com mais facilidade porque viabilizam a obtenção de sistema de estrutura muito flexível. Tal flexibilidade estrutural impacta em maior extensibilidade, maior reusabilidade e maior facilidade de manutenção. As classes de objetos encapsulam dados e serviços, o que permite a construção de software modular. Quando bem empregado, esse recurso permite o mínimo de acoplamento entre as classes, o que contribui significativamente para a redução de custos de manutenção e para a reusabilidade. Somado a isso, o polimorfismo, que é a habilidade de se esconder implementações distintas atrás de interfaces comuns, torna os softwares orientados por objetos menos sensíveis a alterações, o que os tornam mais extensíveis [16].

Por tudo isso, como temos por objetivo contribuir para a construção de software de qualidade, fácil de manter e estruturado com baixa conectividade, o objeto de estudo desta dissertação é paradigma OO.

2.6 Conclusão

Software de boa qualidade é sinônimo de software que atende às necessidades dos usuários e é construído de forma que seja fácil de entender, estender, reutilizar e manter. Esses fatores dependem primordialmente do nível de independência entre os módulos do software.

O caminho principal para a obtenção de software com essa característica é construí-lo a partir de módulos fortemente coesos e com o menor grau de acoplamento entre si [38]. Conforme descrito neste capítulo, a literatura sobre este assunto aborda métodos, regras e princípios que levam a obtenção de software constituído por módulos o mais independentes possível uns dos outros.

A tese desta dissertação é que a conectividade, a medida de interconexões entre módulos de um software, fornece uma avaliação primária da qualidade de sua estrutura e, conseqüentemente, da facilidade de sua manutenção. A alta conectividade é sinal de que o software não foi construído de forma a manter a independência entre os seus módulos, o que compromete a sua qualidade e a sua manutenibilidade. Esta

idéia é abordada em maiores detalhes no Capítulo 6. Neste contexto, surge a necessidade de se identificar meios de se avaliar a conectividade em sistemas, em particular, aqueles desenvolvidos no paradigma orientado por objetos, que é o foco de estudo deste trabalho.

Muitas vezes, a qualidade de um software é avaliada na base do senso comum, o que não propicia uma avaliação objetiva do software. No processo de desenvolvimento de software, é importante a existência de métodos de controle de qualidade objetivos, o que garante uma gerência no processo de software mais segura. É preciso quantificar, comparar, avaliar para que se possa controlar. As métricas de software assumem papel de extrema importância, pois permitem a avaliação quantitativa dos produtos, das pessoas e do próprio processo.

O próximo capítulo apresenta uma revisão bibliográfica sobre métricas de software orientado por objetos, a fim de se identificar um conjunto de métricas a serem utilizadas na avaliação de conectividade em sistemas orientados por objetos.

Capítulo 3

Métricas de Software

Avaliar é uma necessidade em vários segmentos da produção humana. A avaliação quantitativa fornece ao homem uma representação matemática do mundo real, possibilita-lhe aferir, comparar, acompanhar e saber quando é necessário interferir para melhorar, de maneira a garantir maior qualidade na sua produção. Essa necessidade faz-se presente também na produção de software e tem-se evidenciado com o aumento da demanda por sistemas cada vez mais complexos.

Alguns termos são comumente utilizados na literatura que trata sobre métricas de software, dentre eles: *medição (measurement), métrica (metric), medida (measure).* Staa [54] define esses termos da seguinte maneira: *metrica* é um padrão de medição, a unidade de grandeza a ser medida; *medição* é o ato de medir algo de acordo com uma métrica; *medida* é o efeito, o resultado da medição.

No contexto da Engenharia de Software, métrica é um padrão de medição para avaliar determinado atributo de algo que esteja relacionado ao software. De acordo com Berard [9], métricas de software são utilizadas para avaliar produtos, processos e pessoas. Assim, métricas de software têm papel fundamental, pois a avaliação de tais elementos possibilita, dentre outros aspectos: a definição quantitativa do sucesso ou a falha de determinado atributo; a identificação da necessidade de melhorias do atributo avaliado; a tomada de decisão gerencial e técnica; a realização de estimativas. Para que uma métrica seja realmente útil é preciso ter em mente o que exatamente pretende-se medir, que tipo de informação tal medida fornecerá e a que papel ela se prestará, como a métrica será coletada e como será avaliada [9] [37].

Como Meyer avalia em [37], há uma extensa literatura sobre métricas de software [1] [2] [7] [8] [9] [12] [13] [18] [20] [21] [25] [26] [32] [37] [50]. O interesse pela área vem de algumas décadas. O primeiro livro sobre assunto [25], como o próprio autor destaca na obra, data de meados da década de 70. Um estudo realizado por Xenos et al. [61] mostra que o universo de métricas que têm sido propostas chega a algumas centenas. Como destaca Berard [9], neste universo, embora as métricas tradicionais sejam úteis no processo de desenvolvimento de software, elas não são suficientes, algumas inclusive são inadequadas, para se avaliar software OO devido às características peculiares deste paradigma tais como herança, polimorfismo, ocultação de informação e encapsulamento. Assim, existe um grupo específico de métricas propostas para avaliar software orientado por objetos, dentre as quais destacam-se os conjuntos CK [13] e MOOD [2].

Fenton e Neil [18] apontam que o desafio na área de métricas de software é utilizar as métricas de software existentes, que sejam relativamente simples, na construção de ferramentas de apoio à decisão. O trabalho desenvolvido nesta dissertação segue essa idéia. Propomos um Modelo de Avaliação de Conectividade em Sistemas Orientados por Objetos – MACSOO – cujo objetivo é ser um instrumento de apoio à decisão no processo de software para a redução de conectividade. Este modelo está descrito no Capítulo 6.2. Propomos também uma ferramenta, apresentada no Capítulo 7, que visa a implementação do modelo proposto para sistemas desenvolvidos em Java.

Como o propósito desta dissertação é a avaliação da conectividade de sistemas orientado por objetos, buscamos, neste capítulo, realizar uma revisão da literatura sobre métricas de software para este paradigma, construindo conhecimento sobre o assunto, a fim de selecionar, dentre tais métricas, um conjunto delas a serem utilizadas em MACSOO.

3.1 Métricas de Software Orientado por Objetos

Dois conjuntos de métricas para orientação por objetos são amplamente citados na literatura: o conjunto de métricas proposto por Chidamber e Kemerer [13], conhecido como CK, e o proposto por Abreu e Carapuça [2], conhecido como MOOD. Dada a importância de tais conjuntos de métricas, a seguir eles são descritos em maiores detalhes.

3.1.1 Métricas CK

Chidamber e Kemerer [13] propõem um conjunto de métricas para projeto orientado por objetos, referenciadas na literatura como métricas CK. Nesse trabalho, os autores apresentam seis métricas, validam-nas usando os critérios de avaliação de métricas propostos por Weyukers [59] e relatam os resultados de experimentos realizados com as métricas propostas. Para a realização dos experimentos, foram utilizados dois sistemas, um escrito em C++ e outro em Smalltalk. O conjunto CK é constituído pelas seguintes métricas: WMC (Weighted Methods per Class), DIT (Depth of Inheritance Tree), NOC (Number of Children), CBO (Coupling between Object Class), RFC (Response for a Class) e LCOM (Lack of Cohesion in Methods).

- **WMC *(Weighted Methods per Class)*:** é uma métrica que representa a complexidade da classe por meio de seus métodos. O cálculo da métrica é dado pelo somatório das complexidades dos métodos que constituem a classe. Fica em aberto a definição para complexidade. Os autores não determinam um cálculo específico da complexidade dos métodos visando à flexibilidade da métrica WMC. Segundo Chidamber e Kemerer, esta métrica é um indicador de custo de desenvolvimento e manutenção de uma classe, assim como do grau de reúso da classe.
- **DIT *(Depth of Inheritance Tree)*:** indica a posição de uma classe na árvore de herança de um software, que é dada pela distância máxima da classe até a raiz da árvore. Essa métrica é considerada um indicador da complexidade de desenho e de predição do comportamento de uma classe, visto que quanto maior a profundidade da classe na árvore de herança, mais classes, e portanto mais métodos e atributos, estarão envolvidos na análise.
- **NOC *(Number of Children)*:** indica a quantidade de sub-classes imediatas de uma classe. É um indicador da importância que uma classe tem no sistema, pois quanto mais sub-classes possuir uma classe, maior a importância de seu teste no sistema.
- **CBO *(Coupling between Object Class)*:** é um totalizador do número de classes às quais uma determinada classe está acoplada. Para Chidamber e Kemerer, o acoplamento entre duas classes existe quando métodos de uma delas usa métodos ou variáveis de instância da outra. A razão da existência desta métrica é justificada pelos autores pela necessidade de redução de acoplamento entre classes de objetos para atender fatores como melhoria de modularidade e aumento de reusabilidade.

- **RFC *(Response for a Class)*:** apresenta o resultado do número de métodos que podem ser executados em resposta a uma mensagem recebida por um objeto da classe. Este resultado é dado pela quantidade de métodos da classe somada à quantidade de métodos invocados por cada método da classe. Visto que RFC considera a ativação de métodos de outras classes, ela é, como CBO, um indicador de conectividade de uma classe. Enquanto CBO mostra a quantas outras classes uma classe está conectada, RFC é um detalhamento desta informação, pois apresenta por quantos caminhos uma classe está conectada a outras classes.

- **LCOM *(Lack of Cohesion in Methods)*:** é uma métrica da ausência de coesão entre os métodos de uma classe. Chidamber e Kemerer consideram que a coesão entre os métodos de uma classe é definida pela similaridade entre eles. A avaliação da similaridade entre dois métodos é determinada pelo uso de variáveis de instância da classe por eles. Seja P o conjunto formado pelos pares de métodos que não possuem variáveis de instância em comum e Q o conjunto formado pelos pares de métodos que possuem variáveis de instância em comum. Se nenhum método da classe utiliza variáveis de instância da classe, P = ϕ. O cálculo de LCOM é dado por:

  $LCOM =\mid P \mid - \mid Q \mid$, se $\mid P \mid > \mid Q \mid$
  $LCOM = 0$, caso contrário

  LCOM indica o número de pares de métodos de uma classe que têm similaridade 0, isto é, que não têm em comum variável de instância alguma da classe. Baixos valores para essa métrica indicam bom nível de similaridade, portanto de coesão, entre os métodos da classe avaliada.

3.1.2 Métricas MOOD

O conjunto de métricas MOOD (Metrics for Object Oriented Design) foi proposto por Abreu e Carapuça [2]. Nesse trabalho, além de serem definidas as métricas que compõem o conjunto MOOD, os autores apresentam sete critérios, a seguir, que os orientaram na definição de tais métricas.

- Definição formal de métricas: o significado de uma métrica deve ser formalmente definido. A definição formal de uma métrica elimina possíveis subjetividades, o

que garante maior confiabilidade na avaliação de métricas de software.

- Independência de tamanho de software: métricas que não são relacionadas a tamanho de software devem ser independentes desse fator. O valor resultante de uma métrica deve proporcionar uma avaliação do software sem a necessidade de consideração do seu tamanho.

- Unidade de medida consistente: o resultado de uma métrica deve ser representado por uma unidade de medida que possibilite a sua análise de forma objetiva.

- Obtenção precoce: quanto mais cedo puder obter-se métricas no processo de software, melhor, pois problemas identificados nas fases iniciais custam menos do que aqueles identificados em fases tardias do processo de software. Cabe ressaltar que, embora seja um fato que problemas identificados em fases tardias do processo de software tenham custo muito elevado, com demonstra Boehm [10], isso não quer dizer que somente as métricas coletadas nas fases iniciais sejam importantes. Métricas são instrumentos úteis em todo o processo de software.

- Escalabilidade: uma métrica deve servir para avaliar tanto um sistema inteiro como parte dele.

- Facilidade de computação: a computação de uma métrica deve ser fácil para que a sua utilização seja viável.

- Independência de linguagem: uma métrica não deve ser definida em termos de uma linguagem específica.

As métricas MOOD avaliam os aspectos de herança, ocultação de informação, acoplamento, polimorfismo e reusabilidade em um software orientado por objetos. Compõem o conjunto MOOD as seguintes métricas: Fator Herança de Método (*Method Inheritance Factor*), Fator Herança de Atributo (*Attribute Inheritance Factor*), Fator Acoplamento (*Coupling Factor*), Fator Agrupamento (*Clustering Factor*), Fator Polimorfismo (*Polymorphism Factor*), Fator Ocultação de Método (*Method Hiding Factor*) , Fator Ocultação de Atributo (*Attribute Hiding Factor*), Fator Reúso (*Reuse Factor*).

O cálculo de uma métrica MOOD é dado por uma razão, onde o numerador é o número de ocorrências encontradas no sistema para o aspecto avaliado e o denominador é o maior número possível de ocorrências no sistema para tal aspecto. Desta forma, o resultado de qualquer métrica MOOD é sempre um valor entre 0 e 1, o que representa um percentual de ocorrência para o aspecto avaliado no sistema. Esse tipo

de resultado é apropriado porque fornece uma dimensão para a métrica independente do tamanho do sistema avaliado, o que torna possível comparar sistemas que possuam tamanhos e características distintos. A seguir, cada métrica é apresentada e exemplificada. O diagrama de classes apresentado na Figura 3.1 será utilizado para exemplificar o cálculo de algumas dessas métricas. Ele foi construído utilizando-se notação UML [49] e mostra a estrutura simplificada de classes de objetos encontrados em um sistema acadêmico: pessoas, alunos, professores, disciplinas, matrícula e turmas. Neste diagrama, as classes são representadas por retângulos divididos em três compartimentos: o primeiro contém o nome da classe; o segundo, os atributos da classe e o terceiro contém a assinatura dos métodos no formato nome do método (lista de parâmetros): tipo de retorno. O símbolo "+" que precede o nome de um membro da classe indica que ele é público e o símbolo "-" indica que o membro é privado.

1. **Métricas para Avaliação de Herança**

   Herança é o recurso da orientação por objetos que permite criar classes a partir de classes já existentes. Por meio da herança obtém-se a reutilização de estruturas que já estão definidas e possivelmente depuradas e testadas, o que é um ponto considerável na redução do custo da produção do software. Porém, outro aspecto deve ser observado em relação a herança, como apontam Chidamber e Kemerer [13]: árvores de herança muito profundas conferem maior complexidade ao software, o que é um fator negativo para a sua manutenção.

   Um indicador que permita a avaliação da herança em um sistema é, então, um instrumento útil para a predição do esforço de sua manutenção. As seguintes métricas para este aspecto fazem parte de MOOD: MIF (Fator Herança de Métodos) e AHF(Fator Herança de Atributos). Elas são descritas a seguir.

   - **MIF (Fator Herança de Método):** essa métrica indica o percentual de métodos herdados no sistema. Para a definição da métrica, são considerados os seguintes conceitos:
     - Métodos herdados: são os métodos que uma classe possui em decorrência de herança e que não foram redefinidos na classe.
     - Métodos novos: são métodos criados na classe, que não foram herdados nem redefinidos.
     - Métodos redefinidos: são métodos herdados que têm uma redefinição na classe.
     - Métodos definidos: englobam os métodos novos e os métodos redefinidos na classe.

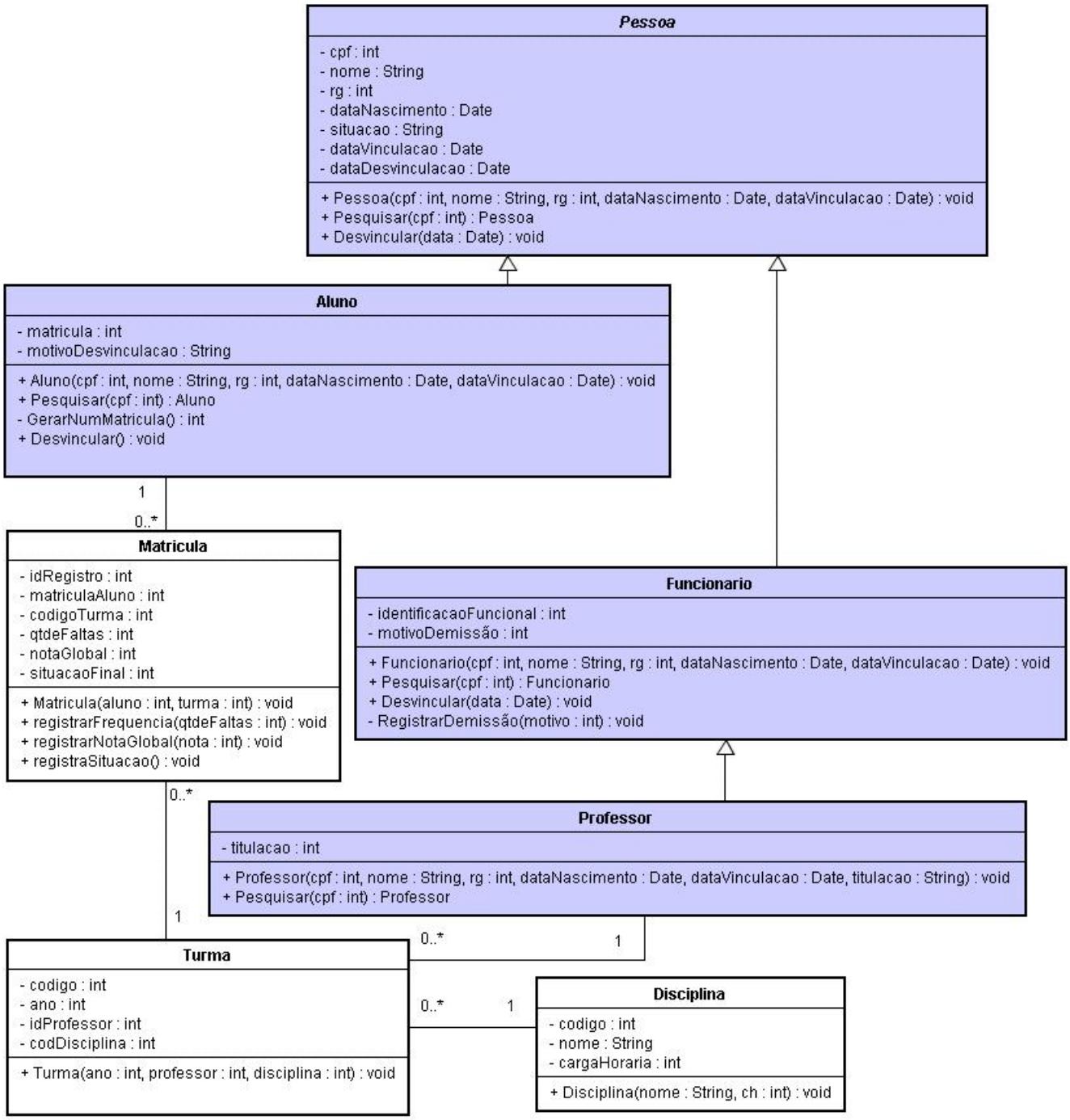


Figura 3.1: Exemplo de estrutura de classes em um sistema acadêmico

- Métodos disponíveis: é a totalidade de métodos que uma classe possui, o que engloba métodos definidos nela e os métodos herdados por ela.

O cálculo de MIF é realizado da seguinte forma: para cada classe do sistema, verifica-se a quantidade de métodos herdados e a quantidade de métodos disponíveis. O valor de MIF é dado pela razão entre o somatório de métodos herdados de cada classe do sistema e o somatório de métodos disponíveis de cada classe do sistema.

A Tabela 3.1 detalha o cálculo de MIF para o sistema representado no diagrama de classes da Figura 3.1.

Classe	Métodos				
	Herdados	*Novos*	*Redefinidos*	*Definidos*	*Disponíveis*
Pessoa	0	3	0	3	3
Aluno	1	2	2	4	5
Funcionario	1	1	2	4	5
Professor	4	1	1	2	6
Matricula	0	4	0	4	4
Turma	0	1	0	1	1
Disciplina	0	1	0	1	1
Total do sistema	6	-	-	-	25
MIF = Total de métodos herdados / Total de métodos disponíveis = 6/25 = 0.24					

Tabela 3.1: Exemplo de cálculo da métrica MIF

MIF com valor igual a 0 indica que no sistema em questão não houve utilização efetiva do recurso de herança de métodos, o que significa que não existe relacionamento algum de herança entre as classes do sistema ou se existe, todos os métodos herdados foram redefinidos. Valores de MIF próximos de 1 indicam alta utilização do recurso de herança de métodos no sistema. Um valor igual a 1 para MIF indica que todos os métodos disponíveis em todas as classes do sistema são herdados. Esta situação parece estranha, mas sua ocorrência é possível, visto que uma métrica pode ser utilizada para avaliar um conjunto de classes particular de um sistema.

MIF indica, portanto, se o recurso de herança de métodos foi explorado amplamente no sistema, o que é um instrumento na predição no custo de manutenção do sistema. Quanto menor o valor de MIF, maior o custo de manutenção do sistema, pois valores baixos para esta métrica indicam que há pouca reutilização de métodos já definidos e possivelmente depurados

e testados.

- **AIF (Fator Herança de Atributo):** indica o percentual de atributos herdados no sistema. Um raciocínio similar ao realizado no cálculo do fator herança de métodos é realizado para o fator herança de atributos AIF. O valor de AIF é dado pela razão entre o somatório de atributos herdados de cada classe do sistema e o somatório de atributos disponíveis de cada classe do sistema.

  A Tabela 3.2 detalha o cálculo de AIF para o sistema representado no diagrama de classes da Figura 3.1.

Classe	Atributos				
	Herdados	*Novos*	*Redefinidos*	*Definidos*	*Disponíveis*
Pessoa	0	7	0	7	7
Aluno	7	2	0	2	9
Funcionario	7	2	0	2	9
Professor	9	1	0	1	10
Matricula	0	6	0	6	6
Turma	0	4	0	4	4
Disciplina	0	3	0	3	3
Total do sistema	23	-	-	-	48
AIF = Total de atributos herdados / Total de atributos disponíveis = 23/48 = 0.48					

Tabela 3.2: Exemplo de cálculo da métrica AIF

As conclusões a cerca dos valores obtidos para esta métrica são similares às referentes à métrica MIF: valores próximos de 0 indicam pouca utilização do recurso de herança de atributos e o oposto, valores próximos de 1, indicam boa utilização de tal recurso. Entretanto, a importância da métrica AIF é menos relevante do que a da métrica MIF, pois, como apontam Abreu e Carapuça [2], o custo de manutenção dos métodos das classes que compõem o sistema tem peso muito maior no custo de manutenção do sistema do que os custos de manutenção decorrentes dos atributos das classes.

2. **Métricas para Avaliação de Ocultação de Informação**

   A ocultação de informação é um conceito importante relacionado a modularidade, pois a sua aplicação potencializa a independência de módulos. Quanto mais as informações e os serviços de uma classe estiverem confinados dentro

dela, menor é a necessidade de as demais classes conhecerem sua organização interna e mais fraco é o nível de interdependência entre elas. Uma classe deve ser conhecida somente pelos serviços que ela disponibiliza. Na orientação por objetos, a ocultação de informação é obtida pelo uso de atributos e métodos privados nas classes.

Uma métrica de ocultação de informação em um sistema é um indicador que influencia a avaliação da modularidade do sistema porque reflete quão restritas estão as informações pertencentes aos módulos do software. As seguintes métricas para avaliação de ocultação de informação em sistemas orientados por objetos fazem parte de MOOD: MHF (Fator Ocultação de Métodos) e AHF (Fator Ocultação de Atributos). Elas são descritas a seguir.

- **MHF (Fator Ocultação de Método):** esta métrica representa o percentual de métodos ocultos no sistema. Para o seu cálculo, os seguintes conceitos são considerados:
  - Métodos visíveis: são os métodos que constituem a interface da classe.
  - Métodos ocultos: são os métodos privados da classe.
  - Métodos definidos: são os métodos visíveis mais os métodos ocultos da classe.

  MHF é dado pela razão entre o número de métodos ocultos em todas as classes e o número de métodos definidos em todas as classes.

  A Tabela 3.3 detalha o cálculo de MHF para o sistema representado no diagrama de classes da Figura 3.1.

Classe	Métodos		
	Visíveis	*Ocultos*	*Definidos*
Pessoa	3	0	3
Aluno	3	1	4
Funcionario	3	1	4
Professor	2	0	2
Matricula	4	0	4
Turma	1	0	1
Disciplina	1	0	1
Total do sistema	-	2	19
MHF = Total de métodos ocultos / Total de métodos definidos = 2/19 = 0.11			

Tabela 3.3: Exemplo de cálculo da métrica MHF

Valores próximos de 1 para a métrica MHF indicam um alto nível de ocultação de métodos das classes do sistema. Esse tipo de resultado reflete que, de uma forma geral, as classes do sistema exportam poucos serviços, o que deve propiciar baixa conectividade entre as classes do sistema. O contrário ocorre quando se obtém valores próximos a 0 para essa métrica, o que indica que as classes do sistema exportam muitos serviços, portanto favorecem alto grau de conectividade entre as classes do sistema.

- **AHF (Fator Ocultação de Atributo):** essa métrica é o percentual de atributos ocultos no sistema. Similarmente a MHF, o cálculo de AHF é dado pela razão entre o número de atributos ocultos em todas as classes e o número de atributos definidos em todas as classes.

  A Tabela 3.4 detalha o cálculo de AHF para o sistema representado no diagrama de classes da Figura 3.1.

Classe	Atributos		
	Visíveis	*Ocultos*	*Definidos (visíveis + ocultos)*
Pessoa	0	7	7
Aluno	0	2	2
Funcionario	0	2	2
Professor	0	1	1
Matricula	0	6	6
Turma	0	4	4
Disciplina	0	3	3
Total do sistema	0	25	25
AHF = Total de atributos ocultos / Total de atributos definidos = 25/25 = 1			

Tabela 3.4: Exemplo de cálculo da métrica AHF

A ocultação de atributos é característica de extrema importância para garantir a independência entre módulos, pois impossibilita a ocorrência dos tipos mais graves de acoplamento que podem existir entre duas classes. Quando uma classe torna público um atributo, outras classes do sistema podem alterar o valor desse dado e, então, perde-se a garantia da sua integridade e estabelece-se uma forte dependência entre todas as classes que fazem uso de tal atributo. Conhecer o grau de ocultação de informação de atributos de um sistema é saber quão propenso é o surgimento de acopla-

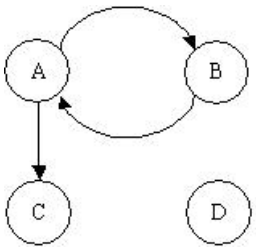


Figura 3.2: Exemplo de conexões em um sistema

mentos desse tipo no sistema.

Valores de AHF próximos a 1 indicam que poucos atributos no sistema em questão são públicos. A situação ideal é que nenhum atributo seja público, o que resulta em AHF igual a 1. O pior caso é um valor igual a 0 para esta métrica, que indica que todos os atributos de todas as classes do sistema são públicos.

3. **Métricas para Acoplamento**

   Métricas que possibilitam análise sobre os acoplamentos existentes em um sistema são úteis na predição do custo da manutenção do mesmo. MOOD contém as seguintes métricas para acoplamento: COF (Fator Acoplamento) e CLF(Fator Agrupamento). Elas são descritas a seguir.

   - **COF (Fator Acoplamento):** para a avaliação de acoplamento, Abreu e Carapuça [2] consideram o conceito de relação *cliente-servidor* entre as classes constituintes de um software. Segundo esse conceito, uma classe A é cliente de uma classe servidora B quando A referencia pelo menos um membro de B, seja este membro uma variável de instância ou um método. Uma relação cliente-servidor entre duas classes corresponde à existência de uma conexão entre elas. A Figura 3.2 é um exemplo de representação de conexões existentes entre classes de um software. Nesse exemplo, A é cliente de B, B é cliente de A e A é cliente de C.

     Em um software com n classes, o maior número possível de conexões é $n^2 - n$. A métrica COF é dada pela razão entre o número total de conexões existentes entre as classes do software e o maior número possível de conexões para o software. Assim, o cálculo de COF para o software representado na Figura 3.2 é dado por $3/(4^2 - 4) = 0.25$. Um software totalmente conectado possui COF = 1.

     COF é uma métrica importante pois indica quão conectado é um software. Um software fortemente conectado possui estrutura rígida, baixo grau de

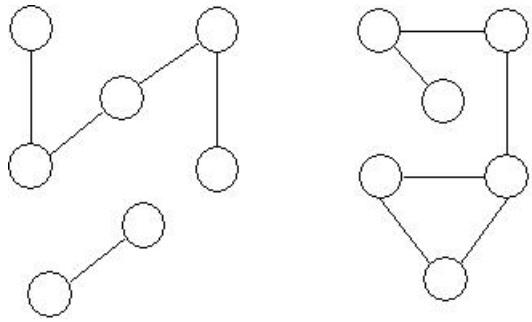


Figura 3.3: Agrupamentos de classes

independência entre os módulos e, conseqüentemente, o custo na sua manutenção é explosivo.

- **CLF (Fator Agrupamento):** esta métrica é um indicador do nível agrupamento de classes – *clustering* - em um software. Abreu e Carapuça consideram a representação de softwares por meio de um grafo para definir essa métrica, onde as classes são representadas pelos nodos do grafo e as relações entre as classes - relações cliente-servidor e de herança - são representadas pelas arestas. Os grafos disjuntos obtidos representam agrupamentos de classes (*clustering*). Um agrupamento de classes é, então, um conjunto de classes independente dos demais agrupamentos de classes do software. Esse grau de independência do agrupamento de classes propicia seu reúso. A Figura 3.3 mostra um exemplo simples de um software no qual se observam agrupamentos de classes.

  A métrica CLF representa a proporção entre a quantidade de agrupamentos de classes e o total de classes do software. No software da Figura 3.3, o cálculo para essa métrica é CLF = 3/13 = 0,23.

  O fator CLF é, então, um indicador de reusabilidade de agrupamentos de classes de um software. Um CLF baixo indica que há baixo potencial de reúso de agrupamentos de classes do software. O pior caso para essa métrica é quando não se observam agrupamentos de classes no software, o que resulta em valor 0 para CLF. O outro extremo se dá quando o número de agrupamentos é igual ao número de classes, o que corresponde a um software constituído por classes totalmente independentes entre si, resultando em valor 1 para CLF.

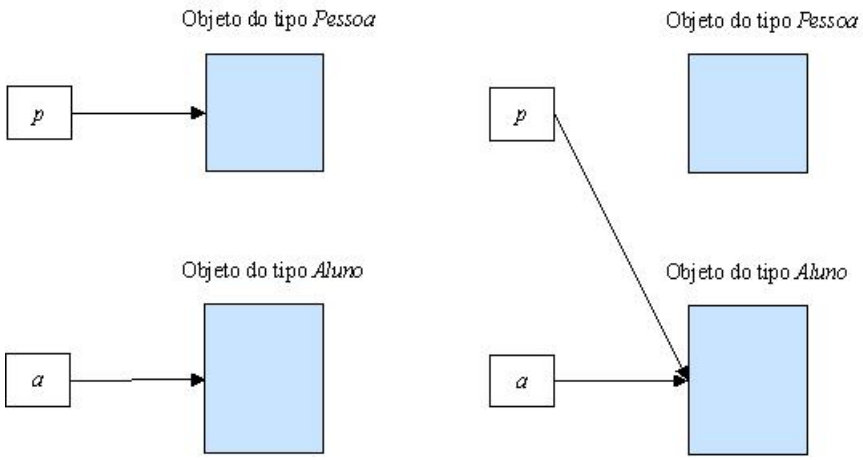


Figura 3.4: Exemplo de situação polimórfica

4. **Métricas para Polimorfismo**

   Na orientação por objetos, a herança é um dos recursos centrais de reusabilidade, pois possibilita a criação de novas classes a partir de classes já existentes. Em conjunto com a herança, a orientação por objetos conta com dois recursos principais para a construção de softwares extensíveis: a redefinição de características (métodos e atributos) e o polimorfismo.

   Uma classe herdeira pode redefinir características herdadas de suas ascendentes. Por exemplo, no diagrama de classes da Figura 3.1, a classe *Funcionário* herda da classe *Pessoa* o método *Desvincular* e dá a esse método uma implementação diferente daquela definida na classe *Funcionário*. Assim, outras classes podem ser adicionadas à hierarquia de classes e, caso seja necessário, as características herdadas por elas podem ser redefinidas.

   A redefinição de características aumenta a extensibilidade de um software. Mas, o grande ganho no uso de herança e de redefinição é conseqüência do polimorfismo. Como exemplo da ocorrência de uma situação polimórfica, consideremos a estrutura de classes da Figura 3.4. Seja p uma referência para o tipo *Pessoa* e a uma referência para o tipo *Aluno*. A atribuição $p = a$ é válida, pois a é um objeto da classe *Aluno*, que é herdeira da classe *Pessoa*. A Figura 3.4 esquematiza o que ocorre antes e depois desta atribuição polimórfica. Primeiro, p é criado e referencia um objeto do tipo *Pessoa* e a é criado e referencia um objeto do tipo *Aluno*. Após a execução de $p = a$, p passa a referenciar um objeto do tipo *Aluno*. *Pessoa* é o tipo estático de p e *Aluno* é o seu tipo dinâmico.

Uma mensagem *Desvincular* enviada ao objeto p antes da atribuição polimórfica, resulta na execução conforme definição do método na classe *Pessoa*. A mesma mensagem enviada a p após a atribuição polimórfica, resulta em uma execução conforme definição do método na classe *Aluno*. Assim, uma mesma mensagem pode assumir diferentes formas em tempo de execução, pois o método apropriado para processar a mensagem é definido de acordo com o objeto receptor da mensagem.

O polimorfismo, na orientação por objetos, confere grande flexibilidade e extensibilidade ao software. A possibilidade de se agregar novas funcionalidades a um software tem impacto no seu custo total. Assim, métricas de avaliação sobre polimorfismo em um software são importantes na avaliação de seu custo.

MOOD conta com a métrica para polimorfismo em software orientado por objetos, referenciada aqui como PF (Fator Polimorfismo), descrita a seguir.

- **PF (Fator Polimorfismo):** o cálculo de PF é baseado na redefinição de métodos, porque se não há redefinição de um método para processar determinada mensagem, então não há possibilidade de ocorrência de poliformismo no comportamento desta mensagem. PF é dada pela razão entre a quantidade de métodos redefinidos no sistema e o total de possibilidades de redefinição de métodos. O cálculo desses dois itens é realizado das seguintes formas:

  (a) *Total de possibilidades de redefinição de métodos* : é dado pelo somatório dos produtos entre *quantidade de métodos definidos na classe* e *total de descendentes da classe* de cada classe do sistema. Esse valor corresponde ao que os autores de MOOD chamam de *número máximo de situações polimórficas diferentes possíveis.*

  (b) *Quantidade de métodos redefinidos no sistema*: para cada classe, computa-se o somatório da quantidade de métodos redefinidos em suas descendentes. Somam-se, então, os valores obtidos para cada classe. Esse valor corresponde ao que os autores de MOOD chamam de *número de situações polimórficas diferentes possíveis.*

  A Tabela 3.5 detalha o cálculo de PF para o sistema representado no diagrama de classes da Figura 3.1. A métrica PF é um indicador do potencial de emprego de polimorfismo no software. O valor 0 para esta métrica indica que nenhum método foi redefinido e, conseqüentemente, não há possibilidade de comportamento polimórfico no envio de mensagens no software. O valor 1 para esta métrica indica que uma mensagem enviada a um objeto de uma classe c que possua n descendentes pode se comportar

de n formas distintas daquela definida na classe c, isto é, a possibilidade de ocorrência de comportamento polimórfico no processamento de uma mensagem é máxima.

Classe	*Métodos definidos*	*Número de descentes*	*Redefinições de métodos possíveis*
Pessoa	3	3	9
Aluno	4	0	0
Funcionario	4	1	4
Professor	2	0	0
Matricula	4	0	0
Turma	1	0	0
Disciplina	1	0	0
Total de possibilidades de redefinição de métodos			13
Classe	*Descendente*	*Qtde. de métodos redefinidos no descendente*	*Total métodos redefinidos nos descendentes*
Pessoa	Aluno Funcionário Professor	2 2 1	5
Aluno	-	-	0
Funcionário	Professor	1	1
Professor	-	-	0
Matricula	-	-	0
Turma	-	-	0
Disciplina	-	-	0
Quantidade de métodos redefinidos no sistema			6
PF = Quantidade de métodos redefinidos no sistema / Total de possibilidades de redefinição de métodos = 6 / 13 = 0.46			

Tabela 3.5: Exemplo de cálculo da métrica PF

5. **Métricas para Reusabilidade**

- **RF (Fator Reúso):** o projeto de software direcionado a maximizar reusabilidade é essencial na redução de custo de produção de softwares. MOOD contém uma métrica para avaliar o reúso em software orientado por objetos. No cálculo desta métrica, duas formas de reúso são consideradas: o reúso de classes já existentes em uma biblioteca de classes e o reúso por

meio de herança de métodos. O fator herança de atributos não entra no cálculo desta métrica porque os seus autores consideram que a herança de métodos impacta em maior custo de construção e manutenção do que a herança de atributos.

O cálculo de RF para um software é dado pela soma de dois fatores:

(a) *Reúso de classes na biblioteca*: razão entre o número de classes reutilizadas de bibliotecas de classes e o número total de classes do software.

(b) *Reúso por herança de métodos*: calcula-se o produto dado por MIF e a quantidade de classes do software que não estão na biblioteca. O *Reúso por herança de métodos* é dado pela razão entre esse produto e o número total de classes do software.

A Tabela 3.6 detalha o cálculo de RF para o sistema representado no diagrama de classes da Figura 3.1. Como RF é um indicador da reusabilidade aplicada no software em questão e não uma avaliação do potencial de reusabilidade do mesmo, ele é uma variável a ser considerada na avaliação da facilidade de teste e do custo de manutenção do software.

Reúso de classes na biblioteca = Número de classes reutilizadas de bibliotecas / Número total de classes do software = 0 / 7 = 0
Reúso por herança de métodos = (MIF x Quantidade de classes que não estão na biblioteca) / Número total de classes do software = (0.24 x 7) / 7 = 0.24
RF = Reúso de classes na biblioteca + Reúso por herança de métodos = 0 + 0.24 = 0.24

Tabela 3.6: Exemplo de cálculo da métrica RF

3.2 Conclusão

Neste capítulo, buscamos realizar um estudo sobre métricas de software orientado por objetos, com o objetivo de, posteriormente, identificar um conjunto de métricas que possam ser utilizadas na avaliação de conectividade de sistemas construídos neste paradigma.

Os dois conjuntos de métricas estudados, CK e MOOD, são amplamente citados na literatura sobre métricas de software orientado por objetos. De acordo com Pressman

[44], embora essas métricas sejam debatidas na literatura e alguns considerem que elas não possuem um grau de formalismo adequado, elas fornecem informações úteis para a avaliação de software.

O conjunto CK caracteriza-se por possuir métricas para classes. Já o conjunto MOOD fornece métricas para o sistema como um todo ou para um conjunto de classes. CK contém métricas para avaliar os fatores: complexidade de métodos de uma classe, profundidade da classe na sua árvore de herança, número de filhos de uma classe, quantidade de classes a qual determinada classe está acoplada, o conjunto resposta de uma classe e ausência de coesão em métodos de uma classe. MOOD conta com métricas para avaliar os seguintes fatores de um sistema: herança, acoplamento, agrupamento, polimorfismo, ocultação de informação e reúso. Desta forma, CK e MOOD mostram-se complementares, pois um fornece uma avaliação em um grão mais fino e o outro, métricas para avaliação geral do sistema. Além disso, um aborda aspectos que não são abordados no outro.

No Modelo de Avaliação de Conectividade em Sistemas Orientados por Objetos (MACSOO), proposto nesta dissertação, utilizamos algumas das métricas dos conjuntos CK e MOOD. A seleção dessas métricas justifica-se por suas utilidades na avaliação dos fatores que impactam na conectividade de um software orientado por objetos. A identificação desses fatores é detalhada na Seção 6.1, e as métricas utilizadas em MACSOO são apresentadas na Seção 6.2.1.

Capítulo 4

Trabalhos Relacionados

Considerando a conectividade como fator preponderante na avaliação da estabilidade de um software, sendo o indicador principal da facilidade de sua manutenção, propomos o Modelo de Avaliação de Conectividade em Sistemas Orientados por Objetos(MACSOO), cujo ponto central é a redução da conectividade, bem como uma ferramenta que viabilize a implementação de tal modelo para sistemas desenvolvidos em Java.

Neste capítulo, apresentamos os seguintes trabalhos relatados na literatura que se relacionam ao tema desta dissertação:

- Um modelo de estabilidade de programa: este modelo foi proposto por Glenford Myers em [38]. Basicamente, este modelo é a proposta de uma métrica que avalia a quantidade média de módulos que podem sofrer impacto em decorrência da alteração em um módulo qualquer do sistema. Esta informação é de grande interesse para o modelo proposto nesta dissertação. Por esta razão, o trabalho de Myers é descrito em detalhes neste capítulo.
- Ferramentas para coleta de métricas: referenciamos algumas ferramentas desse tipo, a fim de estabelecermos um parâmetro para construção da ferramenta proposta nesta dissertação.

4.1 Um Modelo de Estabilidade de Programas

Myers [38] propôs um modelo de avaliação para a estabilidade de software. Este modelo resulta em uma métrica global para o sistema que fornece a seguinte informação: ao alterar um módulo no sistema, quantos módulos serão alterados no total. O modelo tem como base teórica Probabilidade e Teoria dos Grafos. Nesse modelo, o sistema é representado por um grafo não direcionado. Módulos do sistema são representados pelos vértices do grafo. Uma aresta entre dois vértices representa a existência de comunicação entre os dois módulos. Uma matriz m X m, onde m é total de módulos do sistema, é utilizada para o cálculo do valor global para a estabilidade do sistema. Para efetuar os cálculos, são utilizadas duas tabelas nas quais são associados valores a cada escala de acoplamento entre módulos e coesão interna de módulos, mostradas na Tabela 4.1. Como Myers destaca, os valores indicados nessas tabelas não têm base empírica; os valores foram sugeridos por ele com base na análise dos problemas ocasionados por cada tipo de coesão e acoplamento. O autor chama a atenção para o valor atribuido à coesão lógica, que foge à linearidade dos demais. Isso é justificado por ele pelo fato de os problemas da coesão lógica ficarem confinados no módulo e não causarem problemas entre módulos.

Coesão	*Valor*
Coincidental	0,95
Lógica	0,4
Clássica	0,6
Procedural	0,4
Comunicacional	0,25
Informacional	0,2
Funcional	0,2

Acoplamento	*Valor*
Conteúdo	0,95
Dado comum	0,7
Externo	0,6
Controle	0,5
Dado local	0,35
Informação	0,2

Tabela 4.1: Pesos de Coesão e Acoplamento no Modelo de Myers

O cálculo é feito em duas fases: a primeira gera uma matriz de dependência de primeira ordem, que considera os fatores coesão e acoplamento para cada par de módulos. A segunda fase deriva uma matriz de dependência completa a partir da matriz de dependência de primeira ordem.

Seja m o número de módulos de um sistema. A matriz de dependência de primeira ordem é obtida da seguinte forma:

1. Constrói-se uma matriz C, com dimensões m X m. Avalia-se o tipo de acopla-

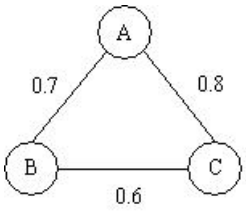


Figura 4.1: Um exemplo simplificado de estrutura de sistema

mento existente entre cada par de módulo e preenche-se a posição C_{ij} da matriz, correspondente ao par analisado, com o valor associado ao tipo de acoplamento.

2. Constrói-se um vetor S de m posições. Avalia-se o tipo de coesão interna de cada módulo e preenche-se a posição S_i correspondente do vetor como o valor associado ao tipo de coesão.

3. Constrói-se a matriz D de dependência de primeira ordem, a partir da seguinte fórmula:
   $D_{ij} = 0.15(S_i + S_j) + 0.7C_{ij}$, se $C_{ij} \neq 0$
   $D_{ij} = 0$, se $C_{ij} = 0$
   $D_{ii} = 1$ para todo i

De acordo com Myers, esta fórmula não foi derivada empiricamente; foi definida como uma equação linear baseada em acoplamento e coesão devido ao fato de que estes dois aspectos têm impacto na estabilidade de programas. Os dois fatores utilizados na fórmula, 0.15 associado à coesão e 0.7 ao acoplamento, representam a idéia de que o acoplamento é mais importante do que a coesão para a estabilidade do software.

A matriz D resultante é baseada em acoplamento e coesão considerando apenas os pares de módulos. Ela fornece a probabilidade de se alterar um módulo j em decorrência de uma alteração ocorrida em um módulo i. Porém, a partir da matriz D, é possível responder esta questão considerando-se apenas a relação de dependência direta existente entre i e j, isto é, ela considera que uma alteração em um módulo pode afetar apenas os módulos que estão conectados diretamente a ele, e despreza os impactos da alteração nos demais módulos do sistema.

Seja o exemplo da Figura 4.1 e a Tabela 4.2. Uma alteração no módulo A pode acarretar uma alteração no módulo B por dois caminhos: A-B e A-C-B. A probabilidade

	A	**B**	**C**
A	1.0	0.7	0.8
B	0.7	1.0	0.6
C	0.8	0.6	1.0

Tabela 4.2: Tabela do exemplo simplificado de estrutura de sistema

de cada caminho é dada pelo produto das probabilidades associadas a cada aresta que compõe o caminho. Seja x a probabilidade de ocorrência do primeiro evento, e y, a probabilidade do segundo. Tais eventos não são mutuamente exclusivos. Assim, a fórmula para se calcular a probabilidade de B ser alterado devido a uma alteração ocorrida em A deve ser:

$$\text{P (alterar B)} = \text{P(x)} + \text{P (y)} - \text{P(x) P(y)}$$

Desta forma, deriva-se uma nova matriz E de dependência completa, considerando os efeitos indiretos de alteração. Em sistemas reais, a estrutura é geralmente complexa e pode existir inúmeros caminhos de se chegar de um módulo i a um módulo j. Esse modelo de avaliação propõe utilizar os três caminhos com maior peso, isto é, os três caminhos com maior probabilidade de impacto de uma alteração em j devido a uma alteração ocorrida em i. A probabilidade de um caminho é dada pelo produto das probabilidades de cada aresta do caminho.

A matriz E é obtida da seguinte forma: para cada par de módulos i e j:

1. Encontram-se todos os caminhos de i para j.
2. Se há apenas um caminho de i para j, $E_{ij} = E_{ji} = \text{P(x)}$, onde P(x) é a probabilidade do caminho.
3. Se há dois caminhos de i para j, $E_{ij} = E_{ji} = \text{P(x)} + \text{P(y)} - \text{P(x)P(y)}$, onde P(x) e P(y) são as probabilidades dos dois caminhos.
4. Se há três caminhos de i para j, $E_{ij} = E_{ji} = \text{P(x)} + \text{P(y)} + \text{P(z)} - \text{P(x)P(y)} - \text{P(x)P(z)} - \text{P(y)P(z)} + \text{P(x)P(y)P(z)}$, onde P(x), P(y) e P(z) são as probabilidades dos três caminhos.

5. Se há mais de três caminhos de i a j, encontram-se os caminhos de maiores probabilidades. Aplica-se a regra do item anterior para esses três caminhos.

A matriz E de dependência completa obtida fornece as seguintes informações:

- Uma nota global para o sistema. Essa nota é obtida somando-se os valores de todas as posições da matriz e dividindo-se o resultado pelo total de módulos do sistema. A interpretação da nota obtida se resume em determinar quantos módulos serão alterados no total em decorrência de uma alteração em um determinado módulo. Quanto menor essa nota, melhor o projeto do sistema.
- A soma dos elementos de cada linha da matriz E fornece o total de módulos que são alterados em conseqüência da alteração do módulo representado pela linha.
- A ordenação dos elementos de cada linha i em ordem decrescente de valores fornece uma lista ordenada dos módulos que são alterados, caso o módulo representado por i seja alterado.

4.1.1 Análise Crítica

Esta proposta de Glenford Myers fornece uma avaliação de sistema baseado em conectividade, considerando não apenas a existência de conexão entre os módulos, mas a qualidade das conexões, visto que considera os graus de acoplamento e coesão para realizar os cálculos. Porém, observam-se alguns problemas neste modelo de avaliação:

- O grafo utilizado no modelo para representar o sistema não é direcionado. Isso é um problema, pois simplifica a análise de conexões entre dois módulos. Sejam dois módulos A e B. O fato de que uma alteração em A impacta em uma alteração em B nem sempre implica que uma alteração em B impacta também em uma alteração em A. Isso depende do tipo de acoplamento existente entre A e B. Assim, o modelo que melhor representaria as conexões existentes entre módulos de um sistema é um grafo direcionado.
- Os parâmetros básicos para efetuar os cálculos neste modelo foram determinados de maneira intuitiva: os valores das Tabelas 3.1 e 3.2 e os fatores 0.15 e 0.7 da fórmula para obter a matriz D.

- Não define como proceder no caso de um módulo estar acoplado a outro por mais de uma forma.

4.2 Ferramentas para Coleta de Métricas de Software

A coleta de métricas em softwares é uma tarefa que demanda ferramentas. Iniciativas neste segmento tiveram início marcante no ínício dos anos 90 [1] [22]. Existem hoje no mercado uma série de ferramentas para coleta automática de métricas [33]. Buscamos identificar aquelas que destinam-se a coletar métricas de software orientados por objetos.

Understand [27] é um conjunto de ferramentas comerciais que coletam métricas em softwares escritos em Ada, Delphi, Fortran, Java e C++. Entre as métricas coletadas pela ferramenta estão: número de linhas, número de linhas de código, número de linhas de comentários; para um sistema, coleta métricas como número de classes, número de linhas de código, número de linhas de comando e número de linhas de declarações. Embora esta ferramenta analise códigos escritos em linguagens orientadas por objetos como C++ e Java, ela não coleta métricas relevantes para a avaliação de software neste paradigma.

Krakatau Essencial Metrics [30] é uma ferramenta que coleta métricas em programas escritos em Java e C/C++. A sua principal funcionalidade é prover meios de comparações de versões de software. Baseia-se na coleta das seguintes métricas: linhas de código alteradas, adicionadas e excluídas. Apesar de ser uma ferramenta que vise a análise de programas escritos em linguagens orientadas por objetos, não provê métricas específicas a esse paradigma.

A ferramenta *ObjectDetail* [40] coleta métricas específicas do paradigma orientado por objetos em softwares desenvolvidos em C++. Dentre as métricas coletadas por esta ferramenta, destacam-se: percentual de atributos públicos, percentual de atributos privados, percentual de métodos públicos, percentual de métodos privados, profundidade da árvore de herança e acoplamento de classe. Esta métrica é dada pelo número de classes que uma classe particular usa. De acordo com [40], um valor alto para essa métrica indica dificuldade de manutenção da classe em questão.

MOODKIT [3] é uma ferramenta desenvolvida pelo grupo de estudos relacionado à

proposta do conjunto de métricas MOOD. Esta ferramenta coleta as métricas MOOD em softwares escritos em linguagens como C++, Eiffel e Java. A principal característica desta ferramenta é o uso de uma espécie de linguagem intermediária denominada GOODLY, proposta pelo mesmo grupo. A idéia basica é converter o código fonte a ser analisado em um código equivalente GOODLY, sobre o qual ocorre a coleta das métricas.

Together [56] é uma ferramenta que possui, dentre outros recursos, a coleta de métricas em códigos escritos em Java, de forma integrada à ferramenta de desenvolvimento.

Q.Metrics [46] extrai métricas de códigos escritos em C# e ASP.NET. A ferramenta é integrada ao ambiente de desenvolvimento e permite que o usuário selecione as métricas que deseja coletar. Os resultados podem ser visualizados em HTML ou PDF. A versão desta ferramenta para análise de código em C# coleta cerca de 20 métricas, dentre elas, métricas de orientação por objetos como *acoplamento aferente* e *acoplamento eferente*. Estas duas métricas são propostas por Robert Martin [34], que as considera como indicadores da independência entre agrupamentos de classes. De acordo com Martin, *acoplamento aferente* é dado pelo número de classes externas ao agrupamento em questão que dependem das classes constituintes do agrupamento; *acoplamento eferente* é dado pelo número de classes constituintes do agrupamento que dependem de classes externas a ele.

Essas ferramentas possuem em comum o fato de que realizam análise em código de linguagens orientadas por objetos. Algumas são simples relatórios de métricas coletadas e outras possuem recursos gráficos de visualização. A análise de ferramentas de coletas de métricas em códigos escritos em linguagens orientadas por objetos é importante para que se possa identificar se há alguma delas que possa ser utilizada no contexto de MACSOO. Concluimos que as ferramentas disponíveis não são adequadas para a implementação de MACSOO, visto que não têm o propósito específico de avaliar conectividade e seus aspectos correlatos, fazendo-se necessária a implementação de uma ferramenta com este propósito. Entretanto, a observação das características das ferramentas avaliadas fornece uma base para se estabelecer os requisitos da ferramenta a ser construída neste trabalho, como interface gráfica de usuário e armazenamento dos resultados para futuras consultas.

4.3 Conclusão

Buscamos, neste trabalho, uma forma de avaliar software orientado por objetos de maneira a fornecer um indicador do grau de dificuldade de sua manutenção. O Modelo de Myers tem como resultado uma informação de grande importância para o objetivo desta dissertação; ele indica quantos módulos de um sistema poderão sofrer impacto em decorrência de uma alteração, tendo como base principal para este cálculo, as conexões entre os módulos do sistema. Ele resulta, portanto, no grau de *estabilidade* de um software. Entretanto, para que possa ser utilizado na orientado por objetos, necessita ser adaptado a este paradigma. Realizamos tal adaptação e empregamos o resultado em MACSOO.

Para que a implementação de MACSOO seja viável, é necessária a utilização de ferramentas de coletas de métricas de software orientado por objetos. Existem ferramentas disponíveis no mercado que coletam métricas em códigos escritos em linguagens orientadas por objetos. Realizamos um levantamento de algumas delas, relatadas neste capítulo, visando identificar suas características e, se possível, alguma que pudesse ser utilizada para a implementação de MACSOO. Algumas destas ferramentas não trabalham com as métricas específicas para o paradigma OO e aquelas que o fazem, não coletam todas métricas necessárias a MACSOO. Como solução para este problema, construimos uma ferramenta com este objetivo.

Capítulo 5

Coesão e Acoplamento na Orientação por Objetos

Os conceitos de coesão interna de módulos e acoplamento entre módulos são instrumentos úteis na avaliação do grau de modularidade de um sistema. As definições realizadas para as escalas de coesão e acoplamento definidas por Myers [38], apresentadas nas Seções 2.2.1 e 2.2.2 , respectivamente, desta dissertação, foram realizadas em uma época na qual a orientação por objetos ainda não era um paradigma consolidado. Como aquelas definições não abordam as características específicas da OO, elas necessitam ser revisitadas para que possam se adequar a esse paradigma. É importante conhecer como os diversos graus de coesão e acoplamento apresentam-se na construções da OO, pois isso é um conhecimento útil na avaliação do grau de modularidade em sistemas neste paradigma. Em particular, a avaliação desses dois fatores auxilia na obtenção de sistemas com menor nível de conectividade, pois atuar na estrutura dos módulos de um sistema e na forma como eles interagem entre si é o ponto essencial para aumentar a independência entre eles.

A avaliação do grau de coesão interna de módulos e do grau de acoplamento entre módulos tem como pré-requisito a definição do que será considerado *módulo*. Conforme descrito na Seção 2.2, as seguintes definições genéricas para módulos são encontradas na literatura, dentre outras:

- conjunto de um ou mais arquivos que possa ser compilado separadamente [54];
- menor unidade na qual o sistema pode ser decomposto [36].

Booch [11] aponta que a *classe* é a unidade primária de decomposição em sistemas orientados por objetos, e não o algoritmo, como ocorre em sistema desenvolvidos no paradigma estruturado. Para Bertrand Meyer [36], a classe desempenha o papel central na OO, representando além de um tipo de objetos, uma unidade modular. Ele enfatiza esta idéia apontando que em ambientes orientados por objetos as classes devem ser os únicos módulos.

Neste capítulo propomos uma adaptação dos critérios de coesão interna de módulos e acoplamento entre módulos em sistemas orientados para objetos, que é uma releitura das escalas propostas por Myers [38] à luz da OO. Para isso, consideraremos *classe* como módulos em sistemas orientados por objetos, em concordância como Booch[11] e Meyer [36].

5.1 Coesão Interna de Classes

A coesão interna de um módulo é definida pelo grau de relacionamento existente entre seus elementos. Desta forma, a coesão interna de uma classe é determinada pelo grau de relacionamento existente entre seus constituintes: seus membros de dados e seus métodos. A avaliação da coesão de um módulo não é tarefa trivial, pois, como ressaltado em [51], essa tarefa requer conhecimento técnico do domínio da aplicação do sistema em questão, dentre outros requisitos.

Em [51], destacam-se os seguintes aspectos como indicadores do bom nível de coesão de uma classe:

- a classe deve representar um conceito completo e coerente, e não uma coleção aleatória de informações;
- cada método de sua interface pública deve realizar uma única função bem definida.

Este último ponto ressalta que, além da coesão entre os elementos da classe, é necessário avaliar a coesão interna de seus métodos, sobretudo daqueles que constituem a sua interface.

Em sistemas orientados por objetos, a principal via de comunicação entre classes se dá por meio de suas interfaces. Conforme Meyer [36] define, a interface de uma

classe corresponde ao seu contrato, que inclui os contratos individuais dos métodos exportados pela classe bem como os seus membros de dados públicos. Staa [54] define o conceito de *conector* como a via de comunicação entre dois módulos e define a *coesão de um conector* como o grau de interdependência semântica entre os seus elementos. Desta forma, para sistemas orientados por objetos, definiremos, aqui, *coesão da interface da classe* como o grau de interdependência semântica entre os elementos que compõem a interface de uma classe.

Assim, identificamos três níveis de avaliação de coesão em sistemas orientados por objetos:

- coesão interna da classe: grau de relacionamento entre os elementos internos da classe;
- coesão interna de método: grau de relacionamento entre os elementos internos do método: seus dados locais e suas instruções;
- coesão da interface da classe: refere-se ao grau de relacionamento entre os elementos da interface da classe.

A seguir, são apresentadas uma classificação de grau de coesão para esses três níveis.

5.1.1 Coesão Interna de Métodos

Métodos são estruturalmente próximos a procedimentos e funções do paradigma estruturado. Assim, a maior parte da classificação de coesão interna de módulos proposta por Glenford Myers [39] pode ser utilizada ou adaptada à avaliação de coesão no caso de métodos, resultando na seguinte escala, do pior para o melhor caso:

- **Coesão Coincidental**

  É o pior tipo de coesão, pois não há relacionamento com significado relevante entre os elementos do método.

- **Coesão Lógica**

  O método realiza mais de uma função logicamente relacionadas, com uma única interface com o usuário do método. A função a ser executada é determinada por meio de parâmetros.

- **Coesão Temporal**

  Corresponde à coesão clássica defina por Myers. As funções realizadas pelo método, além de serem logicamente relacionadas, são temporalmente dependentes, pois precisam ser executadas em conjunto e em determinada seqüência.

- **Coesão Procedimental**

  É similar à coesão temporal, somado-se o fato de que as funções desempenhadas pelo método são relacionadas ao domínio do problema a ser solucionado.

- **Coesão Comunicacional**

  Um método com esse tipo de coesão possui coesão procedimental além de suas funções se comunicarem por meio de dados comuns.

- **Coesão Funcional**

  Um método com este tipo de coesão desempenha uma única função bem definida.

Como destaca Pressman [44], um módulo coeso realiza uma única função, ocasionando pouca interação com outras partes do software. Trazendo esta leitura para o universo de *métodos* na orientação por objetos, podemos dizer que métodos coesos, por realizarem uma função bem definida, ocasionam poucas conexões, seja com outros métodos de sua própria classe, seja com outras classes.

Cabe, aqui, uma análise particular dos menores níveis de coesão: coincidental e lógica. Um método com coesão coincidental pode realizar várias tarefas sem relacionamento entre si. Desta forma, tende a ser mais usado por outras partes do software, ou ser cliente de várias outras partes do software. Um método com esta característica é fonte para o estabelecimento de alta conectividade da classe. Da mesma forma, no caso da coesão lógica, como o método realiza mais de uma função, ele é determinante para a conectividade.

Além disso, métodos com coesão baixa podem denotar um problema estrutural da classe, como o baixo grau de coesão interna da classe à qual pertencem. Se um método realiza mais de uma função, é preciso investigar se isso decorre do fato de a classe implementar mais de um contrato, por exemplo. Como a coesão interna de uma classe impacta na sua conectividade, e é influenciada pela coesão interna de seus métodos, em última análise, concluimos que a coesão interna dos métodos de uma classe tem impacto na conectividade dessa classe.

Desta forma, a criação de métodos altamente coesos é indispensável para a obtenção de software orientado por objetos de baixa conectividade.

5.1.2 Coesão Interna de Classes

O grau de coesão interna de uma classe é definido a partir da observação do relacionamento existente entre seus elementos - seus membros de dados e seus métodos. Um conceito importante na avaliação de coesão interna de classes é o *contrato* [36]. Em um sistema orientado por objetos, classes podem ser vistas como *fornecedoras* ou *clientes* de serviços. Esta relação de cliente-fornecedor é regida por um *contrato*, que define regras a serem seguidas tanto por quem usa quanto por quem fornece o serviço. A classe fornecedora compromete-se a fornecer um determinado serviço a partir de uma especificação do problema e publica a interface a ser utilizada por aquelas que precisarem utilizar o serviço. Para benefício da modularidade, é importando a criação de classes que implementam um único contrato.

Os critérios de organização dos elementos internos de uma classe são apresentados a seguir, do pior para o melhor caso:

- **Coesão Coincidental**

  Uma classe tem coesão coincidental se não se observa relacionamento relevante entre os seus elementos. A ausência de relacionamento relevante entre os elementos de uma classe ocorre nos seguinte casos:

  - A classe implementa mais de um contrato. Um exemplo é uma classe na qual são agrupadas as implementações dos tipos abstratos de dados pilha e lista.
  - Não há relacionamento entre os métodos da classe ou não há relacionamento entre os dados da classe. Um exemplo deste caso ocorre em classes que contêm definições de tipos, dados e métodos sem qualquer relacionamento entre si. Em [57], este caso é referenciado como *utility cohesion*, pois agrupam-se utilitários que não pertencem logicamente a nenhuma outra parte do software. O exemplo em Java a seguir ilustra este caso:

```
public class Geral {
  int ponto;
```

```
    int usuário;
    int codigoLivro;

    public void LimpaTela() { ... }

    public float CalculaImpostoRenda() { ... }

    public int CalculaNotaGlobal() { ... }

  }
```

- **Coesão Lógica**

  A coesão lógica, tal como é definida por Myers [38], ocorre quando um módulo desempenha um conjunto de funções relacionadas logicamente e uma delas é selecionada por meio de um indicador passado como parâmetro para o módulo. Essa situação não ocorre no caso de classes, pois o serviço a ser realizado é chamado diretamente pelo cliente da classe. Uma classe tem coesão lógica quando implementa mais de um contrato relacionados logicamente. A classe *Modem*, cuja estrutura é mostrada a seguir, exemplifica este caso. No exemplo, a classe implementa dois contratos distintos: a gerência da conexão, constituída pelos métodos *discar()* e *desligar()*, e a comunicação de dados, constituída pelos métodos *enviar(char c)* e *receber()*. Há uma relação lógica entre os contratos, pois ambos referem-se à interface de um modem.

```
public class Modem {

  public void Discar() { ... }

  public void Desligar() { ... }

  public void enviar(char c) { ... }

  public void receber() { ... }

}
```

  Um melhoria em coesão interna de classes seria obtida com a construção de duas classes, em substituição a *Modem*, cada uma delas realizando um contrato. As classes *ModemConexao* e *ModemComunicacao*, a seguir, mostram o resultado dessa separação de responsabilidades.

```
public class ModemConexao {

  public void Discar() { ... }

  public void Desligar() { ... }

}

public class ModemComunicacao {

  public void enviar(char c) { ... }

  public void receber() { ... }

}
```

- **Coesão Temporal**

  A coesão temporal corresponde à coesão clássica, definida por Myers [38], caracterizada por um conjunto de funções relacionadas logicamente e que necessitam ser executadas em conjunto e em seqüência determinada. Um exemplo desta situação é uma classe cujos métodos destinam-se a iniciar o ambiente de execução da aplicação. A classe *AmbienteInicial*, a seguir, exemplifica este caso. No exemplo, o método *AbreArquivoParametros(Strin nomeArq)* abre um arquivo que contém parâmetros a serem utilizados na aplicação, entre eles, o nome e a localização de um arquivo de *log* da aplicação; o método *AbreArquivoLog(String nomeArq)* abre o arquivo de *log*. Há um relacionamento lógico entre os métodos, pois ambos destinam-se à inicialização do ambiente da aplicação. Além disso, *AbreArquivoLog* precisa ser executado após *AbreArquivoParametros*.

```
public class Ambiente{
  ...

  public void AbreArquivoParametros(String nomeArq) { ... }

  public void AbreArquivoLog() { ... }

}
```

- **Coesão Procedimental**

  Uma classe tem coesão procedimental se entre os seus elementos internos observa-se relacionamento procedimental, isto é, os métodos da classe precisam ser executados em uma seqüência determinada pelo domínio do problema.
  Um exemplo desta situação ocorre quando utiliza-se métodos de inicialização para determinar o estado inicial do objeto em vez de utilizar um método construtor. Nesta situação, os métodos de inicialização devem ser executados antes dos demais.

- **Coesão Comunicacional**

  A coesão comunicacional ocorre quando os elementos do módulo são dependentes de membros de dados comuns da classe. Em uma classe, essa é uma característica essencial, pois é importante que os métodos operem de maneira relevante sobre os dados da classe.

- **Coesão Contratual**

  É o melhor nível de coesão interna de uma classe. Uma classe tem coesão contratual se implementa um único contrato. Um exemplo é uma classe que implementa o tipo abstrato de dados Pilha.

5.1.3 Coesão de Interface de Classe

A avaliação do grau coesão da interface de uma classe tem os mesmos preceitos daquela realizada para a classe como um todo. A diferença é que no caso da coesão de interface os elementos avaliados são exclusivamente aqueles exportados pela classe. Então, a coesão de interface de classe pode ser classificada de acordo como os mesmo critérios da coesão interna de classes, descrita no item anterior.

5.2 Acoplamento

Como, na orientação por objetos, uma classe corresponde a um módulo, o acoplamento entre módulos neste paradigma é definido pelo relacionamento entre as classes de um sistema. A forma como duas classes se relacionam é definida pela forma como seus elementos se relacionam. O grau de acoplamento entre classes pode ser definido de

acordo com os seguintes critérios, do nível mais forte de acoplamento para o mais fraco:

- **Acoplamento por Conteúdo**

  O acoplamento por conteúdo ocorre quando uma classe modifica sub-repticiamente dados de objetos de outra classe. Isso ocorre principalmente em decorrência do uso de atributos públicos. Identificar este tipo de acoplamento em sistemas orientados por objetos pode ser difícil, pois envolve saber em que situação o acesso ao dado se dá de forma não autorizada, isto é, se o dado acessado é público em decorrência da aplicação ou por um eventual descuido do projetista.
  Como exemplo, seja uma classe A que define o tipo abstrato de dados pilha via suas operações. Entretanto, o projetista, por descuido, deixa um de seus atributos públicos. Se uma classe B modificar o valor deste atributo, o fará de forma não prevista pelo projetista e as classes A e B estarão acopladas por conteúdo. A ocorrência deste tipo de acoplamento pode ser evitada por meio do uso exclusivo de atributos privados.

- **Acoplamento por Dado Comum**

  Este tipo de acoplamento ocorre entre duas ou mais classes que têm acesso à estrutura de um objeto comum declarado globalmente. Neste caso, o acesso é previsto pelo projetista.

  Alterações na estrutura do objeto comum impacta em todos os módulos que a utilizam, assim como uma alteração em algum desses módulos pode ter impacto em todos os demais. Como exemplo deste tipo de acoplamento, sejam as classes *Protocolo*, *Remetente* e *Receptor* a seguir, definidas em Java:

```
public class Protocolo {
  public static int[]inf = new int[3];
}

public class Remetente{
...
  public void GravaInformacao(){
    Protocolo p;
    ...
    p.inf[1] = x;
    ...
```

```
  }
...
}

public class Receptor{
...
  public void LeInformacao(){
    Protocolo p;
    ...
    y = p.inf[2];
    ...
  }
...
}
```

Neste exemplo, a classe *Protocolo* define um vetor que representa um protocolo de comunicação entre as classes *Remetente* e *Receptor*. Se a estrutura do objeto comum for modificada na classe *Protocolo*, as demais classe sofrerão impacto desta alteração. Da mesma forma, se houver uma alteração na estrutura da classe *Remetente* ou *Receptor* que esteja relacionada ao objeto compartilhado, as demais classes também sofrerão impacto.

Como no caso do acoplamento por conteúdo, esse tipo de acoplamento pode ser contornado evitando-se o uso de dados públicos, utilizando-se métodos públicos para acesso a tais estruturas.

- **Acoplamento por Inclusão**

  Em [57] e [51] é descrito um tipo de acoplamento distinto daqueles inicialmente identificados por Myers [38]: o acoplamento por inclusão[57] ou por conteúdo léxico[51]. Este tipo de acoplamento ocorre quando um módulo inclui o código de outro módulo. Todos os módulos que incluiram um determinado módulo estão acoplados por inclusão entre si e ao módulo incluído. Na orientação por objetos, é representado pelo uso da diretiva *include* do C/C++. Este tipo de acoplamento foge do conceito estabelecido aqui para módulo na orientação por objeto - uma classe - e pode ser visto como um tipo particular de acoplamento por dado comum. Porém merece destaque por dois motivos: diferencia-se do acoplamento por dado comum porque o que é compartilhado neste caso é código e não dado; embora não esteja diretamente relacionado ao conceito de classe, é

um tipo de acoplamento que impacta na manutenção do sistema, visto que gera forte dependência entre os módulos envolvidos.

- **Acoplamento por Elemento Externo**

  Este tipo de acoplamento ocorre entre duas ou mais classes que têm acesso a elementos individuais da estrutura de um objeto comum. Neste caso também, o acesso é previsto pelo projetista. Como o acesso é feito a elementos individuais e não à totalidade da estrutura, como ocorre no acoplamento por dado comum, as conseqüências desse tipo de acoplamento são menores do que o por dados comum. Uma alteração na estrutura comum afeta somente aqueles que utilizam os elementos alterados, assim como uma alteração em uma classe que usa parte da estrutura afeta somente as classes que utilizam os mesmos elementos da estrutura. Porém, ainda é um nível inadequado de acoplamento.

  Um exemplo da ocorrência deste tipo de acoplamento, são classes que possuem definições de dados e suas clientes. As classes D, A e B a seguir, definidas em Java, ilustram este exemplo:

```
public class D{
  public int x;
  public float z;
  public int y;
}

public class A{
  ...
  D d;
  ...
  d.x = 10;
  ...
  a = d.y;
  ...
}

public class B{
  ...
  D d;
  ...
  b = d.x;
```

```
  ...
  d.z = 50;
  ...
}
```

No exemplo, a classe D contém a definição do objeto comum e as classe A e B fazem acesso a componentes individuais do objeto comum.

Conforme apontado em [57], o uso do padrão *façade* [23] contribui para a redução deste tipo de acoplamento.

- **Acoplamento por Controle**

  O acoplamento de controle entre classes ocorre quando o método de uma classe invoca o método de outra classe e conhece o funcionamento deste, utilizando parâmetro que controla o seu fluxo de execução. No exemplo a seguir, a classe B invoca o método m da classe A, e controla a execução deste método por meio do parâmetro p.

```
public class A{
  ...
  public void m(int p){
    if (p == 1)
       desenhaCirculo();
    else
       desenhaRetangulo();
  }
  ...
}

public class B{
  ...
    A a;
    ...
    a.m(2);
  ...
}
```

Este tipo de acoplamento é indesejável porque implica que uma classe conheça os detalhes de implementação da outra. Ele pode ser contornado realizando-se

chamadas diretas às funções desejadas. No exemplo anterior, na classe *B*, a chamada *a.m(2)* poderia ser substituída por uma chamada direta ao método *desenhaRetangulo*.

- **Acoplamento por Referência**

  Duas classes estão acopladas por referência se um método de uma delas invoca um método da outra passando-lhe parâmetros por referência.

  Em Java, a passagem de parâmetros de tipos primitivos ocorre por valor. Varejão [58] destaca que a passagem de parâmetros do tipo referência ocorre também por valor; entretanto, neste caso, passa-se uma cópia da referência do parâmetro real. Assim, alterações em componentes do parâmetro formal têm efeito no objeto apontado pelo parâmetro real. Portanto, a passagem de um tipo referência como parâmetro em Java tem efeitos de passagem de parâmetro por referência. As classes *A* e *B*, definidas em Java, a seguir exemplificam este tipo de acoplamento:

```
public class A{
  B b;
  ...
  public void p (){
     ClasseQualquer obj = new ClasseQualquer();
     ...
     b.metodo(obj);
     ...
  }
  ...
}

public class B{
   private int s = 1000;
   ...
   public void metodo(ClasseQualquer x){
      ...
      x.atualizaCampo(s);
      ...
   }
   ...
}
```

No exemplo, a alteração no valor de um campo de x, realizada pela classe B, é percebida na classe A e ambas estão acopladas por referência.

- **Acoplamento por Informação**

  O acoplamento entre duas classes é por informação se não é nenhum dos descritos anteriormente e se o método de uma delas invoca o método da outra passando-lhe parâmetros por valor.

  As classes A e B, definidas em Java, a seguir exemplificam este tipo de acoplamento:

```
public class A{
  private int x;

  public void fazAlgo(int s){
     x = s + 200;
     ...
  }
  ...
}

public class B{
   private A a;
   private int q = 1000;
   ...
   public void m(){
      ...
      a.fazAlgo(q);
      ...
   }
   ...
}
```

  No exemplo, a alteração no valor de s realizada pela classe A não é percebida na classe B. A única dependência entre ambas está no significado do dado passado como parâmetro.

- **Desacoplado**

  Duas classes estão desacopladas se não há tipo algum de comunicação entre elas.

5.3 Conclusão

Sistemas orientados por objetos são construídos basicamente por classes que se conectam entre si, seja por colaboração ou por herança [11]. A forma como essas conexões ocorrem determinam o grau de acoplamento entre as classes. Por exemplo, se uma classe B é herdeira de A, e uma classe C usa um serviço de A, a dependência de B em relação a A é mais forte do que aquela de C em relação a A.

O grau de dependência entre duas classes decorre do critério utilizado para o estabelecimento da conexão entre elas. No exemplo supracitado, a construção da classe B como herdeira de A seguiu o critério de forte acoplamento; e a decisão de que C utilizasse um serviço de A, seguiu o critério de estabelecer baixo grau de acoplamento entre C e A. O ideal é que o estabelecimento de conexões entre classe seja orientado pelo critério de menor grau de acoplamento possível.

Um fator importante na construção de uma classe é a determinação de sua coesão interna, a forma como os seus elementos internos se relacionam. Por exemplo, sejam duas classe X e Y. No caso de X, o projetista decidiu implementá-la como um TAD; já Y, fora implementada com uma classe depositária. Como resultado, o grau de interrelacionamento entre os elementos de X é maior do que aquele entre os elementos de Y. Como a coesão interna de uma classe é determinante para a modularidade do sistema, é de fundamental importância a construção de classes com alta coesão interna.

Como as estruturas responsáveis pela realização das operações em sistemas orientados por objetos são os *métodos*, deve-se observar cuidado ao defini-los. Assim como as classes, os métodos devem ser definidos com o maior grau de coesão possível, para que sua legibilidade e manutenibilidade sejam melhores. Método pouco coeso é fonte para o estabelecimento de alta conectividade da classe à qual pertence com outras classes do sistema, o que impacta na conectividade geral do sistema.

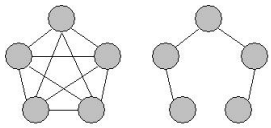
Neste contexto, faz-se necessário conhecer os diversos critérios de acoplamento entre classes, de coesão interna de classes e de coesão interna de métodos. Com este objetivo, este capítulo apresentou uma revisão sobre os acoplamento e coesão no paradigma orientado por objetos que é utilizada em MACSOO.

Capítulo 6

A Conectividade e a Estabilidade de Sistemas

Uma situação ideal na produção de um software seria aquela em que o projetista ou o implementador pudesse realizar uma alteração em determinada parte do software e tal alteração não afetasse as demais partes deste software. Entretanto, ao contrário desta situação ideal, é comum ocorrer o que Martin [35] chama de apodrecimento de projeto (*rotting design*), quando o software se transforma em um conjunto de código de difícil manutenção. Martin identifica a rigidez como um dos sintomas principais deste processo. Em um software de estrutura rígida, uma alteração em um módulo causa uma cascata de outras alterações nos demais. De acordo com Martin[35], a rigidez de um software determina a sua degradação e se estabelece devido à existência de alto grau de interdependência entre os módulos do software.

Gilb [25] denomina a característica de um sistema que se mantém relativamente inalterado diante de uma alteração em seu ambiente como *estabilidade*. Para Gilb [25], a medida da estabilidade de um sistema S diante de uma alteração A é dada pelo percentual de alterações realizadas em S em decorrência de A. A estabilidade é, então, o fator que indica a amplitude do impacto que uma alteração tem em um sistema. Poder conhecer a estabilidade de um sistema é um recurso poderoso no processo de software, pois isso significa, dentre outros benefícios, um forte dado para a predição do esforço real necessário para a realização de alterações em sistemas. O âmbito do problema investigado nesta dissertação diz respeito à identificação de um meio de avaliação da estabilidade de sistemas, em particular aqueles desenvolvidos no paradigma OO.



(a) (b)

Figura 6.1: (a) Alto grau de conectividade (b) Menor grau de conectividade

Conforme Meyer [36] aponta, o caminho para se obter software constituído por módulos o mais independentes entre si possível é que a comunicação entre eles ocorra de forma disciplinada. Para alcançar isso, ele define, dentre outras regras, a regra de *poucas interfaces*, a qual determina que todo módulo deve se comunicar com o menor número possível de outros módulos. Com isso, uma alteração em determinado módulo propaga-se para poucos módulos. Um software cuja estrutura é planejada segundo essa regra tem estrutura mais flexível e possui alta estabilidade. A Figura 6.1*a* representa de forma simplificada um sistema no qual cada módulo do sistema comunica-se com um número alto de outros módulos e a Figura 6.1*b*, um sistema com menor grau de comunicação entre os módulos.

Staa [54] diz que os módulos de um sistema comunicam-se entre si por meio de *conectores*, que são vistos como canais por meio dos quais os módulos trocam dados ou sinais de controle. São exemplos de conectores: variáveis globais, membros de dados e métodos públicos de uma classe.

Definimos, aqui, o conceito de *conectividade* como o grau de intercomunicação entre os módulos de um sistema. Defendemos a tese de que a conectividade é o fator que sobrepuja os demais na estabilidade do sistema e na sua manutenção. A conectividade é o aspecto principal a ser analisado para obter a resposta de questões importantes no processo de software, como: dificuldade de manter o sistema e amplitude do impacto de determinada modificação no sistema.

Myers [38] modela a importância da conectividade para a estabilidade de um sistema por meio de um conjunto de 100 lâmpadas, no qual uma lâmpada representa um módulo, e uma ligação entre duas lâmpadas representa a existência de comunicação entre dois módulos. Uma alteração em um módulo corresponde a ligar a lâmpada que representa o módulo. No esquema de conexões das lâmpadas, a probabilidade de uma lâmpada passar do estado de ligada para desligada no próximo segundo é de 50%; se está desligada, a probabilidade de ser ligada no próximo segundo é de 50%

se pelo menos outra lâmpada que esteja a ela conectada for ligada. Por outro lado, se uma lâmpada estiver desligada, permanecerá assim enquanto todas as lâmpadas conectadas a ela estiverem desligadas. Diz-se que o circuito atingiu o equilíbrio quando todas as lâmpadas estiverem desligadas. Nesse Modelo das Lâmpadas, Myers analisa três situações possíveis de arranjo das conexões:

- Inexistência de conexões entre as lâmpadas: neste caso, o tempo de equilíbrio do circuito é de 7 segundos.
- Circuito constituído por agrupamentos de 10 lâmpadas: os agrupamentos são independentes entre si e cada um deles é totalmente conectado. Neste caso, o tempo de equilíbrio do circuito é de 20 minutos.
- Circuito totalmente conectado: nesta configuração, cada lâmpada possui uma conexão com todas as demais lâmpadas do circuito. Esse é o pior caso, pois tempo para atingir o equilíbrio do circuito é de 10^{22} anos.

As situações mostradas ilustram quão importante a conectividade é para a estabilidade e manutenção de sistemas. O efeito de uma alteração em um módulo de um sistema com alto grau de conectividade é explosivo, o que torna a sua manutenção um problema intratável.

O Modelo das Lâmpadas é a inspiração para a nossa tese de que a conectividade é o indicador primordial para a avaliação da estabilidade de um sistema e, conseqüentemente, é fator determinante para a facilidade de manutenção de sistemas. Outros fatores como complexidade do tipo de aplicação, rotatividade de pessoal nas equipes e qualidade da documentação contribuem para o aumento do custo de manutenção [43], entretanto perdem importância frente à conectividade.

Diante da necessidade de uma modificação qualquer no sistema, por exemplo em decorrência de uma alteração de um requisito ou de uma correção de um erro identificado, em primeiro lugar, deve ser possível a análise completa do impacto desta modificação. Deve ser possível que o projetista ou o implementador possa identificar quais outros módulos do sistema sofrerão impacto em conseqüência da modificação a ser realizada. Deixar de analisar esse impacto favorece o surgimento de problemas em outros pontos do sistema e, na pior das hipóteses, o sistema pode passar a operar de forma incorreta. A conectividade é o aspecto principal nesta análise, pois o fato de um módulo estar conectado a outro implica que uma alteração em um dos módulos pode gerar impacto no outro.

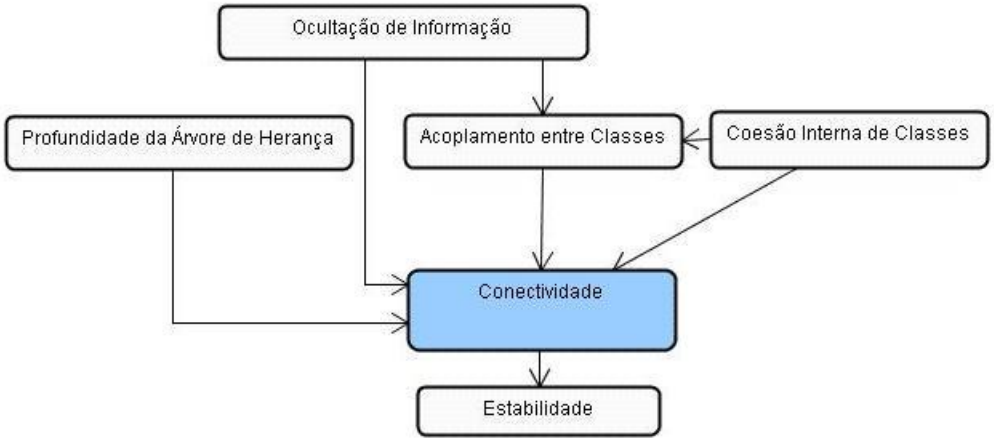


Figura 6.2: Fatores que contribuem para a conectividade em sistemas OO

Diante de um nível de conectividade alto, deve ser possível realizar uma análise sobre a estrutura do sistema, a estrutura de seus módulos e a forma como as conexões se estabelecem nele e atuar nos fatores que elevam a conectividade com o objetivo de reduzi-la.

A seção seguinte identifica os principais fatores que têm impacto na conectividade em sistemas orientados por objetos.

6.1 Fatores de Impacto na Conectividade

Quatro importantes fatores, dentre outros, relacionados à estrutura de software orientado por objetos contribuem para a sua conectividade: profundidade da árvore de herança, ocultação de informação, coesão interna de classes e acoplamento entre classes. O diagrama da Figura 6.2 representa o relacionamento existente entre esses fatores. Os retângulos representam os fatores e as arestas direcionadas entre os retângulos representam uma relação de causa e efeito, onde a causa é o retângulo na origem da aresta e o efeito é o retângulo no final da aresta. As relações entre esses fatores e o aspecto *conectividade* são descritas a seguir.

- **Ocultação de Informação**

  Este é um fator muito importante a ser observado para se manter baixo grau de conectividade em sistemas orientados por objetos. Manter as informações de determinada classe o mais ocultas possível dentro dela contribui para a diminuição da conectividade do sistema, uma vez que minimiza o número de portas abertas para conexão com uma classe. Além disso, a ocultação de informação contribui para a diminuição do grau de acoplamento, pois evita o surgimento de acoplamentos de alto grau decorrentes do acesso direto a dados de um módulo por outros módulo.

- **Acoplamento entre Classes**

  O principal caminho para obter diminuição do grau de conectividade do software é a avaliação das conexões estabelecidas nele. Em ambientes orientados por objetos, duas formas de conexões diretas básicas podem ser estabelecidas entre duas classes: por herança, quando uma classe herda características de outra, ou por uso, quando uma classe usa serviços ou dados de outra, o que caracteriza uma relação do tipo cliente-servidor [54].

  Nas conexões estabelecidas por uso, as classes envolvidas são estruturadas de tal forma que objetos de uma fazem referência a objetos da outra. Gamma et al. [23] denominam este tipo de relacionamento como composição de objetos. Idealmente, neste tipo de conexão, um objeto deve utilizar o outro apenas por meio de suas interfaces, o que garante o respeito à ocultação de informação e ao encapsulamento.

  Para conhecer o impacto real das conexões estabelecidas em um sistema, é preciso conhecer as características de tais conexões. Faz-se necessário conhecer a largura de cada conexão, que corresponde ao grau de acoplamento existente nelas. Uma conexão fina é aquela que resulta em baixo grau de acoplamento, por exemplo, quando um método de uma classe invoca um método da outra com passagem de parâmetro por valor; uma conexão grossa resulta na existência de acoplamento forte entre duas classes, por exemplo, quando um atualiza diretamente um dado da outra. Uma conexão fina não representa para o sistema o mesmo grau de dependência de uma conexão grossa. Os efeitos de alterações em um sistema cujas conexões sejam finas são mais controláveis do que as alterações em um segundo sistema com o mesmo nível de conectividade do primeiro, porém com conexões mais grossas.

  A espessura das conexões – o grau de acoplamento – é um fator de importante impacto na conectividade de sistemas, porque atuando-se sobre o grau de acoplamento das conexões do tipo cliente-servidor do sistema, obtém-se um ganho

quando se consegue diminuir a largura das conexões ou quando, na melhor das hipóteses, consegue-se eliminar conexões.

- **Coesão Interna de Classes**

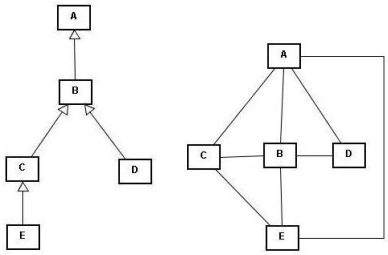
  Há outro ponto a ser trabalhado para a diminuição da conectividade do software: a coesão interna dos módulos. Módulos pouco coesos tendem a realizar tarefas diversas e a quantidade de conexões com os demais módulos tende a ser maior. A reestruturação de uma classe de baixa coesão interna pode resultar na criação de duas ou mais classes de alto grau de coesão interna. A conectividade das classes resultantes tende a ser menor do que a da classe original, o que impacta na diminuição da conectividade do sistema.

- **Profundidade da Árvore de Herança**

  A conexão de herança surge quando uma classe B é herdeira de uma classe A. Neste caso, existe entre A e B uma relação estreita que determina que uma alteração em A impacta em alteração em B. Isso porque a relação de herança é uma relação do tipo "é um" [36]. Se B é herdeira de A, então B é uma classe A, ou seja, B possui as características definidas para A. Se uma característica de A sofre alteração, seja um membro de dado ou um membro função, esta alteração é percebida por seus herdeiros.

  Na hierarquia de classes representada na Figura 6.4a, qualquer alteração em A é percebida imediatamente em B, mesmo que a característica alterada tenha sido redefinida em B, pois, pelo emprego de polimorfismo, B pode se comportar de acordo com a definição estabelecida na superclasse. Por exemplo, se um método for incluído, excluído ou alterado na superclasse, a subclasse é afetada. As classes C e D, sofrem impacto tanto das alterações ocorridas em A quanto daquelas ocorridas em B. A classe E, que está no nível mais profundo da hierarquia, sofre impacto das alterações ocorridas em A, B e C. A Figura 6.3b representa as conexões das classes em decorrência da hierarquia existente entre elas mostrada na Figura 6.3a.

  Embora seja um consenso na literatura que a herança é um recurso poderoso da orientação por objetos para a obtenção de reusabilidade [36] [16] [44], estudos de alguns pesquisadores revelam que seu uso deve ser cauteloso. Gamma et al. [23] denominam a reutilização obtida por herança como reutilização caixa-branca, pois os detalhes de implementação de uma classe ficam expostos às suas descendentes e uma alteração na superclasse impacta em alterações nas suas descendentes. Baseado em uma comparação das vantagens e desvantagens da reutilização obtidas por herança e por composição de objetos, eles definem



(a) (b)

Figura 6.3: (a) Hierarquia de classes (b) Representação das conexões existentes na hierarquia de classes

o princípio de projeto orientado por objetos "Favoreça a composição de objetos em relação à herança". Este princípio não defende que todo uso de herança deve ser abandonado, mas que este é um recurso que deve ser usado com cuidado.

Um estudo de Daly et al. [15] mostra que árvores de herança profundas provocam perda de manutenibilidade de software, pois quanto mais profunda a árvore, maior o tempo e maior a dificuldade para compreendê-la. Sommerville [53] também aponta que a herança introduz dificuldades para análise e entendimento do comportamento dos objetos, o que torna mais difícil solucionar erros nos sistemas.

Tendo em vista os efeitos da herança em sistemas orientados por objetos, Beyer et al. [8] apontam que a herança deve ser considerada na avaliação de métricas referentes a tamanho, acoplamento e coesão em sistemas orientados por objetos. Ao avaliar uma classe, devem ser considerados não só as características definidas nela, mas também aquelas que são herdadas de seus ascendentes.

Conclui-se, então, que a profundidade da árvore de herança impacta na conectividade de um sistema. Se a conectividade de um sistema está alta, as relações de herança devem ser avaliadas. O ponto a ser avaliado neste caso é a profundidade da árvore de herança, pois este é um fator que contribui diretamente para a quantidade de conexões existentes no sistema. Quanto mais profunda a árvore, maior o número de conexões envolvidas na hierarquia.

A seguir, será descrito MACSOO. Este modelo destina-se a medir a conectividade e, diante de um alto grau deste aspecto, avaliar os principais fatores que têm impacto nele. MACSOO é um conjunto de heurísticas que fazem uso de métricas para avaliação

desses fatores e indicam os passos do processo decisório para redução de conectividade de sistemas orientados por objetos.

6.2 Modelo de Avaliação de Conectividade em Sistemas Orientados por Objetos

MACSOO foi concebido da seguinte forma:

- Identificaram-se os principais fatores de impacto na conectividade. Esses fatores são aqueles descritos na Seção 6.1: ocultamento de informação, acoplamento entre classes, coesão interna de classes e profundidade da árvore de herança.
- Levantaram-se as métricas que podem ser utilizadas para medir cada um destes fatores.
- Com foco na redução da conectividade de software, foram propostas heurísticas que têm como elemento principal a avaliação dos fatores impactantes na conectividade por meio de métricas.

A seguir, são identificadas as métricas necessárias no processo de avaliação de conectividade de sistemas orientados por objetos utilizadas em MACSOO. Posteriormente, é apresentada uma adaptação ao paradigma OO do Modelo de Estabilidade de Programas de Myers[38]. Finalmente, é apresentado o Modelo de Avaliação de Conectividade de Sistemas Orientados por Objetos proposto nesta dissertação.

6.2.1 Métricas dos Fatores de Conectividade

Cinco níveis de avaliação mostram-se relevantes na avaliação da estrutura de um sistema orientado por objetos: sistema, agrupamentos de classes, classe, método e atributo. O nível de sistema é uma avaliação geral do sistema. Métricas neste nível indicam uma avaliação para o sistema como um todo. O nível de agrupamento de classes representa uma parte do sistema. De acordo com os critérios definidos por Abreu e Carapuça [2], as métricas para avaliar o sistema devem ser aplicáveis também

a parte dele. No nível de classe, ocorre uma avaliação específica para uma determinada classe. Métricas referentes a uma classe devem ser um indicador de algum aspecto relativo àquela classe. Classes são constituídas por métodos e atributos. Um detalhamento da avaliação de uma classe chega à avaliação de seus atributos e seus métodos.

A idéia central de MACSOO é avaliar o sistema e, diante de um resultado considerado indesejável, detalhar o nível de avaliação a fim de se identificar os pontos carentes de melhoria.

Como o objetivo neste trabalho é a avaliação da conectividade de um software orientado por objetos, selecionou-se, na literatura, as principais métricas a serem utilizadas para a avaliação dos fatores que têm impacto nesse aspecto. Tais métricas foram classificadas quanto ao nível em que podem ser empregadas. Adicionalmente, propomos duas outras métricas para avaliar aspectos não abordados pelas métricas selecionadas na literatura: *conexões aferentes* e *peso da conexão aferente*. *Conexões aferentes* é uma métrica baseada em CBO (*Coupling between Object Class*) do conjunto CK. *Peso da conexão aferente* é uma métrica que propomos aqui como indicador da importância da conexão existente entre duas classes no sistema. Indicamos, também, a avaliação dos critérios acoplamento e coesão empregados na construção das classes. O resultado desta seleção é apresentado na Tabela 6.1.

Na Tabela 6.1, a métrica COF (Fator Acoplamento) faz parte do conjunto de métricas MOOD, proposto por Abreu e Carapuça [2]. As métricas DIT (*Depth of Inheritance Tree*), NOC (*Number of Children*) e LCOM (*Lack od Cohesion in Methods*) fazem parte do conjunto de métricas CK, proposto por Chidamber e Kemerer [13]. Elas foram apresentadas no Capítulo 3. A métrica para Estabilidade proposta por Myers [38] foi apresentada na Seção 4.1 desta dissertação. Grau de acoplamento entre classes, o grau de coesão interna de classe e o grau de coesão interna de método referem-se aos critérios utilizados na construção das classes do sistema e do estabelecimento dos relacionamentos entre elas. Eles seguem as respectivas escalas propostas no Capítulo 5.

A seguir, é apresentada a justificação da utilização de cada uma das métricas e critérios da Tabela 6.1 em MACSOO.

- **Adaptação da métrica para Estabilidade proposta por Myers:** essa métrica reflete o grau de estabilidade do sistema, pois indica a quantidade média de módulos do sistema que serão alterados em decorrência da alteração de um

Fator	***Nível***	***Métrica ou Critério***
Estabilidade	Sistema	Adaptação da métrica para Estabilidade proposta por Myers
Conectividade	Sistema	COF (Fator Acoplamento)
	Classe	Conexões aferentes
Ocultação de Informação	Classe	Quantidade de métodos públicos Quantidade de atributos públicos
Profundidade da Árvore de Herança	Classe	DIT (*Depth of Inheritance Tree*)
Acoplamento entre Classes	Agrupamento de classes	Grau do acoplamento entre classes Peso da conexão aferente
Coesão Interna de Classes	Classe	LCOM (*Lack of Cohesion in Methods*) Grau de coesão da interface da classe Grau de coesão interna de classe
	Método	Grau de coesão interna de método

Tabela 6.1: Métricas dos fatores de conectividade

módulo qualquer no sistema. O seu cálculo baseia-se nas conexões entre os módulos do sistema e considera também o grau de acoplamento entre os módulos bem como o grau de coesão interna dos mesmos. Como esta métrica foi proposta para o paradigma estruturado, é necessário que a forma de seu cálculo seja adaptada ao paradigma orientado por objetos, para que possa considerar as formas de acoplamento e coesão peculiares da orientação por objetos, bem como considerar as conexões decorrentes de relações de herança entre as classes do sistema. Esta adaptação é descrita na próxima seção.

- **COF(Fator Acoplamento):** é um indicador da conectividade do sistema. Esta métrica é a razão entre a quantidade de conexões existentes no sistema e a quantidade máxima possível de conexões no sistema. Assim, seu valor varia de 0 a 1, sendo que 0 indica um software sem conexão alguma entre as classes e 1, um sistema no qual cada classe possui uma conexão com todas as demais classes.

- **Conexões aferentes:** diante de um alto grau de conectividade do sistema, é preciso identificar as classes críticas que contribuem para esse fato. CBO (*Coupling Between Object Class*) é a métrica que indica quantas conexões uma

classe possui, considerando tanto as conexões eferentes - aquelas que se originam na classe - quanto as aferentes - aquelas que chegam até ela. Do ponto de vista de avaliação de conectividade, é importante conhecer as classes de maior impacto no sistema, ou seja, aquelas que têm grande número de conexões aferentes. Desta forma, será utilizada em MACSOO uma métrica que denominamos *conexões aferentes*. Esta métrica é dada pela quantidade de classes que conectam-se à classe em questão.

- **Quantidade de métodos públicos e quantidade de atributos públicos:** um dos fatores que têm impacto na conectividade é a ocultação de informação, pois quanto menor o número de membros públicos das classes, menor o número de caminhos pelos quais conexões entre elas podem ocorrer. Desta forma, é útil um indicador que aponte as classes que possuem o maior número de *conectores*, pois estas, possivelmente, são as que mais contribuem para a alta conectividade do sistema. A quantidade de métodos públicos e a quantidade de atributos públicos são utilizados para identificar estas classes.

- **DIT (*Depth of Inheritance Tree*):** a profundidade das árvores de herança presentes na estrutura de um software impacta na sua conectividade, pois quanto mais profunda uma árvore de herança, maior o número de conexões envolvidadas na hierarquia, conforme discutido na Seção 6.1. Assim, diante de um alto grau de conectividade do software, a profundidade das árvores de herança é um aspecto importante a ser analisado. DIT indica a posição de uma classe na árvore de herança da qual faz parte. Com DIT, é possível identificar as hierarquias de maior profundidade e, a partir daí, verificar a necessidade de reestruturação das mesmas.

- **Grau do acoplamento entre classes:** a importância de se avaliar o grau de acoplamento entre classes está no fato de que conexões cujo grau de acoplamento seja alto determinam um grau de dependência maior do que aquelas com grau de acoplamento mais fraco. Identificar as conexões no sistema com alto grau de acoplamento é ponto chave para auxiliar na redução da conectividade e no aumento da estabilidade do sistema.

- **Peso da conexão aferente:** dada a importância de se conhecer as conexões de maior impacto no sistema, propomos aqui uma métrica para este aspecto. Duas classes podem estar conectadas por mais de um caminho e cada uma

dessas conexões possui uma largura, ou seja, a cada um desses caminhos corresponde um grau de acoplamento. Se uma classe A conecta-se a B, seja usando um método ou atributo desta ou sendo herdeira desta, o *peso desta conexão aferente* de B é dado em função dos pesos atribuídos ao tipos de acoplamentos que ocorrem nas relações de A para B. Propomos utilizar o maior peso. Por exemplo, se A usa um método de B com passagem de parâmetros por referência e outro com passagem de parâmetro por valor, atribuindo-se peso 0,8 ao primeiro tipo de acoplamento e 0,2 ao segundo, o *Peso da conexão aferente* de B a partir de A é dado por 0,8, que é o maior valor dentre aqueles atribuídos às suas conexões aferentes.

- **LCOM (*Lack of Cohesion in Methods*) e grau de coesão interna da classe:** a coesão interna de uma classe impacta na conectividade porque classes pouco coesas tendem a realizar mais serviços. Assim, para a redução da conectividade, é importante identicar as classes com baixo grau de coesão no sistema, para que se possa analisar a necessidade de reestruturação das mesmas. LCOM é uma métrica útil neste aspecto, pois indica a ausência de coesão entre os métodos de uma classe.

- **Grau de coesão interna de classe, de método e grau de coesão de interface da classe:** MACSOO é baseado no processo de medir, avaliar e intervir na estrutura do sistema com o propósito de reduzir a conectividade no mesmo. Neste processo de melhoria da conectividade do sistema, certamente é necessária a avaliação e reestruturação das classes que o constituem. A avaliação de uma classe inclui a avaliação do relacionamento entre os seus elementos, entre os elementos de suas interface e de seus métodos.

6.2.2 Adaptação do Modelo de Estabilidade de Myers

O modelo de estabilidade de programas proposto por Myers [38] e descrito na Seção 4.1 mostra-se como um instrumento capaz de indicar o grau de estabilidade de um sistema. Porém, foi pensado sob os conceitos de projeto e programação vigentes na época, o paradigma estruturado. A idéia deste modelo mostra-se igualmente útil na avaliação de estabilidade de sistemas orientados por objetos se submetido a adaptações que atendam às características deste paradigma. Tais adaptações abrangem os seguintes aspectos:

Acoplamento	*Valor*
Conteúdo	0,95
Dado comum	0,7
Inclusão	0,7
Elemento externo	0,6
Controle	0,5
Referência	0,35
Informação	0,2

Coesão	*Valor*
Coincidental	0,95
Lógica	0,4
Temporal	0,6
Procedimental	0,4
Comunicacional	0,25
Contratual	0,2

Tabela 6.2: Pesos de Acoplamento e Coesão na OO

- Escalas de acoplamento e coesão à luz da orientação por objetos.
- Consideração das conexões entre classes estabelecidas por herança.

As escalas de acoplamento entre classes e de coesão interna de classes são aquelas propostas no Capítulo 5. Myers associa a cada grau de acoplamento e coesão um valor que é utilizado nos cálculos do modelo. A Tabela 6.2 mostra os valores associados às escalas de acoplamento entre classes e os valores associados às escalas de coesão interna de classes em sistemas orientados por objetos.

O critério utilizado para definir tais valores foi manter, no caso da orientação por objetos, os valores usados por Myers para o paradigma estruturado. O motivo é que o papel das escalas não mudou no paradigma orientado por objetos; o que mudou foi a forma como elas se apresentam. Entretanto, vale ressaltar que a atribuição desses valores foi feita por Myers de forma intuitiva.

O fato novo é que, no paradigma OO, manifesta-se a forma de acoplamento por Inclusão e a coesão interna de classe do tipo Contratual substitui os tipos Informacional e Funcional do paradigma estruturado. Nesses novos casos, os valores associados a eles foram definidos em função da posição que o tipo ocupa na escala. Ao acoplamento por Inclusão é associado o valor 0,7 devido ao fato de que este tipo de acoplamento tem efeito comparável ao acoplamento por dado comum. O tipo de coesão Contratual, o melhor nível de coesão interna de classe, recebe o valor 0,2, valor associado ao melhor nível no caso do paradigma estruturado.

Para que os cálculos realizados no modelo de estabilidade de Myers considerem as conexões decorrentes de relacionamentos de herança, um valor deve ser atribuído a

esse tipo de conexão. O valor definido aqui é 0,7. O critério utilizado para definir esse valor é o fato de que o relacionamento de herança entre duas classes introduz uma dependência muito estreita em que alterações têm grande propagação.

A forma de cálculo utilizada no modelo original é mantida. A única ressalva é quanto ao preenchimento da matriz C, que representa os valores dos acoplamentos existentes entre os módulos do sistema:

- Esta matriz deve considerar também as conexões decorrentes de herança.
- Myers não define como proceder no caso de um módulo estar acoplado a outro por mais de uma forma. Em nossa proposta, define-se que deve ser considerado o valor dado pela métrica **Peso da conexão aferente**, descrita anteriormente.

6.2.3 Descrição do Modelo

O Modelo de Avaliação de Conectividade de Sistemas Orientados por Objetos visa medir a conectividade em sistemas orientados por objetos. Este modelo propõe-se a ser um instrumento de apoio à decisão de utilidade no processo de softwares orientado por objetos, na obtenção de software mais estável e mais fácil de manter.

Para atingir o seu objetivo de apoiar a decisão no sentido de obter sistemas com baixo grau de conectividade, o modelo, resumidamente, consiste em avaliar a estabilidade do software por meio de sua conectividade e diante de um grau insatisfatório de conectividade, buscar a melhoria deste aspecto por meio da avaliação e atuação nos aspectos que a influenciam. O modelo conta com o uso de métricas de produto de software orientado por objetos para a avaliação dos aspectos envolvidos.

O Modelo de Avaliação de Conectividade de Sistemas Orientados por Objetos proposto nesta dissertação não tem por objetivo simplesmente a coleta e o relatório de métricas. Ele também realimenta o processo de software à medida que propõe um conjunto de passos que guia por um caminho de decisões a serem tomadas diante de um valor indesejado para determinada métrica, em especial, para a conectividade.

MACSOO aplica-se nas seguintes situações principais:

1. Durante as fases de construção de sistema, MACSOO deve ser utilizado para a avaliação do software em construção. As regras e métodos constituintes de

MACSOO empregam-se para evitar que o produto final possua alto grau de conectividade, diminuindo, assim, a dificuldade de manutenção futura.

2. No caso de produtos já construídos, emprega-se MACSOO na fase de manutenção. Nesta fase, é de extrema importância que se possa identificar a amplitude de uma modificação no sistema. MACSOO conta com métricas que auxiliam na avaliação deste impacto.

3. MACSOO pode ser utilizado como instrumento de apoio na reestruturação de sistemas.

4. O modelo pode ser utilizado também como instrumento na predição de esforço de manutenção de sistemas orientados por objetos, uma vez que conta com métricas que refletem o grau de estabilidade do software.

Para descrever o Modelo de Avaliação de Conectividade de Software Orientado por Objetos, será utilizado o diagrama de atividade da UML [49]. Esse tipo de diagrama é semelhante a um fluxograma e visa esquematizar fluxos de atividades [42]. Essa abordagem de descrição foi escolhida a fim de facilitar a compreensão do modelo. As Figuras 6.4 a 6.9 são os diagramas de atividades de MACSOO. Este modelo é estruturado em estágios e passos. Um estágio no modelo corresponde a avaliação de um dos fatores envolvidos na avaliação da estabilidade do software. Cada estágio é constituído de passos, que são as etapas necessárias para concluir a avaliação e a melhoria do fator avaliado.

Os estágios em MACSOO correspondem à avaliação dos fatores relacionados à estabilidade e à conectividade. São eles:

1. Avaliação de estabilidade,

2. avaliação de conectividade,

3. avaliação de acoplamento,

4. avaliação de coesão,

5. avaliação de ocultação de informação,

6. avaliação de profundidade de árvore de herança

Cada estágio é constituído por passos nos quais coletam-se métricas, avalia-se o resultado e atua-se no respectivo fator com o objetivo de redução da conectividade do sistema. A Tabela 6.3 relaciona os estágios abordados em MACSOO, seus respectivos passos e as métricas utilizadas.

A seguir, são descritos em detalhes os estágios, e seus respectivos passos, constituintes do Modelo de Avaliação de Conectividade de Software Orientado por Objetos proposto neste trabalho.

1. **Avaliação de estabilidade**

   A Figura 6.4 representa o primeiro estágio de MACSOO. Neste estágio, verifica-se o grau de estabilidade do sistema a ser avaliado. A métrica utilizada aqui é a resultante da *métrica para Estabilidade proposta por Myers adaptada à OO* , descrita na Seção 6.2.2. O resultado desta métrica indica quantos módulos em média sofrerão impacto em virtude de uma modificação em um dos módulos do sistema. De acordo com a tese de que a conectividade é o fator de supremacia na determinação da estabilidade de um sistema, diante de um resultado que denote um nível indesejável de estabilidade do sistema, deve-se, então, avaliar o grau de conectividade do mesmo.

   Os passos desse estágio são:

   (a) **Avalie a estabilidade do software:** este passo consiste na coleta e avaliação da *métrica para Estabilidade proposta por Myers adaptada à OO*.

   (b) **Atue na conectividade:** a partir do resultado obtido no passo anterior, caso a estabilidade seja considerada baixa, passa-se para o próximo estágio de *Avaliação da Conectividade*, descrito a seguir.

2. **Avaliação de Conectividade**

   A Figura 6.5 representa o estágio de *Avaliação de Conectividade*, ponto central de MACSOO. Neste estágio, coleta-se a métrica que indique o grau de conectividade do sistema. A métrica utilizada aqui é COF (Fator Acoplamento). Obtendo-se um resultado considerado alto para esta métrica, deve-se buscar identificar e atuar nas causas deste elevado grau de conectividade. Para isso, é importante identificar as classes que contribuem de forma crítica para esta situação. A métrica utilizada para este fim em MACSOO é CBO (*Coupling between Object Class*). Identificadas essas classes, deve-se buscar avaliá-las e,

Estágio	***Passos***	***Métricas***
Avaliação de estabilidade	Avalie a estabilidade do software.	Métrica para Estabilidade proposta por Myers adaptada à OO
	Atue na conectivade.	–
Avaliação de conectividade	Obtenha métricas de conectividade do sistema.	COF
	Identifique as classes mais conectadas do sistema.	Conexões aferentes
	Avalie os acoplamentos entre as classes.	–
	Avalie a coesão interna das classes.	–
	Avalie a ocultação de informação.	–
	Avalie a profundidade da árvore de herança.	–
Avaliação de acoplamento	Identifique as conexões de grau de acoplamento elevado.	Grau de acoplamento entre classes Peso da conexão aferente
	Reestruture as classes envolvidas.	–
Avaliação de coesão	Avalie a coesão da classe.	LCOM Grau de coesão interna da classe Grau de coesão de interface da classe
	Reestruture a classe.	Grau de coesão interna da classe Grau de coesão interna dos métodos
Avaliação de ocultação de informação	Obtenha métricas de ocultação de informação da classe.	quantidade de atributos públicos da classe quantidade de métodos públicos da classe
	Reestruture a classe.	–
Avaliação da profundidade de árvore de herança	Obtenha métrica da profundidade da árvore de herança.	DIT
	Reestruture a árvore de herança.	–

Tabela 6.3: Estágios e passos de MACSOO

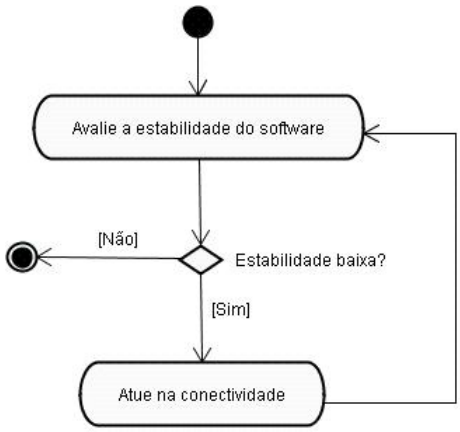


Figura 6.4: Avaliação da estabilidade do sistema

sempre que possível, reestruturá-las visando a diminuição da conectividade. Tal reestruturação deve priorizar os aspectos que determinam a conectividade.

O processo proposto em MACSOO é realimentado, pois baseia-se na análise da conectividade, intervenção no sistema e reavalição da conectividade para verificar o impacto da intervenção realizada. O diagrama da Figura 6.5 representa esta idéia. Esse estágio é constituído pelos seguintes passos:

(a) **Obtenha métricas de conectividade do sistema:** neste passo, a métrica COF é coletada. O resultado desta métrica indica o grau de conectividade do sistema. Se o resultado obtido neste passo indicar alta conectividade, deve-se realizar o passo seguinte a fim de indentificar as classes críticas e avaliá-las.

(b) **Identifique as classes mais conectadas no sistema:** obtém-se para cada classe do sistema uma métrica que informe o grau de conectividade da classe. A métrica utilizada nesse passo é *conexões aferentes*. Este é um passo importante pois nele identificam-se as classes cujas alterações têm grande amplitude de impacto no sistema. Deve-se analisar tais classes sob os aspectos que impactam na conectividade: a ocultação de informação, o acoplamento com outras classes, a coesão interna e a profundidade na árvore de herança.
Neste ponto, então, para cada classe identificada como crítica, deve-se escolher o aspecto a ser analisado. Esta análise é representada em MACSOO

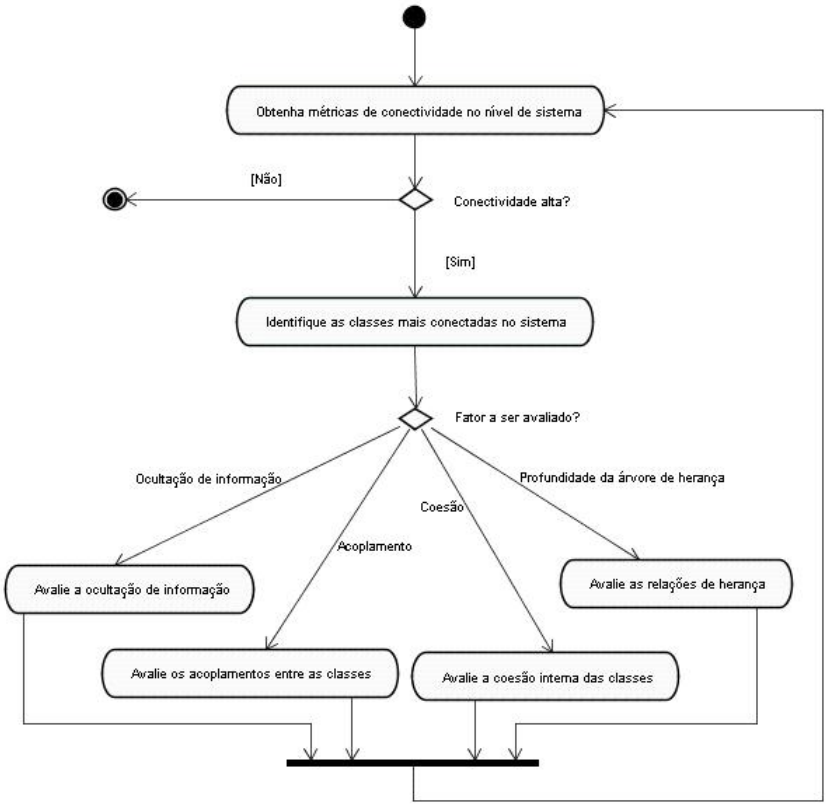


Figura 6.5: Avaliação do fator conectividade

como um estágio. Por exemplo, os passos da análise do aspecto profundidade na árvore de herança é descrita no estágio *Avaliação de profundidade da árvore de herança*.

(c) **Avalie os acoplamentos entre as classes:** este passo representa a realização dos passos descritos no estágio de *Avaliação de acoplamento*.

(d) **Avalie a coesão interna das classes:** este passo representa a realização dos passos descritos no estágio de *Avaliação de coesão interna*.

(e) **Avalie a ocultação de informação:** este passo representa a realização dos passos descritos no estágio de *Avaliação de ocultação de informação*.

(f) **Avalie a profundidade da árvore de herança:** este passo representa a realização dos passos descritos no estágio de *Avaliação de profundidade da árvore de herança*.

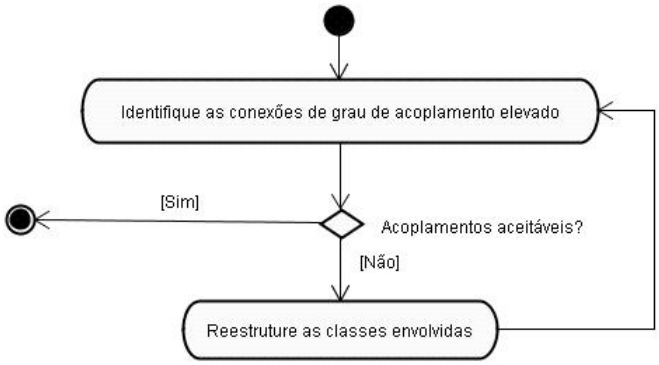


Figura 6.6: Avaliação do fator Acoplamento

3. **Avaliação de Acoplamento**

   Nesse estágio, realiza-se a análise do grau de acoplamento entre uma classe e as demais. O objetivo desta análise é identificar as conexões com maior grau de acoplamento. A contribuição disso para a diminuição da conectividade deve-se principalmente ao fato que para conhecer a real importância de uma conexão na estabilidade de um sistema, é necessário avaliar o nível de dependência entre as classes envolvidas na conexão. Como conexões mais fortes, de alto grau de acoplamento, resultam em maior grau de interdependência entre as classes do sistema, deve-se buscar diminuir o acoplamento das conexões ou, na melhor hipótese, eliminá-las.
   Outro ponto importante da avaliação desse fator é que o grau do acoplamento da conexão é um bom caminho para diminuir conectividade porque pode refletir a existência que classes mal projetadas no sistema. A partir da reestruturação destas classes, pode-se obter ganho na conectividade do sistema.

   Esse estágio, representado na Figura 6.6, é constituído pelos seguintes passos:

   (a) **Identifique as conexões de grau de acoplamento elevado:** para a realização deste passo, utiliza-se a métrica *peso de conexão aferente*. Esta métrica indica as conexões de maior impacto na estabilidade do sistema.

   (b) **Reestruture as classes envolvidas:** este passo consiste em analisar as classes envolvidas, bem como os relacionamento existentes entre elas. Para atingir o objetivo de diminuição de conectividade, a reestruturação das classes convém seguir as regras definidas por Meyer [36] para a construção

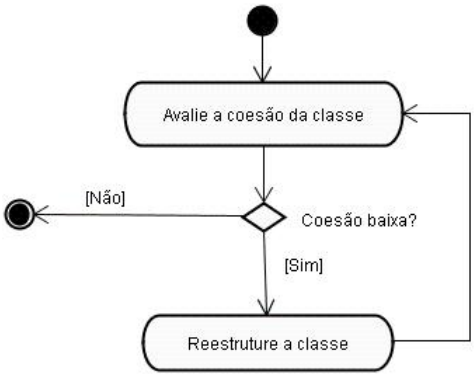


Figura 6.7: Avaliação do fator Coesão

de software modular, em especial, as seguintes: poucas interfaces, interface pequena e ocultação de informação. Estas regras foram apresentadas na Seção 2.2.3 desta dissertação.

4. **Avaliação de coesão**

   Uma classe pode possuir muitas conexões por desempenhar um papel pouco específico, por exemplo, por realizar mais de um contrato. Este aspecto pode ser verificado a partir da análise da coesão entre os elementos da classe. Constatando-se que a classe é pouco coesa, deve-se reestruturá-la; possivelmente, outras classes surgirão a partir desta reestruturação. Esse estágio, representado na Figura 6.7, é constituído pelos seguintes passos:

   (a) **Avalie a coesão da classe:** neste passo são obtidas as métricas de coesão interna das classes. Dois indicadores são utilizados nesse passo: a métrica LCOM, do conjunto CK, e o grau de coesão da classe. A escala para este fator é aquela proposta no Capítulo 5 desta dissertação.

   (b) **Reestruture a classe:** a reestruturação da classe aqui deve visar principalmente o aumento da coesão da interface da classe, tendo em vista que é por meio de sua interface que uma classe se conecta com as demais.

5. **Avaliação de ocultação de informação**

   Os membros públicos de uma classe constituem os meios pelos quais as demais classes podem se conectar a ela. Melhorar a ocultação de informação de uma classe contribui para a obtenção de sistema de baixa conectividade à medida que diminui os caminhos possíveis de estabelecimento de conexão. Esse estágio

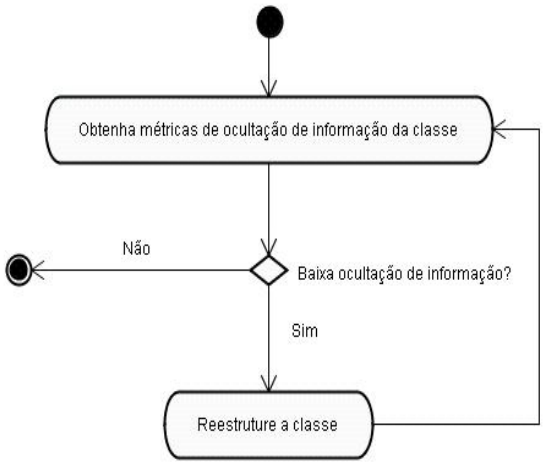


Figura 6.8: Avaliação do fator Ocultamento de Informação

consiste na avaliação da ocultação de informação de uma classe, visando a melhoria deste aspecto como um meio de diminuir a conectividade.

(a) **Obtenha métricas de ocultação de informação da classe:** os indicadores utilizados neste passo são a quantidade de atributos e métodos da classe acessíveis às demais classes do sistema. Um valor alto para esses indicadores pode denotar a causa do alto grau de conectividade da classe.

(b) **Reestruture a classe:** caso o resultado obtido no passo anterior não seja satisfatório, é necessário reestruturar a classe com o objetivo de ocultar o máximo possível as suas informações. Neste caso, é possível que outras classes do sistema sejam afetadas e demandem reestruturação também.

6. **Avaliação de profundidade de árvore de herança**

Conforme discutido na Seção 6.1, a herança é um fator que contribui para a conectividade em softwares. A profundidade das classes nas árvores de herança é uma métrica que pode ser utilizada como indicador da necessidade de reestruturação das árvores visando redução da conectividade. A Figura 6.9 representa o estágio de avaliação da profundidade da árvore de herança da classe, cujos passos são:

(a) **Obtenha métrica da profundidade da árvore de herança:** neste

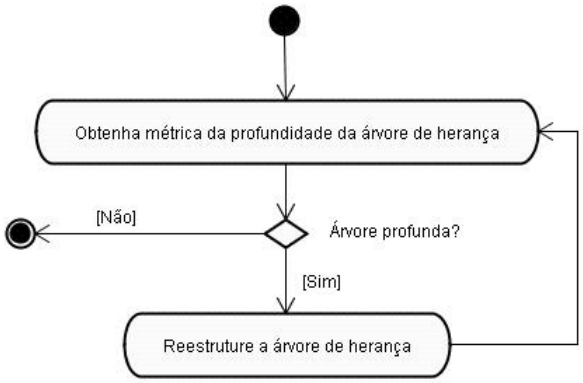


Figura 6.9: Avaliação do Fator Herança

passo, coleta-se a métrica DIT (*Depth of Inheritance Tree*) para a classe em questão.

(b) **Reestruture a árvore de herança:** um valor alto para esta métrica indica que a árvore na qual a classe está inserida pode precisar ser reestruturada, em benefício da diminuição da conectividade do sistema. Deve-se, então, avaliar a hierarquia à qual a classe pertence, buscando obter hierarquias de profundidade menores.

6.3 Conclusão

O Modelo de Avaliação de Conectividade de Software Orientado por Objetos, MACSOO, proposto nesta dissertação e apresentado neste capítulo visa a solução de um dos problemas mais graves vivenciados pela comunidade produtora de software: a instabilidade de sistema, que é sinônimo de custo elevado. Ele tem como base a conectividade como fator de ordem superior na determinação da estabilidade de um software.

MACSOO é um modelo que visa a diminuição da conectividade em sistemas orientados por objetos a partir da avaliação e restruturação de sistemas sob os aspectos que determinam a conectividade. É importante salientar que fazem parte de MACSOO questionamentos do tipo "o fator avaliado é satisfatório?". O modelo proposto aqui enfatiza a importância do impacto dessse fatores na conectividade, mas não determina

os valores considerados satisfatórios.

Nesta dissertação, desenvolvemos uma ferramenta que viabiliza a implementação de MACSOO, a fim de que ele seja exemplificado. A ferramenta avalia sistemas escritos em Java. A ferramenta e os resultados obtidos são apresentados no Capítulo 7.

Capítulo 7

Connecta - Uma Ferramenta para Implementação de MACSOO

Conforme apresentado na Seção 6.2, MACSOO é um modelo que visa a avaliação da conectividade em sistemas orientados por objetos. Para isso, ele conta com a coleta de métricas no software para aspectos como estabilidade, conectividade, ocultação de informação, herança, dentre outras.

Há, hoje, algumas ferramentas disponíveis no mercado e propostas do meio acadêmico que realizam a coleta de métricas em softwares orientados por objetos. Entretanto, tais ferramentas não se mostram adequadas à implementação de MACSOO, visto que não abordam todas as métricas necessárias ao modelo, como por exemplo o cálculo de estabilidade do Modelo de Myers. Além disso, a idéia básica de MACSOO é um conjunto de heurísticas para tomada de decisão no processo de construção de software no sentido de reduzir a conectividade do mesmo, cujos passos são, resumidamente:

1. avaliar a estabilidade do software;
2. diante de um baixo grau de estabilidade, analisar a conectividade do software, tendo este aspecto como determinante da estabilidade do software;
3. diante de um alto de grau de conectividade, investigar os pontos de impacto neste aspecto, como ocultação de informação, acoplamento entre as classes e coesão interna das mesmas.

Desta forma, faz-se necessário uma ferramenta de propósito específico para MACSOO, que realize a coleta de métricas em software OO e forneça ao usuário as informações necessárias para que seja possível que ele aplique as heurísticas propostas no modelo. Nesta dissertação, desenvolvemos uma ferramenta com esse propósito, para análise de softwares desenvolvidos em Java.

Este capítulo descreve o objetivo de *Connecta*, suas funcionalidades, arquitetura e os detalhes mais importantes de sua implementação. Nesta descrição, é utilizada a notação UML. Os diagrama apresentados aqui foram elaborados na ferramenta Jude [29].

7.1 Especificação dos Requisitos

A especificação dos requisitos de Connecta baseia-se no artefato ERSw proposto por Paula [42], sendo constituída pelo objetivo da ferramenta, pelo diagrama de casos de uso, pelos atores, pela lista de casos de uso e pela descrição das interfaces de usuário.

Connecta tem por objetivo a coleta das métricas necessárias a MACSOO, bem como a geração e a consulta de histórico dos resultados das análises realizadas.

7.1.1 Casos de Uso

A Figura 7.1 mostra o Diagrama de Casos de Uso de *Connecta*. O ator da ferramenta é denominado *Usuário*. A Tabela 7.1 lista os casos de uso principais da ferramenta e seus respetivos objetivos.

Caso de Uso	***Descrição***
Analisar Software	Coleta de métricas de um software
Consultar Histórico	Consulta de dados de coletas realizadas

Tabela 7.1: Lista de casos de uso de Connecta

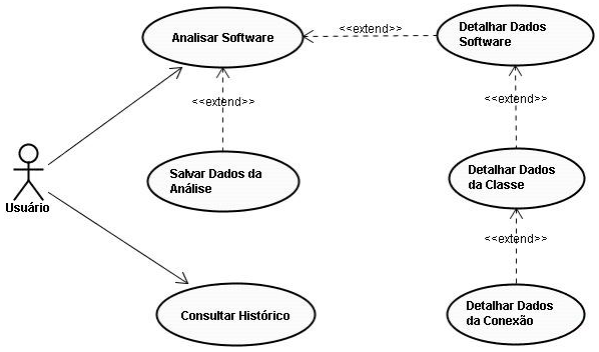


Figura 7.1: Diagrama de Casos de Uso de Connecta

7.1.2 Interfaces de Usuário

Connecta conta com as seguintes interfaces de usuário: *Principal, Seleção de Classes para Análise, Métricas do Sistema, Detalhes do Sistema, Detalhes da Classe, Detalhes da Conexão, Consulta de Resultado de Análise - Seleção de Arquivo* e *Consulta de Resultado de Análise*. A seguir estas interfaces são descritas em detalhes.

- *Principal*: é a tela inicial da aplicação. A Figura 7.2 apresenta sua interface. A Tabela 7.2 descreve os comandos desta interface e suas respectivas ações.

Comando	***Ação***
Nova Análise	Exibe interface de usuário *Seleção de Classes para Análise* .
Consulta de Resultado de Análise	Exibe interface de usuário *Consulta de Resultado de Análise - Seleção de Arquivo*.
Sair	Finaliza a aplicação.

Tabela 7.2: Comandos da interface de usuário *Principal*

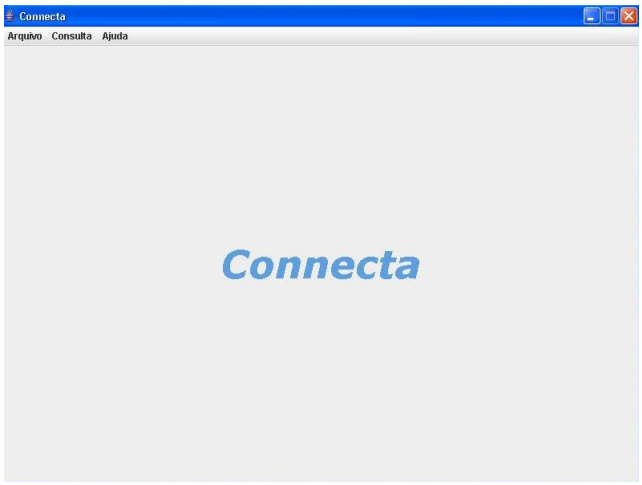


Figura 7.2: Tela *Principal*

- *Seleção de Classes para Análise:* interface para o usuário selecionar as classes a serem analisadas pela ferramenta. A Figura 7.3 apresenta sua interface. A Tabela 7.3 descreve os campos desta interface. A Tabela 7.4 descreve os comandos e suas respectivas ações.

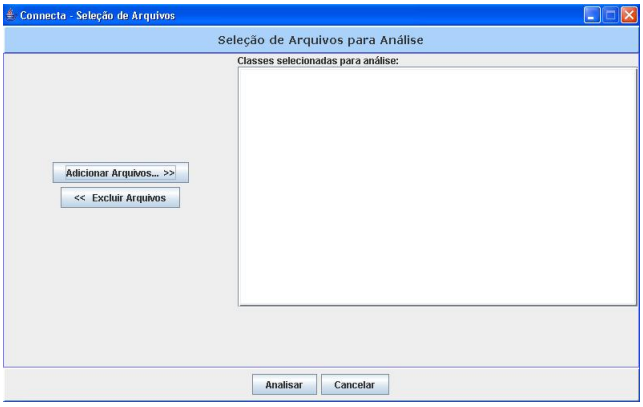


Figura 7.3: Tela *Seleção de Classes para Análise*

- *Métricas do Sistema:* interface para exibir as métricas do sistema. A partir desta interface é possível gravar os dados da análise. A Figura 7.4 apresenta sua interface. A Tabela 7.5 descreve os campos desta interface. A Tabela 7.6 descreve os comandos e suas respectivas ações.

Campo	***Descrição***	***Tipo***
Classes selecionadas para análise	Classes selecionadas para análise	Lista de seleção múltipla

Tabela 7.3: Campos da interface de usuário *Seleção de Classes para Análise*

Comando	***Ação***
Adicionar Arquivos	Abre diálogo para que o usuário selecione os arquivos .class e os insira na lista de arquivos a serem analisados. *Métricas do Sistema*.
Excluir Arquivos	Retira arquivos selecionados da lista de arquivos .class a serem analisados. *Métricas do Sistema*.
Analisar	Realiza análise das classes selecionadas e exibe resultado na interface de usuário *Métricas do Sistema*.
Cancelar	Fecha a interface de usuário.

Tabela 7.4: Comandos da interface de usuário *Seleção de Classes para Análise*

Figura 7.4: Tela *Métricas do Sistema*

Campo	***Descrição***	***Tipo***
Total de classes	Quantidade de classe no sistema analisado	Texto não editável
Total de conexões	Quantidade de conexões no sistema	Texto não editável
COF	Valor da métrica COF do software	Texto não editável
Estabilidade	Valor da métrica de estabilidade calculada a partir do Modelo de Myers	Texto não editável

Tabela 7.5: Campos da interface de usuário *Métricas do Sistema*

Comando	***Ação***
Detalhar	Exibe a interface de usuário *Detalhes do Sistema*.
Salvar histórico	Exibe diálogo solicitando nome do arquivo para gravação e grava os resultados da coleta em tal arquivo.
Fechar	Fecha a interface de usuário.

Tabela 7.6: Comandos da interface de usuário *Métricas do Sistema*

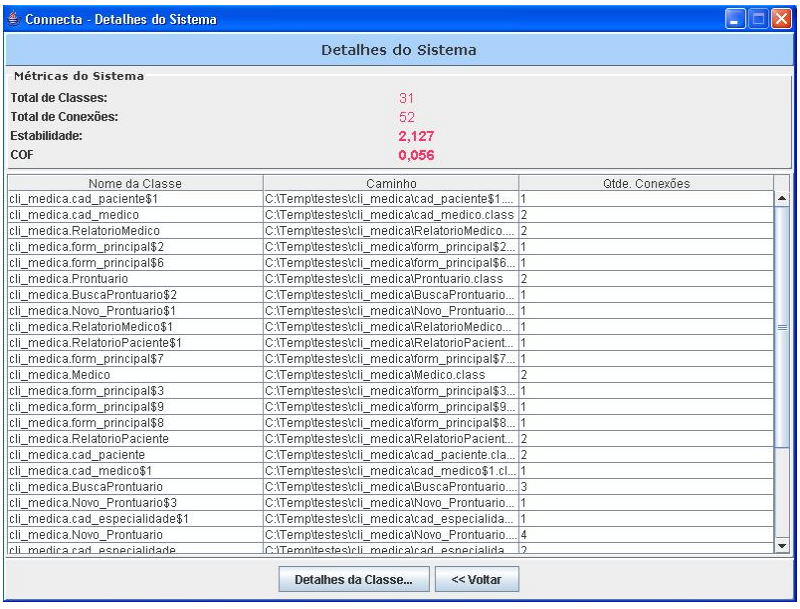


Figura 7.5: Tela *Detalhes do Sistema*

- *Detalhes do Sistema:* interface para exibir os detalhes de análise do sistema. A Figura 7.5 apresenta sua interface. A Tabela 7.7 descreve os campos desta interface. A Tabela 7.8 descreve os comandos e suas respectivas ações.

Campo	***Descrição***	***Tipo***
Total de classes	Quantidade de classes selecionadas para análise	Texto não editável
Total de conexões	Quantidade de conexões no sistema	Texto não editável
COF	Valor da métrica COF do software	Texto não editável
Estabilidade	Valor da métrica de estabilidade calculada a partir do Modelo de Myers	Texto não editável
Nome da classe	Nome das classes do sistema	Texto editável
Caminho	Caminhos das classes do sistema	Texto não editável
Qtde. Conexões	Quantidade de conexões aferentes de cada classe do sistema	Texto não editável

Tabela 7.7: Campos da interface de usuário *Detalhes do Sistema*

Comando	***Ação***
Detalhes da Classe	Exibe a interface de usuário *Detalhes da Classe*.
Voltar	Exibe a interface de usuário *Métricas do Sistema*

Tabela 7.8: Comandos da interface de usuário *Detalhes do Sistema*

- *Detalhes da Classe:* interface para exibir os detalhes da análise de uma classe. A Figura 7.6 apresenta sua interface. A Tabela 7.9 descreve os campos da interface. A Tabela 7.10 descreve os comandos e suas respectivas ações.

- *Detalhes da Conexão:* interface para exibir os detalhes de uma conexão no sistema. A Figura 7.7 apresenta sua interface. A Tabela 7.11 descreve os campos desta interface. A Tabela 7.12 descreve os comandos e suas respectivas ações.

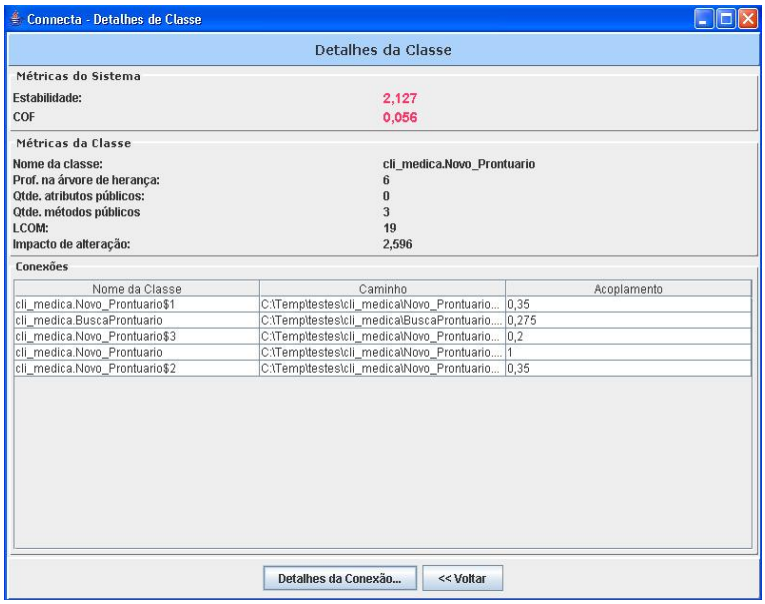


Figura 7.6: Tela *Detalhes do Sistema*

Figura 7.7: Tela *Detalhes da Conexão*

Campo	***Descrição***	***Tipo***
COF	Valor da métrica COF do software	Texto não editável
Estabilidade	Valor da métrica de estabilidade calculada a partir do Modelo de Myers	Texto não editável
Nome da classe	Nome da classe cujos dados são detalhados na interface	Texto não editável
Profundidade na árvore de herança	Valor da métrica DIT para a classe	Texto não editável
Qtde. de atributos públicos	Quantidade de atributos públicos da classe	Texto não editável
Qtde. de métodos públicos	Quantidade de métodos públicos da classe	Texto não editável
LCOM	Valor da métrica LCOM para a classe	Texto não editável
Impacto de alteração	Expectativa de número de classes alteradas em decorrência de uma alteração na classe.	Texto não editável
Nome da classe	Nome das classes às quais a classe em tela está conectada.	Texto não editável
Caminho	Caminhos das classes às quais a classe em tela está conectada.	Texto não editável
Acoplamento	Valor do peso da conexão aferente da classe.	Texto não editável

Tabela 7.9: Campos da interface de usuário *Detalhes da Classe*

Comando	***Ação***
Detalhes da Conexão	Exibe a interface de usuário *Detalhes da Conexão.*
Voltar	Exibe a interface de usuário *Detalhes do Sistema*

Tabela 7.10: Comandos da interface de usuário *Detalhes da Classe*

Campo	***Descrição***	***Tipo***
Classe Origem	Nome da classe origem, isto é, da classe usuária.	Texto não editável
Classe destino	Nome da classe destino, isto é, da classe servidora.	Texto não editável
Assinatura do campo	Nomes dos campos pelos quais ocorrem conexão entre a classe origem e a classe destino.	Texto não editável
Assinatura do método (Por Informação)	Nomes dos métodos com passagem de parâmetro sem efeito de referência pelos quais ocorrem conexão entre a classe origem e a classe destino.	Texto não editável
Assinatura do método (Por Referência)	Nomes dos métodos com passagem de parâmetro com efeito de referência pelos quais ocorrem conexão entre a classe origem e a classe destino.	Texto não editável

Tabela 7.11: Campos da interface de usuário *Detalhes da Conexão*

Comando	***Ação***
Voltar	Exibe a interface de usuário *Detalhes da Classe*

Tabela 7.12: Comandos da interface de usuário *Detalhes da Conexão*

- *Consulta de Resultado de Análise - Seleção de Arquivo:* interface para o usuário selecionar o arquivo que contém os dados da análise que deseja consultar. A Figura 7.8 apresenta sua interface. A Tabela 7.13 descreve os campos da interface. A Tabela 7.14 descreve os comandos e suas respectivas ações.

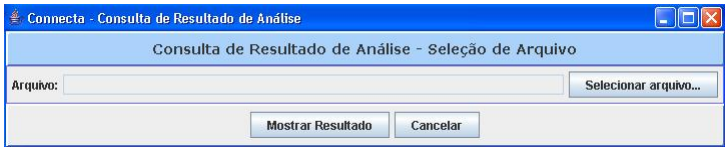


Figura 7.8: Consulta de Resultado de Análise - Seleção de Arquivo

Campo	***Descrição***	***Tipo***
Arquivo	Nome do arquivo que contém os dados a serem consultados.	Texto não editável

Tabela 7.13: Campos da interface de usuário *Detalhes do Sistema*

Comando	***Ação***
Selecionar Arquivo	Exibe diálogo para informação do arquivo que contém os dados a serem consultados.
Mostrar Resultado	Exibe diálogo para informação do arquivo que contém os dados a serem consultados.
Cancelar	Fecha a interface.

Tabela 7.14: Comandos da interface de usuário *Detalhes da Classe*

- *Consulta de Resultado de Análise:* interface para exibir dados de resultado de uma análise já realizada. A Figura 7.9 e 7.10 apresentam sua interface. A Tabela 7.15 descreve os campos desta interface. A Tabela 7.16 descreve os comandos e suas respectivas ações.

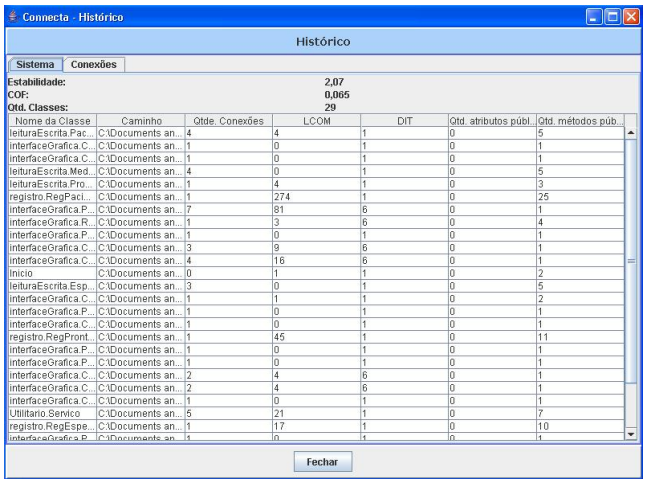


Figura 7.9: Tela *Consulta de Resultado de Análise - Sistema*

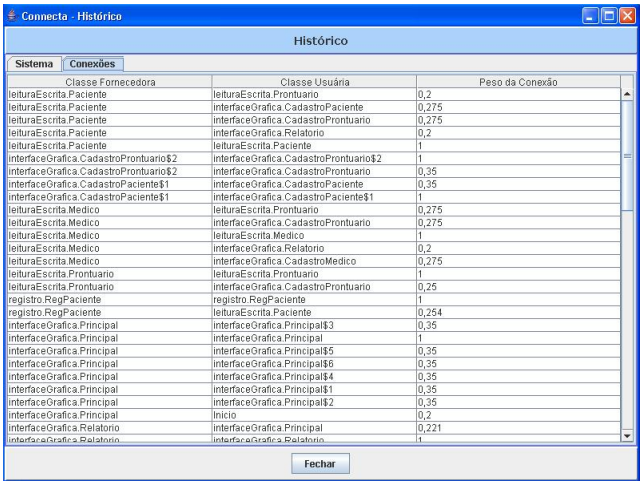


Figura 7.10: Tela *Consulta de Resultado de Análise - Conexões*

Campo	***Descrição***	***Tipo***
Total de classes	Total de classes analisadas	Texto não editável
Total de conexões	Total de conexões no sistema	Texto não editável
COF	Valor da métrica COF do software	Texto não editável
Estabilidade	Valor da métrica de estabilidade calculada a partir do Modelo de Myers	Texto não editável
Nome da Classe	Nomes das classes analisadas	Texto não editável
Caminho	Caminhos das classes analisadas	Texto não editável
Qtde. Conexões	Quantidade de conexões aferentes de cada classe analisada	Texto não editável
LCOM	Valor da métrica LCOM de cada classe analisada.	Texto não editável
DIT	Valor da métrica DIT de cada classe analisada.	Texto não editável
Qtde. atributos públicos	Quantidade de atributos públicos de cada classe analisadas.	Texto não editável
Qtde. métodos públicos	Quantidade de métodos públicos de cada classe analisada.	Texto não editável
Classe Fornecedora	Nomes das classes fornecedoras nas conexões do sistema.	Texto não editável
Classe Usuária	Nomes das classes usuárias nas conexões do sistema.	Texto não editável

Tabela 7.15: Campos da interface de usuário *Consulta de Resultado de Análise*

Comando	***Ação***
Fechar	Fecha a interface de usuário

Tabela 7.16: Comandos da interface de usuário *Consulta de Resultado de Análise*

7.2 Arquitetura

O diagrama da Figura 7.11 mostra a arquitetura da ferramenta *Connecta*. Neste diagrama, as classes estão agrupadas em pacotes lógicos: *Classes de Apresentação*, que contém as classes responsáveis pela interação com o usuário; *Classes de Negócio*, que contém as classes que realizam o processamento necessário para os cálculos das métricas; *Classes de Persistência*, que contém as classes responsáveis pelo armazenamento e recuperação de dados. Além disso, foram utilizadas classes com o objetivo de encapsular os dados a serem trafegados entre as classes do software; estas estão em destaque no diagrama. Na Tabela 7.17 são descritos os objetivos das classes do sistema.

7.3 Implementação

Connecta foi desenvolvida em Java, utilizando-se para tal a ferramenta IDE NetBeans [39] Versão 4.1. Possui 20 classes perfazendo um total de 5200 linhas de código.

O funcionamento da ferramenta baseia-se na análise do código das classes de um sistema. Essa análise é realizada diretamente no *bytecode* das classes, ao invés de analisar o código fonte das mesmas. Esta decisão de implementação tem por objetivo prover maior portabilidade da ferramenta em relação às diferenças entre as versões de Java. Para isso, foi utilizada a biblioteca BCEL (*Byte Code Engineering Library*) [6]. Dentre outros serviços, BCEL fornece recursos para análise de arquivos de *bytecode* de Java, o que atende a estratégia de implementação de *Connecta*.

O ponto central da implementação da ferramenta é a estrutura de dados utilizada para representar o sistema. O sistema foi modelado e representado por um grafo direcionado, no qual:

- um nodo representa uma classe do sistema;
- uma aresta de A para B representa uma conexão na qual A é a classe usuária e B, a classe servidora; no caso de relações de herança, A é a classe herdeira e B, uma superclasse de A.

O grafo foi implementado por meio de listas de adjacências [62]. Para ilustrar sua implementação, seja o grafo da Figura 7.12. A Figura 7.13 mostra o esquema da

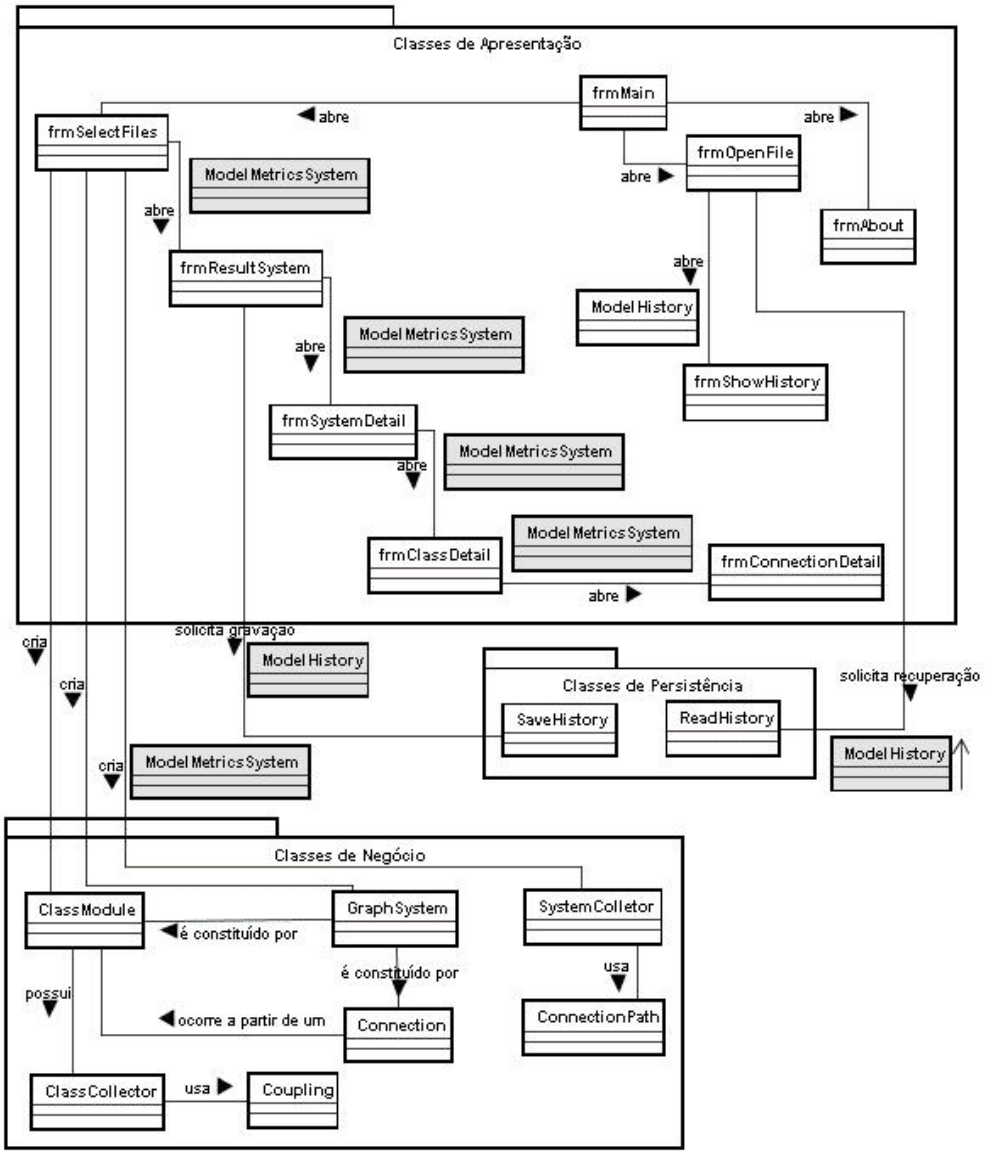


Figura 7.11: Arquitetura de Connecta

Nome da classe	***Objetivo***
ClassCollector	Realiza coleta das métricas necessárias em uma classe do sistema sob análise.
ClassModule	Representa uma classe no sistema em análise.
Connection	Representa uma conexão no sistema em análise.
ConnectionPath	Representa um caminho entre duas classes no sistema.
Coupling	Classe depositária que armazena valores associados aos diversos tipos de acoplamento entre classes de um sistema.
frmAbout	Classe de apresentação de informações sobre a ferramenta.
frmClassDetail	Classe de apresentação de informações sobre uma classe do sistema em análise.
frmConnectionDetail	Classe de apresentação de informações sobre conexões entre duas classes.
frmMain	Classe de apresentação da interface principal da ferramenta.
frmOpenFile	Classe de apresentação que permite a seleção de arquivo que contém dados de análise já realizada para consulta.
frmResultSystem	Classe de apresentação das métricas do sistema e permite a gravação do resultado da análise em histórico.
frmSelectFiles	Classe de apresentação que permite a seleção de classes a serem analisadas.
frmShowHistory	Classe de apresentação de dados de uma análise realizada.
frmSystemDetail	Classe de apresentação de detalhes da análise de um sistema.
GraphSystem	Classe usada para modelar o sistema em análise como um grafo direcionado.
ModelHistory	Classe que contém dados do resultado da análise de um sistema a serem armazenados ou recuperados para consulta.
ModelMetricsSystem	Classe utilizada para realizar o transporte de informações sobre as métricas coletadas no sistema entre as classes da ferramenta.
ReadHistory	Classe responsável por recuperar dados de determinada coleta de métricas de um sistema.
SaveHistory	Classe responsável por armazenar dados de determinada coleta de métricas de um sistema.
SystemCollector	Classe responsável por realizar coleta de métricas no nível do sistema.

Tabela 7.17: Objetivos das classes de Connecta

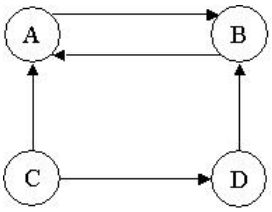


Figura 7.12: Exemplo de grafo

estrutura de dados correspondente. Na lista de adjacência de cada nodo, ou seja, de cada classe do sistema, são incluídas as suas conexões eferentes. Assim, no caso do nodo *B*, por exemplo, são incluídas as conexões que partem das classes *A* e *D* e chegam a *B*. Optou-se pelo registro das conexões aferentes de cada módulo e não das eferentes devido ao fato de que o cálculo da Métrica de Estabilidade de Myers é realizado a partir desta informação. Além disso, do ponto de vista de análise de estabilidade e conectividade, é importante conhecer as classes críticas, isto é, aquelas cujas alterações têm impacto em outras classes.

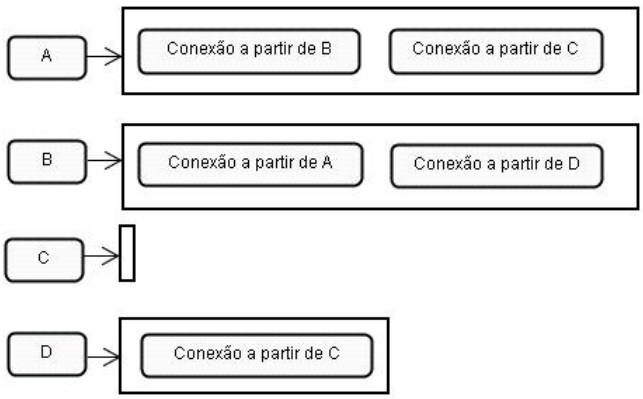


Figura 7.13: Esquema da estrutura de dados Grafo utilizada

7.4 Conclusão

Neste capítulo descrevemos *Connecta*, a ferramenta de coleta de métricas desenvolvida neste trabalho. *Connecta* é caracterizada por realizar coleta das métricas necessárias

a MACSOO, em códigos escritos em Java, apresentar relatórios dos resultados obtidos com a coleta, prover o armazenamento e a consulta de histórico de tais dados.

A ferramenta foi utilizada para a realização de experimentos que visam a validação de MACSOO. Tais experimentos são apresentados no Capítulo 8.

Capítulo 8

Experimentos e Resultados

Com o objetivo de validar as funcionalidades da ferramenta *Connecta* e avaliar a aplicação de MACSOO, realizamos um conjunto de experimentos que são relatados neste capítulo.

O estudo de caso apresentado na Seção 8.1 tem por objetivo ilustrar o comportamento de *Connecta*, prover uma leitura dos resultados reportados pela ferramenta e exemplificar a utilização de MACSOO. Neste caso, utilizamos uma aplicação constituída de poucas classes, para que se possa analisar os resultados em um nível de detalhes pequeno. Como metodologia, criamos, inicialmente, uma aplicação dotada de transgressões a princípios de modularidade de software, como a utilização de dados públicos. Gradualmente, alteramos a aplicação de forma a introduzir melhorias na sua estrutura. Para cada uma das versões da aplicação, coletamos suas métricas utilizando *Connecta*. A aplicação em questão é constituída por classes relacionadas ao Controle Acadêmico de uma escola.

Para prover uma análise de MACSOO, realizamos experimentos em sistemas maiores cujos resultados são relatados nas Seções 8.2, 8.3 e 8.4.

8.1 Caso 1: Controle Acadêmico

8.1.1 Primeira Versão do Controle Acadêmico

A primeira versão deste estudo de caso é constituída pelas classes: `Constantes`, que possui todas as constantes utilizadas na aplicação; `ProfessorAluno`, que contém dados e métodos aplicáveis a professores e alunos; `Disciplina` e `Turma`. Esta versão é caracterizada por possuir classes de baixa coesão e um número grande de conexões entre elas, com forte acoplamento, devido ao uso de atributos públicos. Para esta versão, *Connecta* gerou os resultados reportados nas Tabelas 8.1 e 8.2.

Métrica	***Valor***
Total de Classes	5
Quantidade de Conexões	6
COF	0,30
Estabilidade	2,1

Tabela 8.1: Resultados da primeira versão do Caso 1

A aplicação possui 5 classes e há 6 conexões no total entre elas, o que resulta em COF igual a 0,30. A métrica de estabilidade da aplicação, que corresponde ao *impacto de manutenção*, resultou em 2,1, o que indica que a cada necessidade de manutenção na aplicação, 2,1 classes sofrerão alterações. Considerando-se que o sistema possui apenas 5 classes, este número indica que 42% do sistema será alterado, o que é uma situação crítica. Desta forma, como orienta MACSOO, deve-se buscar formas de reduzir a conectividade por meio da intervenção nos fatores que a determinam.

Um problema estrutural grave desta versão é a utilização de dados públicos. Em todas as classes na aplicação, todos os atributos são públicos. Este fato, por si só, não determina alta conectividade. Entretanto, as classes utilizam os dados públicos umas das outras, o que origina o surgimento de conexões patológicas nas quais o grau de acoplamento entre as classes é altíssimo. Esta característica das classes da aplicação é refletida pelos valores obtidos para as métricas *conexões aferentes* e *impacto de manutenção*. A classe que possui o maior número de conexões aferentes é também aquela cuja alteração tem maior impacto: `Constantes`; para ela, obteve-se o valor 3,8 para a métrica *impacto de manutenção*. Este valor indica que para cada necessidade de manutenção nesta classe, no total, 3,8 classes serão afetadas, incluindo

Classe	***Métrica***	***Valor***
Classe Constantes	Quantidade de atributos públicos	12
	LCOM	1
	Conexões aferentes	3
	Impacto de manutenção	3,8
Classe ProfessorAluno	Quantidade de atributos públicos	6
	LCOM	3
	Conexões aferentes	1
	Impacto de manutenção	1,9
Classe Turma	Quantidade de atributos públicos	3
	LCOM	1
	Conexões aferentes	1
	Impacto de manutenção	1,9
Classe Disciplina	Quantidade de atributos públicos	3
	LCOM	0
	Conexões aferentes	1
	Impacto de manutenção	1,9

Tabela 8.2: Resultados da primeira versão do Caso 1 - classes

a própria classe. Novamente, considerando que o sistema possui possui 5 classes, este valor representa 76% do total de classes do sistema, o que é um valor alto.

Outra questão de grande impacto neste estudo de caso é a coesão interna das classes. Em uma análise qualitativa da estrutura desta versão, pode-se verificar que a classe `ProfessorAluno` agrega características de duas classes do domínio do problema, professor e aluno, o que caracteriza baixa coesão interna. Esta classe tem LCOM igual a 3. Isso significa que a diferença entre a quantidade de pares de métodos sem similaridade e a quantidade de pares com similaridade é igual a 3. Considerando-se que a classe em questão possui três métodos, este número é alto, o que reflete o fato de a classe possuir baixa coesão interna. Da mesma forma, observa-se que a classe `Constantes` apresenta problemas no aspecto de coesão. Embora a métrica LCOM tenha resultado igual a 1 para esta classe, o que poderia levar a se pensar que a classe tem um bom nível de coesão interna, a observação de sua estrutura, associada ao conhecimento do domínio do problema, revela que sua coesão é baixa, visto que trata-se de uma classe depositária de todas as constantes utilizadas na aplicação.

Considerando-se que a coesão interna das classes de um sistema impacta na sua conectividade, a classe `ProfessorAluno` e `Constantes` foram reestruturadas. Os

resultados das versões após a reestruturação são descritos nas seções seguintes.

8.1.2 Segunda Versão do Controle Acadêmico

Na segunda versão da aplicação, a classe `ProfessorAluno` foi reestruturada, sendo substituída por duas classes de coesão interna maior: `Professor` e `Aluno`. Para esta versão, *Connecta* gerou os resultados reportados nas Tabelas 8.3 e 8.4.

Métrica	***Valor***
Total de Classes	6
Quantidade de Conexões	8
COF	0,27
Estabilidade	2,2

Tabela 8.3: Resultados da segunda versão do Caso 1

Classe	***Métrica***	***Valor***
Classe Constantes	Quantidade de atributos públicos	12
	LCOM	1
	Conexões aferentes	4
	Impacto de manutenção	4,7
Classe Professor	Quantidade de atributos públicos	4
	LCOM	1
	Conexões aferentes	1
	Impacto de manutenção	1,9
Classe Aluno	Quantidade de atributos públicos	3
	LCOM	1
	Conexões aferentes	1
	Impacto de manutenção	1,9

Tabela 8.4: Resultados da segunda versão do Caso 1 - classe

A melhor coesão interna das classes `Professor` e `Aluno` refletiu-se no resultado obtido para a métrica LCOM, que é igual a 1 para estas classes. A reestruturação da classe `ProfessorAluno` diminuiu levemente o grau de conectividade da aplicação, que passou para 0,27. O impacto de manutenção das classes resultantes continuou com o mesmo valor da classe na versão anterior, o que se explica pelo fato de as conexões ainda ocorrerem por meio de atributos públicos.

As métricas coletadas indicam melhoria também na estabilidade do sistema. Nesta versão, para cada necessidade de manutenção do sistema, 2,2 classes serão afetadas, o que corresponde a 37% do total de classes do sistema.

8.1.3 Terceira Versão do Controle Acadêmico

A classe `Constantes` contém todas as constantes utilizadas na aplicação. Na terceira versão desta aplicação, essa classe foi reestruturada, sendo substituída por três classes de coesão interna maior: `ConstantesAluno`, `ConstantesProfessor` e `ConstantesDisciplina`. Para esta versão, *Connecta* gerou os resultados reportados nas Tabelas 8.5 e 8.6.

Métrica	***Valor***
Total de Classes	8
Quantidade de Conexões	9
COF	0,16
Estabilidade	2,1

Tabela 8.5: Resultados da terceira versão do Caso 1

Classe	***Métrica***	***Valor***
Classe ConstantesAluno	Quantidade de atributos públicos	6
	LCOM	1
	Conexões aferentes	2
	Impacto de manutenção	2,9
Classe ConstantesProfessor	Quantidade de atributos públicos	4
	LCOM	1
	Conexões aferentes	2
	Impacto de manutenção	2,9
Classe ConstantesDisciplina	Quantidade de atributos públicos	2
	LCOM	1
	Conexões aferentes	1
	Impacto de manutenção	2,7

Tabela 8.6: Resultados da terceira versão do Caso 1 - classes

A reestruturação da classe `Constantes` resultou na diminuição do grau de conectividade do sistema, que passou de 0,27 para 0,16. A métrica para estabilidade, nesta

versão, aponta que 2,1 classes demandarão alterações, no caso de uma necessidade de manutenção do sistema, o que corresponde a 26% do total de classes.

As intervenções realizadas na segunda e na terceira versão da aplicação mostram que classes coesas têm impacto positivo na conectividade do sistema e na sua estabilidade.

8.1.4 Quarta Versão do Controle Acadêmico

As versões anteriores desta aplicação fazem uso de atributos públicos nas classes. Nesta versão, eliminamos a utilização de dados públicos nas principais classes. A única exceção foram as classes depositárias de constantes, que permaneceram com atributos públicos. Para esta versão, *Connecta* gerou os resultados reportados nas Tabelas 8.7 e 8.8.

Métrica	***Valor***
Total de Classes	8
Quantidade de Conexões	7
COF	0,13
Estabilidade	1,5

Tabela 8.7: Resultados da quarta versão do Caso 1

Com a eliminação de dados públicos, o impacto de manutenção das classes `Aluno`, `Turma` e `Professor` diminiu. Entretanto, estas classes ainda têm impacto relevante no sistema, apontado pelo valor 1,4 para a métrica *impacto de manutenção*. Isto deve-se ao fato de que as conexões com tais classes não foram eliminadas; o que mudou foi a forma como as conexões ocorrem, pois, nesta versão, a comunicação ocorre por invocação de métodos. Foram eliminadas também as conexões que as classes `Professor`, `Aluno` e `Disciplina` mantinham com as classes `ConstantesProfessor`, `ConstantesAluno` e `ConstantesDisciplina`, respectivamente. A utilização destas classes passou a se concentrar na classe `Principal`.

Esta versão da aplicação apresenta grau de conectividade igual a 0,13 e a métrica de estabilidade aponta que 1,5 classes sofrerão impacto, no caso de uma manutenção no sistema, o que corresponde a 19% do total de classes.

Nível	***Métrica***	***Valor***
Classe ConstantesAluno	Quantidade de atributos públicos	6
	LCOM	1
	Conexões aferentes	1
	Impacto de manutenção	1,9
Classe ConstantesProfessor	Quantidade de atributos públicos	4
	LCOM	1
	Conexões aferentes	1
	Impacto de manutenção	1,9
Classe ConstantesDisciplina	Quantidade de atributos públicos	2
	LCOM	1
	Conexões aferentes	1
	Impacto de manutenção	1,9
Aluno	Quantidade de atributos públicos	0
	LCOM	45
	Conexões aferentes	1
	Impacto de manutenção	1,4
Turma	Quantidade de atributos públicos	0
	LCOM	21
	Conexões aferentes	1
	Impacto de manutenção	1,4
Disciplina	Quantidade de atributos públicos	0
	LCOM	21
	Conexões aferentes	1
	Impacto de manutenção	1,4
Professor	Quantidade de atributos públicos	0
	LCOM	55
	Conexões aferentes	1
	Impacto de manutenção	1,4

Tabela 8.8: Resultados da quarta versão do Caso 1 - classes

8.1.5 Conclusão

Comparando-se a versão da Seção 8.1.4 com a primeira versão, Seção 8.1.1, com apoio da avaliação qualitativa do software em questão, conclui-se que a intervenção nos aspectos de qualidade estrutural de software, como coesão, ocultação de informação e acoplamento, contribui para redução de conectividade em sistemas. Os dados coletados neste experimento refletem a relação entre a conetividade e a estabilidade de um software, estando de acordo como a idéia de que o grau de estabilidade de um sistema pode ser aferido pelo grau de sua conectividade.

8.2 Caso 2: Análise de um Conjunto de Aplicações

O objetivo deste estudo de caso é demonstrar empiricamente que a estabilidade de um software é afetada por sua conectividade. A metodologia utilizada neste caso foi a coleta e a análise das métricas de um conjunto de aplicações para o mesmo domínio de problema. Como objeto desta análise, foram utilizados os produtos dos trabalhos realizados por alunos de graduação de um curso de computação. O grupo de alunos é heterogêneo, contando com indivíduos com proficiência baixa, média e alta em programação orientada por objetos. Foi solicitado que eles desenvolvessem uma aplicação para a gestão de uma clínica médica, cujo objetivo é manter os cadastros de especialidades médicas, médicos, pacientes e prontuários de pacientes. Neste estudo de caso, foram coletadas métricas em 11 aplicações. Os resultados são mostrados na Tabela 8.9.

O gráfico da Figura 8.1, gerado a partir dos dados da Tabela 8.9, mostra a relação entre a conectividade e o impacto de manutenção no sistema. O gráfico evidencia que a dificuldade de manutenção de uma aplicação é diretamente proporcional à sua conectividade.

As aplicações implementadas pelos alunos foram avaliadas qualitativamente pela professora. O resultado desta avaliação é o seguinte:

- Aplicação 1: não são utilizados dados públicos. No geral as classes estão bem construídas, com responsabilidades específicas, exceto uma delas, que deveria implementar as características de prontuários, mas possui também dados da classe `Paciente`.

No.	*Classes*	*Conexões*	*Conectividade (COF) (%)*	*Estabilidade (número de classes)*	*Estabilidade (%)*
1	64	133	3	2,7	4,2
2	48	93	4	2,7	5,6
3	31	52	6	2,2	7,1
4	29	52	6	2,1	7,2
5	28	52	7	2,3	8,2
6	14	13	7	1,3	9,3
7	25	47	8	2,2	8,8
8	11	17	16	1,8	16,4
9	14	29	16	2,7	19,3
10	10	15	17	2,6	26,0
11	5	4	20	1,7	34,0

Tabela 8.9: Resultados do Caso 2

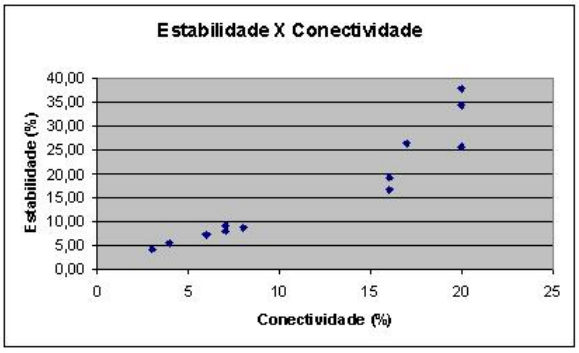


Figura 8.1: Gráfico Impacto de manutenção X Conectividade

- Aplicação 2: não são utilizados dados públicos. As classes de interface realizam algumas funções de leitura e gravação de dados em disco, resultando em baixa coesão.

- Aplicação 3: a aplicação está bem construída. A única ressalva é quanto a utilização de vetor estático para o armazenamento de objetos da classe `Paciente` dentro da própria classe `Paciente`. Visto que este não é um dado desta classe, esse não é um local apropriado para a sua utilização. O mesmo ocorre com as demais classes de negócio da aplicação.

- Aplicação 4: a estrutura desta aplicação é muito boa, pois não foram utilizados dados públicos, o relacionamento entre as classes estão apropriados e as classes possuem papéis bem definidos.

- Aplicação 5: é a melhor aplicação, do ponto de vista estrutural, dentre as avaliadas aqui. As classes foram separadas em pacotes lógicos, são altamente coesas, não são utilizados dados públicos. Foi construída uma classe utilitária para realizar funções como formatação de CEP, formatação de telefone, etc.

- Aplicação 6: embora as classes tenham sido divididas em pacotes lógicos, as classes de negócio realizam também funções de interface de usuário.

- Aplicação 7: nesta aplicação, foi utilizada conexão com banco de dados. Há problemas de coesão das classes. As funcionalidades de conexão com banco de dados e recuperação de dados estão espalhadas por todas as classes do sistema.

- Aplicação 8: nesta aplicação, foi utilizado o armazenamento e a recuperação de dados em arquivos. As funcionalidades de gravação de dados em arquivos estão localizadas nas classes de negócio. Foram construídas classes para realizar a recuperação de dados de arquivos, mas estas também realizam funções de interface de usuário. Nas classes de negócio também foram incluídas funções referentes a interação com o usuário.

- Aplicação 9: foi implementada uma única classes para realizar todo o tratamento de interface gráfica com o usuário, o que confere baixa coesão a esta classe. As demais classes estão bem construídas.

- Aplicação 10: não foram aplicados os conceitos de programação orientada por objetos. As classes de negócio possuem apenas dados público e um método construtor. As funcionalidades de interface de usuário foram distribuídas em várias classes.

- Aplicação 11: também neste caso, não foram aplicados os conceitos de programação orientada por objetos, pois as classes possuem apenas dados, todos eles públicos, e um método construtor. Foi construída uma classe principal que realiza todas as funcionalidades do sistema.

Como resultado desta avaliação qualitativa, a avaliadora atribuiu notas aos trabalhos, em uma escala de 0 a 10, apresentadas na Tabela 8.10.

Aplicação	***Nota***	***Conceito***
1	9	Ótimo
2	8	Bom
3	9	Ótimo
4	10	Excelente
5	10	Excelente
6	8	Bom
7	6	Razoável
8	6	Razoável
9	6	Razoável
10	3	Fraco
11	2	Fraco

Tabela 8.10: Avaliação qualitativa do Caso 2

Pela análise dos dados da Tabela 8.9, a aplicação 11 possui grau de conectividade igual a 20% e, para cada manutenção realizada no sistema, 34% de suas classes sofrerão impacto. De fato, avaliando-se a estrutura desta aplicação, observa-se que foram utilizados somente atributos públicos nas classes e as mesmas possuem apenas os métodos construtores. A classe principal realiza todas as demais funcionalidades do sistema. Já a aplicação 1, pelos dados coletados, apresenta bom nível de conectividade, o que leva a crer que seja uma aplicação construída segundo bons princípios de programação orientada por objetos. Isso é comprovado pela análise de sua estrutura. Na aplicação, não são utilizados dados públicos e as classes podem ser agrupadas em classes de negócio, de interface e de persistência, cada uma delas com responsabilidades bem definidas, com exceção de uma classe.

Comparando-se os resultados da avaliação qualitativa e os resultados gerados por *Connecta*, conclui-se que, embora não se possa afirmar que há uma concordância perfeita entre ambos, as aplicações consideradas como boas na avaliação qualitativa

são as que foram melhor avaliadas também quantitativamente, no aspecto de impacto de manutenção. Da mesma forma, aquelas aplicações consideradas razoáveis e fracas na avaliação qualitativa foram também apontadas como possuidoras de alto impacto de manutenção.

8.2.1 Conclusão

Os resultados obtidos neste estudo de caso são significativos na demonstração empírica dos impactos da conectividade de um sistema na dificuldade de sua manutenção. O gráfico gerado com os resultado obtidos mostra que os impactos de manutenção em um software cresce proporcionalmente ao aumento da conectividade.

A análise comparativa entre uma avaliação qualitativa das aplicações em questão e a avaliação quantitativa das mesmas, gerada por *Connecta*, revela um bom nível de concordância entre ambas. Entretanto, vale ressaltar a necessidade de uma quantidade maior de experimentos, a fim de se indentificar possíveis ajustes finos nos cálculos empregados em *Connecta*.

8.3 Caso 3: JFLAP

JFLAP (*Java Formal Language and Automata Package*) [28] é uma ferramenta visual auxiliar no estudo de Teoria de Linguagens. Entre outras funcionalidades, ela permite a criação e simulação de diversos tipos de autômatos. Foi realizado um experimento com um de seus pacotes, denominado *automata*. Este experimento visa mostrar a viabilidade de se aplicar a avaliação a parte de um sistema.

Os experimentos realizados com o pacote *automata* tiveram os resultados relatados na Tabela 8.11. O pacote tem um total de 64 classes e 154 conexões; a métrica COF é 0,04, que indica que as conexões entre as classes do pacote são de 4%; a estabilidade deste pacote é de 3,2, que indica que a cada necessidade de manutenção neste pacote, 3,2 de suas classes necessitarão de alterações. Este valor corresponde a 5% do total de classes do sistema. Desta forma, tendo por base os resultados dos estudos de casos anteriores, pode-se inferir que o pacote *automata* de JFLAP tem uma estrutura adequada.

Métrica	***Valor***
Total de Classes	64
Quantidade de Conexões	154
COF	0,04
Estabilidade	3,2

Tabela 8.11: Resultados do Caso 3

Os dados das classes mais conectadas do pacote *automata* são apresentados na Tabela 8.12. A classe *automata.Automaton* possui 29 conexões aferentes e seu impacto de manutenção é de 18,2 classes. Uma análise dos dados de tais conexões apontou que, entre elas, estão algumas em decorrência de herança e invocação de métodos com efeitos de acoplamento por referência. A classe *automata.Transition* possui 12 conexões aferentes e impacto de manutenção em 16,1 classes. A classe *automata.State* possui 11 conexões aferentes e impacto de manutenção igual a 12,4 classes. Esta classe também apresenta métodos com efeitos de acoplamento por referência.

Classe	***Métrica***	***Valor***
automata.Automaton	Quantidade de métodos públicos	25
	Quantidade de atributos públicos	0
	DIT	1
	LCOM	262
	Conexões aferentes	29
	Impacto de manutenção	18,2
automata.Transition	Quantidade de métodos públicos	16
	Quantidade de atributos públicos	0
	DIT	1
	LCOM	16
	Conexões aferentes	12
	Impacto de manutenção	16,1
automata.State	Quantidade de métodos públicos	11
	Quantidade de atributos públicos	0
	DIT	1
	LCOM	62
	Conexões aferentes	11
	Impacto de manutenção	12,4

Tabela 8.12: Resultados do Caso 3 - classes de maior conectividade aferente

8.4 Caso 4: Arcademis

Arcademis [5] é um arcabouço implementado em Java para o desenvolvimento de *middleware* orientado por objetos. Um *middleware* é uma camada de software cujo objetivo é realizar a interface entre o sistema operacional e aplicações distribuídas. *Arcademis* é caracterizado por possuir um conjunto de classes abstratas e interfaces.

Os resultados dos experimentos realizados com as classes de *Arcademis* são mostrados na Tabela 8.13. A Tabela 8.14 lista os dados das classes com maiores conectividades aferentes, que correspondem às classes cujas alterações têm grande impacto no sistema.

Esse arcabouço possui um total de 74 classes, com 2% de conectividade entre elas; a cada necessidade de manutenção, em média 2,0 classes necessitarão de manutenção. Este valor corresponde a 2,7% do total de classes do sistema, o que pode ser considerado um valor baixo. Com isso, pode-se concluir que *Arcademis* foi construído de forma a privilegiar a facilidade de manutenção.

Métrica	***Valor***
Total de Classes	74
Quantidade de Conexões	86
COF	0,02
Estabilidade	2,0

Tabela 8.13: Resultados do Caso 4

A classe com maior grau de conectividade aferente apresenta 22 conexões deste tipo; uma necessidade de manutenção nesta classe impacta em 24,9, incluindo a própria classe. O seu alto grau de conectividade aferente é explicado por esta classe ser uma interface implementada por 22 classes no sistema. A segunda classe com maior grau de conectividade aferente possui 11 conexões deste tipo e seu impacto de manutenção é de 11,6 classes. Também, neste caso, o alto grau de conectividade aferente ocorre por esta classe ser uma interface.

Classe	*Métrica*	*Valor*
arcademis.Marshalable	Quantidade de métodos públicos	2
	Quantidade de atributos públicos	0
	DIT	1
	LCOM	0
	Conexões aferentes	22
	Impacto de manutenção	24,9
arcademis.EventHandler	Quantidade de métodos públicos	1
	Quantidade de atributos públicos	0
	DIT	1
	LCOM	0
	Conexões aferentes	11
	Impacto de manutenção	11,6

Tabela 8.14: Resultados do Caso 4 - classes de maior conectividade aferente

8.5 Conclusão

MACSOO é um modelo que faz uso de métricas para orientar à redução da conectividade entre as classe do software. O uso de *Connecta* viabiliza a implementação de MACSOO, pois foi construída de forma a seguir os passos essenciais previstos no modelo: avaliação da estabilidade do software e de sua conectividade, fator determinante do primeiro; diante de um valor insatisfatório para esta métrica, detalha-se o nível de análise, identificando-se pontos como: classes cujas alterações têm grande impacto no software, ocultação de informação das classes, indicador de grau de coesão interna das classes, grau de acoplamento de conexões, etc.

Relatamos e analisamos, neste capítulo, os resultados de quatro experimentos realizados com MACSOO. O primeiro experimento ilustra que a aplicação de MACSOO, apoiada pela utilização de *Connecta*, leva à diminuição de conectividade em um sistema. Os resultados dos experimentos, sobretudo dos dois primeiros, confirmam que a dificuldade da manutenção de um sistema é aferida pelo seu grau de conectividade.

Na Seção 8.2 apresentamos o relato de um experimento no qual comparamos a avaliação quantitativa de um conjunto de aplicações, gerada por *Connecta*, com uma avaliação qualitativa das mesmas. Os resultados desta comparação nesse experimento sugere que a faixa de valores apresentada na Tabela 8.15 pode ser utilizada na avaliação de conectividade em sistemas orientados por objetos. Contudo, salientamos

a necessidade de realização de experimentos em larga escala, envolvendo a coleta de métricas e o acompanhamento de um número considerável de aplicações. Experimentos deste porte poderiam contribuir para a indicação de valores, com maior precisão, a serem considerados satisfatórios ou insatisfatórios para o aspecto conectividade e os demais abordados em MACSOO.

Faixa de valor (%)	***Conceito***
0 a 7	Satisfatório
7 a 15	Razoável
acima de 15	Insatisfatório

Tabela 8.15: Valores sugeridos para a avaliação de conectividade

Capítulo 9

Conclusão

A atividade de manutenção de software é ponto crítico na Engenharia de Software, pois tem impacto em questões chave desta disciplina, como custo e prazo. Essa atividade comumente é difícil de ser realizada e é responsável pela maior parcela do custo total de um sistema. Desta forma, é imperativo investir em recursos que proporcionem alcançar softwares com maior nível de manuntenibilidade.

Obter software de fácil manutenção depende de investimento em qualidade da sua produção, sobretudo no que diz respeito à sua estrutura. Um software com bom nível de manutenibilidade é aquele que diante da necessidade de alteração, tem poucos módulos alterados. Esta característica, denominada *estabilidade*, depende da forma como o software é estruturado. Defendemos a tese de que a *conectividade* é o fator preponderante na determinação da estabilidade de um software e, conseqüentemente, de sua manutenibilidade. Os experimentos realizados neste trabalho confirmam esta tese, uma vez que apontam que a conectividade pode ser utilizada como indicador da dificuldade de manutenção de um sistema. A relação observada entre estes dois fatores indica que o impacto de manutenção é diretamente proporcional ao grau de conectividade.

Buscando contribuir com a produção de software de manutenção mais fácil, foi proposto neste trabalho o MACSOO - Modelo de Avaliação de Conectividade em Sistemas Orientados por Objetos. Este modelo é uma heurística que visa a diminuição da conectividade em sistemas orientados por objetos. Para isso ele conta com um conjunto de métricas de software OO selecionadas dentre aquelas proposta na literatura. Em especial, conta com uma Métrica de Estabilidade, proposta inicialmente

por Myers[38], que foi revisitada nesta dissertação para adequar-se à OO.

Com o objetivo de viabilizar a implementação de MACSOO, desenvolvemos neste trabalho a ferramenta *Connecta*, que tem por funções básicas a coleta de métricas em software desenvolvido em Java, bem como a geração e consulta de histórico de coletas realizadas.

As contribuições deste trabalho são:

1. Avaliação da relação entre conectividade e estabilidade de sistemas.
2. Adaptação dos conceitos de acoplamento e coesão à luz da orientação por objetos.
3. Identificação das métricas de software orientado por objetos impactantes no aspecto conectividade de sistema.
4. Proposta de duas métricas auxiliares na avaliação da conectividade: *conexões aferentes* e *peso de conexão aferente*.
5. Proposta de MACSOO, o Modelo de Conectividade em Sistemas Orientados por Objetos.
6. A construção de uma ferramenta de coleta de métricas de software orientado por objetos baseada em MACSOO, destinada à avaliação de softwares desenvolvidos na linguagem Java.

9.1 Trabalhos Futuros

- **Estudos experimentais em larga escala:** o acompanhamento de sistemas de grande porte orientados por objetos a fim de obter dados estatísticos que possam fornecer a indicação de valores adequados para as métricas utilizadas em MACSOO.
- **Melhorias da ferramenta Connecta:** por exemplo, a inclusão de uma funcionalidade que permita a visualização gráfica do sistema de forma a facilitar a identificação das conexões de maior impacto na estabilidade do mesmo. Outra melhoria importante é a integração de *Connecta* a uma ferramenta IDE.

- **Desenvolvimento de ferramentas similares a Connecta para outras linguagens:** considerando que *Connecta* foi construída para a coleta de métricas em softwares implementados em Java.
- **Avaliação de Conectividade em Sistemas Orientados por Aspectos:** a realização de um estudo similar ao realizado neste trabalho para o paradigma da Orientação por Aspectos.

Referências Bibliográficas

[1] ABREU, Fernando Brito e. *As Métricas na Gestão de Projetos de Desenvolvimento de Sistemas de Informação*. In: Actas das Sextas Jornadas para a Qualidade de Software, APQ, Lisboa,Dezembro de 1992.

[2] ABREU, Fernando Brito e CARAPUÇA, Rogério. *Object-Oriented Software Engineering: Measuring and Controlling the Development Process.*. In: Proceedings of 4th Int. Conf. of Software Quality, McLean, VA, USA, 3-5 October 1994.

[3] ABREU, Fernando Brito e; OCHOA, Luis; GOULO, Miguel. *The GOODLY Design Language for MOOD Metrics Collection*. Portugal: ISEG/INESC. ECOOP Workshops, 1997.

[4] AMBLER, Scott W., *Análise e Projeto Orientado a Objeto*. Volume 2, IBPI Press, Livraria e Editora Infobook AS, 1988.

[5] ARCADEMIS. Disponível em http://www2.dcc.ufmg.br/laboratorios/llp/ arcademis/. Último acesso em Maio de 2006.

[6] BCEL- Byte Code Engineering Library. Apache Software Foundation, 2003. Disponível em http://jakarta.apache.org/bcel/. Último acesso em Junho de 2005.

[7] BELLIN, David;Tyagi, Manish; Tyler, Maurice, *Object-Oriented Metrics: an Overview*. Proceedings of the 1994 Conference of the Centre for Advanced Studies on Collaborative Research, p.4, October 31-November 03, 1994, Toronto, Ontario, Canada.

[8] BEYER, D.; Lewerentz, C.; Simon, F. *Impact of Inheritance on Metrics for Size, Coupling, and Cohesion in Object-Oriented Systems*. In: Dumke/Abran: New Approaches in Software Measurement, LNCS 2006, Springer Publ., 2001, pp. 1-17.

[9] BERARD, Edward V. *Metrics for Object-Oriented Software Engineering*. Disponível em [http://www.toa.com/pub/moose.htm], em Setembro de 2004.

[10] BOEHM, Barry W. *Software Engineering Economics*. Estados Unidos: Prentice-Hall, 1981.

[11] BOOCH, Grady. *Object Solutions - Managing the object-oriented project*. Estados Unidos: Addison-Wesley , 1996. ISBN: 0805305947.

[12] CHIDAMBER, Shyam. R. e Kemerer, Chris F. *Towards a metrics suite for object oriented design*. Conference on Object Oriented Programming Systems Languages and Applications, 1991, pp. 197-211.

[13] CHIDAMBER, Shyam R.; Kemerer, C.F. *A Metrics Suite for Object Oriented Design*. IEEE Transactions on Software Engineering, 20(1994)6, pp. 476-493

[14] CMM - *Capability Maturity Model*. Carnegie Mellon University, Software Engineering Institute. Disponível em http://www.sei.cmu.edu/cmm/. Último acesso em Maio de 2006.

[15] DALY, John; BROOKS, Andrew; MILLER, James; ROPER, Marc; WOOD, Murray. *An Empirical Study Evaluating Depth of Inheritance on the Maintainability of Object-Oriented Software*. ISERN, Scotland, 1996.

[16] DEITEL, H. M.; DEITEL, P. J. *Java - Como Programar*. 3. Ed. Porto Alegre: Bookman, 2001. 1201 p. ISBN 0-13-012507-5.

[17] ECLIPSE. Disponível em: http://www.eclipse.org/. Acesso em Janeiro de 2005.

[18] FENTON, Norman; NEIL, Martin. *Software Metrics: Roadmap*. In: Proceedings of the Conference on the Future of Software Engineering, Maio de 2000.

[19] FERREIRA, Aurélio Buarque de Holanda. *Miniaurélio Século XXI Escolar: O minidicionário de alíngua portuguesa*. Rio de Janeiro: Nova Fronteira, 2000.

[20] FRAKES, Willian; TERRY, Carol. *Software Reuse: Metrics and Models*. In: ACM Computing Surveys, Vol. 28, No. 2, Junho de 1996.

[21] FRANCA, Luiz Paulo Alves; Staa, Arndt von; Fonte II, Hamilton José Sales. *Um modelo de Classes para um Ambiente de Geração de Programas de Medição de Software Baseados na Web*. In: XIII SBES. Florianópolis, 1999.

[22] FRANCA, Luiz Paulo Alves; Staa, Arndt von; Lucena, Carlos José Pereira de. *Medição de Software para Pequenas Empresas: Uma Solução Baseada na Web*. In: Anais XII SBES. Rio de Janeiro: SBC, 1998. pp. 71-86.

[23] GAMMA, Erich; HELM, Richard; JOHNSON, Ralph; VLOSSIDES, John. *Padrões de Projeto - Soluções Reutilizáveis de Software Orientado a Objetos*. Porto Alegre: Bookman, 2000. 364 p. ISBN 0-201-63361-2.

[24] GHEZZI, Carlo e JAZAYERI, Medhi. *Programing Language Concepts. Estados Unidos*. Ed John Wiley & Sons, 1998. ISBN: 0-471-10426-4

[25] GILB, Tom. *Software Metrics*. Estados Unidos: Winthrop Publishers, 1977. 282 p. ISBN 0-87626-855-6.

[26] HASSOUN, Youssef; JOHNSON, Roger; COINSELL, Steve. *A Dynamic Runtime Coupling Metric for Meta-Level Architetures*. In: Proceedings of Eighth European Conference on Software Maintenance and Reengineering, 2004.

[27] JavaCount . Disponível em http://csdl.ics.hawaii.edu/Tools/JavaCount/ JavaCount.html. Último acesso em Maio de 2006.

[28] JFLAP. Disponível em http://www.jflap.org/. Último acesso em Março de 2006.

[29] JUDE - *UML Modelling Tool*. Disponível em http://jude.change-vision.com/jude-web/index.html. Acesso em Maio de 2006.

[30] Krakatau Essencial Metrics. Disponível em http://www.powersoftware.com/. Último acesso em Maio de 2006.

[31] LINDHOLM, Tim e Yellin, Frank. *The Java Virtual Machine Specification*. Second Edition. Disponível em http://java.sun.com/docs/books/vmspec/2nd-edition/html/VMSpecTOC.doc.html. Acesso em Janeiro de 2005.

[32] MORRIS, Kenneth L. *Metrics for Object-Oriented Software Development Environments*. Master's Thesis, M.I.T. Sloan School of Management, 1989.

[33] Metrics Tools. Disponível em http://www.laatuk.com/tools/metric-tools.html. Último acesso em Maio de 2006.

[34] MARTIN,Robert; *OO Design Metrics - An analysis of dependencies*. Outubro de 1994.

[35] MARTIN, Robert. *Design Principles and Design Patterns*. Disponível em: www.objectmentor.com. Acessso em Agosto de 2004.

[36] MEYER, Bertrand. *Object-oriented software construction*. 2. Ed. Estados Unidos: Prentice Hall International Series in Computer Science, 1997. 1254 p. ISBN 0-13-629155-4.

[37] MEYER, Bertrand. *The role of object oriented metrics*. Disponível em http://archive.eiffel.com/doc/manuals/technology/bmarticles/computer /metrics/page.html. Acesso em Agosto de 2004

[38] MYERS, Glenford J. *Reliable software through composite design*. Nova York: Petrocelli/Charter, 1975. 159 p. ISBN 0-88405-284-2.

[39] NETBEANS. Disponível em www.netbeans.org ou www.sun.com. Último acesso em Maio de 2006.

[40] ObjectDetail. Disponível em http://www.obsoft.com/Product/ObjDet.html e http://www.obsoft.com/Product/DetailPaper.html. Último acesso em Maio de 2006.

[41] PARNAS, D. L. *On the criteria to be used in decomposing systems into modules* Communications of ACM 15, 12, Dezembro, 1972. pp. 1053-1058.

[42] PAULA FILHO, Wilson de Pádua. *Engenharia de Software - Fundamentos, Métodos e Padrões*. Rio de Janeiro: LTC, 2001. 584 p. ISBN 85-216-1260-5.

[43] PFLEEGER, Shari Lawrence. *Software engineering theory and practice*. Upper Saddle River: Prentice-Hall, 1998. 576p. ISBN 013624842X

[44] PRESSMAN, Roger S. *Engenharia de Software*. Rio de Janeiro: MacGraw Hill, 2002. 843 p. ISBN 85-86804-25-8.

[45] PURAO, Sandeep; Vaishnavi, Vijay. *Produt Metrics for Object-Oriented Systems*. ACM Computing Surveys, Vol. 35, June 2003, pp. 191-221

[46] QUALITI METRICS. Disponível em: http://www.qualiti.com.br/home.html. Acesso em Janeiro de 2005

[47] QUASAR - QUantitative Approaches on Software Engineering And Reengineering. Disponpivel em http://www-ctp.di.fct.unl.pt/QUASAR/index.html. Acesso em Maio de 2006.

[48] ROCHE, J.M. *Software Metrics and Measurement Principles*. In: Software Engineering Notes, ACM, Vol. 19, No. 1, January 1994, pp. 76-85.

[49] RUMBAUGH, James; JACOBSON, Ivar; BOOCH, Grady. *The unified modeling language reference manual*. Reading, Mass.: Addison-Wesley, c1999. 550 p. ISBN 020130998X

[50] SANT'ANNA, Cláudio Nogueira; Garcia, Alessandro Fabrício; Chaves, Christina Von Flach Garcia; Lucena, Carlos José Pereira; Staa, Arndt Von Staa. *On the Reuse and Maintenance of Aspect-Oriented Software: an Assessment Framework*. In: 17º SBES. Anais. Porto Alegre: SBC, 2003.

[51] SAN Diego State University. *Advanced Object-Oriented Design and Programming*. Disponível em http://www.eli.sdsu.edu/courses/spring98/cs635/notes/. Acesso em Fevereiro de 2005.

[52] SHILDT, Robert. *C - Completo e Total*. São Paulo: Mc GraW Hill, 1990.

[53] SOMMERVILLE, Ian. *Engenharia de software*. 6. ed. São Paulo: Addison Wesley, 2003. 592p. ISBN 8588639076.

[54] STAA, Arndt von. *Programação Modular - Desenvolvendo programas complexos de forma organizada e segura*. Rio de Janeiro: Editora Campus, 2000. 690p. ISBN: 85-352-0608-6

[55] TEGARDEN, D.P.; Sheetz, S.D.; Monarchi, D.E. *A Software Complexity Model of Object-Oriented Systems*. In: Journal of Decision Support Systems, 1994. p.241-262.

[56] Together. Disponível em http://www.borland.com.br/together. Último acesso em Fevereiro de 2006.

[57] UNIVERSITY of Ottawa. *Object Oriented Software Engineering*. Disponível em http://www.site.uottawa.ca:4321/oose/index.html. Acesso em Fevereiro de 2005.

[58] VAREJÃO, Flávio Miguel. *Linguagens de Programação - Conceitos e Técnicas*. Rio de Janeiro: Elsevier, 2004.

[59] WEYUKER, E. *Evaluating software complexity measures*. IEEE Transactions Software Engineering, Vol. 14, pp. 1357-1365, 1988.

[60] XAVIER, Carlos Magno Da S.; PORTILHO, Carla. *Projetando com Qualidade a Tecnologia em Sistemas de Informação*. Rio de Janeiro: LTC, 1995. 117 p. ISBN:85-216-1047-5.

[61] XENOS, M.; Stavrinoudis, D.; Zikouli, K.; Christodoulakis, D. *Object-Oriented Metrics - A Survey*. Proceedings of the FESMA 2000, Madrid, Spain, 2000.

[62] ZIVIANI, Nivio. *Projeto e Análise de Algoritmos*. 2. ed. rev e ampl. São Paulo: Pioneira Thomson Learning, 2004.