UM MODELO DE PREDIÇÃO DE AMPLITUDE DA PROPAGAÇÃO DE MODIFICAÇÕES CONTRATUAIS EM SOFTWARE ORIENTADO POR OBJETOS

KECIA ALINE MARQUES FERREIRA

UM MODELO DE PREDIÇÃO DE AMPLITUDE DA PROPAGAÇÃO DE MODIFICAÇÕES CONTRATUAIS EM SOFTWARE ORIENTADO POR OBJETOS

Tese 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 Doutor em Ciência da Computação.

Orientadora: Mariza Andrade da Silva Bigonha
Co-orientador: Roberto da Silva Bigonha
Colaborador: Bernardo Nunes Borges de Lima

Belo Horizonte

Fevereiro de 2011

© 2011, Kecia Aline Marques Ferreira.
Todos os direitos reservados.

Ferreira, Kecia Aline Marques
F383m Um Modelo de Predição de Amplitude da Propagação de Modificações Contratuais em Software Orientado por Objetos / Kecia Aline Marques Ferreira. — Belo Horizonte, 2011
xxii, 202 f. : il. ; 29cm

Tese (doutorado) — Universidade Federal de Minas Gerais. Departamento de Ciência da Computação.
Orientadora: Mariza Andrade da Silva Bigonha
Co-orientador: Roberto da Silva Bigonha
Colaborador: Bernardo Nunes Borges de Lima

1. Computação - Teses. 2. Engenharia de Software - Teses. I. Orientador. II. Título.

CDU 519.6*32
(043)

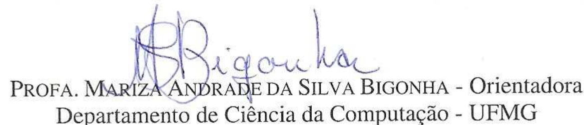
UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Um modelo de predição de amplitude da propagação de modificações contratuais em software orientado por objetos

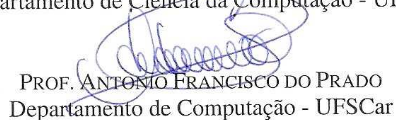
KÉCIA ALINE MARQUES FERREIRA

Tese defendida e aprovada pela banca examinadora constituída pelos Senhores:

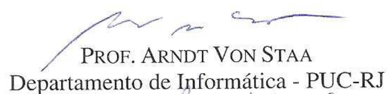


PROFA. MARIZA ANDRADE DA SILVA BIGONHA - Orientadora
Departamento de Ciência da Computação - UFMG

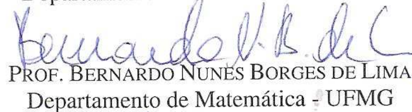
PROF. ROBERTO DA SILVA BIGONHA - Co-orientador
Departamento de Ciência da Computação - UFMG



PROF. ANTONIO FRANCISCO DO PRADO
Departamento de Computação - UFSCar



PROF. ARNDT VON STAA
Departamento de Informática - PUC-RJ



PROF. BERNARDO NUNES BORGES DE LIMA
Departamento de Matemática - UFMG

PROF. PAULO CÉSAR MASIERO
Departamento de Sistemas de Computação - ICMC

Belo Horizonte, 22 de fevereiro de 2011.

Aos meus avós.

Agradecimentos

A Deus, por todas as bênçãos que me concedeu.

Aos meus pais, pelo amor e pelo incentivo aos estudos.

Aos meus irmãos, pelo carinho, pela amizade e pelo apoio incondicionais.

Ao meu querido sobrinho Pedro Lucas, por tornar tudo muito mais alegre.

À Professora Mariza Bigonha e ao Prof. Roberto Bigonha, pela confiança e pela orientação cuidadosa e generosa.

Ao Prof. Bernardo de Lima, pela contribuição com seus preciosos conhecimentos de Matemática e pela cooperação na realização deste trabalho.

Aos alunos da Iniciação Científica, Bárbara Malta, Luiz Felipe Mendes e Heitor Corrêa, pela dedicação e seriedade com que realizaram seus trabalhos.

Ao Prof. Frederico Campos pelo incentivo a observar os números.

À equipe do Synergia, especialmente Vitor Alcântara, Daniela Cascini e Sophia, pela solicitude em cooperar com este trabalho.

A todos os meus professores. O papel de um professor na formação e na vida de um aluno é grandioso. A todos vocês que se dispuseram a doar conhecimento, tempo e paciência para que eu e meus colegas pudéssemos evoluir, meus sinceros agradecimentos. Agradeço especialmente ao Prof. José Wilson, com quem tive o privilégio de estudar no curso técnico e na graduação, que me motivou a avançar nos estudos e que me abriu as portas para a carreira de docente.

Ao Prof. Lúcio Mauro Pereira, pela confiança no meu trabalho docente.

Ao CEFET-MG, onde iniciei meus estudos em Computação e onde trabalho.

À PUC-MG, pela formação que recebi em Computação e pelos anos maravilhosos que passei lá, estudando e lecionando.

Ao Prof. Sílvio Alves, meu colega no CEFET-MG, pelo apoio em Estatística.

A todos os meus amigos, pela torcida.

Aos colegas do DCC-UFMG, especialmente César Couto e Evandrino Barros.

Aos meus alunos, pela parceria e pelos ótimos momentos em sala de aula.

À FAPEMIG e ao CNPQ, pelo suporte na realização deste trabalho.

Resumo

Este trabalho apresenta um novo modelo de avaliação e predição de manutenibilidade de software. O modelo, denominado K3B, realiza predição de amplitude da propagação de modificações contratuais em software orientado por objetos. Modificação contratual é aquela ocorrida em uma classe de forma a alterar o seu contrato, que corresponde aos serviços da classe, e às pré e pós-condições desses serviços. Pré-condição é uma condição que deve ser satisfeita para que o serviço seja realizado com sucesso. As pós-condições de um serviço correspondem ao estado do sistema após a realização do serviço. K3B fornece o número esperado de passos de modificações no software dado que um número inicial de módulos sofrerá modificações. O modelo é definido em termos do número total de módulos do software, do número de módulos que serão inicialmente modificados e de métricas do software, tais como coesão, acoplamento e conectividade.

Para dar suporte ao uso de K3B e à sua concepção, este trabalho também propõe uma métrica para avaliação de coesão de classes denominada Coesão de Responsabilidade, e apresenta os resultados de um estudo que identifica valores referência para um conjunto de métricas de software orientado por objetos, em particular aquelas que podem ser associadas à K3B. Além disso, relata os resultados de um estudo sobre os grafos de dependência entre módulos que revela importantes propriedades da natureza evolutiva de softwares e identifica a estrutura macroscópica desses grafos.

O modelo K3B foi avaliado empiricamente a partir de dados de um conjunto de softwares abertos. Os valores de K3B foram comparados aos obtidos por meio de simulação de propagação de modificações contratuais em software Java. A avaliação mostra que os valores gerados por K3B aproximam-se fortemente dos valores observados na simulação.

Palavras-chave: manutenibilidade de software, modelos de predição, orientação por objetos, métricas de software.

Abstract

This work presents a novel model to assess and predict software maintainability quantitatively. The model, named K3B, measures the propagation of contractual modifications in object-oriented software systems. We define contractual modification as a modification in a module of the software system which changes the contract of the module. The contract of a class corresponds to its public services, and the preconditions and the postconditions of these services. Precondiction is a condiction to be guaranteed in order to the service be executed successfully. A postcondiction corresponds to the expected status of the system after the service execution.

K3B predicts the number of modification steps which will be taken until the modification process ends. The model is defined in terms of the number of modules within the program, the number of modules which will be initially modified, and software metrics such as coupling, cohesion and connectivity. For the purposes of defining and using K3B, we also defined a new class cohesion metric called Cohesion of Responsibility, we carried out an experimental study to identify thresholds for the software metrics applied with K3B, and we performed a study whose results show the macroscopic structure of software systems and reveal important properties of the evolutive nature of software systems.

K3B was evaluated on a sample of open-source software systems. The evaluation of K3B was carried out by comparing the values generated by K3B and the values observed in simulation of modification in Java software. The results of this evaluation showed that the values reported by K3B are quite close to the data from simulation.

Keywords: software maintainability, predicting model, object-oriented software, software metrics.

Lista de Figuras

1.1 Curvas de falhas idealizada e real para software. Fonte: Pressman (2006) . 2

5.1 COF – (a) frequência e (b) ajuste à distribuição de Weibull. 71
5.2 Conexões aferentes – (a) frequência, (b) frequência em escala log-log, (c) ajuste à distribuição Weibull e (d) frequência detalhada. 72
5.3 LCOM – (a) frequência, (b) frequência em escala log-log, (c) ajuste à distribuição Weibull e (d) frequência detalhada. 73
5.4 DIT – (a) frequência, (b) frequência em escala log-log, (c) ajuste à distribuição Poisson e (d) frequência detalhada. 74
5.5 Atributos públicos – (a) frequência, (b) frequência em escala log-log, (c) ajuste à distribuição Weibull e (d) frequência detalhada. 75
5.6 Métodos públicos – (a) frequência, (b) frequência em escala log-log, (c) ajuste à distribuição Weibull e (d) frequência detalhada. 76

6.1 Frequência de valores de COR 104

7.1 O modelo *bow-tie* da Web 112
7.2 Hibernate (versão 3.5.1) modelado pelo *little house* 123
7.3 JUnit (versão 4.8.1) modelado pelo *little house* 123
7.4 Kolmafia (versão 13.7) modelado pelo *little house* 124
7.5 A camada de fronteira do software proprietário (versão 1.18) modelado pelo *little house* 124
7.6 A camada de modelo do software proprietário (versão 1.18) modelado pelo *little house* 125
7.7 *Little house* – A estrutura macroscópica genérica das redes de software . . 125

8.1 Exemplo de propagação de modificações contratuais. 131
8.2 Exemplo de propagação de modificações contratuais - continuação 132

8.3 Passeio Aleatório do processo de modificação de software: i é o número de módulos em modificação em determinado instante 136
8.4 Matriz de probabilidades 139
8.5 Forma da matriz de probabilidades 140
8.6 Padrão de formação dos polinômios obtidos 144

9.1 Tela Principal – ferramenta de simulação 157
9.2 Tela Seleção de Arquivos para Análise – ferramenta de simulação 158
9.3 Tela Simulação de Modificações em Métodos – ferramenta de simulação . . 158
9.4 Tela Simulação de Modificações em Métodos - Resultado – ferramenta de simulação 159
9.5 Tela Caminho de Modificações – ferramenta de simulação 160
9.6 Correlação entre dados 164
9.7 Correlação entre dados da simulação e K3B, para $i = 3$ e $i = 5$ - JUnit 3.4 166

Lista de Tabelas

3.1 Pesos de Coesão e Acoplamento no Modelo de Myers 29
3.2 Pesos de Acoplamento e Coesão na OO 30

5.1 Softwares e seus domínios de aplicação, tipos, tamanhos, número de conexões entre as classes e a métrica COF. 65
5.2 Softwares e seus domínios de aplicação, tipos, tamanhos, número de conexões entre as classes e a métrica COF. 68
5.3 Valores Referência Gerais para Métricas de Software OO 76
5.4 Valores Referência para Métricas de Software OO por Tipo 79
5.5 Valores Referência para Métricas de Software OO por Domínio de Aplicação 79
5.6 Valores Referência para Métricas de Software OO por Tamanho 80
5.7 Classes do conjunto de softwares e valores de suas métricas #CA (conexões aferentes), LCOM, DIT, #AP (atributos públicos) e #MP (métodos públicos) 82
5.8 Classes de JHotDraw e valores de suas métricas #CA (conexões aferentes), LCOM, DIT, #AP (atributos públicos) e #MP (métodos públicos) 89

6.1 Avaliação da métrica COR 103

7.1 Softwares analisados no estudo sobre evolução de software 115
7.2 Dados da evolução dos softwares abertos 116
7.3 Dados da evolução dos softwares abertos - continuação 117
7.4 Dados sobre a evolução do software proprietário 118
7.5 Evolução do grau de entrada - JEdit 2.4 and 4.3.18 119
7.6 Evolução do grau de entrada - Kolmafia 2.0 and 13.7 119
7.7 Evolução do grau de entrada - JUnit 4.0 and 4.8.1 120
7.8 Evolução do grau de entrada - Software proprietário 1.0 and 1.18 120
7.9 Instabilidade da classe net.sourceforge.kolmafia.KoLmafia 121
7.10 Instabilidade das classes com alto grau de entrada do software proprietário 121
7.11 Evolução dos tamanhos dos componentes LSCC 127

8.1 Último termo do polinômio $E(t_i)$ 142

9.1 Comparação entre K3B e dados de simulação - Valores para $1 \leq i \leq 3$ . . . 170
9.2 Comparação entre K3B e dados de simulação - Valores para $1 \leq i \leq 3$ . . . 171
9.3 Comparação entre K3B e dados de simulação para $i = 4$ e $i = 5$ 172

Sumário

Agradecimentos **ix**

Resumo **xi**

Abstract **xiii**

Lista de Figuras **xv**

Lista de Tabelas **xvii**

1 Introdução **1**

1.1 Objetivos 5
1.2 Contribuições 6
1.3 Metodologia 7
1.4 Organização do Trabalho 7

2 Métricas de Software **9**

2.1 Métricas e Qualidade de Software 10
2.2 Princípios de Medição 13
2.3 Categorias de Métricas de Software 14
2.3.1 Métricas de Qualidade de Produto 15
2.3.2 Métricas de Qualidade de Processo 16
2.3.3 Métricas para Custo de Manutenção de Software 16
2.4 Conclusão 17

3 Métricas de Software Orientado por Objetos **19**

3.1 Métricas CK 19
3.2 Métricas MOOD 22
3.3 Análise das Métricas CK e MOOD 27
3.4 Métrica de Estabilidade 28

3.4.1 Métrica de Instabilidade . . . 28
3.4.2 Métrica de Estabilidade . . . 29
3.4.3 Avaliação das Métricas de Estabilidade . . . 30
3.5 Ferramentas de Coleta de Métricas . . . 31
3.6 Conclusão . . . 33

4 Pesquisas em Medição de Software 35
4.1 Medição de Coesão . . . 35
4.2 Medição de Software Orientado por Aspectos . . . 38
4.3 Reestruturação de Software . . . 40
4.4 Predição de Falhas de Sistemas . . . 42
4.5 Manutenibilidade de Software . . . 44
4.5.1 Modelos de Avaliação de Manutenibilidade . . . 45
4.5.2 Modelos Baseados em Métrica . . . 46
4.5.3 Modelos Baseados em Características de Manutenibilidade . . . 52
4.5.4 Propagação de Modificações . . . 53
4.5.5 Avaliação do Estado da Arte . . . 56
4.6 Conclusão . . . 57

5 Valores Referência para Métricas de Software Orientado por Objetos 61
5.1 Trabalhos Relacionados . . . 62
5.2 Metodologia . . . 65
5.2.1 Métricas de Software Avaliadas . . . 65
5.2.2 Dados Analisados . . . 68
5.2.3 Ajuste dos Dados . . . 69
5.2.4 Análise dos Dados . . . 70
5.3 Resultados . . . 71
5.3.1 Ajuste dos Dados de Todos os Software do Estudo . . . 71
5.4 Valores Referência para as Métricas de Software OO . . . 76
5.4.1 Valores Referência para Métricas de Software OO por Tipo . . . 77
5.4.2 Valores Referência para as Métricas de Software OO por Domínio de Aplicação . . . 78
5.4.3 Valores Referência para Métricas de Software OO por Tamanho 79
5.5 Avaliação dos Valores Referência . . . 80
5.5.1 Estudo de Caso 1 . . . 81
5.5.2 Estudo de Caso 2 . . . 88
5.5.3 Discussão . . . 90

5.6 Limitações . . . 91
5.7 Conclusões . . . 92

6 Métrica de Coesão de Responsabilidade 95
6.1 Trabalhos Relacionados . . . 96
6.2 Definição da Métrica de Coesão de Responsabilidade . . . 97
6.2.1 Comparação de COR com Outras Abordagens . . . 99
6.3 Avaliação da Métrica COR . . . 100
6.4 Valores Referência da Métrica COR . . . 104
6.5 Conclusão . . . 105

7 Evolução de Software - Uma Abordagem de Redes Complexas 107
7.1 Referencial Teórico . . . 109
7.1.1 Métricas de Software . . . 109
7.1.2 Métricas de Rede . . . 110
7.1.3 Redes Complexas . . . 110
7.1.4 O Modelo *Bow-tie* . . . 111
7.2 Trabalhos Relacionados . . . 112
7.3 Metodologia . . . 114
7.4 Experimentos e Resultados . . . 117
7.4.1 Crescimento dos Softwares . . . 118
7.4.2 Densidade da Rede de Software . . . 118
7.4.3 Grau de Entrada . . . 118
7.4.4 Evolução de Classes com Alto Grau de Entrada . . . 121
7.4.5 Diâmetro . . . 122
7.4.6 *Little house* – A Estrutura Macroscópica Genérica das Redes de Software . . . 124
7.5 Conclusões . . . 127

8 K3B - Um Modelo de Predição de Amplitude da Propagação de Modificações Contratuais em Software Orientado por Objetos 129
8.1 Enunciado do Problema . . . 130
8.2 Modelo de Myers - Software Visto como um Circuito de Lâmpadas . . . 133
8.3 Metodologia . . . 134
8.4 Definição do Modelo K3B . . . 135
8.4.1 Resultados . . . 140
8.5 Aproximação do Modelo Proposto a Software Real . . . 142
8.6 K3B . . . 143

8.6.1 Implementação . . . 145
8.6.2 A Escolha de Métricas para α e β . . . 147
8.6.3 A Relação entre α e β . . . 149
8.6.4 Implicações de K3B . . . 150
8.6.5 Aplicações de K3B . . . 150
8.6.6 Limitações de K3B . . . 151
8.6.7 Conclusões . . . 151

9 Avaliação do Modelo K3B 153
9.1 Metodologia . . . 153
9.2 Tipo de Modificação Avaliada . . . 155
9.3 Ferramenta de Simulação de Propagação de Modificações Contratuais . 157
9.3.1 Algoritmo de Simulação . . . 161
9.4 Coeficiente de Correlação . . . 162
9.5 Resultados . . . 165
9.6 Limitações da Avaliação . . . 167
9.7 Conclusões . . . 168

10 Conclusão 173

Referências Bibliográficas 179

A Polinômios para Predição de Propagação de Modificações Contratuais em Software 189
A.1 Resultados . . . 189
A.2 Resultados Obtidos Considerando-se a Conectividade . . . 201

Capítulo 1

Introdução

Software tornou-se um produto de grande importância social, cada vez mais complexo e sofisticado. Apesar da notável evolução tecnológica, desenvolver software de alta qualidade, com custo e prazo aceitáveis não é uma tarefa simples. Traduzindo a dimensão dos problemas envolvidos nessa atividade, Pressman [2006] os caracteriza como uma *aflição crônica*[1] vivenciada ainda hoje.

O produto *software* é particularmente complexo. O seu desenvolvimento envolve uma gama de tecnologias, conceitos, métodos, técnicas, linguagens de programação, ferramentas e outros recursos disponíveis, fatores ambientais, como hardware e outros softwares, fatores econômicos, humanos, gerenciais etc. Trata-se de um produto de profundo impacto nas atividades humanas atuais, que não se desgasta como os demais, porém deteriora-se. O processo de deterioração de software, ilustrado pela Figura 1.1, é decorrente das manutenções realizadas ao longo de sua vida. Cada manutenção realizada pode gerar efeitos colaterais ou necessidades de novas manutenções, aumentando a taxa de falhas do software. Antes de esse processo se estabilizar, o que significa que a taxa de falhas estabilizou-se, outra manutenção pode ser realizada, iniciando um novo aumento na taxa de falhas. Com isso, a taxa de falhas mínima do software tende a aumentar, o que caracteriza a sua deterioração. A complexidade da produção de software não se restringe ao seu processo de criação e à sua disponibilização para os usuário. Um grande desafio é amenizar sua deterioração.

Mais de 70% do custo total de um sistema corresponde à atividade de manutenção [Meyer, 1997; Sommerville, 2003] apud Lientz e Swanson (1980)[2] e Nosek e Palvia

[1] Pressman destaca o significado desta expressão: algo que causa dor ou sofrimento e que tem longa duração ou volta frequentemente.

[2] Lientz, B. P e Swanson, E. B. *Software maintenance management.* Reading, MA: Addison Wesley. 1980

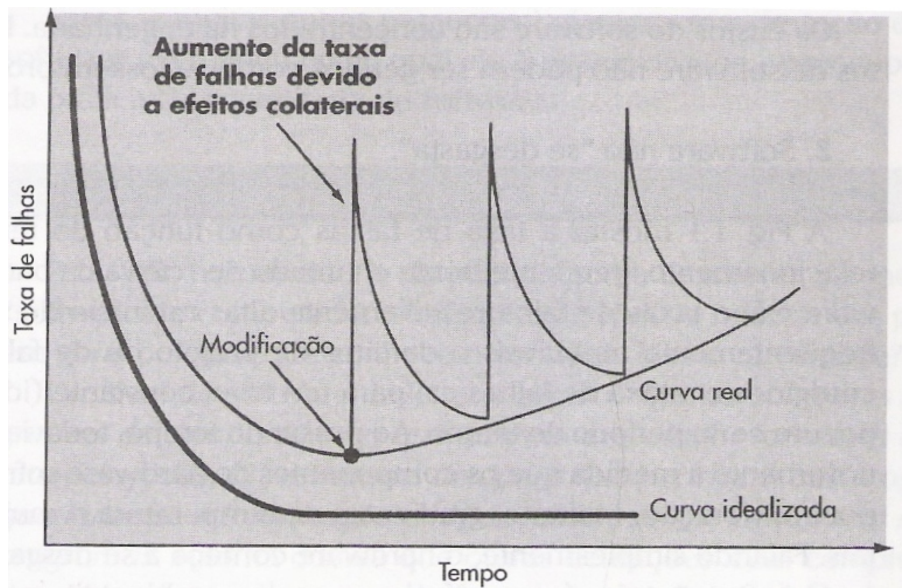


Figura 1.1. Curvas de falhas idealizada e real para software. Fonte: Pressman (2006)

(1990)[3]. Desse custo, 17% corresponde a reparo de defeitos, 18% a adaptação do software a um novo ambiente e 65% a inclusão ou modificação de requisitos. Diante disso e do processo de deterioração potencial do software, é imperativo a construção de software com alto grau de *manutenibilidade*, definida como a facilidade de realizar manutenções corretas e completas em um software. Manutenções em software podem ser: corretivas, aquelas realizadas para corrigir defeitos no software; adaptativas, aquelas realizadas para adaptar o software a mudanças no seu ambiente, por exemplo, a suas regras de negócios; perfectivas, com o objetivo de aperfeiçoar o software; e preventivas ou de reengenharia, com o objetivo de facilitar outras manutenções no software [Pressman, 2006; Filho, 2009].

Vários fatores influenciam a manutenibilidade de um software, por exemplo a complexidade do domínio do problema, a rotatividade de pessoal, a qualidade da documentação, a idade e a estrutura do software [Pfleeger, 1998; Sommerville, 2003]. A complexidade do domínio do problema é inerente à aplicação, e a rotatividade de pessoal pode ser um problema comum nas equipes de desenvolvimento de software. Investir na qualidade de documentação e na qualidade estrutural do software são formas de se contornar ou amenizar as dificuldades de manutenção de software, inclusive no caso de dificuldades trazidas por rotatividade de pessoal e complexidade da aplicação. Sommerville [2003] avalia que a atividade de manutenção de software é geral-

[3]Nosek, J. T. e Palvia, P. (1990). *Software Maintenance management: changes in the last decade.* Software Maintenance: Reasearch and Practice, Vol. 2, n. 3, p. 157-174. 1990.

mente subestimada entre os engenheiros de software, que imaginam que esta atividade demanda menos habilidade e experiência do que a fase de desenvolvimento. O *Software Engineering Institute* (SEI), em *Software Technology Roadmap* [SEI, 2009], aponta como essencial que durante o desenvolvimento do software ocorra o planejamento e o preparo para a sua manutenção. O SEI identifica os seguintes pontos principais a serem considerados neste aspecto: a rastreabilidade de requisitos em código permite reduzir o tempo para se identificar o código que sofrerá manutenção; a documentação do software deve conter informações consistentes com o código para prover informações de suporte à manutenção; a possibilidade de medir diretamente a complexidade e a manutenibilidade de software permite a predição de custos e riscos futuros.

A medição, a avaliação e o controle da manutenibilidade do software são importantes durante as fases de desenvolvimento e operação para que problemas futuros sejam evitados. Muitos trabalhos, de diferentes abordagens, têm essa questão como objeto de estudo [Subramanyam & Krishnan, 2003; Hordijk & Wieringa, 2005; Tsantalis et al., 2005; Chhabra & Aggarwal, 2006; Ferreira, 2006; Gyimothy et al., 2005; Nagappan et al., 2006; Schröter et al., 2006; Zhou & Leung, 2006; Deissenboeck et al., 2007; Heitlager et al., 2007; Li et al., 2010]. Um dos trabalhos mais conhecido sobre medição de manutenibilidade é a métrica *Índice de Manutenibilidade* (MI) [Ash et al., 1994; Pearse & Oman, 1995]. Esta métrica provê um indicador do grau de manutenibilidade a partir de coleta de medidas de complexidade do código fonte do software. Foi proposta e validada com softwares estruturados, adotada pelo SEI e utilizada em organizações como Hewllet-Packard e Departamento de Defesa Americano (DoD) [SEI, 2009]. Porém, o uso dessa métrica tem sido criticado [Heitlager et al., 2007; Sarwar et al., 2008]. Uma observação que se pode fazer sobre MI é que o seu cálculo baseia-se em métricas como número de linhas código, número de linhas de comentários em código e complexidade. Essas métricas não capturam características de softwares orientados por objeto, para o qual há métricas específicas [Abreu & Carapuça, 1994; Chidamber & Kemerer, 1994]. Heitlager et al. [2007] afirmam que, com base em suas experiências com o uso de MI em um grande número de softwares de diversos propósitos e tecnologias, eles observaram fortes limitações do uso desta métrica, dentre as quais destaca-se a dificuldade de se identificar o fator que causa determinado valor de MI, o que torna o seu uso difícil tanto em nível gerencial como técnico.

Apesar do grande investimento em estudos para identificar uma maneira de medir a manutenibilidade de software, ainda não há uma solução satisfatória para esse problema. Uma das causas que contribuem para isso é que a manutenibilidade é uma característica multidimensional, dificilmente capturada por uma única métrica. De

acordo com a ISO 9126, essa característica envolve as seguintes subcaracterísticas: facilidade de identificação das partes do software que devem ser modificadas e de realização de tais modificações, a capacidade do software evitar efeitos colaterais dessas modificações e a facilidade de testar o software modificado. Outra causa que pode ser observada é a dificuldade de validação das soluções propostas, devido à carência de dados experimentais de softwares reais.

Nesse contexto, o objetivo principal deste trabalho é a definição de um modelo de predição de amplitude da propagação de modificações contratuais em software orientado por objetos. Define-se aqui modificação contratual como aquela ocorrida dentro de uma classe e que altere o seu contrato. O contrato [Meyer, 1997] de uma classe corresponde aos seus serviços públicos, as pré e pós condições desses serviços. O conceito de contrado baseia-se na ideia do relacionamento cliente-servidor entre classes: um classe servidora compromete-se a prestar corretamente um serviço às suas classes usuárias desde que estas cumpram determinadas pré-condições. As usuárias de determinada classe são as classes que dependem dela. As pré-condições de um serviço são as condições que devem ser satisfeitas para que o serviço seja realizado com sucesso. As pós-condições de um serviço correspondem ao estado do sistema após a execução do serviço. Se uma modificação não altera uma pré-condição e não altera uma pós-condição de uma classe, as classes usuárias dessa classe não deverão sofrer impacto. Contudo, se as pré ou as pós-condições sofrem alterações em decorrência da modificação realizada, as classes usuárias do serviço podem ser afetadas, podendo também necessitar ser modificadas para adaptarem-se, o que também pode gerar necessidades de modificações em suas respectivas classes usuárias.

O propósito desse modelo é obter-se, a partir da estrutura do software, por exemplo por meio de seu código, um indicador da sua manutenibilidade, de tal forma que com esse indicador, tanto gerentes quanto técnicos possam atuar no software de maneira a tornar a sua manutenção mais fácil. O modelo proposto nesta tese, denominado K3B, estima o número médio de passos de modificações, *E(n,i)*, para que um software com n módulos alcance *estabilidade* quando for necessário modificar i de seus módulos. No modelo, considera-se que durante o processo de modificação de um software, um módulo pode assumir dois estados distintos: *em modificação*, que corresponde à situação em que uma dada modificação está sendo realizada no módulo, ou *atualizado*, que indica que o módulo não está sofrendo modificação alguma no momento. Define-se um *passo de modificação* como uma transição entre esses estados em um dos módulos do software. Um passo de modificação ocorre, então, quando uma modificação é iniciada ou concluída em um módulo do software. Desta forma, um passo de modificação corresponde ao evento que aumenta ou diminui, em uma unidade, o número de módulos

em modificação durante um processo de modificação. A *estabilidade* ocorre quando, a partir de um conjunto de modificações iniciadas em i módulos, todos os módulos do sistema estão no estado *atualizado*. Neste modelo, o grau de interconexão entre os módulos do software, a sua *conectividade*, é considerada como um dos principais fatores, pois é determinante na propagação das modificações. O modelo proposto é genérico, podendo ser aplicado em diferentes paradigmas. Neste trabalho é enfatizada a orientação por objetos, por ser um dos paradigmas mais amplamente difundidos e utilizados atualmente.

Para dar suporte à definição e ao uso do modelo, nesta tese foram conduzidos dois estudos experimentais: um com o objetivo de identificar propriedades das estruturas reais de software e como elas evoluem, e outro com o objetivo de identificar valores referência para um conjunto de métricas de software relacionadas a aspectos que têm impacto em manutenibilidade de software. É definida também uma métrica para avaliação de coesão interna de classes, denominada Coesão de Responsabilidade, que pode ser usada em associação com K3B.

A avaliação do modelo K3B mostra que os valores que ele gera como resultado são muito próximos daqueles observados em simulações de modificações em software orientado por objetos. O modelo pode ser utilizado para avaliar ou comparar softwares sob o ponto de vista de manutenibilidade, bem como auxiliar gerentes e desenvolvedores na alocação de recursos às atividades de modificações. Se aplicado em fases iniciais do desenvolvimento de software, pode permitir a predição de dificuldades de modificação do software antes que o software seja implementado.

1.1 Objetivos

Esta tese tem por objetivos:

1. Definição de um modelo de predição de amplitude da propagação de modificações contratuais em software orientado por objetos. O modelo tem como entrada o software, representado como um conjunto de medidas coletadas nele, e gera como saída a estimativa do número de passos de modificações $E(n,i)$, onde n é o número de classes do software e i é o número de classes que serão inicialmente modificadas.

2. Identificação de valores referência para métricas de software orientado por objetos. Esta é uma questão em aberto na literatura, e sua solução é de grande valia no processo decisório na produção de software [Lanza & Marinescu, 2006; Tempero, 2008]. No presente trabalho são identificados valores referência para

um conjunto de métricas relacionadas a fatores que geram impacto na manutenibilidade de software e que podem ser utilizadas em associação a K3B.

3. Definição de uma métrica de coesão interna de classe, denominada Coesão de Responsabilidade (COR). Esta métrica foi definida com o objetivo de prover uma informação de fácil interpretação sobre o grau de coesão da classe. Neste trabalho, a métrica COR é utilizada com um dos parâmetros de K3B.

4. Identificação de propriedades das estruturas reais de software. No presente trabalho, é conduzido um estudo para caracterizar a evolução de software por meio de métricas de software e identificar a figura macroscópica dos softwares. O objetivo desse estudo é obter empiracamente subsídios para as premissas assumidas na definição de K3B.

1.2 Contribuições

Este trabalho gerou as seguintes contribuições principais:

1. Definição de K3B, um modelo de predição de amplitude da propagação de modificações contratuais em software orientado por objetos.

2. Formalização e avaliação do modelo proposto.

3. Identificação e avaliação de valores referência para seis métricas de software orientado por objetos.

4. Definição de uma métrica de coesão interna de classe, denominada Coesão de Responsabilidade (COR).

5. Identificação de propriedades das estruturas reais de software e da evolução dessas estruturas.

6. Identificação da figura macroscópica das estruturas reais de software.

7. Aperfeiçoamento da ferramenta *Connecta* [Ferreira, 2006]. A ferramenta foi modificada para prover funcionalidades de cálculo de K3B e da métrica Coesão de Responsabilidade.

8. Desenvolvimento de uma ferramenta de simulação de propagação de modificações em software orientado por objetos. Esta ferramenta foi utilizada na avaliação de K3B.

1.3 Metodologia

O trabalho para identificar valores referência de métricas de software foi conduzido experimentalmente. Foram coletadas métricas de um conjunto de 40 softwares abertos, desenvolvidos em Java, de diversos tamanhos, domínios de aplicação e tipos, perfazendo um total de mais de 26.000 classes. Os dados foram analisados estatisticamente e os valores referência foram derivados com base nas propriedades das distribuições dos valores das métricas. Foram realizados dois estudos de caso para avaliar os valores referência propostos.

O estudo sobre a estrutura real de software e sua evolução foi conduzido também experimentalmente. Foram analisados dados de 16 softwares abertos e de um software proprietário desenvolvidos em Java, em um total de 129 versões. Os softwares abertos foram selecionados segundo os critérios de idade, popularidade e diversidade de tamanho, tipo e domínio de aplicação.

A avaliação da métrica Coesão de Responsabilidade (COR) proposta foi realizada por meio da comparação dos valores resultantes de COR com os de outra métrica bem conhecida para avaliar coesão e com os resultados de avaliação qualitativa em um conjunto de classes.

O modelo K3B foi definido com base no raciocínio teórico sobre a natureza e implicações das relações entre os módulos de um software e nos conceitos de Probabilidades, em particular, Cadeias de Markov. O modelo foi matematicamente definido e implementado em um programa no Matlab. A partir da análise dos resultados da execução desse programa, foi identificada uma expressão matemática que corresponde ao modelo K3B. O modelo foi implementado em Java. Sua avaliação foi conduzida por meio de simulação. Para isso, foi implementada um ferramenta que simula propagação de modificações, contando os passos de modificações resultantes a partir de um grupo de modificações iniciais. Os resultados de K3B foram comparados aos da simulação em 37 programas, com o objetivo de se averiguar a existência de correlação entre os valores gerados por K3B e aqueles resultante da simulação.

1.4 Organização do Trabalho

Além deste capítulo introdutório, este trabalho possui outras duas partes:

- **Parte I**: essa parte apresenta uma revisão da literatura relacionada ao assunto tratado neste trabalho. Inicialmente é apresentada uma revisão sobre métricas de software, abordando os seguinte aspectos: conceito e tipologia de métricas; princi-

pais métricas de software orientado por objetos; em particular, são aprofundados os estudos e análise de trabalhos sobre medição de coesão interna em software orientados por objetos; principais trabalhos e áreas de pesquisa em medição de software. Na sequência, é apresentado o conceito de manutenibilidade de software e suas subcaracterísticas. É realizada uma revisão da literatura sobre os principais trabalhos relacionados à avaliação da manutenibilidade de software. Essa parte abrange os seguinte capítulos:

Capítulo 2: descreve os principais conceitos da área de métricas de software, tais como sua relação com qualidade de software, classificações de métricas e princípios de medição.

Capítulo 3: apresenta uma análise das métricas de software orientado por objetos e de ferramentas de coleta de métricas.

Capítulo 4: avalia o estado da arte da área de medição de software, identificando os principais trabalhos realizados e necessidades de trabalhos futuros. Em particular, são estudados aqueles trabalhos relacionados à avaliação de manutenibilidade e à estimativa de propagação de modificações em software.

- **Parte II**: essa parte apresenta a solução proposta para alcançar os objetivos desta tese.

Capítulo 5: apresenta um estudo realizado com o objetivo de identificar valores referência para um conjunto de métricas de software orientado por objetos.

Capítulo 6: apresenta a definição e a avaliação da métrica Coesão de Responsabilidade.

Capítulo 7: apresenta o trabalho realizado sobre a estrutura real de software e aspectos de sua evolução.

Capítulo 8: define o modelo K3B proposto nesta tese.

Capítulo 9: avalia o modelo K3B.

Capítulo 10: apresenta as conclusões deste trabalho e identifica trabalhos futuros.

Capítulo 2

Métricas de Software

Um dos primeiros livros publicados na área de métricas de software, escrito por Gilb [1977], data de meados da década de 70. Desde então, muitos pesquisados têm se dedicado a esse assunto [Boehm, 1981; V. Basili & Rombach, 1994b,a; Chidamber & Kemerer, 1994; Abreu & Carapuça, 1994; Kitchenham, 2009] e o número de métricas que têm sido propostas chega a algumas centenas. Como avaliam Fenton & Neil [2000], um dos desafios nessa área é utilizar as métricas propostas na construção de ferramentas de apoio à decisão. Este capítulo apresenta os principais conceitos relacionados à medição de software.

Três termos principais são utilizados na literatura sobre Métricas de Software: métrica, medida e medição. Embora estes termos sejam usados no mesmo sentido por alguns autores, eles têm significados diferentes, como definem von Staa [2000] e Pressman [2006]:

- métrica (*metric*): é um padrão de medição para avaliar um atributo de algo relacionado a um software.

- medida (*measure*): é o resultado da medição. Uma medida indica quantitativamente a presença de um atributo em determinado software.

- medição (*measurement*): é o ato ou efeito de medir algo de acordo com uma métrica.

Métrica de software é o conjunto de regras que presidem a avaliação de um atributo de algo relacionado ao software. Uma métrica constitui um indicador que é usado pelos engenheiros de software para ajustar o produto, o projeto ou o processo.

Neste capítulo são apresentados os seguintes conceitos relacionados à área de métricas de software: métrica e sua relação com qualidade de software, princípios de medição e categorias de métricas de software.

2.1 Métricas e Qualidade de Software

Pressman [2006] aponta como principal meta da Engenharia de Software a produção de software de alta qualidade e destaca o uso de medição como meio de avaliar a qualidade dos artefatos produzidos ao longo do ciclo de vida do software. A importância da medição de software para a sua qualidade é destacada também pela sua referência em modelos de qualidade mundialmente reconhecidos como o CMM *Capability Maturity Model* e a ISO 9000 [Kan, 2003].

O CMM [Humphrey, 1995] define cinco níveis de maturidade para uma organização desenvolvedora de software: o Nível 1 é o *inicial*, caracterizado como caótico, no qual poucos processos são definidos e o sucesso depende de esforços individuais; no Nível 2 os processos são *repetíveis* em projeto com aplicações similares; o Nível 3, caracterizado como *definido*, o processo é padronizado, documentado e utilizado por toda a organização; o Nível 4 é *gerenciável*, definido como quantitativo, havendo um controle estatístico da qualidade do produto; o Nível 5 é o ideal, caracterizado como *otimizado*, no qual a base quantitativa é utilizada, por exemplo, para previsão de defeitos e investimento na automação de processos.

A ISO 9000 é um conjunto de padrões para a garantia da qualidade na gerência de sistemas. Ela define, entre outros requisitos, o uso de técnicas estatísticas para alcançar qualidade. Esse requisito abrange o uso de métricas de processo e de produto. No caso de métricas de processo, as medidas devem ser utilizadas para propósitos como identificar se o desenvolvimento do processo é efetivo na redução da probabilidade de ocorrência de falhas ou de não detecção dessas. No caso de métricas de produto, as medidas devem ser utilizadas para reportar falhas, agir de forma corretiva caso o nível da métrica seja considerado insatisfatório e estabelecer objetivos de melhoria em termos de métricas.

Qualidade de software é comumente definida como a conformidade do software com os seus requisitos. Esta definição, como aponta Kan [2003], é expressa numericamente por duas métricas principais: taxa de defeitos e confiabilidade. Um exemplo de métrica para taxa de defeitos é a quantidade de defeitos por quantidade de linhas de código. Um exemplo de métrica de confiabilidade é o número de falhas por horas de operação. Entretanto a idéia da qualidade de software vista apenas como a con-

formidade com os seus requisitos não reflete de forma completa os seus aspectos. Na avaliação de Meyer [1997], a qualidade de software pode ser observada por fatores externos, aqueles do ponto de vista do usuário, e por fatores internos, aqueles do pontos de vista da estrutura do software. Outra classificação similar à de Meyer para fatores de qualidade foi dada por McCall et al.[1] (1977 apud Pressman [2006]). Os fatores externos são:

- Correção: é a conformidade das tarefas realizadas pelo software com a sua especificação de requisitos.
- Robustez: é a característica de um software que realiza suas tarefas de forma correta mesmo quando submetido a situações anormais.
- Extensibilidade: é a característica de um software no qual pode-se facilmente incluir ou alterar requisitos.
- Reusabilidade: é a característica de um software que pode ser reutilizado ao todo ou em parte na construção de outros softwares.
- Compatibilidade: é a caracterítica de um software de poder ser facilmente combinado com outros softwares.
- Eficiência: um software é dito eficiente se faz bom uso de recursos como memória e processadores.
- Portabilidade: refere-se à facilidade de se utilizar o software em diferentes ambientes de hardware e software.
- Verificabilidade: é a facilidade de se verificar se um software está em conformidade com a sua especificação de requisitos.
- Integridade: é a capacidade de proteger componentes como dados, programas e documentos contra acessos não autorizados.
- Facilidade de uso: é a facilidade com que o software pode ser utilizado.

Os fatores internos são determinantes para atingir os fatores externos. Dentre eles destaca-se a modularidade, que é a característica de um software construído por unidades elementares denominadas módulos. Um software é construído de forma modular quando seus módulos são o mais independente possível uns dos outros, de maneira

[1] McCALL, J.; Richards, P.; Walters, G. *Factors in Software Quality.* three volumes, NTIS AD-A049-014, 015, 055, November 1977.

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. Para isso, é necessário que o projeto de um software seja direcionado a manter baixo grau de interdependência entre módulos, o que se denomina acoplamento, e alto grau de relacionamento entre os elementos internos de um módulo, o que se denomina coesão [Myers, 1975].

De acordo com Pressman [2006], há fatores que podem ser medidos diretamente e há fatores que podem ser medidos apenas indiretamente. Mas em todos os casos, medições devem ocorrer para que se possa chegar a uma indicação de qualidade do software. McCall et al.[2] (1977 apud Pressman [2006]) propõem que um conjunto de métricas sejam utilizadas para avaliar os fatores de qualidade de acordo com a seguinte relação: $F_q = c_1 \times m_1 + c_2 \times m_2 + ... + c_n \times m_n$, onde F_q é um fator de qualidade, c_i é um peso dado a um métrica e m_i é o valor da métrica. As métricas que podem ser avaliadas apenas subjetivamente têm seus valores graduados em uma escala de 0 a 10, que indicam, respectivamente, valores baixos e altos para a métrica. Dentre as métricas propostas por McCall et al., destacamos a *modularidade*, definida como o grau de independência funcional entre os módulos do software, pois, de acordo com os autores, serve para avaliar indiretamente a maior parte dos fatores de qualidade: confiabilidade, manutenibilidade, flexibilidade, testabilidade, portabilidade, reutilização e interoperabilidade. Na definição de Meyer [1997], confiabilidade corresponde a robustez, flexibilidade corresponde a extensibilidade, testabilidade corresponde a verificabilidade, portabilidade tem o mesmo significado nas duas propostas, e interoperabilidade corresponde a compatibilidade. Manutenibilidade é definida genericamente como o esforço necessário para localizar e eliminar um erro no software. Além da modularidade, as seguintes métricas são indicadas para medir a manutenibilidade de um software:

- Autodocumentação: grau em que o código fonte possui documentação significativa.
- Concisão: indica quão compacto é um software em termos de número de linhas de código.
- Consistência: uso uniforme das técnicas de documentação e projeto ao longo de todo o desenvolvimento do software.

[2]McCALL, J.; Richards, P.; Walters, G. *Factors in Software Quality.* three volumes, NTIS AD-A049-014, 015, 055, November 1977.

- Instrumentação: grau em que o software monitora sua própria operação e é capaz de identificar as falhas ocorridas.
- Simplicidade: grau de facilidade com que o software pode ser compreendido.

2.2 Princípios de Medição

Medição de software é realizada para a obtenção de dados para que possa: obter entendimento quantitativo; avaliar e controlar um produto, um processo ou uma organização; realizar estimativas e planejamentos [Humphrey, 1995]. Encontram-se na literatura duas classificações principais para medições: uma dada por Humphrey [1995] e a outra dada por Kan [2003]. A seguir estas duas classificações são abordadas.

Humphrey [1995] define as seguinte categorias principais para medição:

Objetiva ou subjetiva: uma medida objetiva é quantitativa. Uma medida subjetiva baseia-se no julgamento humano, por exemplo *bom, ruim, baixo, alto, aceitável, satisfatório, etc.*

Absoluta ou relativa: uma medida absoluta não varia em decorrência da adição de novos itens. O contrário ocorre quando uma medida é relativa. O tamanho de um programa é um exemplo de uma medida absoluta, já a média de tamanhos de programas é uma medida relativa.

Explícita ou derivada: uma medida explícita é obtida de forma direta, enquanto uma medida derivada é obtida a partir da combinação de outras medidas.

Dinâmica ou estática: uma medida dinâmica varia com o tempo, o que não ocorre com uma medida estática. Número de defeitos por mês é uma medida dinâmica, pois seu valor varia de acordo com o mês. LOC (número de linhas de código) e número total de defeitos são exemplos de medidas estáticas.

Preditiva ou explanatória: uma medida explanatória é obtida após a ocorrência de um fato, enquanto a obtenção de uma medida preditiva é adiantada, ou seja, ocorre antes do fato.

Kan [2003] classifica medições em quatro níveis: escala nominal, escala ordinal, escala de intervalo e escala de taxa.

Escala nominal: também denominada classificação, é a escala mais simples utilizada. Consiste em classificar um conjunto de elementos em categorias. As categorias

são mutuamente exclusivas e o conjunto de todas as categorias deve cobrir todo o universo de discussão. Nesse tipo de escala, os nomes e as sequência das categorias não definem relações entre as categorias.

Escala ordinal: este tipo de escala é um nível maior do que a escala nominal. Na escala ordinal, os elementos podem ser comparados de acordo com uma ordem. Por exemplo, uma organização pode ser classificada em nível no CMM; uma família pode ser classificada de acordo com sua situação sócio-econômica. Neste caso é possível não apenas categorizar um grupo de elementos mas também ordenar as categorias. A relação entre as categorias é simétrica, ou seja, se $A > B$ então B não é maior do que A, e transitiva, ou seja, se $A > B$ e $B > C$, então $A > C$.

Escala de intervalo: uma escala de intervalo indica a diferença entre dois pontos de medida. As operações matemáticas aplicáveis a intervalos de dados são adição e subtração.

Escala taxa: é o maior nível de medição e todas as operações matemáticas são aplicáveis, inclusive multiplicação e divisão. Por exemplo, se um software A tem 4 defeitos por KLOC (*kilo lines of code*) e B tem 2 defeitos por KLOC, pode-se dizer que A tem duas vezes mais defeitos que B.

O tipo de medição a ser utilizada em um contexto depende da aplicação desejada. Sobre a qualidade da medição, dois critérios mais importantes são identificados: validade e confiabilidade. A validade assegura se a métrica realmente mede o que ela pretende medir. Confiabilidade refere-se à consistência da medição e do método empregado na medição [Kan, 2003].

2.3 Categorias de Métricas de Software

Métricas de software são classificadas em três categorias: métricas de processo, métricas de projeto e métricas de produto [Pressman, 2006; Kan, 2003; Humphrey, 1995].

Métricas de processo: permitem a organização avaliar o processo empregado. Humphrey [1995] define processo como a sequência de passos necessários no desenvolvimento ou na manutenção de software. O processo define técnicas, métodos gerenciais, ferramentas, pessoas e tarefas relacionadas à produção do software. Métricas de processo são utilizadas para promover melhorias no desenvolvimento e na manutenção de software.

Métricas de projeto: métricas de projeto permitem avaliar o andamento de um projeto; descrevem as características de um projeto e de sua execução. Número de desenvolvedores, custo, cronograma e produtividade são exemplos de métricas de projeto. Estas métricas são denominadas por Humphrey [1995] como métricas de recursos.

Métricas de produto: descrevem as características de produtos de software. São exemplos dessas características tamanho, desempenho e complexidade.

Um subconjunto de métricas de software são as denominadas métricas de qualidade de software, que têm foco na qualidade de produto, processo e projeto. Métricas de qualidade de software subdividem-se em métricas de qualidade de produto final e métricas de qualidade de processo, também denominadas métricas *in-process*. A *Engenharia de Qualidade de Software* tem por objetivo investigar a relação entre métricas *in-process*, características de projeto e métricas de qualidade de produto final, construindo melhorias em processo e produto com base nos dados obtidos. Além disso, uma vez que a qualidade deve ser avaliada por todo o ciclo de vida de um software, faz-se necessária a utilização de métricas que avaliem a qualidade do processo e de manutenção de software [Kan, 2003]. Estas categorias de métricas são discutidas a seguir.

2.3.1 Métricas de Qualidade de Produto

A qualidade do produto de software é usualmente medida em função da quantidade de erros, defeitos e falhas do mesmo. Um erro é resultado de uma ação humana que resulta no funcionamento incorreto do software; um defeito é uma anomalia no produto; uma falha ocorre quando uma unidade do software não é capaz de desempenhar a sua função (IEEE/ANSI 982.2). Kan [2003] exemplifica as seguintes métricas de qualidade produto:

1. Tempo médio de falha (MTTF - *mean time to failure*): mede o tempo médio entre falhas de um software.

2. Densidade de defeitos: medida da quantidade de defeitos de um sofware em relação ao seu tamanho. O tamanho pode ser dado por exemplo por número de linhas de código, quantidade de pontos de função, etc.

3. Problemas de usuários: denotam os defeitos encontrados na utilização do software por seus usuários. É dada pelo *total de problemas reportados pelo usuário em*

período / total de número de licenças-mês durante o período, onde *número de licenças-mês = número de licenças instaladas do software × número de meses no período.*

4. Satisfação de usuário: frequentemente medida via uma escala de cinco pontos: *muito satisfeito, satisfeito, neutro, insatisfeito, muito insatisfeito*. A partir dessa escala, muitas outras métricas podem ser obtidas, por exemplo: percentual de usuários insatisfeitos, percentual de usuários satisfeitos, etc.

2.3.2 Métricas de Qualidade de Processo

Partindo da ideia de que os erros identificados durante a fase de teste indicam que erros foram introduzidos no software durante seu desenvolvimento, Kan [2003] exemplifica as seguintes métricas de qualidade processo (métricas *in-process*):

1. Densidade de defeitos durante testes: uma boa métrica para essa densidade é o número de defeitos por KLOC ou pontos de função.

2. Remoção de defeitos baseada em fases: é uma extensão da métrica de densidade de defeitos para as demais fases do ciclo de vida de sistema. Reflete a capacidade geral de remoção de defeitos por fases do processo de desenvolvimento.

2.3.3 Métricas para Custo de Manutenção de Software

A fase de manutenção do ciclo de vida de um software inicia-se quando o desenvolvimento de um software é finalizado e este entra em operação. Kan [2003] descreve métricas para a fase de manutenção que se baseiam na identificação de defeitos por intervalo de tempo e chamados de problemas de usuários, que podem significar defeitos ou não. As métricas para a fase de manutenção descritas por Kan [2003] são apresentadas a seguir.

Acúmulo de consertos (*fix backlog*): é uma soma simples do número de problemas reportados que permanecem sem consertos no fim de cada período, por exemplo, cada semana ou cada mês.

Índice gerencial de acúmulo de consertos: BMI (*backlog management index*) é dada pela razão entre *o número de problemas resolvidos durante o período* e *o número de problemas identificados durante o período* multiplicado por 100. Se BMI é maior do que 100, significa que o acúmulo de problemas foi reduzido no período, caso contrário, o acúmulo de problemas aumentou.

Tempo de resposta de consertos: é o tempo médio de resposta entre a abertura de um problema e o seu fechamento. A abertura do problema ocorre quando ele é identificado; o seu fechamento ocorre quando ele é solucionado.

Qualidade de consertos: do ponto de vista do usuário, é ruim que um software apresente defeitos e é igualmente ruim, ou ainda pior, se uma manutenção introduz defeitos no software. Uma manutenção gera defeitos se ela não corrige o problema inicialmente identificado ou se ela corrige o problema, mas introduz um novo defeito. Isso é denominado por Kan como conserto defeituoso. A métrica de qualidade de consertos é dada pelo percentual de todos os consertos defeituosos realizados em um intervalo de tempo.

Estas métricas são coletadas já na fase de manutenção do software. Entretanto, é importante a possibilidade de preparo e planejamento da manutenção já em fases iniciais do ciclo de vida do sistema. Para isso, é necessária a identificação de uma métrica, ou de um conjunto de métricas, que possa ser utilizada em fases inicias do ciclo de vida do software para indicar o esforço na fase de manutenção. Com isso, problemas que acarretariam dificuldades na fase de manutenção poderiam ser identificados e sanados previamente, reduzindo o custo de manutenção. As Seções 4.4 e 4.5 descrevem estudos que têm sido realizados nesse sentido.

2.4 Conclusão

Qualidade de software é uma das principais questões na produção de software. Métricas e qualidade de software são duas áreas correlatas. Métricas têm papel fundamental na qualidade, pois fornecem aos engenheiros de software dados que lhes permitem avaliar processos, projetos e produtos, bem como controlá-los, melhorá-los e planejá-los.

A importância desse tópico pode ser avaliada pela quantidade de trabalhos e publicações na área. As primeiras publicações datam de meados dos anos 70 [Gilb, 1977]. Um levantamento realizado por Xenos et al. [2000] conta dezenas de métricas propostas na literatura, dentre as quais a mais famosa talvez seja LOC (número de linhas de código). Fenton & Neil [2000] avalia que o desafio da área de medição de software é utilizar métricas simples, já propostas, para a construção de ferramentas de apoio à decisão no desenvolvimento de software.

Nos próximos capítulos são identificadas as principais métricas propostas na literatura para software orientado por objetos, bem como as principais linhas de trabalhos realizadas atualmente nesta área.

Capítulo 3

Métricas de Software Orientado por Objetos

A estrutura de um software orientado por objetos é essencialmente diferente de software construído no paradigma estruturado. Desta forma, métricas utilizadas para avaliar este tipo de software não são suficientes e algumas delas são inadequadas para avaliar software orientado por objetos. Como exemplo, podemos citar a métrica LOC (*lines of code*), que refere-se ao número de linhas de código de um software, uma das mais conhecidas métricas para indicar tamanho de software. No caso de um software OO, esta métrica fornece um nível de informação muito baixo. Métricas de software OO devem avaliar especificamente as características deste paradigma, tais como: coesão interna de classes, acoplamento entre classes, ocultação de informação e herança.

Dois dos conjuntos de métricas mais referenciados na literatura são aquele proposto por Chidamber e Kemerer [Chidamber & Kemerer, 1994], denominado conjunto CK, e aquele proposto por Abreu & Carapuça [1994], denominado MOOD. Ambos foram propostos em 1994, mas são independentes. Este capítulo apresenta estes dois conjuntos de métricas e duas métricas propostas para avaliação de estabilidade de sistema: a de Martin [1994] e a de Myers [1975] adaptada por Ferreira [2006], bem como um conjunto de ferramentas de coletas de métricas em sistemas OO.

3.1 Métricas CK

Chidamber & Kemerer [1994] 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 Weyuker [1988] e relatam os resultados de experimentos reali-

zados 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: Métodos Ponderados por Classe (WMC - *Weighted Methods per Class*), Profundidade de Árvore de Herança (DIT - *Depth of Inheritance Tree*), Número de Filhos (NOC - *Number of Children*), Acoplamento entre Classes de Objetos (CBO - *Coupling between Object Class*), Resposta de Classe (RFC - *Response for a Class*) e Ausência de Coesão em Métodos (LCOM - *Lack of Cohesion in Methods*).

- *WMC (Métodos Ponderados por Classe):* 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 tornar a aplicação desta métrica mais geral. 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. A quantidade de métodos de uma classe e a complexidade de tais métodos constituem um indicador do esforço de manutenção da classe. Além disso, uma classe com um grande número de métodos tem potencial de reúso limitado, pois tende a ter um uso específico da aplicação da qual fazem parte.

- *DIT (Profundidade de Árvore de Herança):* 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 (Número de filhos):* 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 (Acoplamento entre Classes de Objetos):* 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 (Resposta de Classe):* é o 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 (Ausência de Coesão em Métodos):* 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. 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

  Por exemplo, seja uma classe que possua dez pares de métodos sem variáveis de instância em comum e dois pares de métodos com variáveis de instância em comum. Assim, P = 10 e Q = 2. Então, LCOM = P - Q = 8. Uma classe que possua dez pares de métodos sem variáveis de instância em comum e quatro pares de métodos com variáveis de instância em comum tem LCOM = 6. Em comparação com a classe do exemplo anterior, esta classe apresenta melhor grau de coesão entre os seus métodos.

  LCOM indica a diferença entre a quantidade de pares de métodos sem similaridade, isto é, pares de métodos que não possuem variáveis de instância em comum, e a quantidade de pares de métodos com similaridade. Valores baixos dessa métrica indicam bom nível de similaridade, portanto de coesão, entre os métodos da classe avaliada. Um valor alto para LCOM indica que a classe não provê uma funcionalidade bem específica. Todavia, essa métrica tem sido criticada na literatura Kitchenham [2009] pelo fato de gerar valores que não permitem uma interpretação adequada da coesão interna de classes.

3.2 Métricas MOOD

O conjunto de métricas MOOD (Metrics for Object Oriented Design) foi proposto por Abreu & Carapuça [1994]. 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 (MIF - *Method Inheritance Factor*), Fator Herança de Atributo (AIF - *Attribute Inheritance Factor*), Fator Acoplamento (COF - *Coupling Factor*), Fator Agrupamento (CLF - *Clustering Factor*), Fator Polimorfismo (PF - *Polymorphism Factor*), Fator Ocultação de Método (MHF - *Method Hiding Factor*) , Fator Ocultação de Atributo (AHF - *Attribute Hiding Factor*), Fator Reúso (RF - *Reuse Factor*).

O cálculo de uma métrica MOOD é dado por uma razão na qual 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 o nível de ocorrência do 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, são apresentadas as principais métricas do conjunto MOOD.

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 & Kemerer [1994]: 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 da dificuldade 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. Seja C_i uma classe do sistema a ser avaliado. Para a definição da métrica MIF, são consideradas as seguintes métricas básicas:

- Métodos herdados: $M_h(C_i)$. São os métodos que uma classe possui em decorrência de herança e que não foram redefinidos na classe.
- Métodos novos: $M_n(C_i)$. São métodos criados na classe, que não foram herdados nem redefinidos.
- Métodos redefinidos: $M_r(C_i)$. São métodos herdados que têm uma redefinição na classe.
- Métodos definidos: $M_d(C_i)$. Englobam os métodos novos e os métodos redefinidos na classe.
- Métodos disponíveis: $M_{dis}(C_i)$. É a totalidade de métodos que uma classe possui, o que engloba métodos definidos nela e os métodos herdados por ela.
- Total de classes do sistema: TC.
- Total de métodos novos definidos no sistema: é o somatório do número de métodos novos de cada classe do sistema, dado pela Equação 3.1.

$$TM_n = \sum_{k=1}^{TC} M_n(C_k) \tag{3.1}$$

- Total de métodos novos redefinidos no sistema: é o somatório do número de métodos redefinidos em cada classe do sistema, dado pela Equação 3.2.

$$TM_r = \sum_{k=1}^{TC} M_r(C_k) \tag{3.2}$$

- Total de métodos definidos no sistema: é o somatório do número de métodos definidos em cada classe do sistema, que englobam tanto os métodos novos das classes quanto os redefinidos nelas. Esse total é dado pela Equação 3.3.

$$TM_d = TM_n + TM_r = \sum_{k=1}^{TC} M_d(C_k) \tag{3.3}$$

- Total de métodos herdados no sistema: é o somatório do número de métodos herdados em cada classe do sistema, dado pela Equação 3.4.

$$TM_h = \sum_{k=1}^{TC} M_h(C_k) \tag{3.4}$$

- Total de métodos disponíveis no sistema: é o somatório do número de métodos disponíveis em cada classe do sistema, dado pela Equação 3.5.

$$TM_{dis} = \sum_{k=1}^{TC} M_{dis}(C_k) \tag{3.5}$$

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 do número de métodos herdados de cada classe do sistema e o somatório do número de métodos disponíveis de cada classe do sistema. Assim, o cálculo de MIF é dado pela Equação 3.6

$$MIF = \frac{TM_h}{TM_{dis}} \tag{3.6}$$

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.

- *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 do número de atributos herdados de cada classe do sistema e o somatório do número de atributos disponíveis de cada classe do sistema. Assim, o cálculo de AIF é dado pela Equação 3.7, onde TA_h é o total de atributos herdados no sistema e TA_{dis} é o total de atributos disponíveis no sistema. O cálculo de TA_h e TA_{dis} são similares aos de TM_h e que são dados pelas Equações 3.4 e 3.5 respectivamente.

$$AIF = \frac{TA_h}{TA_{dis}} \tag{3.7}$$

A interpretação dos valores desta 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 alta utilização de tal recurso.

2. *Métricas Para Avaliação de Ocultação de Informação*

   A ocultação de informação é um conceito importante relacionado à modularidade, pois a sua aplicação potencializa a independência de módulos. Quanto mais ocultos os serviços e as informações de uma classe, menor a necessidade de as demais classes conhecerem sua organização interna e, consequentemente, mais fraco é o nível de interdependência entre as classes. Uma classe deve ser conhecida somente pelos serviços que ela disponibiliza. Na orientação por objetos, a ocultação de informação é obtida principalmente 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, as seguintes métricas básicas são definidas, considerando-se C_i uma classe qualquer do sistema a ser avaliado.

     - Métodos visíveis: $M_v(C_i)$. São os métodos que constituem a interface da classe.
     - Métodos ocultos: $M_o(C_i)$. São os métodos privados da classe.
     - Métodos definidos: $M_d(C_i)$. São os métodos visíveis e os métodos ocultos da classe. Essa métrica é dada pela Equação 3.8.

     $$M_d(C_i) = M_v(C_i) + M_o(C_i) \tag{3.8}$$

     MHF é a razão entre o número de métodos ocultos em todas as classes e o número de métodos definidos em todas as classes, dado pela Equação 3.9

     $$MHF = \frac{\sum_{k=1}^{TC} M_o(C_k)}{\sum_{k=1}^{TC} M_d(C_k)} \tag{3.9}$$

     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 são obtidos valores próximos a 0 para essa métrica, o que indica que as classes do sistema exportam muitos serviços. Isso pode favorecer o surgimento de 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, conforme a Equação 3.10. Nesta equação, A_o corresponde ao número de atributos ocultos na classe e A_d, ao número de atributos disponíveis na classe; o cálculo de A_d é similar ao de M_d, que é dado pela Equação 3.8.

$$AHF = \frac{\sum_{k=1}^{TC} A_o(C_k)}{\sum_{k=1}^{TC} A_d(C_k)} \tag{3.10}$$

  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 acoplamentos 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. *Métrica Para Acoplamento*

   O conjunto MOOD define a métrica COF (Fator Acoplamento) como um indicador do grau de conectividade do software.

- *COF (Fator Acoplamento):* para a avaliação de acoplamento, Abreu & Carapuça [1994] 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 um atributo ou um método. Uma relação cliente-servidor entre duas classes corresponde à existência de uma conexão entre elas.

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

3.3 Análise das Métricas CK e MOOD

Os dois conjuntos de métricas, CK e MOOD, visam a avaliação quantitativa de sistemas orientados por objetos, porém com abordagens distintas. As métricas CK avaliam fatores de classes no sistema, por exemplo, a falta de coesão da classe, o número de filhos de uma classe, a profundidade na árvores de herança de uma classe, etc. Já as métricas MOOD buscam avaliar não classes particulares no sistema, mas o sistema como um todo ou em parte. As duas abordagens são complementares, pois na avaliação de um sistema estes dois níveis de avaliação - sistema e classe - são importantes. A primeira permite avaliar condições gerais do sistema, a segunda permite detalhar o nível da avaliação.

Nenhum dos dois conjuntos possui métricas para avaliação de aspectos como estabilidade de um software. Para este aspecto, destaca-se a métrica proposta por Martin [1994]. Uma outra forma de avaliar estabilidade de software foi proposta por Myers [1975], porém para o paradigma estruturado. Esta proposta foi adaptada ao paradigma orientado por objetos por Ferreira [2006]. A Seção 3.4 apresenta estas duas propostas.

Em especial, métricas para avaliação de coesão interna de classe apresentam maior grau de dificuldade em sua elaboração do que os demais fatores, pois a avaliação de

coesão pode depender da compreensão do domínio do problema do sistema, cuja avaliação automática é muito difícil. Isso tem sido tema de trabalhos atuais nesta linha, conforme é discutido na Seção 4.1.

3.4 Métrica de Estabilidade

Gilb [1977] define a estabilidade como a capacidade de um sistema manter-se inalterado diante de uma alteração em seu ambiente. Para Martin [1994] também a estabilidade está relacionada a esta habilidade: quando um módulo é independente dos demais, alterações nestes não demandam alterações no primeiro, garantindo sua estabilidade. Pressman [2006] dá um significado mais amplo para estabilidade, definindo-a como a característica de um software no qual modificações não são frequentes, são controladas e não invalidam os testes realizados. Avaliamos que os conceitos não são distantes, mas complementares, pois a estabilidade, tal como é definida por Gilb e Martin, é que permite atingir as características apontadas por Pressman.

A seguir são apresentadas duas métricas para avaliação de estabilidade em sistemas orientados por objetos: a primeira, proposta por Martin [1994], mede o grau de *instabilidade* de um software, e a segunda, denominada *métrica de estabilidade*, proposta originalmente por Myers [1975] para avaliação de softwares no paradigma estruturado e adaptada por Ferreira [2006] para avaliar software orientado por objetos.

3.4.1 Métrica de Instabilidade

Martin [1994] alerta que quando a extensão de uma alteração em um sistema não é predizível, o seu impacto não pode ser estimado. Com base neste problema, ele propõe uma métrica de *instabilidade* que baseia-se na análise de dependências entre o que ele chama de categorias de classes. Uma categoria de classes é um grupo coeso de classes no qual: se uma classe da categoria é alterada, as demais classes da categoria têm grandes chances de serem alteradas; as classes são reutilizadas em conjunto; as classes têm um objetivo em comum ou realizam funções interdependentes.

Para Martin [1994], a independência e a estabilidade de uma categoria de classes podem ser medidas a partir do número de conexões da categoria com classes externas a ela. Desta forma, a métrica de instabilidade I é definida da seguinte maneira:
$I = Ce/(Ca + Ce)$, onde:

- Ca: é o número de acoplamentos aferentes (*afferent couplings*). Corresponde ao número de classes externas à categoria que usam as classes da categoria.

- Ce: é o número de acoplamentos eferentes (*efferent couplings*). Corresponde ao número de classes da categoria que usam classes externas à categoria.

A métrica representa a razão entre os acoplamentos eferentes da categoria e total de acoplamentos nos quais a categoria está envolvida. Os valores da métrica de instabilidade vão de 0 a 1. Martin aponta que quando I é igual a zero, indica que a categoria está com a estabilidade máxima; um valor igual a 1, indica que a categoria tem instabilidade máxima.

3.4.2 Métrica de Estabilidade

Myers [1975] propõe *um modelo de estabilidade de programas* que resulta em métricas que fornecem os seguintes indicadores: quantos módulos serão alterados em decorrência da alteração de um módulo qualquer no sistema; quantos módulos serão alterados em decorrência da alteração de um módulo específico do sistema. Para alcançar isso, o sistema é modelado como um grafo não direcionado no qual os vértices representam os módulos do sistema e as arestas representam as conexões entre os módulos. Para cada módulo, deve ser avaliado o grau de sua coesão, assim como para cada conexão avalia-se o grau do acoplamento envolvido entre os módulos. Estas avaliações seguem uma escala proposta pelo autor, como mostra a Tabela 3.1. Um descrição dessa escala pode ser obtida no trabalho de Ferreira [2006].

Coesão	*Valor*
Coincidental	0,95
Lógica	0,4
Clássica	0,6
Procedimental	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 3.1. Pesos de Coesão e Acoplamento no Modelo de Myers

A ideia do cálculo desta métrica é obter para cada par de módulos a probabilidade de um ser alterado em decorrência da alteração do outro. O cálculo desta probabilidade considera o grau de coesão dos módulos envolvidos, o grau de acoplamento direto e indireto entre eles.

A métrica de estabilidade de Myers [1975] foi adaptada por Ferreira [2006] ao paradigma OO. A adaptação consiste nos seguintes pontos:

- A métrica original modela o sistema como um grafo não direcionado. Ferreira considera que o modelo que melhor representa as conexões existentes entre módulos de um sistema é um grafo direcionado, pois o fato de uma alteração em um módulo A acarretar uma alteração em B, não implica que uma alteração em B também gere alteração em A.

- Ferreira [2006] revisou os tipos de acoplamento e coesão na orientação por objetos, o que resultou nos valores mostrados na Tabela 3.2. A descrição detalhada dessa escala pode ser obtida no trabalho de Ferreira [2006].

- Na métrica original, não é definido como proceder no caso de um módulo estar acoplado a outro por mais de uma forma. Na adaptação de Ferreira, considera-se a conexão que envolve acoplamento mais forte, de acordo com a escala definida na Tabela 3.2

Acoplamento	*Valor*
Conteúdo	0,95
Dado comum	0,7
Inclusão	0,7
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 3.2. Pesos de Acoplamento e Coesão na OO

3.4.3 Avaliação das Métricas de Estabilidade

A métrica de Instabilidade de Martin [1994] fornece uma avaliação da instabilidade de categorias de classes de um sistema e não de classes isoladas, tampouco do sistema como um todo. No que tange ao nível de informação fornecida, a métrica de estabilidade de Myers [1975] adaptada por Ferreira [2006] é mais abrangente, pois avalia a estabilidade do sistema e de classes particulares. Ambas métricas têm base conceitual sólida. A primeira parte do princípio de que é possível avaliar a estabilidade de um conjunto de classes a partir da quantidade de conexões deste conjunto com as demais classes do sistema. A segunda métrica considera, além da quantidade de conexões, o grau de acoplamento entre os módulos, bem como o grau de coesão interna dos módulos.

3.5 Ferramentas de Coleta de Métricas

A aplicação de métricas de software requer o uso de ferramentas que as coletem. Algumas das ferramentas que possuem este propósito são apresentadas a seguir.

Understand [Understand, 2007] é uma ferramenta comercial que coleta métricas em softwares escritos em Ada, Delphi, Fortran, Java e C++. Entre as métricas coletadas pela ferramenta estão: número de classes, número de linhas de código, número de linhas de comentários e número de linhas de declarações.

Krakatau Essencial Metrics [Krakatau, 2006] é uma ferramenta comercial 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 OO, não provê métricas específicas a esse paradigma.

A ferramenta comercial *ObjectDetail* [ObjectDetail, 2006] coleta métricas específicas do paradigma OO em softwares desenvolvidos em C++. Dentre as métricas coletadas, destacam-se: profundidade da árvore de herança, acoplamento de classe, que é o número de classes que uma classe particular usa, e percentuais de atributos públicos, de atributos privados, de métodos públicos e de métodos privados.

MOODKIT [Abreu et al., 1997] 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 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.

Dependency Finder [DependencyFinder, 2007] é uma ferramenta *open source* que permite a análise de código compilado Java, sendo disponível para ambientes Windows, Unix e Web. Fornece o grafo de dependência entre classes e a coleta de um conjunto de métricas OO em nível de métodos, classes, grupos de classes e sistema. Dentre as métricas do conjunto CK e MOOD, coleta apenas a métrica de profundidade da árvore de herança.

JDepend [JDepend, 2007] é uma ferramenta *open source* que coleta métricas de pacotes de classes em Java, tais como número de classes e interfaces, acoplamentos aferentes, que são o número de classes externas ao pacote que usam as classes do pacote, e acoplamentos eferentes, que são o número de pacotes dos quais as classes do pacote dependem.

Metrics [Metrics, 2007] coleta métricas em software implementado em Java. É um *plugin* para o Eclipse. Coleta várias métricas orientadas por objetos, dentre as quais estão: acoplamento aferente, acoplamento eferente e instabilidade para pacotes, LCOM e DIT. Inclui um analisador gráfico de dependências entre pacotes de classes.

Connecta [Ferreira, 2006] é uma ferramenta que coleta métricas em software JAVA, a partir da análise de *bytecode*. A ferramenta coleta as seguintes métricas: COF, número de conexões aferentes, DIT, LCOM e estabilidade Ferreira [2006]. Além disso, a ferramente indica os graus de acoplamentos entre as classes do sistema. Os resultados são apresentados de forma que se possa identificar possíveis pontos de melhoria no sistema: inicialmente são reportados os valores da estabilidade e COF; a partir desses valores, pode-se detalhar o nível de avaliação por classes do sistema, obtendo-se os valores das métricas DIT, número de conexões aferentes, LCOM e grau de acoplamento entre classes. Uma das melhorias a serem realizadas na ferramenta é a inclusão de um recurso que permita a visualização dos resultados de forma gráfica.

Embora exista uma quantidade grande de ferramentas de coleta de métricas disponíveis no mercado e na academia, nenhuma delas apresenta todos os requisitos ideais de uma ferramenta desta categoria. Tais requisitos envolvem os seguintes aspectos, dentre outros:

- *Coleta de métricas de fato orientada por objetos*: algumas das ferramentas disponíveis, embora coletem métricas em software orientado por objetos, não coletam métricas relevantes específicas da orientação por objetos, como é o caso de Understand e Krakatau.

- *Integração com o ambiente de implementação*: este requisito é importante para que o próprio desenvolvedor possa avaliar a qualidade do código que está produzindo com facilidade. Metrics provê esta facilidade.

- *Possibilidade de execução fora do ambiente de implementação*: permite que o código seja avaliado de forma independente do ambiente de execução. Isso facilita a coleta e análise de métricas por gerentes, analistas, etc., sem a necessidade do uso do ambiente de implementação. Connecta e JDepend, por exemplo, funcionam desta forma.

- *Exportação de dados*: o recurso de exportação de dados coletados permite que estes sejam posteriormente manipulados e analisados. Metrics, por exemplo, permite exportação de dados em formato XML.

- *Armazenamento de histórico*: o armazenamento dos dados coletados é essencial para tarefas como a análise comparativa entre resultados de versões distintas do mesmo software e entre softwares diferentes.

- *Multilinguagem*: um requisito desejável em uma ferramenta de coleta de métricas é que ela opere sobre softwares desenvolvidos em linguagens diferentes. Connecta, Metrics, JDepend, Dependency Finder funcionam somente para código Java.

- *Visualização gráfica das dependências entre os módulos do software*: uma das métricas mais importantes da orientação por objetos é a que mede o grau de conectividade do software. Das ferramentas listadas nesta seção, somente MOODKIT e Connecta coletam tal métrica. Um recurso facilitador para identificação de módulos críticos no sistema em relação ao fator conectividade é a visualização gráfica das dependências entre os módulos. Metrics possui este recurso, porém considera como módulos os pacotes do sistema e não as suas classes.

O ideal é que existisse uma ferramenta que agregasse todos esses requisitos. O que se observa é que, ao realizar uma pesquisa que exija a coleta de métricas, os pesquisadores acabam desenvolvendo suas próprias ferramentas. A ausência desses requisitos nas ferramentas prejudica também o seu emprego na indústria.

3.6 Conclusão

Há uma grande quantidade de métricas de software orientado por objetos propostas na literatura. As mais referenciadas atualmente são as do conjunto CK [Chidamber & Kemerer, 1994], que define métricas para avaliar classes. Outro conjunto de métricas muito conhecido é o MOOD [Abreu & Carapuça, 1994], que define métricas para avaliar sistemas. Embora tenha havido grandes avanços na definição e avaliação de métricas de software, ainda há importantes desafios nessa área. Um deles é a identificação dos valores referência das métricas de software [Tempero, 2008], pois a falta de conhecimento sobre esses valores pode dificultar a aplicação de métricas de software na prática [Lanza & Marinescu, 2006].

Capítulo 4

Pesquisas em Medição de Software

A área de pesquisa em métricas de software orientado por objetos é um campo fértil. O interesse pela área abrange questões como a proposta de novas métricas de software, a identificação de necessidades de refatorações por meio de métricas, e a utilização de métricas como intrumentos de predição de falhas e de avaliação de manutenibilidade de software. Este capítulo identifica os principais assuntos que têm sido discutidos na literatura, analisa alguns dos trabalhos realizados e identifica necessidades de trabalhos futuros na área.

4.1 Medição de Coesão

Coesão em software foi definida por Myers [1975] como o grau de intercomunicação entre os elementos internos de um módulo. Myers apresentou a escala de grau de coesão interna de módulos que desde então foi adotada e aceita pela comunidade[1]. Essa escala é qualitativa e baseia-se na observação e na análise do relacionamento entre os elementos de um módulo. Um módulo cujos elementos não apresentam relacionamento algum possui o pior grau de coesão, denominada coesão coincidental; um módulo que realiza uma única função bem definida possui coesão funcional, sendo o melhor tipo de coesão.

Avaliar quantitativamente a coesão de um módulo é difícil, pois dizer se os elementos de um módulo possuem relacionamento ou desempenham uma única função bem definida muitas vezes envolve conhecer o domínio do problema. Como apontam Counsell et al. [2005], esta tarefa tem sido tema de muitos trabalhos pelo fato de a coesão ser um fator de grande importância na compreensão e na manutenção de software.

[1] Alguns autores atribuem a Yourdon e Constantine essa definição de coesão e sua respectiva escala em 1979. Porém, esse conceito foi proposto quatro anos antes, por Glenford Myers.

Em particular, a comparação matemática de propriedades de métricas de coesão está em curso. Esta seção apresenta trabalhos realizados para avaliação de coesão interna em software orientados por objetos.

A métrica mais referenciada para este fator é LCOM proposta por Chidamber & Kemerer [1994], descrita na Seção 3.1. A métrica baseia-se no conceito de que coesão de uma classe é avaliada pelo grau de similaridade entre seus métodos. Esta métrica foi revisada posteriormente por: Bieman e Ott (1994), Henderson-Sellers (1996) e Briand et al. (1998) [Counsell et al., 2005]. Embora essa métrica seja muito criticada na literatura [Kitchenham, 2009], ela é implementada em muitas ferramentas de coleta de métricas [Lincke et al., 2008].

Bansiya[2](1999 apud Counsell et al. [2005]) propõem a métrica CAMC (*cohesion among methods in a class*), uma métrica de coesão baseada no fato de que uma classe é considerada coesa se seus métodos usam o mesmo conjunto de tipos de parâmetros. Counsell et al. [2005] avalia que esta abordagem tem a seguinte vantagem principal em relação àquela utilizada em LCOM: pode ser vista como uma métrica de projeto, podendo ser utilizada em estágios iniciais do desenvolvimento de software. Counsell et al. [2005] propõem outras duas métricas, denominadas NHD (*normalised Hamming distance*) e SNHD (*scaled normalised Hamming distance*), que seguem o mesmo conceito de coesão utilizado em CAMC. Eles realizaram um estudo comparativo entre essas métricas e LCOM e concluíram que, assim como LCOM, os valores gerados pelas métricas são dependentes do tamanho da classe.

Marcus & Poshyvanyk [2005] propõem medir a coesão de classes com base na análise de informação semântica no código fonte, tais como comentários e identificadores. O cálculo da métrica baseia-se em técnicas de recuperação de informação avançadas. A métrica C3, denominada *coesão conceitual*, mede o grau em que os métodos de uma classe estão conceitualmente relacionados. Eles realizaram um estudo de caso em que a coleta da métrica proposta em um software de código aberto foi comparado aos resultados de outras métricas de coesão já propostas na literatura, entre elas LCOM. Marcus & Poshyvanyk [2005] verificaram que os resultados obtidos pelas duas métricas são coerentes. Entretanto, como os próprios autores de C3 avaliam, o sucesso do cálculo da métrica é totalmente dependente da existência de comentários e nomes significativos no corpo do código fonte.

Mäkelä & Leppänen [2007] propõem a medição de coesão externa de classe, com base na premissa de que para o cliente de uma classe, a sua representação interna não é tão relevante quanto a sua interface. A métrica, denominada ELCOM (*external lack*

[2]Bansyia, J., Etzkorn, L. *A class cohesion metric for object-oriented designs.* Journal Object-Oriented Program. 11, 8, 47-52. 1999.

of cohesion in methods) refere-se à forma como classes clientes de uma classe c usa os serviços dela. A métrica é definida conforme a Equação 4.1, onde $M(c)$ é o conjunto de classes clientes de c, $V(c)$ é o conjunto de variáveis de instância de c, e $C(c)$ é o conjunto de pares ordenados (m, v) pertencentes ao produto cartesiano entre $M(c)$ e $V(c)$, tal que m possua algum método que use v . Para avaliar a aplicabilidade da métrica, foi realizado um estudo de caso com 4 softwares de código aberto; no estudo foram coletadas as métricas ELCOM e LCOM. A métrica ELCOM fornece uma evidência que sugere o grau de coesão interna da classe, porém, como destacam os autores, é necessária uma avaliação mais aprofundada da classe para afirmar a qualidade de sua estrutura.

$$ELCOM(c) = 1 - \frac{|C(c)|}{(|M(c)|.|V(c)|)} \tag{4.1}$$

Briand et al. [1998] recomendam que a definição ou a implementação de métricas de coesão devem possuir estratégias para lidar com as seguintes questões: métodos construtores, métodos de acesso a dados da classe (métodos *get* e *set*) e herança. Em algumas métricas de coesão, a inclusão de método construtor na análise pode prejudicar o valor gerado pela métrica. Isso é especialmente importante quando a métrica é baseada na análise do uso de atributos da classe pelos métodos. O ideal é que métodos construtores não sejam considerados nesses casos. Da mesma forma, no caso de algumas métricas propostas na literatura, como é o caso de LCOM, a inclusão de métodos de acesso diminuem artificialmente o valor obtido para a coesão da classe. Entretanto, optar por excluir métodos de acesso pode ser uma alternativa de difícil implementação porque nem sempre a identificação desses métodos é facilmente realizada de forma automática. Em relação à herança, é possível considerar ou não os métodos e atributos herdados. A exclusão de elementos herdados na análise da coesão da classe deve ser realizada quando se pretende avaliar o grau de coesão da extensão da classe. Os elementos herdados devem ser incluídos na análise quando se pretende avaliar a coesão da classe como um todo.

Avaliação

Há outras propostas de métricas de coesão na literatura. Não há, ainda, um consenso na literatura sobre uma métrica padrão para coesão. Há uma grande diversidade de abordagens para medir coesão interna de classes. Em alguns casos, por exemplo, nas métricas CAMC e ELCOM, a complexidade da definição e do cálculo da métrica torna a interpretação dos seus resultados difícil.

Dentre as necessidades de trabalhos futuros nessa linha, destacam-se: estudos experimentais em sistemas maiores, a fim de verificar a aplicabilidade das métricas propostas; a comparação dos valores obtidos para as métricas com a análise qualitativa realizada por desenvolvedores de software; definição de valores ou faixas de valores a serem considerados satisfatórios para o fator coesão.

4.2 Medição de Software Orientado por Aspectos

Em relação à Programação Orientada por Aspectos (POA), tem sido discutido [Griswold et al., 2006; Kiczales & Mezini, 2005; Wand, 2003] o seu impacto na modularidade e no *raciocínio modular*, que refere-se à compreensão de um módulo e a tomada de decisão sobre ele conhecendo-se apenas sua implementação, sua interface e as interfaces dos módulos referenciados por ele [Kiczales & Mezini, 2005]. Os relatos nesses trabalhos baseiam-se em conceitos e aspectos qualitativos. Há uma carência de estudos que avaliem quantitativamente os impactos do uso da orientação por aspectos (OA) na manutenibilidade de software. Esta seção discute dois dos trabalhos realizados nessa linha.

Santa'Anna et al. [2003] propõem um modelo para avaliação de manutenibilidade e reusabilidade de software orientado por aspectos e realizam estudos experimentais. O modelo consiste em um conjunto de fatores considerados como determinantes da manutenibilidade e reusabilidade desse tipo de software, como separação de interesses, tamanho, coesão e acoplamento, bem como a definição de um conjunto de métricas para avaliação de tais fatores. O estudo experimental relatado nesse trabalho usa duas versões de um software: uma que utiliza padrões de projeto na orientação por objetos (OO) e outra que utiliza OA. No experimento, um grupo de implementadores realizaram sete tipos de alterações nos softwares em questão e, para as alterações realizadas, foram coletadas as seguintes medidas: quantidade de componentes de software adicionados, número de componentes alterados, número de relações entre componentes incluídos, número de linhas de código incluídas e alteradas. Santa'Anna et al. [2003] afirmam que os resultados dos experimentos mostram que os fatores utilizados no modelo, como acoplamento, número de componentes e separação de interesses, tem impacto direto na manutenibilidade e na reusabilidade. Entretanto, esta relação não está claramente apresentada nos resultados. Os dados apresentados mostram que, em algumas alterações, o software OO demandou um número maior de inclusão de linhas de código, relacionamentos e de alterações de operações. Esse fato poderia indicar que alterar software OA é mais fácil do que alterar software OO, porém não evidencia que as

métricas de acoplamento, número de componentes e separação de interesses podem ser utilizadas para avaliar a manutenibilidade e reusabilidade de software OA. Sant'anna et al. destacam, ainda, os seguintes pontos que ficam em aberto:

1. O estudo que realizaram considera como esforço de manutenção a quantidade linhas de código incluídas ou alteradas. Faz-se necessário utilizar medidas mais representativas para o aspecto manuntenibilidade, por exemplo, o tempo para realizar as modificações no software.

2. O experimento restringiu-se à comparação de dois softwares apenas: um desenvolvido com OA e outro com OO.

3. O tamanho e a complexidade limitada dos softwares analisados não permitem que os resultados alcançados no experimento sejam generalizados para outros softwares.

Kulesza et al. [2006] apresentam um estudo de caso que compara a manutenibilidade de software OA e OO. No estudo, foi utilizado um sistema baseado na Web, sendo que uma versão utiliza OO e a outra utiliza OA. As métricas utilizadas na avaliação de manutenibilidade são as mesmas utilizadas por Santa'Anna et al. [2003]. As versões originais implementam 13 casos de uso. O experimento consiste em incluir 8 casos de uso ao sistema, que demandam alterações em classes das quatro camadas. As métricas foram coletadas nas versões originais e nas versões alteradas. Na avaliação de Kulesza et al. [2006], os resultados mostram superioridade da OA em relação à OO nos fatores acoplamento, tamanho e separação de interesses. O contrário ocorre no caso de coesão. Uma causa apontada para isso é que a métrica utilizada para avaliação de coesão, que é uma extensão da métrica LCOM, pode não aferir corretamente esse fator. De fato, conforme discutido na Seção 4.1, o problema de medição de coesão em software OO não está satisfatoriamente solucionado e isso vale para a OA também, já que a métrica para avaliação de coesão neste paradigma baseia-se em LCOM.

Avaliação

Os dois estudos discutidos nesta seção baseiam-se em métricas de fatores como acoplamento, coesão e tamanho para avaliar a manutenibilidade de software OA. Não avaliam, por exemplo, o tempo necessário para realizar as manutenções. Outro fato importante a ser destacado é que as métricas utilizadas avaliam componentes do software, tais como classes e aspectos, e não o software como um todo.

4.3 Reestruturação de Software

Refatoração de software (do inglês *software refactoring*) foi introduzida por Fowler [1999] como a alteração da estrutura do software sem afetar o seu comportamento com o objetivo de melhorar a sua estrutura. O processo de realização de uma refatoração consiste em identificar qual parte do software será reestruturada, escolher uma refatoração apropriada como solução e aplicá-la. Identificar pontos de necessidade de refatoração em software, sobretudo nos de grande porte, pode ser uma tarefa inviável. Métricas de software têm sido utilizadas para este propósito.

Pizka [2004] descreve um estudo de caso em que utilizou métricas de software e ferramentas para auxiliar a reestruturação de um software comercial com o objetivo de melhorar sua estrutura. O ambiente de manutenção do software apresentava problemas tais como: instabilidade de requisitos, pressões sofridas pela equipe em relação a prazos para execução das tarefas de manutenção, ausência de documentação e pouca experiência dos desenvolvedores. De acordo com Pizka, o código tornou-se um *(hyper)spaghethi-code*. Motivado por isso, o responsável pelo software aderiu ao experimento, que foi realizado durante 5 meses. Foram realizadas as seguinte tarefas no experimento: identificação de problemas no código utilizando cinco ferramentas de coleta de métricas; seleção e realização das reestruturações necessárias; avaliação dos benefícios das refatorações. Dentre as métricas utilizadas estão LCOM, RFC e CBO, do conjunto CK [Chidamber & Kemerer, 1994]. Os problemas identificados durante o experimento estão relacionados com a dificuldade de se utilizar as ferramentas e o grande tempo que elas demandam para realizar a coleta das métricas. As medidas obtidas para o software mostraram-se insuficientes para identificar necessidades de reestruturações, o que demandou análise subjetiva do código. Algumas das ferramentas utilizadas auxiliam na refatoração automática de software, porém, na avaliação de Pizka [2004], não são suficientemente amadurecidas para esta tarefa. Pizka descreve que, após o experimento, o gerente, os desenvolvedores e ele próprio consideram que não houve melhoria significativa na estrutura do software, porém esta avaliação não foi suportada por dados numéricos.

Marinescu[3] (2002 apud Munro [2005]), em sua tese de doutorado, utilizou métricas de software para definir estratégias para identificar 14 tipos de problemas de desenho e classes que demandam refatorações em código fonte de software OO. Entretanto, como analisa Munro [2005], a escolha das métricas não foi satisfatoriamente justificada. O trabalho de Munro [2005] estende o de Marinescu. Munro utilizou métricas de soft-

[3]Marinescu, R.. *Measurement and Quality in Object-Oriented Design*. Tese de doutorado. Universidade de Timisoara. Outubro de 2002.

ware para identificar automaticamente *bad smell* em código fonte de software OO. Um *bad smell* é informalmente definido por Fowler [1999] como problemas de desenho em software, por exemplo, uma classe pouco coesa. Em seu trabalho, Munro seleciona um conjunto de métricas de software que podem ser utilizadas para identificar um sub-conjunto de *bad smells* em código fonte Java. O arcabouço proposto por ele é constituído por:

Nome do *bad smell*: a descrição informal do problema, definida por Fowler [1999].

Processo de medição: descrição das técninas de medição que podem identificar o problema em código Java.

Interpretação: indica um conjunto de regras a serem aplicadas para definir como as métricas podem ser utilizadas para identificar os candidatos à refatoração.

Munro [2005] apresenta a sua proposta para dois *bad smells*: *Lazy Class*, caracterizada por ser uma classe que não realiza tarefas relevantes e por isso deve ser eliminada; campo temporário, caracterizado como uma variável de instância que não representa uma propriedade legítima de um objeto. A proposta foi avaliada a partir de dois estudos de caso: um com um software de 1500 linhas de código, 13 classes e 124 métodos, e o outro com um software de de 16000 linhas de código, 84 classes e 730 métodos. Embora a pesquisa tenha mostrado resultados que indicam a eficiência do uso de métricas com o propósito de identificar pontos de refatoração em sofware, ainda há pontos importantes a serem estudados, como:

- estender a proposta para os demais problemas de desenho de classe;
- realizar estudos de caso em sistemas de grande porte;
- investigar se há uma métrica, ou um conjunto de métricas, que possa ser utilizado na identificação dos problemas de desenho de classes.

Avaliação

Reestruturação de software é essencial para amenizar a deterioração de softwares. Um dos desafios nessa tarefa é identificar as partes do software que demandam melhorias. Métricas de software têm sido utilizadas para isso. Porém, esse parece ser um campo ainda pouco amadurecido. O estudo de Pizka [2004] ressalta dois problemas importantes: há dificuldades com o uso efetivo de ferramentas durante o processo de

reestruturação de software; o uso das métricas geralmente utilizadas com esse propósito, como as do conjunto CK [Chidamber & Kemerer, 1994], não são eficientes para identificar necessidades de reestruturação, o que pode ser causado pelas próprias definições das métricas ou pela forma como as ferramentas as computam.

4.4 Predição de Falhas de Sistemas

Alguns trabalhos têm sido realizados com o objetivo de identificar meios de predição de falhas em software [Gyimothy et al., 2005; Nagappan et al., 2006; Li et al., 2006; Schröter et al., 2006; Zimmermann & Nagappan, 2008]. Esta seção discute dois desses trabalhos.

O trabalho de Nagappan et al. [2006] tem por objetivo responder o que leva um software a falhar, identificando fatores que possam ser utilizados como instrumentos de predição de falha para um grupo amplo de softwares. O termo *falha*, aqui, refere-se a um erro observado no comportamento do software. Esse trabalho parte das seguintes hipóteses:

1. o aumento no valor das métricas de uma entidade do software, como um módulo, um arquivo ou algum outro componente, está relacionada ao número de falhas nesta entidade do software;

2. há um conjunto de métricas para o qual a Hipótese 1 se aplica a todos os softwares;

3. há uma combinação de métricas que indicam as falhas de novas entidades introduzidas no software;

4. indicadores obtidos a partir de tais métricas em um projeto podem ser utilizados na predição de falhas de entidades de outros softwares.

Visando atingir o objetivo do trabalho, Nagappan et al. realizaram um estudo empírico baseado na coleta de métricas em códigos fontes de cinco softwares produzidos pela Microsoft, desenvolvidos no paradigma orientado por objetos, nas linguagens C++ e C#. Foram coletadas métricas como: número de linhas executáveis em cada método, número de métodos em uma classe, número de superclasses de uma classe, número de classes acopladas a uma classe, entre outras. Os resultados foram comparados com uma base de dados históricos de falhas nos referidos softwares. Os resultados desse estudo foram:

1. para cada software analisado, foi encontrado um conjunto de métricas correlacionadas com as falhas no software, o que confirma a primeira hipótese;

2. a segunda hipótese não foi confirmada, pois os resultados dos experimentos não evidenciam um conjunto comum de métricas que possa ser utilizado como predição de falhas em todos os projetos;

3. a terceira hipótese foi confirmada, pois verificou-se que os indicadores obtidos de um componente principal podem ser utilizados na construção de modelos de regressão para estimar falhas em novas entidades;

4. a quarta hipótese foi parcialmente confirmada com os experimentos, pois verificou-se que os indicadores obtidos em determinado software podem ser utilizados apenas na predição de falhas em softwares similares e não em qualquer software.

De acordo com Nagappan et al. [2006], o trabalho deles avança o estado da arte nos seguintes pontos: é um dos primeiros trabalhos a mostrar como construir indicadores para falhas em software a partir da análise de dados históricos de falhas no software; investiga se as métricas de software orientado por objetos podem ser utilizadas como instrumento de predição de falhas em software; analisa se indicadores obtidos em um software podem ser utilizados para outros softwares; é um dos mais amplos estudos realizados com softwares comerciais, em relação a tamanho do código fonte, tamanho da equipe envolvida na produção de software e quantidade de usuários dos softwares. No entanto, dentre as métricas coletadas, não estão a de *conectividade*, tampouco métricas de coesão interna de módulos.

Gyimothy et al. [2005] realizaram um estudo empírico sobre a predição de falhas em sistemas OO *open source*. O objetivo do trabalho é verificar a utilidade das métricas CK na predição de falhas em sistemas OO. Como estudo de caso, eles utilizaram um banco de dados de falhas detectadas no navegador Mozilla desde a sua versão 1.0 até a 1.7. Os valores das métricas CK foram comparados aos dados de falhas do Mozilla, o que levou a concluir que a métrica CBO Chidamber & Kemerer [1994] parece ser a melhor métrica para predição de falhas.

Avaliação

A possibilidade de se realizar predição de falhas em software seria um poderoso recurso para a obtenção de software mais confiável. A abordagem empregada em alguns dos trabalhos nesse tópico é investigar correlação entre métricas de software e falhas no software. O trabalho de Nagappan et al. [2006], por exemplo, é um dos mais significativos nesta linha, porque avalia softwares comerciais amplamente utilizados.

Porém, ainda não se identificou uma métrica, ou um conjunto de métricas, que possa ser utilizada genericamente para a predição de falhas de software.

4.5 Manutenibilidade de Software

A norma ISO 9126 é um modelo para avaliação de qualidade de software que é tomado como padrão internacional. O modelo especifica seis características internas e externas de qualidade de software:

- Funcionalidade: característica do software que atende os seus requisitos.
- Confiabilidade: característica de um software que provê seus serviços em condições e período de tempo determinados.
- Usabilidade: facilidade de aprendizado, entendimento, utilização e atratividade de um software.
- Eficiência: refere-se ao bom uso que o software faz de recursos de sistema, tais como rede e memória.
- Portabilidade: facilidade com que o software é capaz de se adaptar à mudanças em seu ambiente.
- Manutenibilidade: facilidade de se realizar modificações em um software. Modificações compreendem alterações, correções, adaptações e inclusões de funcionalidades.

Para cada uma dessas características de qualidade, são definidas subcaracterísticas. Para manutenibilidade, a ISO 9126 define as seguintes subcaracterísticas:

- Analisabilidade: facilidade de se identificar as partes do software que deverão ser modificadas em decorrência de uma manutenção.
- Modificabilidade: facilidade de se realizar modificações no software.
- Estabilidade: capacidade de o software evitar efeitos não esperados em decorrência de uma modificação.
- Testabilidade: facilidade de se testar um software modificado.

A ISO 9126 indica métricas internas e externas para avaliar cada uma dessas subcaracterísticas. As métricas externas avaliam aspectos relacionados ao comportamento da equipe de manutenção, dos usuários e do sistema durante a atividade de manutenção. Um exemplo de métrica externa para *modificabilidade* é o *tempo necessário para implementação de alteração*, que indica a facilidade de se alterar o software para solucionar um problema identificado pelo usuário. O tempo de alteração é dado pelo tempo decorrido entre a identificação das causas do problema e remoção dessas causas. A métrica é dada pela razão entre o somatório de tempos de alteração e o número total de falhas registradas e removidas. As métricas internas avaliam aspectos relacionados especificamente ao produto. Um exemplo de métrica interna para *estabilidade* é a *localização do impacto da modificação*, que indica a amplitude de impacto de uma modificação do produto, sendo dada pela razão entre o número de itens afetados pela modificação e o total de itens.

Avaliação

A norma ISO 9126 é um padrão de grande importância e disseminação internacional para a qualidade de software. Ela define, entre outros aspectos, as características, subcaracterísticas e suas respectivas métricas internas e externas. No caso da manutenibilidade de software, as métricas de avaliação abrangem aspectos relativos a existência ou não de fatores considerados importantes para a atividade de manutenção, por exemplo registro de *log* de operações, funções de teste, ou análise dos dados de registros de falhas e suas soluções. No entanto, essa norma não aborda, de forma específica, aspectos relacionados à avaliação da estrutura do software, tais como modularidade, coesão e acoplamento.

4.5.1 Modelos de Avaliação de Manutenibilidade

Esta seção discute trabalhos relacionados ao problema de predição de manutenibilidade de software. Os trabalhos estão agrupados segundo a abordagem utilizada por eles: utilização de métrica ou avaliação de características de manutenibilidade.

4.5.2 Modelos Baseados em Métrica

Índice de Manutenibilidade - MI

Uma das métricas mais conhecidas para avaliação de manutenibilidade de software é MI (*Maintainability Index*) [Ash et al., 1994; Pearse & Oman, 1995]. A métrica é reconhecida e recomendada pelo *SEI - Software Engineering Institute* [SEI, 2009], que a considera como um recurso importante no ciclo de vida de software, uma vez que permite ao desenvolvedor avaliar a dificuldade de manutenção e, então, intervir na estrutura do software a fim de reduzi-la. A proposta inicial desta métrica foi realizada por Ash et al. [1994] e, posteriormente foi aperfeiçoada como resultado de um trabalho conjunto do SEI, da Hewlett-Packard e de outras organizações. A métrica MI é dada pela seguinte expressão:

$171 - 5.2 * ln(aveV) - 0.23 * aveV(g') - 16.2 * ln(aveLOC) + 50 * sin(sqrt(2.4 * perCM))$, onde:

- $aveV$ é a média do Volume de Halstead por módulo,
- $aveV(g')$ é a média da complexidade ciclomática por módulo,
- $aveLOC$ é a média do número de linhas de código por módulo,
- $perCM$ é a média do percentual de linhas de comentários por módulo. O termo $perCM$ é opcional na fórmula.

O *volume de Halstead* avalia a complexidade de módulos de programas por meio da análise de operandos e operadores. *Volume de Halstead* é dado por $V = N * log_2 N$, onde N é o tamanho do programa, que corresponde ao total de operandos e de operadores.

A *complexidade ciclomática* é uma métrica para avaliar complexidade de código que contabiliza o número de caminhos de execução em um módulo de um programa, cujo fluxo de execução é representado por um grafo. A métrica é dada por $CC = E - N + p$, onde E é o número de arestas no grafo, N é o número de nodos no grafo e p é o número de componente conectados no grafo.

No polinômio da métrica MI, os coeficientes foram calibrados de forma empírica a partir da análise de sistemas na Hewlett-Packard. A metodologia empregada para a definição dos parâmetros da métrica foi o seu teste em softwares totalizando cerca de

50 KLOC e a verificação dos resultados a partir da comparação com dados subjetivos obtidos por meio de questionários. Os resultados desse primeiro estudo foram validados por um segundo, com um software de metade do tamanho do primeiro. Em um estudo posterior com softwares da US Air Force, escrito inicialmente em FORTRAN e traduzido para C, verificou-se que o uso de MI favoreceu a redução do esforço de manutenção.

Inicialmente, o autor da métrica desenvolveu um protótipo de uma ferramenta para coleta de medidas de MI em softwares escritos em Pascal e C. Como os resultados de seus estudos foram considerados sólidos, essa métrica passou a ser utilizada na construção de sistemas do Departamento de Defesa Americano. SEI recomenda as seguintes situações para o uso dessa métrica: verificação periódica da manutenibilidade de software; uso da métrica durante o processo de desenvolvimento do software; identificação de código que representa risco para a manutenção; comparação entre softwares.

O uso de métrica de software deve ser um instrumento de tomada de decisão. Diante de um valor não apropriado de uma métrica, deve ser possível intervir no software a fim de melhorar o aspecto avaliado. O uso da métrica MI não favorece isso, pois MI é computada a partir de valores de LOC, complexidade ciclomática e da métrica Halstead. Pode ser difícil para o gerente ou para o desenvolvedor intervir nestes aspectos e melhorar a manutenibilidade. Outro ponto importante é que esta métrica foi proposta e validada para software estruturados, o que não necessariamente indica que ela seja aplicável a software orientado por objetos.

Um Estudo sobre Correlação entre um Conjunto de Métricas e Manutenibilidade

Li & Henry [1993] utilizaram métricas de software OO para avaliar manutenibilidade. O estudo foi realizado com software escrito em Ada. Os dados de esforço de manutenção e métricas OO foram coletados em dois softwares comerciais durante três anos. A métrica considerada como indicador de manutenibilidade foi o número de linhas de código alteradas por classe. O objetivo do estudo foi identificar a relação entre esforço de manutenção e métricas de software. Dentre as métricas coletadas estão: DIT (profundidade na árvore de herança), NOC (número de filhos), LCOM, RFC (conjunto resposta de uma classe) e WMC (métodos ponderados por classe) [Chidamber & Kemerer, 1994]. O estudo teve como conclusões que a avaliação de manutenibilidade, tal como foi considerado, é possível a partir das métricas utilizadas. Contudo, uma observação sobre esse estudo é que a métrica número de linhas de código

alteradas pode não corresponder fielmente à dificuldade de manutenção do software.

Um Estudo sobre Correlação entre Coesão e Modificabilidade

Kabaili et al. [2001] investigaram a correlação da coesão com a modificabilidade (do inglês *changeability*) de software orientado por objetos, partindo da afirmativa de Yourdon e Constantine que a coesão pode ser usada na predição de reusabilidade e manutenibilidade[4] (1979 apud Kabaili et al. [2001]). Modificabilidade é definida pelos autores como a capacidade de um software absorver uma modificação, sendo determinada pelo impacto de uma modificação, ou seja, o conjunto de classes que necessitam de modificação ou correção em decorrência de uma dada modificação no sistema. Para validar a hipótese, foram utilizadas duas métricas de coesão em três sistemas industriais desenvolvidos em C++. Dos 66 tipos de modificações possíveis identificados pelos autores, foram analisados 6: alteração de tipo de variável, alteração de modificador de acesso de atributo de público para protegido, alteração da assinatura de método, alteração do modificador de acesso de classe de público para protegido, e adição de classe abstrata na hierarquia de classes. Inicialmente foram coletadas as métricas de coesão nos sistemas. Para cada um dos tipos de alterações, em cada um dos sistemas, determinou-se o conjunto de classes às quais a alteração era aplicável. Para cada uma destas classes, calculou-se o número de classe que sofreram impacto. Com as métricas e os dados de impacto de alteração coletados, verificou-se a correlação entre coesão e impacto de alteração. Os resultados do experimento não comprovaram a hipótese do trabalho, ou seja, não foi identificada correlação entre coesão e modificabilidade. Os autores consideram que um dos seguinte fatores contribuiram para isso: ou as métricas escolhidas não avaliam coesão corretamente, ou de fato não existe correlação entre coesão e modificabilidade.

Um Modelo de Predição de Estabilidade de Classes

Grosser et al. [2003] apresentaram um modelo de predição de estabilidade de classes em software desenvolvido em Java. O modelo baseia-se em comparar versões de um software para aferir sua estabilidade. Métricas de software são utilizadas no modelo para identificar componentes de software similares. Os autores acreditam que

[4]Edward Yourdon and Lany L. Constantine. *Structured Design*. Prentice Hall, Englewood Cliffs, N.J., 1979.

dois componentes de software similares evoluirão da mesma maneira, ou seja, têm estabilidade similares. O conceito de instabilidade empregado no modelo é definido como o aumento de responsabilidades de uma classe ao longo da vida do software. Estabilidade é avaliada no nível de classe e entre versões do mesmo software. Para realizar esta avaliação, uma classe é representada por uma tupla: $classe = (nome, m_1, m_2, ..., m_n, S_t, e)$, onde:

nome: é a identificação da classe;

m_i: é uma métrica de algum aspecto da classe. No trabalho, foram consideradas 14 métricas que avaliam os seguintes aspectos: coesão, conectividade, herança e complexidade.

S_t: é o chamado *fator de estresse* da classe, definido pelos autores como os fatores que têm impacto na estabilidade da classe. Foram identificados os seguintes fatores de estresse: modificações locais da classe em decorrência da definição de novos métodos; modificações nas classes ancestrais da classe; modificações nas classes descendentes da classe; modificações nas classes às quais a classe está conectada: aquelas que dependem da classe ou das quais a classe depende. Seja $I(c_i)$ a interface da classe na versão i do software, e $I(c_{i+1})$, a interface da classe na versão seguinte do software. Considera-se como interface o conjunto dos métodos públicos e protegidos, locais ou herdados da classe. O fator de estresse S_t é aproximado pelo percentual de métodos adicionados na classe entre duas versões. Formalmente, $S_t(c_i, c_{i+1}) = (I(c_{i+1}) - I(c_i))/(I(c_{i+1})$.

e: é o indicador de estabilidade da classe. O nível de estabilidade da classe é medido pela razão entre $I(c_i)$ e $I(c_{i+1})$, ou seja, é o percentual de $I(c_i)$ em relação a $I(c_{i+1})$.

A hipótese do trabalho é que a estabilidade de uma classe depende de sua estrutura e do *estresse induzido* pela implementação de novos requisitos entre duas versões diferentes do software. A avaliação da estrutura da classe é realizada por meio de métricas de software. Para exemplificar o emprego do modelo proposto, foi realizado um estudo de caso com quatro versões do JDK.

O modelo de estabilidade proposto por Grosser et al. [2003] tem como base a ideia de que dois componentos de software similares evoluirão da mesma maneira. Essa premissa foi utilizada no modelo proposto por eles para estimar a estabilidade de classes em software orientado por objetos a partir da comparação de uma dada classe com classes que sejam similares e que já tenham sido avaliadas anteriormente. Embora

esse método possa ser sólido, o trabalho deixa em aberto ainda os seguintes pontos a serem explorados:

- O modelo avalia estabilidade de classes, mas não a de um sistema como um todo ou de um conjunto de classes. Ele poderia ser estendido para permitir esse nível de avaliação.
- Métricas são utilizadas como parâmetro de comparação entre classes. O trabalho, porém, não investiga correlação entre as métricas utilizadas. Havendo correlação entre as métricas, seria possível reduzir o número de métricas utilizadas.
- O trabalho propõe um modelo de estabilidade, mas não destaca a sua utilidade no processo de software. O modelo proposto não recomenda ações a serem tomadas diante de uma alta instabilidade das classes. Uma das possíveis utilizações desse resultado passível de investigação é a identificação de um conjunto de métricas correlacionadas à estabilidade e sua utilização para reestruturação de software.
- Dentre os resultados apontados pelos autores, está o de que a estabilidade de classes mostrou-se ser predizível. Porém esta afirmativa não é satisfatoriamente sustentada pelo trabalho, uma vez que foi realizado apenas um estudo de caso. Para uma comprovação desse resultado, seria necessário um estudo mais amplo.

Um Modelo de Avaliação de Manutenibilidade por meio da Conectividade do Software

Ferreira et al. [2008] abordam a conectividade como fator principal na determinação da manutenibilidade de software. O trabalho tem as seguintes contribuições principais:

1. Avaliação da relação entre conectividade e estabilidade de sistemas: uma análise sobre o impacto da conectividade na estabilidade de sistema é apresentada. Tal análise destaca a conectividade como fator principal na determinação da estabilidade de software, baseando-se no Modelo das Lâmpadas proposto por Myers [1975]. Neste modelo, Myers representa um software como um conjunto de lâmpadas conectadas entre si, no qual cada lâmpada representa um módulo. A mudança de estado de uma lâmpada de desligada para ligada representa atividade de manutenção no módulo correspondente e afeta o estado das demais lâmpadas conectadas a ela.

2. Análise crítica do Modelo de Estabilidade de Software proposto por Myers: Myers propõe uma métrica que indica a quantidade média de módulos que sofrerão impacto em decorrência de uma alteração em um módulo qualquer no software. No trabalho, é apresentado e analisado este modelo de Myers, e são propostas adaptações de tal métrica à orientação por objetos, visto que a métrica foi inicialmente descrita por Myers para o paradigma estruturado.

3. Adaptação dos conceitos de acoplamento e coesão à luz da orientação por objetos: visto que coesão e acoplamento são determinantes na conectividade de software, realizou-se uma releitura desses conceitos para a orientação por objetos, classificando e exemplificando tipos de coesão e acoplamento em software orientados por objetos.

4. Identificação das métricas de software orientado por objetos impactantes no aspecto conectividade de sistema: é apresentada uma revisão bibliográfica das principais métricas de software orientado por objetos propostas na literatura, identificando-se aquelas a serem utilizadas na avaliação do aspecto de conectividade.

5. Proposta de duas métricas auxiliares na avaliação da conectividade: *conexões aferentes*, que indica o número de conexões que chegam a uma classe, e *peso de conexão aferente*, que corresponde ao grau de acoplamento envolvido em determinada conexão que chega a uma classe.

6. Proposta do Modelo de Conectividade em Sistemas Orientados por Objetos (MACSOO): é um método 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.

7. Experimentos de avaliação de conectividade que resultaram em indícios de que a conectividade pode ser tomada como um indicador na avaliação da manutenibilidade de software.

O trabalho deixa os seguintes pontos a serem explorados:

1. A Métrica de Estabilidade de Myers, utilizada em MACSOO, avalia o impacto de uma alteração qualquer em determinado módulo do software. O modelo, porém, não é capaz de estimar o impacto quando modificações são inicialmente realizadas em mais de um módulo.

2. MACSOO constitui-se de avaliação de fatores de impacto na conectividade e da própria conectividade por meio de métricas. Entretando, o modelo não indica os valores a serem considerados como satisfatórios para os indicadores obtidos.

4.5.3 Modelos Baseados em Características de Manutenibilidade

Mapeamento de Propriedades de Código em Características de Manutenibilidade

Heitlager et al. [2007] definem um modelo de medição de manutenibilidade que baseia-se no mapeamento de propriedades do código, tais como volume e complexidade, em características do sistema relacionadas à manutenibilidade, como estabilidade e testabilidade. O modelo tem por objetivo favorecer um meio mais prático do que aquele obtido pela métrica MI.

O conceito de *unidade* é definido pelos autores como a menor parte de um software que pode ser executada e testada individualmente. As propriedades do código podem ser avaliadas por meio de métricas. *Volume* corresponde ao tamanho do software e pode ser medido por métricas como LOC e pontos de função. *Complexidade* não é um conceito bem definido na produção de software; a métrica indicada pelos autores para avaliar complexidade é *a complexidade ciclomática*. Heitlager et al. [2007] consideram o *tamanho da unidade* um fator devido à premissa de que unidades maiores tendem a ser mais difíceis de manter. A existência de *testes de unidade* é considerada também um fator importante para a manutenibilidade do código no modelo.

O modelo propõe que a *analisabilidade* pode ser avaliada a partir das seguintes propriedades do código: volume, duplicação de código, tamanho da unidade e teste de unidade; *modificabilidade* pode ser avaliada a partir da complexidade por unidade e da duplicação de código; *estabilidade* pode ser avaliada a partir da existência de testes de unidade para o software; e *testabilidade* pode ser avaliada a partir de complexidade por unidade, tamanho da unidade e existência de testes de unidade.

Embora esse modelo seja mais simples do que a métrica MI, eles têm abordagens diferentes: MI é uma métrica representada por uma única expressão, enquanto o modelo proposto subdivide características relacionadas a manutenibilidade e as avalia de forma separada, por meio de várias métricas. Nesse modelo, considera-se que o tamanho do software ou de suas unidades é um fator determinante para a manutenibilidade. Contudo, essa premissa parece ser uma análise superficial sobre esse aspecto, pois

pode ser que um software pequeno seja de difícil manutenção se, por exemplo, sua construção não tiver priorizado aspectos como baixo acoplamento entre módulos e alta coesão interna dos módulos. Além disso, a existência de testes de unidade é considerada no modelo uma propriedade de impacto na analisabilidade, estabilidade e testabilidade do software. Porém, diante de uma modificação no software, os seus testes de unidade poderão também sofrer alterações, e a facilidade de se realizar tais alterações depende da forma como o código a ser testado foi construído.

Um Modelo Bidimensional de Manutenibilidade Baseado em Atividades

Deissenboeck et al. [2007] propõem um modelo bidimensional de manutenibilidade baseado em atividades. O modelo considera que para alcançar manutenibilidade de software é preciso relacionar as propriedades do software e as atividades realizadas na manutenção. No modelo, a manutenção possui as seguintes atividades: conceito, análise de impacto, codificação e modificação. A atividade de análise de impactos, por exemplo, está relacionada às seguintes propriedades do código: concorrência, recursão, clonagem de código, e utilização de um depurador. Embora o modelo indique quais características do software estão relacionadas às atividades de manutenção identificadas, ele não determina como avaliar tais características. Além disso, o modelo não apresenta uma definição das propriedades do software. Por exemplo, o modelo define que a propriedade *formato do código* tem impacto na atividade de codificação, mas não define o que é considerado como formato do código.

4.5.4 Propagação de Modificações

Modificabilidade, a facilidade de se realizar modificações em um software, é uma característica importante da manutenibilidade, especialmente em ambientes nos quais modificações em software são frequentemente realizadas. O processo de *propagação de modificações* é de grande relevância na gestão de modificações em software. *Propagação de modificações* refere-se ao processo em que uma modificação em determinado elemento ou grupo de elementos de um software ocasiona modificações em outros elementos sucessivamente.

Alguns trabalhos têm sido realizados com o objetivo de criar recursos que possam auxiliar a avaliação e a estimativa de propagação de modificações em software. Diferentes abordagens são adotadas nesses trabalhos. As principais delas são baseadas

na análise de dependência entre os módulos do software, na análise de dados históricos sobre modificações realizadas no software e em simulação.

Um dos primeiros trabalhos nessa área foi o de Myers [1975], que define um modelo de estabilidade de programas estruturados. Ferreira [2006] realizou uma adaptação desse modelo ao paradigma orientado por objetos. Para cada módulo do programa, o modelo estima quantos outros módulos serão modificados em decorrência da modificação realizada no módulo. A partir desse resultado dado por módulo do programa, o modelo estima o número de módulos que serão modificados, em média, quando um módulo qualquer do software sofrer modificação. Esse modelo considera os impactos diretos e indiretos entre módulos, e realiza sua estimativa com base na avaliação de coesão interna dos módulos e do grau de acoplamento entre módulos.

O modelo definido por Chaumun et al. [1999] visa avaliar o impacto de modificações em software orientado por objetos. O modelo consiste em contar o número de classes que são diretamente impactadas por uma modificação realizada em uma determinada classe do software. Eles realizaram um estudo de caso no qual o modelo foi avaliado para um tipo de modificação específica: a modificação de assinatura de método. O software usado no estudo de caso foi desenvolvido em C++ e tem 1044 classes. O objetivo do estudo de caso foi investigar a existência de correlação entre o modelo proposto e o *número de métodos* de uma classe, tendo sido identificada fraca correlação entre essas duas métricas. Eles concluíram que as observações desse estudo de caso mostram que uma modificação na assinatura de um método gera impacto direto em não mais de uma classe.

Mirarab et al. [2007] propõem um modelo para gerar uma medida que representa as chances de determinado módulo ser modificado dado que um determinado conjunto inicial de módulos sofrem modificações. Esse modelo baseia-se na análise de dependências entre os módulos e na análise dos dados históricos de modificações no software, que tem por objetivo identificar módulos que sofreram modificações simultaneamente. Essa informação é usada para definir a probabilidade de mudança de um módulo A dado que um módulo B é modificado. O modelo consiste na construção de uma Rede Bayesiana, o que, como os próprios autores do modelo avaliam, não é uma tarefa trivial. Para avaliar o modelo, foi realizado um estudo de caso com o software Azureus em que os resultados gerados pelo modelo foram comparados ao histórico de modificações do software. Os resultados do estudo de caso indicaram correlação entre os dados. Embora o modelo proposto por Mirarab et al. [2007] possa ser sólido, o processo envolvido para sua implementação é de alta complexidade. Isso pode ser um fator inibidor para sua aplicação na prática. Além disso, o modelo não possui recursos que permitam a identificação de forma direta dos fatores que contribuem para a taxa de propagação de

modificações no software.

Li et al. [2010] definem um modelo de avaliação de propagação de modificações em software orientado por objetos baseado em simulação. O modelo considera que um software pode ser representado por um grafo ponderado, no qual os vértices são as classes e interfaces do software, e as arestas, os relacionamentos entre elas. Os pesos das arestas são definidos de acordo com o tipo de relacionamento, por exemplo, herança, implementação, agregação e dependência. O peso de uma aresta representa a probabilidade de uma modificação em uma classe afetar a outra. O modelo proposto por eles considera apenas modificações atômicas, i.e, um único módulo é inicialmente modificado e, a partir daí, o modelo simula a propagação de modificações no software. O algoritmo de simulação do modelo é definido da seguinte forma: uma classe é selecionada aleatoriamente; a partir da classe selecionada, as classes adjacentes são iterativamente afetadas, até que nenhuma outra classe seja mais afetada; o resultado da simulação é o número de classes afetadas. Para um dado software, várias simulações são realizadas, sendo que o resultado final reportado pelo modelo é a média dos resultados de cada simulação. Os autores utilizaram o modelo proposto para avaliar cinco versões do software Apache Ant. Considerando que o modelo pode ser usado para avaliar a qualidade da estrutura do software, os autores concluem que a qualidade do software avaliado é suficientemente boa. Contudo, o estudo relatado não avalia se o modelo proposto de fato avalia adequadamente estruturas de software.

O trabalho realizado nesta tese refere-se especificamente ao tópico de propagação de modificações. O modelo proposto no presente trabalho, denominado K3B, visa estimar como um conjunto de modificações propagam-se em um software. O presente trabalho diferencia-se dos trabalhos anteriores nos seguintes aspectos principais:

- O modelo proposto nesta tese consiste em uma expressão matemática cujos parâmetros são métricas de software. Isso permite que o desenvolvedor possa identificar o aspecto do software que possa estar contribuindo para a alta propagação de modificações e, então, atuar em tal propriedade com o objetivo de melhorar a estrutura do software a fim de reduzir o impacto de modificações.

- A estimativa realizada por K3B não se limita a uma única modificação. O modelo tem como entrada o número de módulos que serão modificados e, a partir daí, estima o número de *passos de modificações*.

- Em um processo de modificação, uma classe pode ser modificada mais de uma vez devido às dependências cíclicas que podem existir no software. O modelo K3B não se limita a contar o número de classes que serão afetadas pelo conjunto

de modificações iniciais. O resultado de K3B representa o número de *passos de modificações* estimado, que abrange o número total estimado de vezes que classes do software serão afetadas por modificações.

4.5.5 Avaliação do Estado da Arte

Há alguns anos, vários trabalhos têm sido realizados com o objetivo de avaliar ou estimar a dificuldade de manutenção de software, o que evidencia que embora essa questão seja de grande importância na produção de software, ainda não foi solucionada. Esta seção realizou um estudo dos principais trabalho nesta linha.

Um dos trabalhos mais conhecidos para avaliação de manutenibilidade é a métrica MI [SEI, 2009], adotada pela SEI. Essa métrica foi proposta e validada utilizando-se softwares construídos no paradigma estruturado com linguagens como FORTRAN e C. Porém, softwares orientados por objetos têm características diferentes dos softwares construídos no paradigma estruturado. O trabalho de Li & Henry [1993] identificou correlação entre um conjunto de métricas OO e a manutenibilidade de software, porém a métrica utilizada para estimar esse fator, número de linhas de código alteradas, pode não corresponder fielmente à dificuldade real de manutenção. O trabalho de Kabaili et al. [2001] busca identificar correlação modificabilidade e coesão, utilizando duas métricas para este fator. Os resultados do trabalho não evidenciam tal correlação, segundo os autores, por um dos dois possíveis motivos: ou realmente a coesão não gera impacto na manutenção ou as métricas utilizadas não avaliam coesão satisfatoriamente. Grosser et al. [2003] definem um modelo de predição de estabilidade de classes baseado no pressuposto que duas classes com estrutura semelhantes tendem a evoluir da mesma maneira. Embora o trabalho tenha concluído que é possível realizar esse tipo de predição com a abordagem proposta, é necessária a realização de estudos mais amplos para confirmar essa conclusão. Além disso, o modelo avalia estabilidade de classes e não de um software como um todo. Ferreira et al. [2008] propõem a avaliação de manutenibilidade por meio de sua conectividade, e definem um método de reestruturação de software baseado nisso. Os resultados do estudo evidenciaram a eficiência do método, porém ainda é necessária a realização de experimentos mais significativos para a comprovação dessa conclusão.

Alguns trabalhos para avaliação de manutenibilidade são baseados nas características desse aspecto. Deissenboeck et al. [2007] definem um modelo que associa as propriedades do código e as atividades realizadas na fase de manutenção. O modelo indica quais características devem ser avaliadas, mas não determina como avaliá-las. Heitlager et al. [2007] definem um modelo que mapeia características de código, repre-

sentadas por métricas, em características de manutenibilidade. A ideia do modelo é consistente porque considera os vários aspectos de manutenibilidade. Contudo, a sua aplicação pode não ser muito prática, pois utiliza uma série de métricas isoladas para avaliar a dificuldade de manutenção. Além disso, as métricas utilizadas podem não ser indicadores fieis de manutenibilidade. Por exemplo, o modelo utiliza o tamanho do software como um dos indicadores, e não necessariamente essa métrica indica facilidade de manutenção. Embora esses dois últimos trabalhos sejam baseados em características de manutenibilidade, a utilização deles é dependente do uso de métricas.

O problema de avaliação de manutenibilidade de software está em aberto e parece ser de difícil solução. Uma das causas para esse cenário é que para realizar a predição de dificuldade de manutenção é preciso avaliar o software e suas características. Essa avaliação é idealmente realizada por meio de métricas. Para alguns fatores, como coesão, não há um consenso sobre uma métrica para sua avaliação. Ocorre que há uma grande quantidade de métricas propostas na literatura, mas ainda não se conhecem os valores a serem considerados como satisfatórios para elas. A solução dessa questão tem papel central para o uso efetivo de métricas na produção de software. O segundo ponto é que, embora haja um número significativo de modelos de predição de dificuldade de manutenção de software, não há um modelo que realize isso satisfatoriamente no caso de softwares orientados por objetos. O modelo mais conhecido com o propósito de avaliar manutenibilidade corresponde à métrica MI. Essa métrica foi proposta e validada para o paradigma estruturado, mas não captura características do paradigma OO.

Em particular, os modelos propostos para avaliar propagação de modificações em software limitam-se a contar os número de classes afetadas em decorência de modificação em uma determinada classe do software. Além disso, eles não fornecem uma indicação direta sobre os aspectos estruturais do software que podem contribuir para a alta propagação de modificações. A presente tese tem por objetivo avançar nesse tópico, propondo um recurso para contornar essas dificuldades.

4.6 Conclusão

Este capítulo mostrou uma revisão dos principais trabalhos na área de medição de software orientado por objetos. As principais linhas de pesquisas atuais em métricas de software orientado por objetos foram identificadas. Dentre elas, destacam-se: a pesquisa por uma métrica de coesão padrão, questão ainda em discussão na literatura; medição de software orientado por aspectos; o uso de métricas na reestruturação de

software; o uso de métricas para predição de falhas e manutenibilidade de software. Este último assunto foi enfatizado por se tratar do tema discutido nesta tese.

A busca por uma métrica de coesão padrão justifica-se pela importância da avaliação deste fator para a qualidade estrutural de um software. A maior dificuldade para alcançar uma métrica que avalie fielmente a coesão interna de classe é decorrente da natureza do fator avaliado, pois como a coesão é o grau de relacionamento entre os elementos internos do módulo, a sua avaliação é fortemente dependente do entendimento do domínio do problema do software. Não há consenso na literatura sobre a métrica de coesão interna de módulos na OO. As principais abordagens utilizadas na definição de métricas para este fator são:

- a coesão interna de uma classe é definida em função do grau de similaridade entre seus métodos, o que é analisado a partir do uso de variáveis de instâncias comuns entre os métodos. Esta abordagem é empregada na métrica LCOM;

- uma classe é considerada coesa se seus métodos usam o mesmo conjunto de tipos de parâmetros. Esta abordagem é utilizada no cálculo das métricas CAMC e NHD;

- a coesão é avaliada a partir da análise de informações semânticas no código fonte, tais como comentários. O cálculo da métrica utiliza técnicas de recuperação de informação. Esta abordagem é empregada na métrica C3.

- a coesão de uma classe é avaliada pela maneira como as suas classes clientes utilizam seus serviços. Esta abordagem é empregada na métrica ELCOM.

Nesta área de pesquisa é necessário definir melhor o que é coesão interna de uma classe. Dado o fato que a avaliação de coesão interna de módulos depende do entendimento do problema tratado pelo software, as métricas propostas para este aspecto devem ser validadas em relação à avaliação qualitativa de especialistas. Outra questão de grande relevância ainda não solucionada é o valor a ser considerado satisfatório para as métricas. Para estas duas questões em aberto, faz-se necessário a realização de experimentos extensivos.

Identificar pontos de necessidades de refatoração em sistemas de grande porte pode ser uma tarefa inviável. Com o objetivo de solucionar este problema, a utilização de métricas de software para a reestruturação de software tem sido investigada. Os trabalhos nesta linha baseiam-se em definir um conjunto de métricas que permite identificar automaticamente um *bad smell* em um código de software. Embora esses

trabalhos tenham encontrado resultados significativos, as seguintes questões ainda precisam ser exploradas: os estudos precisam ainda ser validados em sistemas de grande porte; vale investigar se há um fator, ou um conjunto de fatores, que esteja relacionado a todos os problemas de desenho de classes.

O alto custo da fase de manutenção de software é fato que motiva vários estudos que têm como objetivo encontrar soluções que reduzam custos e esforços nesta fase bem como meios de predição deste esforço. Métricas de software orientado por objetos têm sido investigadas como instrumento de avaliação de manutenibilidade de software. A principal proposta de solução desse problema é a métrica MI. Essa métrica foi proposta e validada para softwares implementados no paradigma estruturado. Os estudos realizados nesta área ainda não identificaram de forma conclusiva um fator, ou um conjunto de fatores, que possa ser utilizado na predição de dificuldade de manutenção em software orientado por objetos. A solução desta questão corresponde a um instrumento poderoso na área de produção de software, pois permitiria ao desenvolvedor realizar uma análise prévia do esforço de manutenção de um software e, mediante esta análise, atuar na estrutura do software de maneira a reduzir seu custo de manutenção. As grandes dificuldades nessa linha de pesquisa são: não há um consenso sobre a melhor forma de se avaliar a manutenibilidade, por exemplo, há estudos que realizam esta avaliação a partir do número de linhas alteradas por classe e outros como o tempo necessário para realizar uma manutenção; a realização de um estudo conclusivo depende de experimentos em larga escala preferencialmente com sistemas reais.

Outra questão relevante ainda não solucionada é a identificação de valores a serem considerados satisfatórios para as métricas. Na literatura consta uma imensa quantidade de métricas de software, porém não há estudos que indiquem os valores considerados apropriados para elas. A indicação desses valores é essencial para o uso efetivo de métricas de software, pois sem esta informação não é possível a tomada de decisão apropriada a partir do uso das métricas.

Diante dos problemas identificados, esta tese tem por objetivos: a definição de um modelo de predição da amplitude de propagação de modificações em software orientado por objetos, a definição de uma métrica de avaliação de coesão interna de classes e a realização de um estudo para identificar valores referência para as métricas utilizadas com o modelo proposto. Para orientar e dar suporte às premissas assumidas na definição do modelo, foi conduzido um estudo sobre as estruturas de software e como elas evoluem. Os capítulos seguintes apresentam os estudos realizados e o modelo proposto nesta tese.

Capítulo 5

Valores Referência para Métricas de Software Orientado por Objetos

Métricas de software permitem a medição, avaliação, controle e melhoria de produtos e processos de software. Dezenas de métricas têm sido propostas e validadas, e um grande número de ferramentas de coleta de métricas têm sido desenvolvidas [Fenton & Neil, 2000; Xenos et al., 2000; Baxter et al., 2006; Kitchenham, 2009]. Apesar da importante contribuição desses trabalhos, o uso efetivo de métricas na Engenharia de Software é inibido pela falta de conhecimento sobre seus valores referência. Alguns estudos têm sido realizados com o objetivo de derivar esses valores [Lanza & Marinescu, 2006]. Todavia, esses trabalhos não são apropriadamente baseados nas propriedades estatísticas dos dados analisados. Além disso, eles não se concentram na investigação de valores referência para métricas de software orientado por objetos especificamente.

O objetivo do estudo apresentado neste capítulo é a identificação de valores referência para um conjunto de métricas de software: LCOM, DIT, COF, número de métodos públicos e número de atributos públicos. Estas métricas foram escolhidas porque são relacionadas a importantes fatores de qualidade de software, tais como coesão, acoplamento e ocultação de informação. Por esta razão, elas podem ser associadas com o uso de K3B para avaliar software do ponto de vista de modificabilidade, auxiliando na identificação de necessidades de melhorias no software direcionadas ao aumento da manutenibilidade do software.

Foi realizado um estudo sobre a estrutura de uma grande quantidade de softwares abertos desenvolvidos em Java, com diferentes tamanhos e domínios de aplicação. Os valores referência foram derivados a partir da análise das propriedades estatísticas dos dados. Eles são baseados na identificação dos valores mais comumente utilizados na

prática. Os resultados dessa análise mostram que os valores de cinco das métricas estudadas são modelados por uma distribuição de cauda pesada (*heavy-tailed distribution*), o que significa que não há um valor típico para essas métricas. Este capítulo apresenta em detalhes o método usado para derivar os valores referência de tal forma que ele possa ser replicado para derivação de valores referência para outras métricas de software futuramente. Foram realizados quatro tipos de análise: com o conjunto de dados completo, o que levou à identificação de valores referência gerais; por tamanho de software em número de classes; por domínio de aplicação; e por tipo de software: ferramenta, biblioteca e *framework*. Os valores referência foram avaliados por meio de dois estudos de caso. Os resultados dos estudos de caso indicam que os valores referência são úteis para auxiliar a avaliar softwares adequadamente.

O restante deste capítulo é organizado da seguinte forma: a Seção 5.1 discute alguns trabalhos relacionados; a Seção 5.2 provê embasamento teórico sobre os conceitos aplicados à análise dos valores das métricas e descreve a metodologia usada nesta pesquisa; a Seção 5.3 apresenta os resultados do estudo; a Seção 5.4 identifica os valores referência para as métricas; a Seção 5.5 avalia os valores referência por meio de dois estudo de caso; a Seção 5.6 discute as limitações e as ameaças para a validade do estudo. As conclusões são apresentadas na Seção 5.7.

5.1 Trabalhos Relacionados

Dezenas de métricas de software têm sido propostas [Abreu & Carapuça, 1994; Chidamber & Kemerer, 1994; Xenos et al., 2000; Kitchenham, 2009]. Apesar do esforço para definir e avaliar métricas de software, há ainda grandes desafios nesta área de pesquisa. A interpretação apropriada dos valores das métricas é essencial para caracterizar, avaliar e melhorar softwares. Entretanto, os valores típicos da maior parte das métricas de software não são conhecidos ainda [Tempero, 2008]. A falta de conhecimento sobre esses valores dificulta a aplicação de métricas de software na prática [Lanza & Marinescu, 2006]. Esta seção discute trabalhos concentrados em caracterizar softwares por meio de métricas e em identificar valores referência para métricas de software.

Tem havido grande interesse na investigação sobre a forma como módulos de um software conectam-se uns com os outros. Uma conclusão dos trabalhos realizados nesta linha é que software parece ser governado pelas denominadas leis de potência (*power laws*) [Baxter et al., 2006; Louridas et al., 2008; Potanin et al., 2005; Puppin & Silvestri, 2006; Wheelson & Counsell, 2003]. Uma lei de potência é uma

função de distribuição de probabilidades na qual a probabilidade de uma variável randômica X assumir um valor x é proporcional a uma potência negativa de x, i.e., $P(X = x) \propto cx^{-k}$. Um lei de potência é uma distribuição de cauda pesada (*heavy-tailed distribution*). Uma característica desse tipo de distribuição é que a frequência de valores altos da variável randômica é muito baixa, enquanto a frequência de valores baixos é alta. Neste tipo de distribuição, o valor médio não é representativo e, então, não há valor que possa ser considerado como típico para a variável randômica [Newman, 2003].

Muitas pesquisas têm identificado leis de potência em grafos que representam relacionamentos entre classes e objetos em softwares orientado por objetos. Potanin et al. [2005] analisaram 60 grafos de 35 softwares e concluíram que os relacionamentos entre objetos, em tempo de execução, constituem um grafo livre de escala (*scale-free graph*). Um grafo dessa natureza é diferente de um grafo nos quais as arestas são distribuídas randomicamente. Em um grafo randômico, o valor médio dos graus dos vértices é representativo, enquanto em um grafo livre de escala esta propriedade não se aplica porque a distribuição dos graus de seus vértices seguem uma lei de potência. Wheelson & Counsell [2003] identificaram leis de potência em grafos de dependência entre classes de programas Java. Os dados analisados no estudo deles são de três software bem conhecidos: JDK (*Java Development Kit*), Apache Ant e Tomcat, em um total de 6.870 classes. Eles também identificaram leis de potência nos valores das seguintes métricas: número de campos, número de métodos e construtores. Louridas et al. [2008] analisaram dados de programas desenvolvidos em C, Perl, Java e Ruby. Um conjunto de 11 softwares foi analisado no trabalho deles, dentre eles J2SE SDK, Eclipse e OpenOffice. O estudo concluiu que os graus de entrada e de saída dos módulos são governados por leis de potência, independente do paradigma de programação.

Baxter et al. [2006] investigaram a estrutura de um grande número de programas Java. O conjunto de dados usado no estudo deles é de 56 programas abertos, com diferentes tamanhos e de diferentes domínios de aplicação. Eles concluíram que algumas das métricas analisadas seguem uma lei de potência enquanto outras não. O estudo deles sugere que grau de entrada e número de subclasses seguem leis de potência, mas grau de saída, número de atributos e número de atributos públicos não. Esta conclusão diverge dos achados do estudo de Louridas et al. [2008], que identificou que grau de saída de classes segue lei de potência. Os achados dos trabalhos de Baxter et al. [2006] e Louridas et al. [2008] trazem informações que podem auxiliar a entender a forma de programas abertos. Entretanto, eles não exploram os seus resultados para identificar valores referência das métricas utilizadas. Além disso, esses trabalhos deixaram de avaliar métricas relacionadas a outros importantes fatores de qualidade, tal como coesão

interna de classe.

Lanza & Marinescu [2006] definem que há duas fontes principais para a identificação de valores referência: informações estatísticas e o conhecimento amplamente aceito. Valores referência baseados em análise estatística são derivados a partir da análise dos dados de uma população ou amostra. Aplicando análise estatística, Lanza & Marinescu [2006] sugerem valores referência para três métricas de software: número de métodos por classe, linhas de código por método e número ciclomático por linha de código. Eles coletaram essas métricas em 37 softwares desenvolvidos em C++ e em outros 45 em Java. A diversidade de tamanhos, domínios de aplicação e tipos (proprietário ou código aberto) foi a base para a seleção da amostra. Os valores referência propostos por eles são dados por: um intervalo de valores típicos para a métrica; um limite inferior e um superior deste intervalo; e um valor que pode ser considerado como *fora do padrão*. Ele consideraram que os valores das métricas seguem uma distribuição Normal e, com nisso, aplicaram cálculo de média e desvio padrão para definirem os intervalos de valores típicos. Eles consideraram um valor como *fora do padrão* quando ele é 50% maior do que o maior valor do intervalo. O método que eles aplicaram para derivar os valores referência é aplicável apenas se os valores das métricas seguirem uma distribuição Normal. Todavia, como indicado por outros estudos discutidos nesta seção, muitas métricas seguem leis de potência. Então, interpretar essas métricas em termos de valores médios pode levar a resultados errôneos.

O presente trabalho tem por objetivo determinar valores referência para seis métricas de software que ainda não foram estudadas desta forma em trabalhos anteriores: LCOM (*lack of cohesion in methods*), DIT (*depth in inheritance tree*) [Chidamber & Kemerer, 1994], COF (*coupling factor*) [Abreu & Carapuça, 1994], conexões aferentes, número de métodos públicos e número de atributos públicos. Essas métricas são descritas na Seção 5.2.1. Os resultados da análise realizada neste trabalho mostram que os valores dessas métricas não seguem uma distribuição Normal. Foi, então, identificada uma distribuição de probabilidades adequada aos valores de cada uma dessas métricas. Além disso, é apresentado o método utilizado para derivação de valores referência de métricas baseado em análise estatística dos dados. Aplicando-se esse método, foram derivados valores referência para as seis métricas avaliadas neste estudo.

Tabela 5.1. Softwares e seus domínios de aplicação, tipos, tamanhos, número de conexões entre as classes e a métrica COF.

Domínio	Software	Tipo	#Classes	#Conexões	COF
Clustering	Essence	framework	182	543	0,016
	Gridsim	tool	214	774	0,017
	JavaGroups	tool	1061	3807	0,003
	Prevayler	library	90	137	0,017
	Super (Acelet-Scheduler)	tool	246	1085	0,018
Database	DBUnit	framework	289	911	0,011
	ERMaster	tool	569	2187	0,007
	Hibernate	framework	1359	5199	0,003
Desktop	Facilitator	tool	2234	6565	0,001
	Java Gui Builder	tool	60	126	0,036
	Java X11 Library	library	318	1146	0,011
	J-Pilot	tool	142	367	0,018
	Scope	framework	214	535	0,012
Development	Code Generation Library	library	226	662	0,013
	DrJava	tool	2766	9684	0,001
	Find Bugs	tool	1019	3108	0,003
	Jasper Reports	library	1233	5610	0,004
	Junit	framework	154	353	0,015
	Spring	framework	2116	7069	0,002
	BCEL	library	373	2111	0,015

5.2 Metodologia

Esta seção descreve as métricas de software analisadas no estudo e apresenta o método que foi aplicado na derivação dos valores referência.

5.2.1 Métricas de Software Avaliadas

Neste trabalho, são estudadas seis métricas de software que foram selecionadas por serem relacionadas a fatores influentes na qualidade estrutural de software, tais como acoplamento, coesão, encapsulamento, ocultação de informação e profundidade da árvore de herança. Essas métricas são descritas a seguir.

COF (*Coupling Factor*) [Abreu & Carapuça, 1994]: esta métrica é calculada em nível de sistema. O sistema é representado por um grafo direcionado sem *self-loops* no qual os vértices são as classes e as arestas, as conexões entre as classes. No cálculo dessa métrica, considera-se que há uma conexão direcionada de A para B quando A usa um serviço ou atributo de B ou é sub-classe de B. Em um software

com n classes, pode haver no máximo $n^2 - n$ conexões. COF é dado por $c/(n^2 - n)$, onde c é o número de conexões existentes no software. Esta métrica é um indicador do grau de conectividade do software. Como afirmado por Meyer [1997], em uma arquitetura de software "cada módulo deve se comunicar com o menor número possível de outros módulos". Quanto maior COF, maior a conectividade do software, o que pode contribuir para a dificuldade de manutenção do software [Abreu & Carapuça, 1994].

Número de atributos públicos: esta métrica é o total de número de atributos públicos definidos em uma classe. Um dos conceitos principais do paradigma orientado por objetos é a ocultação de informação, que define que um módulo deve revelar o mínimo possível de seu funcionamento interno. Para atender esse princípio, é desejável evitar tornar dados públicos. Um atributo público pode ser diretamente acessado e alterado por qualquer outro objeto. Desta forma, o uso de atributos públicos pode levar ao surgimento de forte acoplamento entre classes de um software, reduzindo a modularidade do programa [Fowler, 1999; Meyer, 1997].

Número de métodos públicos: esta métrica é o total de métodos públicos definidos na classe. Como um método público corresponde a um serviço provido pela classe, esta métrica é um indicador da quantidade de serviços realizados pela classe. Como afirmado por Fowler [1999], quando uma classe possui um número grande métodos é um sinal de que ela pode ter muitas responsabilidades, o que pode tornar difícil o seu entendimento e sua manutenção.

LCOM (*Lack of Cohesion in Methods*) [Chidamber & Kemerer, 1994]: esta métrica mede o grau de coesão interna de uma classe considerando o conceito de similaridade entre métodos. Dois métodos são considerados similares se eles usam pelo menos uma variável de instância da classe em comum. LCOM é dada pela diferença entre o número de pares de métodos similares e o número de pares de métodos não similares. Por definição, quando o número de pares sem similaridade é menor do que o número de pares de métodos com similaridade, LCOM é igual a 0. De acordo com Chidamber & Kemerer [1994], quanto maior o valor de LCOM, menor a coesão da classe. Entretanto, o valor zero não corresponde necessariamente à boa coesão.

Há uma grande quantidade de métricas de coesão propostas para a orientação por objetos e LCOM tem sido criticada na literatura [Briand et al., 1998]. Apesar disso, LCOM foi avaliada no presente estudo porque ainda não há uma con-

clusão consensual sobre a melhor forma de se medir coesão interna de classe. Além disso, outras métricas de coesão são baseadas na mesma ideia de similaridade usada por LCOM. Aqui não se faz nenhuma afirmação sobre a validade de LCOM. O escopo deste trabalho é identificar os valores comuns desta métrica na prática, provendo valores referência para aqueles que a utilizam. Um estudo de Lincke et al. [2008] analisou dez ferramentas de coleta de métricas para programas Java. Oito destas ferramentas coletam LCOM de acordo com sua definição original feita por Chidamber & Kemerer [1994], enquanto apenas três coletam uma nova versão da métrica. Desta forma, LCOM parece ser utilizada apesar das críticas que tem sofrido.

DIT (*Depth of Inheritance Tree*) [Chidamber & Kemerer, 1994]: esta métrica é dada pela distância máxima de uma classe à raiz da sua árvore de herança. Herança é uma técnica poderosa de reuso de software. Entretanto, Gamma et al. [2000] afirmam que o uso imoderado de herança pode tornar a estrutura do software complexa. Eles, então, definem o princípio: *favoreça composição de objetos em relação à herança*. Na mesma linha, Sommerville [2003] argumenta que herança introduz dificuldades na compreensão de comportamento de objetos. Um estudo empírico de Daly et al. [1996] mostrou que árvores de herança profundas torna a manutenção de software mais difícil. DIT indica a profundidade de uma classe na sua árvore de herança. Esta métrica é considerada um indicador da complexidade do desenho do software [Chidamber & Kemerer, 1994]. Quanto maior o valor de DIT de uma classe, maior o número de classes envolvidas na sua análise e mais difícil a sua compreensão.

Conexões aferentes: um software pode ser representado como um grafo direcionado, sem *self loops*, no qual os vértices representam as classes, e as arestas, as conexões entre as classes. No cálculo dessa métrica, considera-se que há uma conexão direcionada de A para B quando A usa um serviço ou um atributo de B ou é sub-classe de B. A métrica é dada pelo total de conexões (arestas) que chegam a uma classe. Ela é um indicador do número de classes no software que dependem da classe avaliada. Classes que possuem um alto número de conexões aferentes desempenham um papel central no sistema, pois erros ou modificações nelas podem afetar um grande número de outras classes. Um valor alto de conexões aferentes em uma classe pode ser um sinal de que ela desempenha muitas responsabilidades. Esta métrica ajuda a identificar tais classes, para que se possa ter uma noção melhor dos impactos de modificações realizadas nela ou avaliar a necessidade de reestruturá-las.

5.2.2 Dados Analisados

Os dados usados neste estudo são de 40 softwares abertos, desenvolvidos em Java, obtidos em SourceForge[1], em suas últimas versões em Junho de 2008. Os software têm tamanhos de 14 a 3.586 classes, são de 11 domínios de aplicação e de três tipos: ferramenta, biblioteca e *framework*. Mais de 26.000 classes foram analisadas. Os softwares e seus domínios de aplicação, tipos e tamanhos são descritos nas Tabelas 5.1 e 5.2. As duas últimas colunas destas tabelas mostram, respectivamente, o número de conexões entre as classes e o valor da métrica COF.

A ferramenta *Connecta* [Ferreira et al., 2008], foi utilizada para coletar as métricas. *Connecta* coleta métricas de software orientado por objetos em programas Java a partir do *bytecode*. Por esta razão, um critério para a escolha dos softwares a serem analisados neste estudo foi a disponibilidade de seus *bytecodes*.

Tabela 5.2. Softwares e seus domínios de aplicação, tipos, tamanhos, número de conexões entre as classes e a métrica COF.

Domínio	Software	Tipo	#Classes	#Conexões	COF
Enterprise	Liferay	framework	14	14	0,077
	Talend	tool	2779	3567	0,000822
	uEngine BPM	framework	708	1774	0,004
	YAWL	tool	382	1186	0,008
Financial	JMoney	tool	193	424	0,019
Games	JSpaceConquest	tool	150	424	0,019
	KoLmafia	tool	810	5106	0,008
	Robocode	tool	213	738	0,016
Hardware	Jcapi	library	21	61	0,145
	LibUSBJava	library	35	90	0,076
	ServoMaster	library	55	117	0,039
Multimedia	CDK	library	3586	14711	0,001
	JPedal	tool	539	1533	0,005
	Pamguard	tool	1503	5267	0,002
Networking	BlueCove	library	142	461	0,023
	DHCP4Java	library	18	29	0,095
	jSLP	library	42	156	0,091
	WiKID Strong Authentication	library	50	27	0,011
Security	JSch	library	110	226	0,022
	OODVS	library	171	325	0,011

Foram realizadas quatro análises no conjunto de dados coletados: (1) os dados

[1] www.sourceforge.net

foram analisados como um todo e também por (2) tipo , (3) domínio de aplicação e (4) por tamanho de software. Essas análises foram realizadas com o propósito de identificar se há uma única distribuição de probabilidade que pode modelar os valores das métricas, independente de domínio de aplicação, tipo ou tamanho de software. Com base nas propriedades estatísticas dessas distribuições de probabilidades, foram derivados os valores referência das métricas. A próxima seção descreve como foi realizado o ajuste das distribuições de probabilidades aos dados.

5.2.3 Ajuste dos Dados

Uma ferramenta, denominada EasyFit [Mathwave, 2010], foi utilizada para realizar o ajuste dos dados a várias distribuições de probabilidades, tais como Bernoulli, Binomial, Uniforme, Geométrica, Hipergeométrica, Logarítmica, Binomial, Poisson, Normal, t-Student, Chi-quadrado, Exponencial, Lognormal, Pareto e Weibull. Uma distribuição de probabilidades é descrita por duas funções principais: a função de densidade de probabilidade (*pdf*), $f(x)$, que expressa a probabilidade de uma variável randômica assumir um valor x; e a função de distribuição cumulativa (*cdf*), $F(x)$, que expressa a probabilidade de uma variável randômica assumir um valor menor ou igual a x. No caso de distribuições de probabilidades discretas, a distribuição é descrita por uma *cdf* e por uma função de distribuição de pesos, *pmf*, que expressa a probabilidade da variável randômica assumir um valor x. Nos experimentos realizados neste estudo, as distribuições Poisson e Weibull mostraram-se adequadas para ajustar os dados.

A distribuição Poisson tem *pmf*, $f_p(x)$, e *cdf*, $F_p(x)$, definidas pelas Equações 5.1 e 5.2, respectivamente. O parâmetro λ da distribuição representa o valor médio da variável randômica.

$$f_p(x) = P(X = x) = \frac{e^{-\lambda}.\lambda^x}{x!} \tag{5.1}$$

$$F_p(x) = P(X \leq x_0) = \sum_{x=0}^{x=x_0} \frac{e^{-\lambda}.\lambda^x}{x!} \tag{5.2}$$

A distribuição Weibull, com parâmetros α e β, tem *pdf*, $f_w(x)$, e *cdf*, $F_w(x)$ definidas pelas Equações 5.3 e 5.4, respectivamente. O parâmetro β é chamado *parâmetro de escala*. O aumento do valor de β tem o efeito de diminuir a altura da curva, achatando-a. O parâmetro α é chamado *parâmetro de forma*. Quando esse parâmetro é menor do que um, a distribuição Weibull é uma distribuição de cauda pesada. Essa situação pode ser aplicada nos casos em que a variável randômica

apresenta assimetria à esquerda, i.e., quando há um pequeno número de ocorrências de valores altos e um número muito grande de ocorrências de valores baixos. Nesse caso, o valor médio não é representativo.

$$f_w(x) = P(X = x) = \frac{\alpha}{\beta}(\frac{x}{\beta})^{\alpha-1}e^{-(\frac{x}{\beta})^{\alpha}}, \alpha > 0, \beta > 0 \quad (5.3)$$

$$F_w(x) = P(X \leq x) = 1 - e^{-(\frac{x}{\beta})^{\alpha}}, \alpha > 0, \beta > 0 \quad (5.4)$$

5.2.4 Análise dos Dados

Para cada métrica, os dados foram coletados e dois gráficos foram gerados: um gráfico de dispersão, para exibir a frequência dos valores da métrica, e um gráfico com os mesmos dados, porém em escala logarítmica (escala *log-log*), como o objetivo de se observar se a distribuição dos valores seguem uma lei de potência. Quando plotados em escala log-log, uma distribuição com lei de potência é aproximadamente uma reta inclinada à esquerda. Se os valores de uma métrica seguem uma lei de potência, significa que a frequência de valores altos é muito baixa, enquanto a frequência de valores baixos é extremamente alta.

A ferramenta indica as distribuições que têm os melhores ajustes aos dados. Para cada métrica, considerando-se os resultados reportados pela ferramenta e a análise visual dos gráficos de ajustes, foi identificada a distribuição com melhor ajuste. Se a distribuição de probabilidade possuir valor médio representativo, como é o caso da distribuição de Poisson, este valor representa o valor típico da métrica. Caso contrário, são identificadas três faixas de valores: *bom*, *regular* e *ruim*. A faixa *bom* corresponde a valores com alta frequência. São os valores mais comuns da métrica na prática. Esses valores não necessariamente expressam as melhores práticas em Engenharia de Software. Entretanto, eles expressam um padrão da maior parte dos softwares. Quando comparado a outros softwares, é desejável que um determinado software tenha pelo menos a mesma qualidade dos demais. Contudo, preferencialmente, o desenvolvimento de software deve visar uma qualidade ainda maior. A faixa *ruim* corresponde a valores com baixa frequência, e a faixa *regular* é intermediária, que corresponde a valores que não são nem muito frequentes nem raros. Uma nomenclatura alternativa para essas faixas poderia ser *frequente*, *ocasional* e *raro* em substituição a *bom*, *regular* e *ruim*.

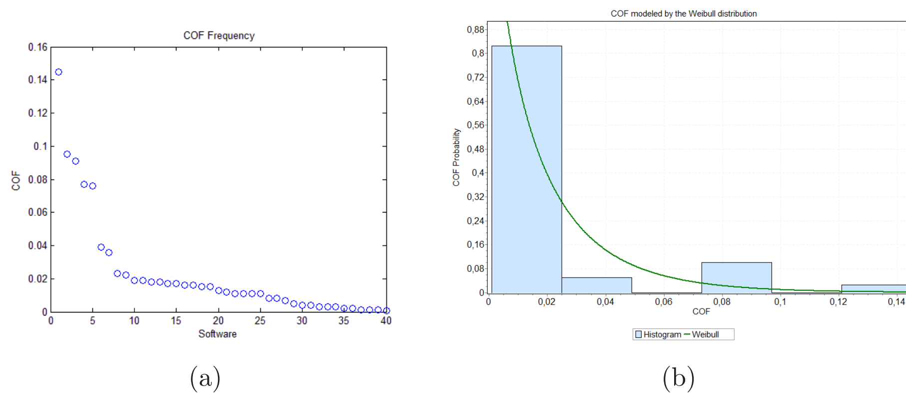
5.3 Resultados

Esta seção apresenta os resultados do estudo. Primeiramente, os resultados obtidos a partir do conjunto completo de dados são descritos. Em seguida, são apresentados os resultados da análise realizada por tipos, domínios de aplicação e tamanhos de software.

5.3.1 Ajuste dos Dados de Todos os Software do Estudo

5.3.1.1 COF

O gráfico de dispersão da Figura 5.1a mostra que os valores menores do que 0,02 são muito mais frequentes do que valores superiores a 0,02. COF pode ser modelado pela distribuição Weibull, com parâmetros $\alpha = 0,91927$ and $\beta = 0,01762$. A Figura 5.1b mostra valores de COF modelados pela distribuição Weibull. Mais de 80% dos softwares têm COF menor do que 0,02. A probabilidade de COF assumir valores acima de 0,02 e até 0,14 é baixa, e a probabilidade de COF assumir valores superiores a 0,14 é desprezível. Este resultado indica que, na maior parte dos casos, software aberto tem grau de conectivadade baixo, o que pode contribuir para sua manutenibilidade.



(a) (b)

Figura 5.1. COF – (a) frequência e (b) ajuste à distribuição de Weibull.

5.3.1.2 Conexões Aferentes

O gráfico de dispersão dos valores de conexões aferentes, mostrado na Figura 5.2a, sugere uma distribuição de cauda pesada. A Figura 5.2b mostra os mesmos dados em um gráfico de escala logarítmica. Neste gráfico, a distribuição mostra-se como uma reta inclinada à esquerda, característica de uma lei de potência. Há um pequeno número de classes com alto número de conexões aferentes e um número muito

grande de classes com poucas conexões aferentes. Como mostrado pela Figura 5.2c, valores dessa métrica podem ser modelados pela distribuição Weibull, com parâmetros $\alpha = 0,78986$ and $\beta = 3,2228$. A distribuição dos valores é detalhada na Figura 5.2d. Aproximadamente 50% das classes têm pelo menos uma conexão aferente. A probabilidade de uma classe ter de 2 a 20 conexões aferentes é baixa, e a probabilidade de uma classe ter mais de 20 conexões aferentes é desprezível. Esta observação indica que a maioria das classes afeta no máximo uma única classe diretamente. Isso pode contribuir para a manutenibilidade de software, pois uma modificação ou um erro em uma classe impactaria diretamente em um número pequeno de classes.

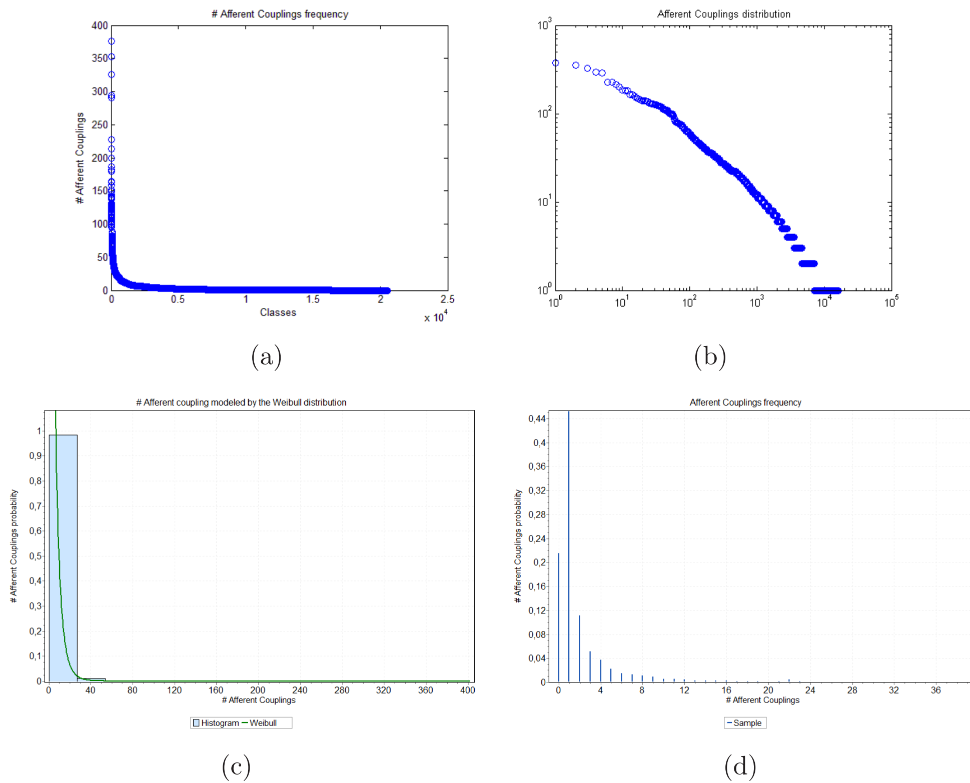
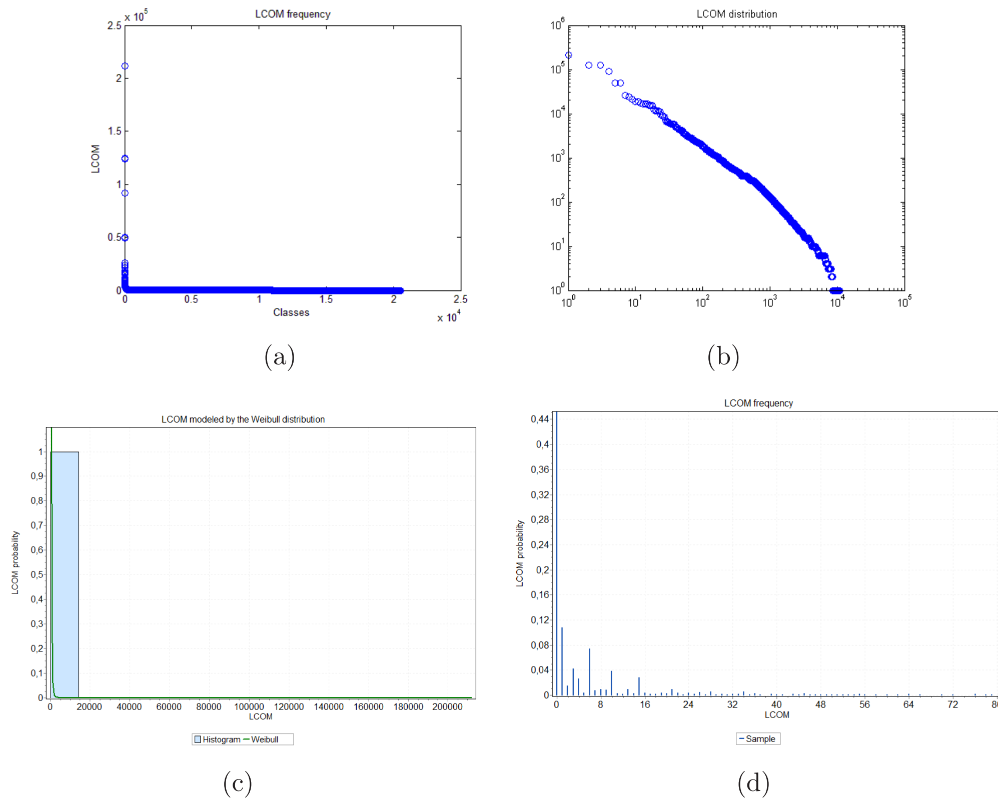


Figura 5.2. Conexões aferentes – (a) frequência, (b) frequência em escala log-log, (c) ajuste à distribuição Weibull e (d) frequência detalhada.

5.3.1.3 LCOM

LCOM também é ajustada por uma distribuição de cauda pesada. A Figura 5.3a mostra um gráfico de dispersão dos valores de LCOM, e a Figura 5.3b mostra os mesmos dados em escala logarítmica. Este gráfico indica que os valores de LCOM seguem uma lei de potência. Os valores de LCOM podem ser modelados pela distribuição de

Weibull, como mostrado na Figura 5.3c, com parâmetros $\alpha = 0,23802$ e $\beta = 1,465$. A Figura 5.3d detalha a distribuição dos valores de LCOM. Aproximadamente 50% das classes têm LCOM igual a zero. Há classes com LCOM de 1 a 20 em uma frequência baixa, inferior a 12%. A probabilidade de uma classe ter LCOM superior a 20 é desprezível.



(a) (b) (c) (d)

Figura 5.3. LCOM – (a) frequência, (b) frequência em escala log-log, (c) ajuste à distribuição Weibull e (d) frequência detalhada.

5.3.1.4 DIT

O gráfico de dispersão na Figura 5.4a mostra a distribuição de valores de DIT, e a Figura 5.4b mostra os mesmos dados em escala logarítmica. Esses gráficos não sugerem características de uma lei de potência. De fato, os valores de DIT podem ser ajustados pela distribuição Poisson, como mostrado na Figura 5.4c, com parâmetro $\lambda = 1,6818$. Na distribuição Poisson, λ é o valor médio da variável randômica. Este resultado mostra que, em um software aberto, a maior distância de uma classe à raiz de sua árvore de herança é 2, em média. Árvores de herança rasas contribuem para a

qualidade estrutural do software diminuindo a sua complexidade [Gamma et al., 2000; Daly et al., 1996; Sommerville, 2003].

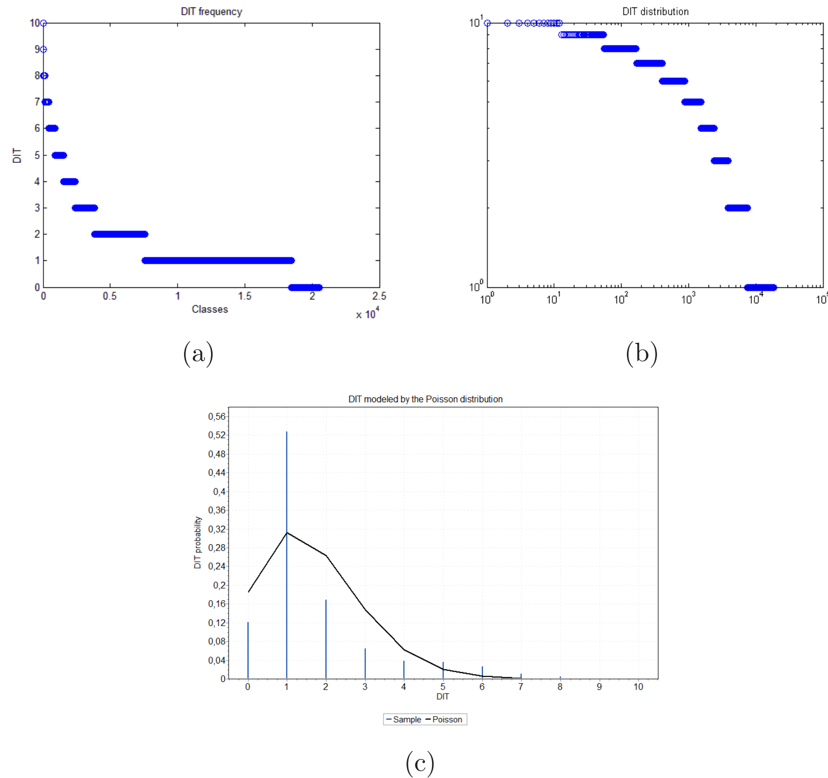
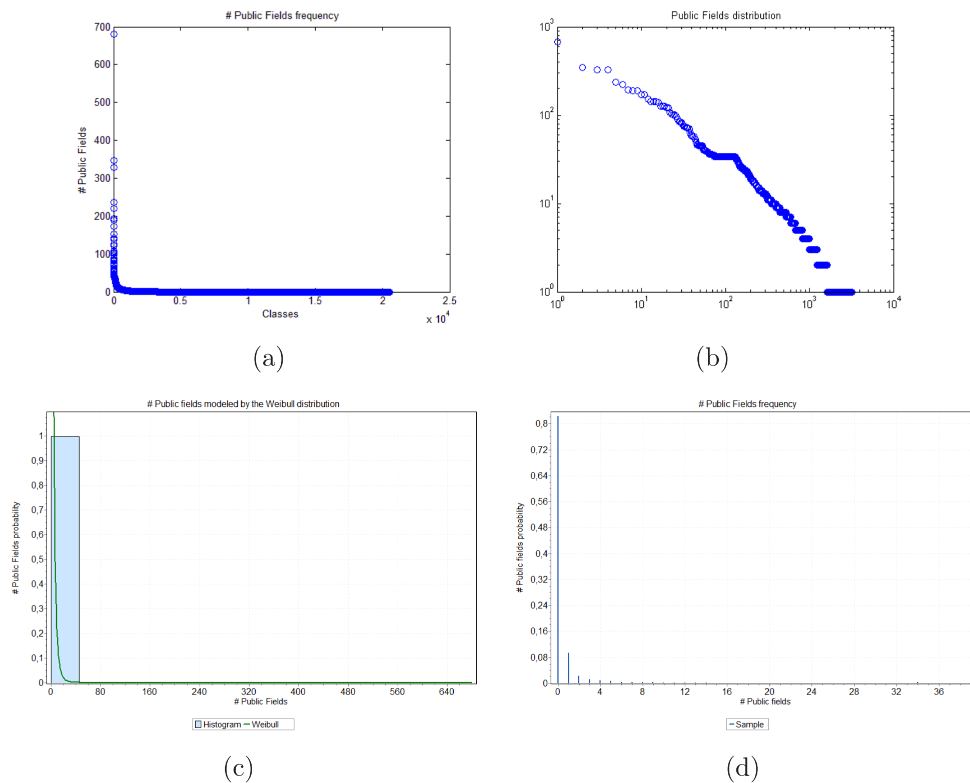


Figura 5.4. DIT – (a) frequência, (b) frequência em escala log-log, (c) ajuste à distribuição Poisson e (d) frequência detalhada.

5.3.1.5 Atributos Públicos

O gráfico de dispersão do número de atributos públicos, mostrado na Figura 5.5a, revela que há poucas classes com um número grande de atributos públicos. Na maioria dos casos, o número de atributos públicos é zero. A Figura 5.5b mostra os dados em escala logarítmica, que indica que o número de atributos públicos em classes também segue uma lei de potência. Esta métrica pode ser modelada pela distribuição Weibull com parâmetros $\alpha = 0,71008$ e $\beta = 4,4001$, o que é mostrado na Figura 5.5c. A Figura 5.5d detalha os dados. Mais de 80% das classes não têm atributos públicos, e a probabilidade de uma classe ter mais de 10 atributos públicos é desprezível. Esta observação revela que a maior parte dos softwares aplica o princípio de ocultação de informação apropriadamente, no que se refere a dados.



(a) (b)

(c) (d)

Figura 5.5. Atributos públicos – (a) frequência, (b) frequência em escala log-log, (c) ajuste à distribuição Weibull e (d) frequência detalhada.

5.3.1.6 Métodos Públicos

A frequência de número de métodos públicos é mostrada na Figura 5.6a e, em escala logarítmica, na Figura 5.6b. Estes gráficos mostram que o número de métodos públicos segue uma lei de potência. Esta métrica pode ser modelada pela distribuição Weibull, com parâmetros $\alpha = 0,85938$ e $\beta = 5,6558$, como mostrado na Figura 5.6c. A frequência de números de métodos públicos é detalhada na Figura 5.6d. Há uma pequena parte das classes com um número alto de métodos públicos, e a maioria das classes tem poucos métodos públicos. A maior parte das classes tem de 0 a 10 métodos públicos. Classes com 11 a 40 métodos públicos são pouco frequentes, e a probabilidade de uma classe ter mais de 40 métodos públicos é desprezível. Esses resultados mostram que, na maior parte dos casos, uma classe provê poucos serviços.

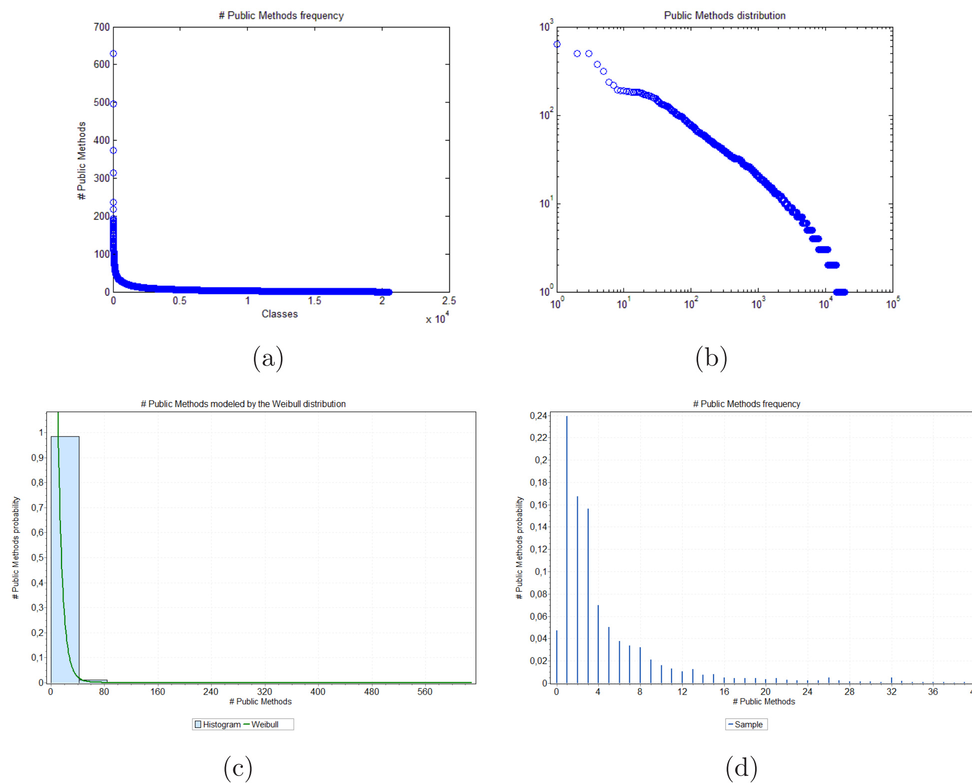


Figura 5.6. Métodos públicos – (a) frequência, (b) frequência em escala log-log, (c) ajuste à distribuição Weibull e (d) frequência detalhada.

Tabela 5.3. Valores Referência Gerais para Métricas de Software OO

Fator	Nível	Métrica	Valor Referência
Conectividade	Sistema	COF	Bom: até 0,02 – Regular: 0,02 a 0,14 – Ruim: superior a 0,14
	Classe	# Conexões aferentes	Bom: até 1 – Regular: 2 a 20 – Ruim: superior a 20
Ocultação de informação	Classe	# Atributos públicos	Bom: 0 – Regular: 1 a 10 – Ruim: superior a 10
Tamanho da interface	Classe	# Métodos públicos	Bom: 0 a 10 – Regular: 11 a 40 – Ruim: superior a 40
Herança	Classe	DIT	Valor típico: 2
Coesão	Classe	LCOM	Bom : 0 – Regular: 1 a 20 – Ruim: superior a 20

5.4 Valores Referência para as Métricas de Software OO

Valores referência são úteis para a interpretação adequada dos valores das métricas. O conhecimento desses valores são de grande importância para favorecer o uso

de métricas na prática e apoiar a tomada de decisão na avaliação de software. Neste estudo, um grande conjunto de softwares abertos orientados por objetos foi avaliado por meio de seis métricas de software. Os resultados da análise realizada levou à identificação de valores referência para essas métricas. Foram identificadas três faixas de valores para as métricas: *bom*, que corresponde aos valores mais comuns encontrados na prática; *regular*, que corresponde a valores com frequência intermediária, mas não irrelevante; e *ruim*, que corresponde a valores com frequência rara. Por exemplo, LCOM é zero em aproximadamente 50% das classes, valores de 1 a 20 ocorrem em uma frequência baixa, e valores superiores a 20 são raros. Desta forma, identificam-se os seguintes valores referência para LCOM: 0 (boa coesão), 1 até 20 (coesão regular) e superior a 20 (coesão ruim).

Esses valores referência não necessariamente denotam as melhores práticas em Engenharia de Software. Todavia, eles representam o nível de qualidade que desenvolvedores usualmente aplicam em software, provendo uma referência na avaliação de software por meio de métricas. Por exemplo, se o COF de um determinado software é 0,5 e a faixa *bom* desta métrica é 0,02, isso significa que o grau de conectividade do sistema é muito mais alto do que o da maioria dos softwares. Com isso, se os desenvolvedores querem que seus softwares sejam tão bons quanto os demais em termos de conectividade, eles devem manter COF abaixo de 0,02.

A mesma análise foi realizada para as outras métricas. Os valores referência sugeridos para COF, LCOM, DIT, conexões aferentes, número de métodos públicos e número de atributos públicos são descritos na Tabela 5.3. Foram realizadas também análises dos dados por tipos, domínios de aplicação e tamanhos de sofware. Essas análises não foram realizadas para a métrica COF porque esta é uma métrica calculada em nível de sistema; o número de software em cada conjunto resultante para essa métrica na amostra utilizada neste estudo é muito pequeno para permitir chegar a conclusões sólidas a respeito do comportamento de COF nesses cenários. Os resultados desta análise são descritos nas Seções 5.4.1, 5.4.2 e 5.4.3, respectivamente.

5.4.1 Valores Referência para Métricas de Software OO por Tipo

Os softwares analisados neste estudo são de três tipos: ferramenta, *framework* e biblioteca. Os dados utilizados foram analisados separadamente para estas três categorias. Os resultados dessa análise revelam que a distribuição de probabilidade que modela os valores de uma métrica no conjunto completo de dados é aplicável também a categorias distintas de softwares. A Tabela 5.4 mostra os valores referência derivados

por tipo de software. A seguir esses resultados são discutidos para cada métrica:

- Atributos públicos: nos três casos, mais de 80% das classes não possuem atributos públicos. A frequência de classes com 1 a 8 atributos públicos é muito baixa, e a frequência de classe com mais de 8 atributos públicos é próxima de zero. Uma pequena diferença é observada no caso de *framework*, cujo limiar superio da faixa *regular* dessa métrica é 10.

- Métodos públicos: há uma pequena diferença entre as distribuições dos valores dessa métrica em ferramentas, bibliotecas e *frameworks*. A distribuição de valores no caso das ferramentas é mais concentrada à esquerda, o que indica que classes de ferramentas tendem a ter menos métodos públicos do que *frameworks* e bibliotecas.

- LCOM: valores dessa métrica têm uma distribuição muito similar nos três casos. Os valores referência nas três categorias são basicamente os mesmos encontrados no conjunto completo de dados.

- DIT: esta métrica pode ser modelada pela distribuição de Poisson nos três casos. Há uma diferença muito sutil nos valores médios encontrados: 1,68 em *framework*, 1,74 em ferramentas e 1,96 em bibliotecas.

- Conexões aferentes: os resultados encontrados na análise dos dados das três categorias são basicamente os mesmos. Além disso, eles são compatíveis com os resultados encontrados no conjunto completo de dados.

5.4.2 Valores Referência para as Métricas de Software OO por Domínio de Aplicação

As métricas de software estudadas foram analisadas para cada domínio de aplicação dos softwares listados na Tabela 5.1. Os resultados dessa análise mostram que os valores de uma métrica de software podem ser modelados por uma única distribuição de probabilidade, independente do domínio de aplicação. Essa distribuição é a mesma que ajusta os dados no conjunto completo de dados. Aplicando-se o mesmo método utilizado para derivar os valores referência gerais, foram derivados os valores referência por domínio de aplicação. Os resultados são mostrados na Tabela 5.5. Há uma pequena diferença entre esses resultados e aqueles encontrados no conjunto completo de dados. Porém, esta diferença não representa discordância entre os resultados. Esta observação é verdadeira mesmo para o caso do domínio de aplicação *Financial*, que

possui apenas um software analisado neste estudo. Desta forma, os valores referência gerais, mostrados na Tabela 5.3, podem ser aplicados independente de domínio de aplicação.

Tabela 5.4. Valores Referência para Métricas de Software OO por Tipo

Tipo	#Conexões aferentes (bom/regular/ruim)	#Atributos públicos (bom/regular/ruim)	#Método públicos (bom/regular/ ruim)	DIT (valor típico)	LCOM (bom/ regular/ruim)
Ferramenta	0 - 1 / 2 - 20 / >20	0 / 1 - 8 / >8	0 - 20 / 21 - 50 / >50	2	0 / 1 - 20 / >20
Framework	0 - 1 / 2 - 20 / >20	0 / 1 - 10 / >10	0 - 25 / 26 - 50 / >50	2	0 / 1 - 20 / >20
Biblioteca	0 - 1 / 2 - 25 / >25	0 / 1 - 8 / >8	0 - 25 / 26 - 40 / >40	2	0 / 1 - 25 / >25

Tabela 5.5. Valores Referência para Métricas de Software OO por Domínio de Aplicação

Domínio	#Conexões aferentes (bom/regular/ruim)	#Atributos públicos (bom/regular/ruim)	#Métodos públicos (bom/regular/ruim)	DIT (valor típico)	LCOM (bom/regular/ruim)
Clustering	0 - 1 / 1 - 20 / >20	0 / 1 - 7 / >7	0 - 20 / 20 - 45 / >45	2	0 / 1 - 20 / >20
Database	0 - 1 / 2 - 20 / >20	0 / 1 - 8/ >8	0 - 20 / 21 - 50 / >50	2	0 / 1 - 20/ >20
Desktop	0 - 1 / 2 - 20 / >20	0 / 1 - 8/ >8	0 - 20 / 21 - 55/ >55	2	0 / 1 - 15 / >15
Development	0 - 1 / 2 - 25 / >25	0 / 1 - 8 / >8	0 - 20 / 21 - 35 / >35	2	0 / 1 - 25 / >25
Enterprise	0 - 1 / 2 - 22 / >22	0 / 1 - 11 / >11	0 - 15 / 16 - 35 / >35	1	0 / 1 - 35 / >35
Financial	0 - 1 / 2 - 16 / >16	0 / 1 - 4 / >4	0 - 13 / 14 - 32 / >32	2	0 / 1 - 25 / >25
Games	0 - 1 / 2 - 22 / >22	0 / 1 - 9 / >9	0 - 20 / 21 - 32 / >32	1	0 / 1 - 35 / >35
Hardware	0 - 1 / 2 - 11 / >11	0 - 1 / 2 - 6 / >6	0 - 20 / 21 - 36 />36	2	0 / 1 - 80 / >80
Multimedia	0 - 1 / 2 - 20 / >20	0 / 1 - 9 / >9	0 - 35 / 36 - 60 / >60	2	0 / 1 - 60 / >60
Networking	0 - 1 / 2 - 20/ >20	0 / 1 - 5 / >5	0 - 15 / 16 - 40 / >40	2	0 / 1 - 40 / >40
Security	0 - 1 / 2 - 16 / >16	0 / 1 - 8/ >8	0 - 25 / 26-50 / >50	1	0 / 1 - 45 / >45

5.4.3 Valores Referência para Métricas de Software OO por Tamanho

Os softwares foram agrupados em três conjuntos de acordo com seus tamanhos em número de classes: até 100 classes, de 101 a 1000 classes e mais de 1000 classes. As métricas de software estudadas foram analisadas para cada um desses conjuntos. Os resultados mostram que uma única distribuição de probabilidades modela valores de uma métrica, independente do tamanho do software. Tal distribuição é a mesma que modela os valores da métrica no conjunto completo de dados.

Tabela 5.6. Valores Referência para Métricas de Software OO por Tamanho

Tamanho (#classes)	#Conexões aferentes (bom/regular/ruim)	#Atributos públicos (bom/regular/ruim)	#Método públicos (bom/regular/ruim)	DIT (valor típico)	LCOM (bom/regular/ruim)
⩽100	0 - 1 / 2 - 20 / >20	0 / 1 - 10 / >10	0 - 20 / 21 - 30 / >30	2	0 / 1 - 25 / >25
101 - 1000	0 - 1 / 2 - 20 / >20	0 / 1 - 8 / >8	0 - 25 / 6 - 50 / >50	2	0 / 1 - 20 />20
>1000	0 - 1 / 2 - 15 / >15	0 / 1 - 5 / >5	0 - 30 / 30 - 60 / >60	2	0 / 1 - 20/ >20

Aplicando-se o mesmo método utilizado no conjunto completo de dados, foram derivados os valores referência para as métricas de software, por tamanho de software. Os resultados são mostrados na Tabela 5.6. Observa-se uma sutil diferença entre os resultados encontrados nos três conjuntos. Uma diferença interessante é em relação ao número de atributos públicos: quanto maior o tamanho do software, menor o número de atributos públicos. Isso sugere que atributos públicos são mais fortemente evitados em softwares de grande porte.

Os resultados obtidos nesta análise não discordam daqueles encontrados na análise do conjunto completo de dados. Com isso, os valores referência gerais encontrados aplicam-se a software em geral, independente de tamanho.

5.5 Avaliação dos Valores Referência

Os valores referência auxiliam especialistas a utilizarem métricas em suas tarefas. Na Engenharia de Software, os valores referência são úteis, por exemplo, para auxiliarem desenvolvedores e gerentes em suas tarefas de avaliação de software e processos. Os valores referência identificados neste estudo foram derivados empiricamente a partir da análise de dados de um grande conjunto de softwares. Esta seção realiza uma avaliação da aplicação dos valores referência por meio de estudos de caso. Os estudos de caso também exemplificam como os valores referência podem ser utilizados no processo de avaliação de software.

A aplicação de um valor referência pode resultar em duas situações possíveis: o valor referência avalia corretamente a classe ou ele falha. A avaliação correta ocorre quando a faixa em que a métrica cai reflete a avaliação real da classe. Nesse estudo de caso, consideram-se duas situações em que é possível falha do valor referência: *falso positivo*, quando o valor classifica a classe como *ruim* e a inspeção da classe não identifica problemas em sua estrutura, ou *falso negativo*, quando o valor não classifica a classe como *ruim*, mas a inspeção da classe identifica problemas em sua estrutura.

Foram realizados dois conjuntos de estudos de caso. O primeiro foi realizado no conjunto completo das classes utilizadas na derivação dos valores referência. Para isso, foram consideradas as classes que resultaram em valores que correspondem à faixa *ruim* de cada métrica. Essas classes foram, então, inspecionadas a fim de se avaliar a eficácia do valor referência para identificar classes com problemas estruturais. Casos *falso positivos* podem ser identificados nesse primeiro estudo de caso. Entretanto, a identificação de *falsos negativos* não é viável, porque demandaria a inspeção de todas as classes envolvidas. No segundo estudo de caso, foi utilizado o software JHotDraw[2]. O motivo pelo qual esse software foi escolhido para esse segundo estudo de caso é que ele é considerado como um software de boa qualidade estrutural e tem sido utilizado em muitos outros estudos [Seng et al., 2006; Czibula & Czibula, 2008; Bertrán, 2009; Jancke, 2010; Kessentini et al., 2010]. JHotDraw foi inicialmente desenvolvido por Erich Gamma e Thomas Eggenschwiler para ser um exercício da aplicação de padrões de projeto, mas atualmente é considerado maduro e aplica extensivamente padrões de projetos [Riehle, 2000]. A hipótese averiguada nesse estudo de caso é a de que a aplicação dos valores referência avaliam o software JHotDraw como detentor de boa qualidade. Esse resultado é sugestivo de que a aplicação dos valores referência não levam a resultados *falso negativos*, ou seja, quando os valores referência avaliam a qualidade do software como boa ou regular, esse resultado pode ser considerado aceitável.

Na avaliação de software, métricas podem ser utilizadas isoladamente ou em conjunto. Os estudos de casos realizados empregam as duas abordagens. No primeiro grupo de estudos de casos, as métricas foram utilizadas em conjunto. Foram selecionadas as classes que atenderam ao seguinte critério: número de conexões aferentes, LCOM, número de métodos públicos e número de atributos públicos na faixa *ruim* e DIT com valor acima da média. Os valores referência utilizados são os gerais, mostrados na Tabela 5.3. No segundo estudo de caso, as métricas foram empregadas da mesma forma e também isoladamente. As seções seguintes apresentam e discutem os resultados dos estudos de caso.

5.5.1 Estudo de Caso 1

O objetivo deste estudo de caso é avaliar a capacidade dos valores referência identificarem classes com deficiências do ponto de vista de sua estrutura. Para isso, foram utilizadas como alvo de avaliação todas as classes do conjunto de software utilizado para derivar os valores referência.

[2]http://www.jhotdraw.org/

Tabela 5.7. Classes do conjunto de softwares e valores de suas métricas #CA (conexões aferentes), LCOM, DIT, #AP (atributos públicos) e #MP (métodos públicos)

Software	Classe	#CA	LCOM	DIT	#AP	#MP
JasperReports	`net.sf.jasperreports.engine.xml.JRXmlConstants`	51	957	1	347	44
KolMafia	`net.sourceforge.kolmafia.textui.DataTypes`	27	261	1	70	27
Facilitator	`org.w3c.tidy.TagTable`	27	41	1	60	10
JavaX11Library	`gnu.x11.Display`	63	67	1	54	31
KolMafia	`net.sourceforge.kolmafia.request.UseItemRequest`	29	67	4	41	18
KolMafia	`net.sourceforge.kolmafia.KoLmafia`	163	2826	1	38	75
Facilitator	`org.w3c.tidy.Lexer`	43	1057	1	36	51
Hibernate	`org.hibernate.Hibernate`	61	91	1	31	17
JavaX11Library	`gnu.x11.Window`	53	1345	3	29	96
KolMafia	`net.sourceforge.kolmafia.KoLAdventure`	25	147	3	25	20
JavaX11Library	`gnu.x11.extension.glx.VisualConfig`	32	878	1	20	45
KolMafia	`net.sourceforge.kolmafia.request.GenericRequest`	119	872	3	19	38
Jpilot	`org.jpilotexam.ui.util.UIUtilities`	24	21	1	18	4
CodeGeneration	`net.sf.cglib.core.CodeEmitter`	59	2175	1	16	98
KolMafia	`net.sourceforge.kolmafia.persistence.ConcoctionDatabase`	39	328	1	16	20
Bcel	`org.apache.bcel.generic.Type`	55	24	1	16	10
YAWL	`org.yawlfoundation.yawl.engine.interfce.WorkItemRecord`	44	2725	1	15	75
KolMafia	`net.sourceforge.kolmafia.AdventureResult`	144	46	1	15	31
KolMafia	`net.sourceforge.kolmafia.request.UseSkillRequest`	26	285	4	15	19
KolMafia	`net.sourceforge.kolmafia.KoLCharacter`	124	15507	2	13	188
JasperReports	`net.sf.jasperreports.engine.util.JRProperties`	37	223	1	13	26
JasperReports	`net.sf.jasperreports.engine.design.JRDesignChart`	31	4449	3	12	128
KolMafia	`net.sourceforge.kolmafia.request.EquipmentRequest`	36	62	5	12	14
KolMafia	`net.sourceforge.kolmafia.persistence.SkillDatabase`	26	206	3	11	22
KolMafia	`net.sourceforge.kolmafia.swingui.panel.ItemManagePanel`	28	53	7	11	15
Super	`com.acelet.lib.CommonPanel`	36	75	5	11	8
Facilitator	`org.w3c.tidy.Configuration`	29	76	1	11	4

O estudo de caso consistiu em inspecionar as classes classificadas na faixa *ruim* com o objetivo de comparar o resultado da avaliação qualitativa com a classificação dada pelo valor referência. Inicialmente foram selecionadas as classes que possuem número de conexões aferentes superior a 20. Este primeiro filtro resultou em 526 classes. Dentre elas foram selecionadas as classes com LCOM superior a 20, o que resultou em 276 classes. Como avaliar manualmente essa quantidade de classes é uma tarefa inibidora, foram realizados três outros filtros: classes com número de métodos públicos superior a 40, que resultou em 59 classes; classes com DIT superior a 2, que resultou em 39 classes; e classes com número de atributos públicos superior a 10, que resultou em 24 classes. Optou-se por trabalhar com este último conjunto porque ele possui o menor número de classes. Os dados das classes avaliadas são mostrados na Tabela 5.7. Apenas as classes do software Facilitator não puderam ser avaliadas porque o seu código fonte não pode ser obtido. A seguir é descrita a avaliação qualitativa de cada uma dessas classes.

- `net.sf.jasperreports.engine.xml.JRXmlConstants`: é uma classe depositária de constantes e utilitária, já que possui uma grande quantidade de métodos estáticos de propósitos distintos. Apenas pela observação dos dados que os métodos manipulam, percebe-se que há conjuntos de métodos entre os quais não há dependência baseada em dados. A classe poderia ser dividida em outras, pois, aparentemente, possui muitas responsabilidades.

- `net.sourceforge.kolmafia.textui.DataTypes`: é uma classe depositária de constantes. É também uma classe utilitária, pois possui vários métodos para conversão de dados.

- `gnu.x11.Display`: possui vários atributos públicos não constantes. A classe fere ocultação de informação, que é preceito básico da orientação por objetos. Há 11 desses atributos que parecem poder ser extraídos para formar outra classe, denominada `Server`, por exemplo, pois referem-se a dados de um servidor em uma conexão. A classe parece ser uma boa candidata a reestruturação, pois é muito grande e é difícil descrever um papel bem definido para ela.

- `net.sourceforge.kolmafia.request.UseItemRequest`: os seus atributos públicos são constantes. A classe parece ser uma candidata à reestruturação, pois mistura pelo menos dois objetivos: a implementação de um tipo abstrato de dados (TAD) e o fornecimento de vários serviços não diretamente relacionado ao TAD implementado, pois possui vários métodos estáticos. Além disso, a sua com-

preensão demanda a análise de sua classe mãe, `GenericRequest`, que também apresenta problemas estruturais, conforme descrito a seguir.

- `net.sourceforge.kolmafia.KoLmafia`: a maior parte dos seus atributos públicos é constante. Porém, a classe possui também atributos públicos que não são constantes. Ela contém o método principal da aplicação e tem muitos métodos estáticos públicos de propósitos distintos como manipular propriedades do jogador e manipular aparência do *display*. Esses métodos poderiam ser extraídos para outras classes.

- `org.hibernate.Hibernate`: a classe possui somente atributos e métodos estáticos. Os métodos operam somente sobre os parâmetros que lhe são passados, ou seja, a classe não representa um tipo abstrato de dados. É uma espécie de classe utilitária.

- `gnu.x11.Window`: possui vários atributos públicos não constantes. A classe fere ocultação de informação, que é preceito básico da orientação por objetos. A classe parece ser uma boa candidata a reestruturação, pois é muito grande, de difícil compreensão e é difícil descrever um papel bem definido para ela.

- `net.sourceforge.kolmafia.KoLAdventure`: os atributos públicos são constantes estáticas. A classe implementa um TAD, mas contém alguns métodos estáticos que não estão diretamente relacionados à implementação do TAD. Eles poderiam ser extraídos para uma outra classe.

- `gnu.x11.extension.glx.VisualConfig`: a classe basicamente possui somente dados e métodos do tipo *get* e *set*. Ela possui dois atributos públicos não constantes, que deveriam ser encapsulados a fim de manter melhor integridade de dados.

- `net.sourceforge.kolmafia.request.GenericRequest`: a classe possui vários atributos estáticos públicos que não são constantes. Eles poderiam ser encapsulados para promover melhor integridade dos dados. A classe é uma candidata à reestruturação, pois possui vários métodos estáticos não relacionados diretamente ao TAD que a classe implementa.

- `org.jpilotexam.ui.util.UIUtilities`: é uma classe depositária de constantes e utilitária. A classe possui métodos estáticos utilitários referentes à manipulação de objetos em interfaces gráficas, por exemplo, para centralizar um componente gráfico em uma tela.

- `net.sf.cglib.core.CodeEmitter`: aparentemente, o único problema é que a classe possui um único método que não está relacionado ao TAD que ela implementa. Este método é estático e tem por objetivo verificar se um vetor de inteiros está ordenado.

- `net.sourceforge.kolmafia.persistence.ConcoctionDatabase`: a classe possui vários atributos públicos estáticos que não são constantes. Eles deveriam ser encapsulados. Todos os métodos da classe são estáticos. Seria necessário maior conhecimento do domínio do problema para afirmar que os metodos têm alguma relação lógica entre si. É dificil perceber a que a classe se destina. Seria interessante verificar se é possível dividir as responsabilidades dela em mais de uma classe para melhorar o entendimento do programa.

- `org.apache.bcel.generic.Type`: os atributos públicos são constantes. A classe pertence à biblioteca BCEL que lida com o problema de analisar *bytecode* de programas Java. Neste contexto, a classe representa tipos de dados. Ela possui atributos e seus respectivos métodos *get* e *set*. Além disso, possui métodos estáticos utilitários. Por exemplo, possui um método para converter dois tipos passados como parâmetro para uma *string* que representa uma assinatura de um método. A classe poderia ser dividida, então, em pelo menos duas.

- `org.yawlfoundation.yawl.engine.interfce.WorkItemRecord`: os atributos públicos são todos constantes. Os métodos são todos do tipo *get* e *set*, sendo que alguns dos métodos do tipo *get* convertem alguns dos dados, por exemplo, para formato XML. Há um grupo de atributos que parecem representar um tipo abstrato de dados particular referente a horário e seus valores correspondentes em milisegundos. Este grupo de dados poderiam ser extraídos para constituir uma nova classe.

- `net.sourceforge.kolmafia.AdventureResult`: os atributos públicos são constantes. A classe implementa um TAD. Entretanto, possui alguns métodos estáticos que não são diretamente relacionados ao TAD implementado. Eles poderiam ser extraídos para uma outra classe.

- `net.sourceforge.kolmafia.request.UseSkillRequest`: a maior parte dos atributos públicos é constante. Porém, a classe possui também atributos públicos que não são constantes. Eles deveriam ser encapsulados. A classe possui vários métodos estáticos não relacionados à implementação do TAD. Eles

poderiam ser extraídos para uma outra classe. Além disso, é herdeira da classe `GenericRequest`, que possui também problemas estruturais.

- `net.sourceforge.kolmafia.KoLCharacter`: a maior parte dos atributos públicos é constante. Porém, a classe possui também atributos públicos que não são constantes. Eles deveriam ser encapsulados. A classe é uma grande coleção de métodos estáticos. É difícil identificar relação lógica entre os métodos. Ela é uma boa candidata à reestruturação.

- `net.sf.jasperreports.engine.util.JRProperties`: é uma classe depositária de constantes, tais como nome do arquivo no qual são armazenadas propriedades do programa. Possui somente vários métodos estáticos. A classe não representa um tipo abstrato de dados. É uma classe utilitária que provê métodos para manipulação das propriedades do aplicativo.

- `net.sf.jasperreports.engine.design.JRDesignChart`: os atributos públicos são todos constantes. Possui basicamente somente métodos *get* e *set*. Há uma seção da classe constituída por 15 atributos privados usados apenas localmente dentro de um método, que utiliza somente esses dados. Ou seja, eles parecem ser casos de variáveis locais e não de variáveis de instância. Além disso, o referido método parece não estar diretamente relacionado ao restante da classe.

- `net.sourceforge.kolmafia.request.EquipmentRequest`: os atributos públicos são constantes. A classe implementa um TAD. Entretanto, possui alguns métodos estáticos que não são diretamente relacionados ao TAD implementado. Eles poderiam ser extraídos para uma outra classe. Além disso, é herdeira de uma classe denominada `PasswordHashRequest`, que herda de `GenericRequest`, que possui problemas estruturais. Como várias classes desta hierarquia possuem problemas estruturais, conforme resultados das avaliações descritas anteriormente, a hierarquia é uma candidata à reestruturação.

- `net.sourceforge.kolmafia.persistence.SkillDatabase`: os atributos públicos são constantes. A classe é uma coleção de métodos estáticos. É difícil afirmar se há relação lógica entre eles. Isso é agravado pelo fato de haver outras duas classes na hierarquia de herança: ela é herdeira de `KolDataBase`, que por sua vez, herda de `StaticEntity`. As classes envolvidas nesta hierarquia não correspondem a tipos abstratos de dados. A herança não parece ter sido aplicada para subtipagem. O relacionamento de herança, então, não parece legítimo, embora seja necessário maior conhecimento do domínio do problema para afirma isso.

- `net.sourceforge.kolmafia.swingui.panel.ItemManagePanel`: a classe possui seis atributos públicos que não são constantes nem estáticos. Eles poderiam ser encapsulados. A compreensão da classe como um todo é dificultada porque há várias outras classes envolvidas na hierarquia de herança: sua classe mãe é `Scrolllabelpanel`, que herda de `Actionpanel`, uma classe reutilizada de outro pacote denominado `net.java.dev.spellcast.utilities`, que por sua vez herda de `JRootPane`, uma classe do pacote `swing` de Java. Embora o uso de herança pareça legítimo, a quantidade de classes envolvidas dificulta a compreensão da classe analisada.

- `com.acelet.lib.CommonPanel`: possui três atributos públicos que não são constantes. Eles poderiam ser encapsulados a fim de evitar que valores inválidos possam ser atribuídos a eles. É uma classe abstrata que representa um componente gráfico genérico. Além de métodos *get* e *set*, possui métodos para realizar tarefas como compor um *grid* no painel e compor os botões comuns.

Todas essas classes foram classificadas dentro da faixa *ruim* das métricas *número de conexões aferentes*, *LCOM* e *número de atributos públicos*. Em todos os casos, a inspeção das classes revela problemas estruturais. Em vários casos, identifica-se que a classe realiza mais de uma função. Com isso, é de se esperar que sua coesão não seja alta e que haja uma tendência para que a classe atenda a um grande número de outras classes. Embora a métrica LCOM não seja considerada ideal para avaliar coesão interna de classes, neste estudo de caso a aplicação de seu valor referência e da métrica *número de conexões aferentes* serviram para identificar classes potencialmente deficientes quanto à coesão. O valor referência da métrica *número de atributos públicos* também mostraram-se úteis para identificar classes que são meras depositárias de dados e classes que realmente possuem indevidamente dados públicos. A primeira categoria de classes não necessariamente representa necessidade de refatoração ou ameaça à estrutura do software. Todavia, a segunda categoria fere boas práticas de projeto de software. Nesses casos, os dados deveriam ser encapsulados.

Dentro do conjunto de classes avaliadas, há oito que possuem *número de métodos públicos* classificados como *ruins*. Uma observação comum entre essas classes é que elas possuem muitas responsabilidades e são de difícil compreensão. Ainda que o número dessas classes possa ser considerado pequeno, a aplicação do valor referência dessa métrica pode ser utilizado, neste estudo de caso, para identificar classes que possuem muitas responsabilidades.

Um grupo também pequeno de classes dentro do conjunto avaliado possui DIT superior a 3, que é maior do que valor típico dessa métrica. A observação comum

na avaliação dessas cinco classes é que tanto elas quanto as classes envolvidas nas suas respectivas hierarquias apresentam problemas estruturais. Em alguns dos casos, observaram-se indícios de que a herança não está legitimamente aplicada. Desta forma, embora o número de classes nesta situação seja muito pequeno, neste estudo de caso o valor referência da métrica DIT pode ser utilizado para identificar classes que podem ter problemas relacionados à aplicação do recurso de herança.

Este estudo de caso baseou-se no uso dos valores referência das métricas das faixas *ruins* para auxiliar a identificação de classes com deficiências estruturais. O objetivo do estudo foi verificar se a aplicação dos valores referência identificam classes com problemas estruturais quando elas realmente os possuem. As classes foram analisadas qualitativamente. Os resultados dessa avaliação indicam que a aplicação desses valores referência auxiliaram adequadamente na identificação de tais classes. Uma ameaça para a validade dessa conclusão é que a avaliação qualitativa pode não ter um nível de acurácia alto, pois pode ser afetada pela falta de grande conhecimento do domínio do problema de cada software avaliado.

5.5.2 Estudo de Caso 2

O objetivo deste estudo de caso é avaliar a acurácia dos valores referência identificados neste trabalho. Em particular, o estudo verifica se os valores referência podem ser utilizados para identificar corretamente classes de boa estrutura. Para isso, foi utilizado um software cuja estrutura tem sido qualitativamente avaliada como boa, JHotDraw. Com isso, é esperado que boa parte dos valores aferidos de suas métricas estejam dentro das faixas recomendadas como *boas* ou pelo menos *regular*. JHotDraw possui 1.095 classes e COF igual 0,003. Este valor de COF está de acordo com a faixa classificada como *boa* para essa métrica. A seguir é descrita a comparação dos valores aferidos em JHotDraw com os valores referência das demais métricas.

- *Conexões Aferentes:* apenas 29 classes do software possuem mais de 20 conexões aferentes, 384 possuem de 2 a 20 conexões aferentes e 682 classes possuem menos do que 2 conexões aferentes. Considerando-se apenas esse aspecto, isso indica que 97% das classes do software têm qualidade boa ou pelo menos regular.

- *LCOM:* 215 classes do software possuem LCOM superior a 20, 335 classes possuem LCOM de 1 a 20 e 545 classes possuem LCOM igual a 0. Isso significa que, considerando-se a métrica LCOM isoladamente, 80% das classes têm coesão boa ou pelo menos regular.

- *DIT:* 344 classes possuem DIT superior a 2, enquanto 751, o que corresponde a 70% das classes, possuem DIT até 2. A média dos valores de DIT do software é 2,3. Isso significa que o uso de herança aplicado no software é compatível com o valor típico identificado para a métrica.

- *Número de atributos públicos:* 945 classes não possuem atributos públicos, 146 classes possuem de 1 a 10 atributos públicos e quatro classes possuem mais de 10 atributos públicos. Considerando-se essa métrica isoladamente, praticamente a totalidade das classes do software são avaliadas como boas ou pelo menos regulares.

- *Número de métodos públicos:* 925 classes possuem até 10 métodos públicos, 156 classes possuem de 11 a 40 métodos públicos e 14 classes possuem mais de 40 métodos públicos. Considerando-se apenas essa métrica, 99% das classes do software são avaliadas como boas ou pelo menos regulares.

Tabela 5.8. Classes de JHotDraw e valores de suas métricas #CA (conexões aferentes), LCOM, DIT, #AP (atributos públicos) e #MP (métodos públicos)

Classe	**#CA**	**LCOM**	**DIT**	**#AP**	**#MP**
`org.jhotdraw.geom.Geom`	50	276	1	4	42
`org.jhotdraw.draw.AbstractFigure`	49	1339	2	0	45
`org.jhotdraw.draw.DefaultDrawingView`	34	2573	4	1	58
`org.jhotdraw.draw.BezierFigure`	25	184	4	0	51
`org.jhotdraw.draw.action.ButtonFactory`	23	496	1	8	69

Considerando-se cada métrica isoladamente, um número relativamente baixo de classes foram classificadas como *ruins*. O software foi analisado também a partir de associação de métricas da forma descrita a seguir. Apenas 18 classes de JHotDraw possuem LCOM e número de conexões aferentes classificados como *ruins*. Dentre essas classes, somente uma delas, `org.jhotdraw.draw.AttributeKeys`, possui número de atributos públicos classificado como *ruim*. Esta classe, porém, trata-se de um repositório de constantes utilizadas no sistema, o que não representa necessariamente uma necessidade de reestruturação. Dentre aquelas 18 classes, apenas seis possuem DIT superior a dois. Apenas seis delas também possuem número de métodos públicos classificados como *ruim*. Os dados destas classes são mostrados na Tabela 5.8. Se todos os fatores correspondentes às métricas forem considerados em conjunto, nenhuma classe do sistema é classificada como *ruim*.

Aplicando-se os valores referências das métricas para avaliar o software JHotDraw, conclui-se que o software possui boa estrutura. Tomando-se JHotDraw como um exemplo de software bem estruturado, os resultados desse estudo de caso, então, indicam que esses valores referência avaliam a estrutura de um software como boa quando ela realmente o é.

5.5.3 Discussão

Os valores referência propostos mostraram-se efetivos nos estudos de caso realizados. Os resultados obtidos mostram que quando uma métrica é classificada como *ruim*, não significa que a classe ou o software definitivamente viola um princípio de projeto. Entretanto, métricas classificadas como *ruim* denotam um sintoma que indica fortemente um problema de projeto. Para DIT, foi identificado um valor típico em vez de faixas de valores referência. Considerando-se que árvores de herança profundas podem tornar a manutenção do software difícil, é desejável manter DIT baixo. Desta forma, classes com DIT superior ao valor típico identificado devem ser tomadas como um sintoma de que a herança talvez não tenha sido usada adequadamente no sistema, o que, então, recomenda reestruturação da árvore de herança.

As faixas *bom* e *regular* não foram avaliadas individualmente porque, com análise qualitativa, é difícil diferenciar com precisão se uma classe tem qualidade *boa* ou *regular*. Considera-se que JHotDraw, o software usado para avaliar as faixas *bom* e *regular*, tem projeto de boa qualidade. Porém, é possível que a sua estrutura possa ser ainda melhorada. Com isso, assumiu-se que a estrutura desse software tem qualidade pelo menos *regular*, e as faixas *bom* e *regular* foram, então, avaliadas em conjunto. Para todas as métricas, a grande parte das classes de JHotDraw foram classificadas na faixa *bom*, o segundo maior grupo de classes foi avaliado como *regular*, e apenas poucas classes foram avaliadas como *ruim*. Esse resultado é consistente com a reconhecida alta qualidade da estrutura do software analisado. As observações realizadas nesse estudo de caso indicam que as faixas *bom* e *regular*, bem como o valor típico de DIT, mostraram-se confiáveis.

Outra observação no estudo de caso realizado com JHotDraw é que as distribuições dos valores das métricas desse software têm as mesmas propriedades observadas no conjunto de softwares utilizados na derivação dos valores referência: o valor médio de DIT em JHotDraw é compatível com o valor típico identificado para essa métrica, e, para as outras métricas, a maior parte das classes apresentam valores baixos, enquanto poucas classes apresentam valores altos. Esse resultado suporta a conclusão de que há uma única distribuição que modela valores de uma métrica,

independente de tamanho, tipo ou domínio de aplicação do software.

A faixa *regular* pode ser considerada como uma recomendação de que a estrutura do software deve ser melhorada, embora ela não represente necessariamente uma ocorrência de anomalia de projeto. Entretanto, a fronteira entre as faixas *regular* e *ruim* é sutil e não é muito precisa. Como o método usado para derivar as faixas baseou-se em análise gráfica, seria possível derivar outros limites inferiores e superiores para a faixa *regular*, aplicando-se a mesma análise sobre os mesmos gráficos. Por exemplo, seria possível concluir que o limite superior da faixa *regular* é 15 ou 25, em vez de 20. Desta forma, valores de métricas próximos do limite superior da faixa *regular* podem ser interpretados também com um indicador de possível violação de princípio de projeto.

Os valores referência foram derivados com base na prática comum. Esse método pode ser considerado por alguns como enganoso, já que a prática comum não necessariamente representa a melhor prática. Contudo, os valores referência derivados são compatíveis com princípios de projeto bem conhecidos, tais como a ocultação de informação, poucas conexões, alta coesão e *favoreça composição de objetos em relação à herança*. Além disso, as observações dos estudos de casos realizados nessa pesquisa mostraram que os valores referência propostos podem auxiliar engenheiros de software na identificação de falhas de projeto relacionadas com esses princípios de projeto.

5.6 Limitações

As diversas ferramentas existentes podem coletar uma mesma métrica de forma diferente. Por exemplo, uma ferramenta pode coletar a métrica COF considerando as relações de herança entre classes, enquanto outras não. Do ponto de vista de aplicação dos resultados encontrados neste estudo, diferentes interpretações das definições das métricas representam uma ameaça à validade do estudo. Para evitar esse problema, foram descritas as interpretações utilizadas na implementação das métricas avaliadas neste estudo.

A amostra de softwares usada é em si uma ameaça para a validade desse estudo. Embora a amostra possua uma grande quantidade de softwares, não é possível garantir que eles são os melhores representantes da prática comum. Além disso, apesar de softwares bem conhecidos fazerem parte do conjunto analisado neste estudo, não se pode afirmar que todos os software da amostra possuem alta qualidade estrutural. Dentre eles, BCEL, uma biblioteca para manipulação de *bytecode* de programas Java, foi avaliado qualitativamente. Esta biblioteca foi utilizada na implementação de *Connecta*. A partir da avaliação qualitativa de BCEL, conclui-se que ele é um software bem cons-

truído, modular e fácil de se utilizar. A avaliação qualitativa, porém, pode ser errônea, especialmente no caso de softwares de grande porte.

Neste estudo foram considerados apenas programas desenvolvidos em Java. Todavia, é possível que os valores referência possam ser influenciados pela linguagem de programação, o que limitaria a generalização dos valores referência propostos.

Alguns podem julgar os valores referência de LCOM como inválidos, já que as imperfeições dessa métrica poderiam levar a valores referência sem significado. Entretanto, as imperfeições de LCOM consistem em limitações da métrica e não dos valores referência identificados para ela ou do método empregado para derivá-los. A principal razão pela qual essa métrica foi avaliada neste estudo é o seu amplo uso. Apesar das limitações dessa métrica, os valores referência identificados para ela podem ser ainda úteis para aqueles que a usam. Outros trabalhos são necessários para identificar valores referência para outras métricas de coesão, a fim de prover recursos para uma melhor avaliação de software no aspecto de coesão.

Os estudos de caso realizados para avaliar os valores referência propostos são baseados em avaliação qualitativa de software. No primeiro estudo de caso, foram inspecionadas classes de vários softwares. Essa avaliação é sujeita a falha porque, em alguns casos, a avaliação adequada das classes requer conhecimento sólido sobre o domínio do problema tratado pelo software. No segundo estudo de caso, com base em relatos de trabalhos anteriores, considera-se que a estrutura do software usado tem alta qualidade. Essa premissa, porém, depende da validade desses trabalhos.

5.7 Conclusões

Este capítulo apresentou os resultados de um estudo realizado com uma grande quantidade de softwares abertos orientados por objetos. Foram analisados 40 softwares desenvolvidos em Java, incluindo ferramentas, bibliotecas e *frameworks*, de 11 domínios de aplicações diferentes, perfazendo um total de mais 26.000 classes. A partir dos resultado obtidos, foram identificados valores referência para seis métricas de software: COF, LCOM, DIT, conexões aferentes, número de atributos públicos e número de métodos públicos. O estudo concluiu que os valores dessas métricas, com exceção de DIT, podem ser modelados por uma distribuição de cauda pesada. Esta propriedade significa que para essas métricas há um número baixo de ocorrência de valores altos, enquanto há uma frequência muito alta de valores baixos. Os valores de DIT podem ser modelados pela distribuição de Poisson, com valor médio igual a 2.

Com base nos valores mais comumente aplicados na prática, foram derivados va-

lores referência gerais para as métricas. Foram derivados também valores referência para as métricas por domínio de aplicação, tamanho e tipo (ferramenta, biblioteca e *framework*) de software. Como não se evidenciaram diferenças relevantes entre os resultados dessas análises, acredita-se que os valores referência gerais possam ser aplicados para sofware orientado por objetos em geral.

Os valores referência foram avaliados por meio de dois estudos de caso, que mostraram que esse valores podem ser aplicados para classificarem satisfatoriamente classes quanto à sua qualidade estrutural. Um dos usos desses valores referência é no auxílio à identificação de classes candidatas à refatoração. Os resultados deste estudo podem ajudar engenheiros de software na tarefa de avaliação de software por meio de métricas.

Há dezenas de métricas de software. No escopo deste trabalho, foram definidos valores referência para seis delas. É necessário um esforço para identificar valores referência para outras métricas de software para que a aplicação delas seja encorajada. Sugere-se adotar a abordagem utilizada neste estudo para definir valores referência de outras métricas de software.

Os valores identificados neste estudo podem ser utilizados em conjunto com o modelo K3B, pois as métricas avaliadas neste estudo avaliam fatores relacionados à dificuldade de manutenção de software. O modelo estima o impacto de modificação em software orientado por objetos. Diante de uma necessidade de reestruturar o software para amenizar os impactos de modificação, é importante identificar os pontos do software que precisam ser melhorados. Realizar esta tarefa em software de grande porte manualmente pode ser extremamente difícil. Essa tarefa pode ser automatizada utilizando-se métricas de software. Entretanto, ainda assim, sem o conhecimento de valores referência das métricas, a tarefa é desafiadora.

Capítulo 6

Métrica de Coesão de Responsabilidade

Modularidade é a característica de um software construído a partir de unidades básicas, denominadas módulos. Há dois caminhos principais para se obter modularidade adequada de software: minimizar o relacionamento entre módulos, o que se denomina acoplamento (*coupling*), e maximizar o relacionamento entre elementos no mesmo módulo, o que se denomina coesão (*strength* ou *cohesion*). Coesão em software é definida como o grau de intercomunicação entre os elementos internos de um módulo. Myers [1975] definiu uma escala de grau de coesão interna de módulos que desde então foi adotada e aceita pela comunidade. Essa escala é qualitativa e baseia-se na observação e na análise do relacionamento entre os elementos de um módulo. O pior nível de coesão ocorre naquele módulo em que não há relacionamento com significado relevante entre os seus elementos. Nesta situação, os elementos do módulo parecem ter sido reunidos ao acaso, portanto não é possível descrever a função do módulo. Esse nível é conhecido como coesão coincidental. O melhor nível de coesão é aquele em que o módulo desempenha uma única função bem definida, o que é conhecido como coesão funcional [Myers, 1975] .

É um consenso que a coesão interna de módulos em um software é determinante de sua qualidade. A avaliação quantitativa dessa característica é essencial para avaliação da qualidade do software. Avaliar quantitativamente a coesão de um módulo é difícil, pois dizer se os elementos de um módulo possuem relacionamento ou desempenham uma única função bem definida muitas vezes envolve conhecer o domínio do problema. Esta tarefa tem sido tema de muitos trabalhos pelo fato de a coesão ser um fator de grande importância na compreensão e na manutenção de software [Chidamber & Kemerer, 1994; Briand et al., 1998; Marcus & Poshyvanyk, 2005;

Counsell et al., 2006; Mäkelä & Leppänen, 2007; Al-Dallal, 2009]. Entretanto, ainda não há um consenso na literatura sobre uma métrica padrão para coesão interna de classes.

Este capítulo apresenta uma nova métrica de coesão interna de classes, denominada Coesão de Responsabilidade (Cohesion of Responsibility - COR). Há duas motivações principais para se definir uma nova métrica de coesão em vez de se utilizar uma das diversas métricas já definidas na literatura. A primeira delas é que o modelo K3B, proposto nesta tese, tem como parâmetro um fator denominado β que pode ser estimado por meio da avaliação da coesão interna das classes do software. Para isso, o ideal é que valores baixos da métrica correspondam a baixa coesão, enquanto valores altos refletam alta coesão. Algumas métricas de coesão já propostas avaliam a ausência de coesão, gerando valores contrários: valor nulo representa alta coesão, enquanto um valor alto representa baixa coesão. Isso não é necessariamente impeditivo para aplicação dessas métricas como fator β, mas pode tornar essa aplicação mais trabalhosa. A outra motivação é que os valores gerados pela maior parte das métricas já propostas não fornecem uma informação direta e simples sobre os problemas de coesão da classe e o que poderia ser feito para melhorá-la. A métrica proposta neste capítulo é baseada no princípio SRP (*the single responsibility principle*), segundo o qual uma classe deve ter uma única razão para ser modificada [Martin, 2002]. Esse princípio é alinhado ao de coesão máxima de classes em um software orientado por objetos: cada classe deve implementar uma única responsabilidade. A métrica proposta visa contar o número de responsabilidades que uma classe possui, fornecendo um indicador de fácil interpretação sobre a estrutura da classe. Os valores dessa métrica estão no intervalo $[0, 1]$. O valor 1 indica que a classe implementa uma única responsabilidade, o que corresponde a alto grau de coesão da classe avaliada.

O restante deste capítulo está organizado da seguinte forma: a Seção 6.1 avalia trabalhos relacionados à medição de coesão interna de classes; a Seção 6.2 define a métrica COR; a Seção 6.3 avalia a métrica proposta; a Seção 6.4 identifica os valores referência da métrica; a Seção 6.5 apresenta as conclusões do trabalho.

6.1 Trabalhos Relacionados

Um estudo comparativo entre métricas de coesão interna de classes realizado por Briand et al. [1998] cita sete métricas com esse propósito. Outras métricas de coesão, de diferentes abordagens, foram propostas depois desse levantamento. A Seção 4.1 apresenta uma revisão de algumas dessas métricas. A grande quantidade de métricas

de coesão interna de classe sinaliza que coesão é notadamente algo muito difícil de se medir com precisão. Avaliar a coesão interna de uma classe é uma tarefa intimamente dependente da compreensão do domínio do problema. Isso inibe a avaliação de coesão interna de classe meramente com base em métrica e dispensando o julgamento do especialista. Por outro lado, o trabalho do especialista pode ser apoiado em métricas. Para que isso seja alcançado, o ideal é a que métrica seja simples, fácil de computar e que seu resultado traga informações significativas ao especialista.

As métricas de coesão propostas na literatura sofrem de três problemas principais: (i) algumas têm definição complexa, de difícil compreensão, como é o caso de ELCOM [Mäkelä & Leppänen, 2007], C3 [Marcus & Poshyvanyk, 2005] e CAMC [Counsell et al., 2005]; (ii) os valores resultantes de boa parte das métricas não se traduzem em indicadores de fácil compreensão, que possam orientar o especialista na decisão de reestruturação da classe; (iii) muitas dessas métricas, como LCOM [Chidamber & Kemerer, 1994], avaliam *ausência de coesão* e não a coesão interna da classe, o que também prejudica a interpretação de seus resultados.

A métrica proposta nesta tese, COR, foi definida com o objetivo de contornar esses problemas. Com base no conceito de que uma classe coesa deve implementar uma única responsabilidade, a métrica foi definida de forma a resultar em valores que indiquem a quantidade de responsabilidades que uma classe implementa. Desta forma, seus resultados apoiam a tarefa de avaliação e reestruturação de classes, que é o objetivo principal de uma métrica de coesão de classe. Neste trabalho, COR é avaliada a partir da comparação de seus resultados com os gerados pela métrica LCOM. Além disso, são identificados valores referência para a métrica.

6.2 Definição da Métrica de Coesão de Responsabilidade

Coesão é o grau de relacionamento entre os elementos internos de um módulo. Os elementos internos de uma classe são atributos e métodos. O grau de coesão interna de uma classe é dado, então, pelo relacionamento entre seus atributos e métodos. Um conceito correlato importante na avaliação de coesão interna de classes é o de *responsabilidade*. Uma classe deve possuir uma única responsabilidade. Por exemplo, uma classe que implementa os tipos abstratos de dados fila e pilha possui duas responsabilidades. Tal classe poderia ser dividida em três: uma que implemente uma lista, e duas classes herdeiras desta, uma que implemente uma pilha e outra que implemente uma fila.

A definição da métrica proposta, Coesão de Responsabilidade (Cohesion of Re-

sponsibility - COR), baseia-se nos seguintes conceitos:

1. *Responsabilidade:* um conjunto de métodos dentre os quais há *relacionamento* representa uma responsabilidade implementada pela classe.

2. *Relacionamento:* dois métodos de uma classe C estão relacionados se utilizam pelo menos um atributo da classe C em comum, ou se um utiliza o outro, ou se usam direta ou indiretamente métodos da classe C em comum. Um método a utiliza diretamente um método b se há uma chamada a b no corpo de a. Um método a utiliza indiretamente um método b se em a há uma chamada a algum método que pode levar a chamada de b.

3. *Propriedade transitiva de relacionamento:* se um método a está relacionado com um método b e b está relacionado com um método c, então a está relacionado com c.

O cálculo da métrica COR é realizado da seguinte forma:

- S é o conjunto de conjuntos disjuntos formados por métodos com relacionamento entre si na classe.
- Número de responsabilidades da classe: é a quantidade de conjuntos disjuntos formados por métodos com relacionamento entre si na classe, dada por $r = |S|$

  Assim,

  $COR = 1/r$, se $r > 0$

  $COR = 0$, caso contrário.

A Coesão de Responsabilidade é dada por $1/r$, onde r é o total de responsabilidades da classe. A métrica pode assumir valores no intervalo $[0, 1]$, sendo que quanto mais próximo de 1, melhor a coesão da classe. COR resulta em zero somente se a classe não possuir método algum. Opcionalmente, pode-se utilizar o valor de r, que indica diretamente o número de responsabilidades da classe.

De acordo com Briand et al. [1998], a definição de uma métrica de coesão deve levar em consideração três aspectos principais: herança, métodos de acesso a dados da classe (métodos *get* e *set*) e métodos construtores. Em relação à herança é possível considerar ou não os métodos e atributos herdados. A definição de COR não restringe

qualquer uma dessas possibilidades. A exclusão de elementos herdados na análise da coesão da classe indica que se pretende avaliar o grau de coesão da extensão da classe. A inclusão dos elementos herdados na análise indica que se pretende avaliar a coesão da classe como um todo. Entretanto, posto que um software é constituído por módulos que se relacionam entre si, que uma classe é um módulo, que herança é um tipo de relacionamento entre classes e que busca-se avaliar coesão interna de módulos (classes), entendemos que a avaliação dos elementos definidos na classe, ou seja, excluindo os elementos herdados, pode favorecer uma avaliação mais fiel da coesão interna de classe.

A inclusão de métodos de acesso no cálculo da métrica também não é restringida na definição da métrica. No caso de algumas métricas propostas na literatura, como é o caso de LCOM, a inclusão de métodos de acesso diminuem artificialmente a coesão da classe. Na métrica COR, entretanto, isso não ocorre, pois tais métodos estarão inseridos no conjunto de outros métodos que usam os respectivos atributos. Classes que possuem apenas campos e métodos de acesso são definidas por Fowler [1999] como o *bad code smell* denominado *Data Class*. Nesse tipo de classe, o resultado da métrica COR indicará a quantidade de atributos da classe, indicando falta de coesão da classe. Esse comportamento é desejável, já que classes com essa característica são consideradas um *bad smell*.

Semelhante ao que ocorre com outras métricas de coesão, a inclusão de método construtor pode aumentar artificialmente o valor da métrica COR, uma vez que é comum que esse tipo de método faça referência a uma grande quantidade de atributos da classe. Desta forma, o ideal é excluir esses métodos do cálculo de COR.

6.2.1 Comparação de COR com Outras Abordagens

Os conceitos empregados em COR não diferem substancialmente dos conceitos empregados nas métricas mais conhecidas para coesão. A ideia de avaliar coesão por meio da análise de relacionamento entre métodos é empregada em métricas como LCOM. Em particular a métrica LCOM4 [Hitz & Montazeri, 1995] modela o relacionamento entre métodos de uma classe como um grafo, no qual os métodos da classe são os vértices e as arestas representam relacionamentos entre dois métodos. Nesse grafo, há uma aresta entre dois métodos se eles usam um atributo da classe em comum, ou se um método chama o outro. A métrica LCOM4 é definida como o número de componentes conectados no grafo. Na abordagem da métrica COR proposta no presente trabalho, um componente conectado na métrica LCOM4 corresponderia a uma responsabilidade. Entretanto, LCOM4 não considera o uso indireto de atributos pelos métodos da classe.

A principal vantagem de COR em relação às demais métricas já propostas na literatura é sua facilidade de interpretação dos resultados. Além disso, COR resulta em valores independentes do tamanho da classe, no intervalo $[0, 1]$. Métricas como LCOM indicam a ausência de coesão, sendo que valores baixos indicam alta coesão e valores altos indicam baixa coesão. Os valores de COR, ao contrário, indicam o nível de coesão da classe: valores baixos indicam baixa coesão, e valores altos indicam alta coesão.

6.3 Avaliação da Métrica COR

A avaliação da métrica COR foi realizada a partir da comparação de seus resultados com a avaliação qualitativa de um conjunto de classes. Os resultados também foram comparados aos valores da métrica LCOM [Chidamber & Kemerer, 1994]. Os resultados das medidas obtidas foram classificados da seguinte maneira: *positivo*, se o valor da medida está de acordo com a avaliação qualitativa; *falso negativo*, se a métrica aponta baixa coesão e a avaliação qualitativa aponta alta coesão; *falso positivo*, se a métrica aponta alta coesão e a avaliação qualitativa aponta baixa coesão.

As seis primeiras classes avaliadas são de aplicações desenvolvidas por alunos aprendizes de orientação por objetos. As três classes seguintes são da ferramenta de coleta de métricas denominada *Connecta* [Ferreira, 2006]. A avaliação qualitativa das classes é descrita a seguir:

1. `Autor`: uma classe que representa um Autor. Possui dados como código, nome e e-mail. A classe possui 7 métodos: um método *get* e *set* para cada atributo, e um método construtor que invoca os métodos *set* para os atributos da classe. Embora a classe represente uma única entidade do domínio do problema, ela enquadra-se como uma *Data Class*, pois possui somente métodos de acesso.

2. `Artigo`: uma classe que implementa um Artigo. Possui os dados código, título, palavras-chave, número de páginas, ano de publicação e lista de autores. A classe possui 14 métodos: um método *get* e *set* para cada atributo, um método construtor que invoca os métodos *set* para os atributos da classe e um método que permite incluir autor na lista de autores. Embora a classe represente uma única entidade do domínio do problema, ela enquadra-se como uma *Data Class*.

3. `GestaoArtigo`: possui métodos para manipular lista de artigos (inclusão, pesquisa, alteração etc.) cadastrados no sistema. Possui também um método que, dados dois artigos, identifica seus autores em comum. A classe implementa duas funcionalidades: a primeira, legítima, que é a manipulação da lista de artigos do sistema; a segunda é a verificação de autores em comum entre dois artigos, que deveria ser implementada na classe `Artigo`. Essa classe possui, então, duas responsabilidades.

4. `Cliente`: possui sete atributos de cliente e seus respectivos métodos *get* e *set*. Não possui método construtor implementado. Embora a classe represente uma única entidade do domínio do problema, ela enquadra-se como uma *Data Class*.

5. `Acomodacao`: possui cinco atributos. O método construtor inicia apenas um dos atributos. Embora a classe represente uma única entidade do domínio do problema, ela enquadra-se como uma *Data Class*.

6. `Hotel`: classe principal da aplicação de reservas de acomodações em um hotel. Possui um método principal que chama outros da mesma classe, que executam funções tais como: incluir reserva, excluir reserva, incluir cliente, excluir cliente, incluir acomodação e excluir acomodação. A classe não implementa um tipo de dado, ela agrega um conjunto de tarefas procedimentais para a execução do software. Poderia ser melhorada se fosse dividida em quatro classes: a principal, que representa o hotel, uma de gestão de cliente, uma de gestão de acomodações e outra de gestão de reservas. A coesão da classe não pode ser considerada alta, pois não representa um único papel bem definido. Contudo, também não é baixa. Dentro da escala de coesão proposta por Myers [1975], avalia-se a coesão dessa classe como comunicacional, o segundo melhor nível de coesão. Em um módulo com este tipo de coesão, seus elementos pertencem ao domínio do problema a ser solucionado e 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. No caso dessa classe, as ações de incluir acomodação e incluir cliente devem ser executadas antes de incluir reserva, e as ações de exclusão são dependentes dos dados incluídos.

7. `ClassModule`: representa uma classe no software avaliado. É uma classe coesa, pois todos seus dados e métodos referem-se à representação deste tipo de dado.

8. `ConnectionPath`: representa um caminho entre duas classes. É uma classe coesa, pois todos seus dados e métodos referem-se à representação deste tipo de dado. Porém, há um método nesta classe, denominado `printPath`, que não executa ação alguma.

9. `ClassCollector`: esta classe implementa pelo menos três responsabilidades. A primeira responsabilidade, legítima, é a coleta de métricas de uma classe. A segunda responsabilidade é implementada pela disponibilização de dois métodos que permitem incrementar o número de conexões aferentes e eferentes da classe. Essas são duas métricas de classes que correspondem, respectivamente, ao número de outras classes do sistema que utilizam a classe avaliada e ao número de outras classes do sistema que são utilizadas pela classe avaliada. Esses são dados que deveriam fazer parte de outra classe do software, `ClassModule`, e não de `ClassCollector`. A classe possui um método, denominado `openLogFile` cujo objetivo é abrir um arquivo de `log` a ser utilizado pelos outros métodos. Entretanto, ele não chegou a ser utilizado em nenhum método da classe.

10. `Env`: é uma classe que implementa uma tabela de símbolos de um compilador [Aho et al., 2008]. A classe é coesa, uma vez que tem um propósito bem definido e todos os seus métodos destinam-se à implementação desse propósito.

Além dessas classes, foram avaliadas quatro outras classes: uma classe denominada `PilhaFilaLista`, que agrega a implementação dos tipos abstratos de dados pilha, lista e fila de inteiros; as classes `Pilha`, `Lista` e `Fila` que implementam os respectivos tipos de abstratos de dados. A classe `PilhaFilaLista` é obviamente pouco coesa, uma vez que implementa três responsabilidades. Os resultados da avaliação da métrica COR são relatados na Tabela 6.1.

Dos 14 casos analisados, os resultados de LCOM discordam da avaliação qualitativa em 4 casos. Os resultados de COR estão de acordo com a avaliação qualitativa. Há também de se observar que o valor fornecido por LCOM não é um indicador fiel do grau de coesão interna da classe, mesmo quando identifica corretamente que a classe possui problemas de coesão. No caso da classe `ClassCollector`, por exemplo, considerada pouco coesa na análise qualitativa, que identificou três responsabilidades na

Tabela 6.1. Avaliação da métrica COR

Classe	*Avaliação*	*LCOM*	*Comportamento LCOM*	*COR*	*Comportamento COR*
Env	Coesa	0	Positivo	1	Positivo
Autor	Data Class	15	Positivo	0,33	Positivo
Artigo	Data Class	73	Positivo	0,077	Positivo
GestaoArtigo	2 responsabilidades	0	Falso positivo	0,5	Positivo
Cliente	Data Class	91	Positivo	0,125	Positivo
Acomodacao	Data Class	46	Positivo	0,2	Positivo
Hotel	Boa coesão, mas não máxima	259	Negativo	0,5	Positivo
ClassModule	Coesa	17	Falso negativo	1	Positivo
ConnectionPath	Coesa, mas poluída por um método que faz nada	0	Falso positivo	0,5	Positivo
ClassCollector	3 responsabilidades	1292	Positivo	0,333	Positivo
PilhaFilaLista	3 responsabilidades	28	Positivo	0,333	Positivo
Pilha	Coesa	0	Positivo	1	Positivo
Fila	Coesa	0	Positivo	1	Positivo
Lista	Coesa	0	Positivo	1	Positivo

classe, LCOM resultou em 1292 e COR em 0,33. Com o valor 1292 é extremamente difícil perceber o que exatamente está errado em relação à coesão interna da classe. O valor 0,33 indica que há três responsabilidades implementadas na classe, o que orienta melhor a identificar os problemas de coesão da classe. O mesmo ocorre nos casos da classe `PilhaFilaLista`.

No caso das classes `Autor` e `Artigo`, que são classes do tipo *Data Class*, o resultado de COR seria falso positivo se o cálculo da métrica considerasse métodos construtores.

A coesão da classe `Hotel` foi considerada boa, mas não máxima na análise qualitativa. Na escala qualitativa, avaliou-se que a classe possui o segundo melhor nível de coesão, coesão comunicacional. O resultado de COR pode ser considerado positivo, pois a métrica identificou a existência de uma responsabilidade, mas não possui recursos para qualificar o nível da coesão neste caso.

Embora a quantidade de casos analisados neste estudo não seja extensa, ela fornece um indício da eficácia de COR na avaliação de coesão interna de classes. COR foi utilizada no processo de reestruturação de um software com 68 classes, tendo auxiliado satisfatoriamente a identificação de classes candidatas à reestruturação. A métrica COR foi coletada para todas as classes do sistema. Com base nos valores da métrica,

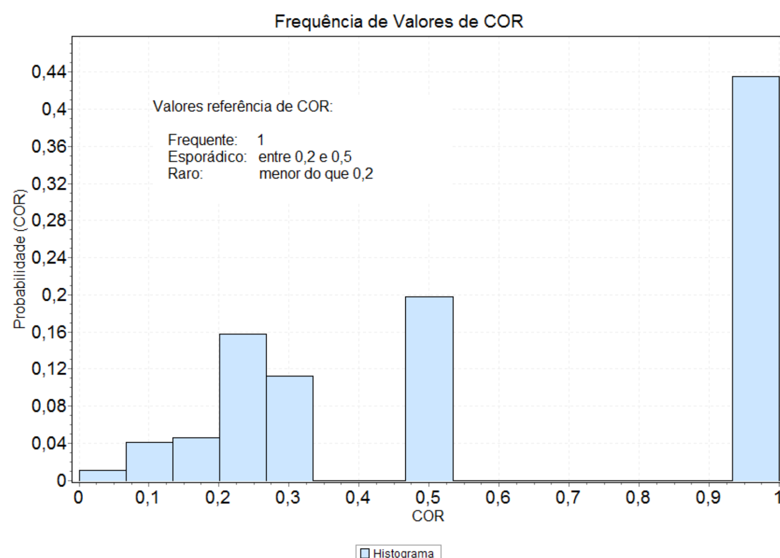


Figura 6.1. Frequência de valores de COR

foram identificadas as classes desse sistema que necessitaram ser subdivididas devido à deficiência de coesão interna.

6.4 Valores Referência da Métrica COR

Esta seção apresenta valores referência para a métrica COR. A metodologia utilizada é a mesma utilizada no estudo que identificou valores referência para um conjunto de métricas de software, apresentado neste trabalho no Capítulo 5. Os valores de COR foram coletados em mais de 20.000 classes de 40 programas abertos desenvolvidos em Java, de diferentes tamanhos e domínios de aplicação. Esse programas são os mesmos utilizados para a identificação dos valores referência das demais métricas de software, no estudo apresentado no Capítulo 5.

A frequência dos valores de COR é mostrada na Figura 6.1. Nesta distribuição de valores, identificam-se três faixas de valores: valores frequentes, valores esporádicos e valores raros. O valor 1 é o mais frequente, com 44% das ocorrências. Pela própria definição da métrica, valores entre 0,5 e 1 não ocorrem. Se tomados em conjunto, valores maiores ou iguais a 0,2 e menores ou iguais a 0,5 ocorrem com uma frequência próxima à do valor 1. Entretanto, se tomados isoladamente, valores maiores ou iguais a 0,2 e menores ou iguais a 0,5 ocorrem com uma frequência intermediária. Por exemplo, o valor 0,5 ocorre em 20% dos casos. Por esta razão, classificamos esta faixa de valores, de 0,2 a 0,5, como esporádicos. Valores menores do que 0,2 são raros.

6.5 Conclusão

Coesão interna de módulo, definida como o grau de interrelacionamento entre os elementos de um módulo, é um importante fator da modularidade e da qualidade de um software. Na orientação por objetos, classes correspondem a módulos. Nos últimos anos, um grande número de métricas de software, de diferentes abordagens, têm sido propostas para avaliação de coesão interna de classes. Apesar das importantes contribuições desses trabalhos, ainda não há um consenso sobre uma abordagem ideal de medição de coesão.

A métrica LCOM (ausência de coesão de métodos) é uma das mais referenciadas na literatura. A métrica tem sofrido muitas críticas, mas também tem sido inspiração para muitas outras propostas. De fato, LCOM não se mostra um indicador eficiente de coesão interna de classes. Contudo, sua base teórica é consistente: baseia-se na avaliação de similaridade entre métodos, de acordo com o qual dois métodos são similares se utilizam pelo menos um atributo em comum da classe.

Este capítulo definiu, avaliou e apresentou os valores referência de uma nova métrica de coesão interna de classes denominada Coesão de Responsabilidade (COR). A métrica baseia-se no princípio de responsabilidade única. COR é um indicador do número de responsabilidades que uma classe implementa. No cálculo dessa métrica, uma responsabilidade corresponde a um conjunto de métodos dentre os quais verifica-se relacionamento. Os valores de COR pertencem ao intervalo $[0, 1]$. A interpretação dos resultados ocorre de acordo com o seguinte padrão: 0 corresponde ao pior nível de coesão; 0,25 indica que a classe possi 4 responsabilidades; 0,33, indica que a classe possui 3 responsabilidades; 0,5 indica que a classe possui 2 responsabilidades; 1, indica que a classe implementa uma única responsabilidade, o que corresponde a alto grau de coesão.

A métrica proposta foi avaliada a partir da comparação de seus resultados com a avaliação qualitativa de um conjunto de 14 classes e com os valores de LCOM. Embora a quantidade de classes avaliadas não seja extensa, o estudo fornece indícios de que a métrica COR é um bom indicador do grau de coesão interna de classes. Na prática, a métrica pode ser utilizada para avaliação da qualidade de software e para auxiliar a identificar classes candidatas à reestruturação.

Capítulo 7

Evolução de Software - Uma Abordagem de Redes Complexas

O fenômeno de degradação de desenho é um problema clássico em Engenharia de Software. Apesar de todos os conhecimentos sobre construção de software de alta qualidade consolidados em princípios, critérios, regras, padrões de projeto e técnicas bem conhecidas [Gamma et al., 2000; Meyer, 1997; Pressman, 2006], sabe-se que à medida que um software evolui e muda, sua arquitetura torna-se mais complexa e rígida e, devido a essa degradação de seu desenho, o software torna-se crescentemente difícil de manter. As Leis de Lehman [Lehman et al., 1997] descrevem esta natureza evolutiva de software, postulando que softwares em geral crescem e sofrem manutenções continuamente, têm complexidade crescente e qualidade decrescente ao longo de sua evolução.

Nos últimos anos, muitas pesquisas têm sido realizadas para caracterizar a evolução de software. Grande parte desses trabalhos destinam-se a investigar se as Leis de Lehman são aplicáveis a software aberto, especialmente nos aspectos de crescimento e complexidade [Koch, 2007; Xie et al., 2009; Israeli & Feitelson, 2010]. Crescimento tem sido usualmente avaliado por meio de métricas como LOC (*lines of code*) ou número de arquivos [Godfrey & Tu, 2001; Herraiz et al., 2006], enquanto complexidade tem sido avaliada por meio das métricas de complexidade McCabe ou Halstead [Mens et al., 2008; Israeli & Feitelson, 2010]. Poucos trabalhos têm usado outras métricas de software para estudar evolução de software, por exemplo: número de arquivos excluídos/incluídos/alterados [Mens et al., 2008], e métricas de acoplamento e coesão [Lee et al., 2007]. Recentemente, os conceitos de Redes Complexas têm sido timidamente aplicados para entender o comportamento e a natureza das estruturas de software [Jing et al., 2006; Jenkins & Kirk, 2007; Louridas et al., 2008; Zimmermann & Nagappan, 2008].

Um achado comum nesses trabalhos é que os *graus de entrada* de vértices nas redes de módulos em um software seguem uma lei de potência, que é característica de redes livres de escala [Newman, 2003]. Entretanto, há ainda uma grande carência de conhecimento sólido sobre a evolução das estruturas de software. Sabe-se que estruturas de software degradam ao longo do tempo e que as redes formadas por classes em um software parecem ser governadas pelo efeito *small-world* [Newman, 2003], que é caracterizado pela distância curta entre pares de vértices em uma rede. Esta característica propicia alta disseminação de informação em uma rede. O fenômeno *small-world* pode, portanto, ocasionar dificuldade de manutenção em software, já que ele pode determinar que uma modificação em um software possa ser amplamente propagada pelo software. Nesse contexto, este capítulo apresenta os resultados de um estudo sobre o processo de evolução de software do ponto de vista de Redes Complexas. O presente trabalho tem por objetivo investigar como a degradação da estrutura de software ocorre e identificar as forças que fazem o fenômeno *small-world* surgir em software. Com esse propósito, foi realizado um estudo experimental para caracterizar essa evolução por meio de um conjunto de métricas de software e de redes, tais como *diâmetro*, *grau de entrada*, coesão interna de classe, *densidade* e tamanho do maior componente fortemente conectado.

Este estudo foi realizado com um conjunto de 16 softwares abertos e um software proprietário desenvolvidos em Java, em um total de 129 versões de programas. Os resultados do estudo revelam fatos interessantes sobre a natureza evolutiva de software, por exemplo: as classes com alto grau de entrada tendem a preservar esta propriedade e a qualidade delas tende a degradar com o crescimento do software; a *densidade* das redes que representam os softwares tende a diminuir, o *diâmetro* de tais redes é pequeno, e seu maior componente fortemente conectado aumenta em número de classes com o crescimento do software. Os resultados do trabalho revelam também a imagem da estrutura macroscópica de softwares. Os achados deste estudo podem ser aplicados para melhorar e ajudar em tarefas de desenvolvimento de software, tais como em manutenção e no planejamento e execução de testes. Além disso, os resultados do estudo descrito neste capítulo fornecem uma melhor compreensão da natureza das estruturas reais de software, constituindo um importante embasamento para as premissas assumidas na definição do modelo K3B. Por exemplo, os resultados deste estudo evidenciam a importância da conectividade entre módulos nas redes que representam relações entre classes em software orientado por objetos.

O restante deste capítulo está organizado da seguinte forma: a Seção 7.1 provê embasamento sobre os conceitos utilizados no estudo; a Seção 7.2 é uma revisão de trabalhos relacionados; a Seção 7.3 descreve o método e os dados utilizados no estudo; a Seção 7.4 descreve os experimentos e relata seus resultados; a Seção 7.5 traz as

conclusões e indicações de trabalhos futuros nessa linha.

7.1 Referencial Teórico

Um software orientado por objetos pode ser modelado como um grafo dirigido (uma rede) no qual as classes são os vértices e uma conexão entre duas classes é uma aresta. Nos experimentos realizados neste estudo, considera-se que uma classe A está conectada a uma classe B se A usa um atributo ou um método de B, ou se A é subclasse de B. Nesta situação, há uma aresta direcionada de A para B. Neste trabalho será utilizada a expressão *rede de software* para designar esse tipo de rede. Esta seção descreve as métricas de software e de rede utilizadas na condução deste estudo, bem como apresenta os conceitos e terminologia de análise de redes utilizados neste estudo.

7.1.1 Métricas de Software

As seguintes métricas são utilizadas para avaliar evolução de classes em um software:

- Número de atributos públicos: é o número de atributos públicos definidos na classe. Esta métrica é utilizada neste estudo para avaliar a evolução do tamanho da classe em termos de atributos públicos.

- Número de métodos públicos: é o número de métodos públicos definidos na classe. Este tipo de avaliação permite verificar a evolução da classe em relação à quantidade de serviços providos por ela.

- COR - Coesão de Responsabilidade: há várias métricas de coesão de classes propostas na literatura, mas não há um consenso sobre a melhor forma de se medir coesão. No presente estudo, é utilizada a métrica COR, apresentada no Capítulo 6. Esta métrica é dada por $1/r$, onde r é o número de conjuntos disjuntos de métodos em uma classe. Cada um desses conjuntos consiste de métodos similares. Dois métodos são considerados similares se usam um atributo ou um método da classe em comum. Por exemplo, se há dois conjuntos na classe, COR resulta em 0,5. Isso é um indicador de que a classe possui duas responsabilidades. Quando há apenas um conjunto na classe, COR resulta em 1, o que é um indicador de alta coesão.

7.1.2 Métricas de Rede

As seguintes métricas de rede são usadas para avaliar a evolução das estruturas de software:

- *Densidade*: em uma rede sem *self-loops* com n vértices e c arestas, esta métrica é dada por $c/(n(n-1))$ [Leskovec et al., 2007]. No contexto de métricas de software, esta métrica é chamada COF (*coupling factor*) [Abreu & Carapuça, 1994]. Neste capítulo, optou-se por utilizar o termo *densidade* em vez de *COF* para manter consistência com a terminologia empregada na análise de redes, já que a abordadem principal aplicada aqui é a de redes complexas.

- *Diâmetro*: o diâmetro de uma rede é o tamanho, dado em número de arestas, do caminho geodésico mais longo na rede. Um caminho geodésico é o caminho mais curto entre um par de vértices. Em uma rede social, por exemplo, esta métrica é um indicador que quão rápido uma informação poderia ser disseminada na rede, pois se a rede possui diâmetro curto significa que as distâncias entre os pares de vértices é curta e, com isso, são necessários poucos passos para uma informação passar de um vértice para outro [Newman, 2003].

- *Grau de entrada*: em um grafo direcionado, o grau de entrada de um vértice v é dado pelo número de outros vértices a partir dos quais há uma aresta direcionada para v [Newman, 2003]. Em uma rede de software, esta métrica representa o número de classes a partir das quais há uma conexão direcionada para a classe avaliada. No contexto de métricas de software, esta métrica corresponde à métrica *número de conexões aferentes*, que corresponde ao número de classes que dependem de determinada classe. Esta métrica é discutida na Seção 5.2.1. Neste capítulo, optou-se por utilizar o termo *grau de entrada* em vez de *conexões aferentes* para manter consistência com a terminologia empregada na análise de redes.

7.1.3 Redes Complexas

A observação empírica de redes reais forneceram valiosa compreensão sobre tais redes. O trabalho de Newman [2003] apresenta uma ampla revisão sobre os avanços nesta área do conhecimento denominada Redes Complexas. Os estudos das propriedades de tais redes incluem conceitos tais como o fenônomeno *small-world*, distribuição de graus de vértices, redes livres de escala (*scale-free networks*) e modelos de crescimento de redes.

As redes em que os graus dos vértices são distribuídos conforme uma *lei de potência* são referenciadas como redes livres de escala. Uma lei de potência é uma função de distribuição de probabilidade na qual a probabilidade de uma variável randômica X assumir um valor x é proporcional a uma potência negativa de x, denotada por $P(X = x) \propto cx^{-k}$. Em uma rede livre de escala há um número muito grande de vértices com grau pequeno, enquanto há uma pequena porção de vértices com grau alto. Tem havido grande interesse em tais redes, visto que tem-se observado que a distribuição de graus em lei de potência está presente em uma grande variedade de redes importantes tais como a Web, a Internet, redes biológicas e químicas, redes sociais e redes de software [Baxter et al., 2006; Louridas et al., 2008; Potanin et al., 2005; Puppin & Silvestri, 2006; Wheelson & Counsell, 2003]. Uma propriedade importante de uma rede livre de escala é sua resiliência à remoção de seus vértices. Um estudo sobre a Internet e a Web mostrou que tais redes são muito resilientes contra falha aleatória dos vértices da redes, enquanto a remoção de vértices com os mais altos graus de entrada tem efeito destrutivo na rede [Newman, 2003]. Como redes de software também são livres de escala, esta propriedade talvez seja também aplicável a elas. No contexto de propagação de modificações, por exemplo, isso significa que uma modificação ou uma falha em um módulo com alto grau de entrada pode afetar amplamente o software como um todo.

O fenômeno *small-world* refere-se à característica de redes nas quais os pares de vértices são conectados por um caminho curto. Esta propriedade é relacionada à facilidade de propagação de informação na rede. Dependendo do tipo de rede em questão, a informação pode assumir diversos significados, tais como disseminação de doença em uma população, disseminação de um boato em uma rede social ou uma modificação em uma rede de software.

Modelos de rede auxiliam a compreender a topologia de redes e os processos que ocorrem em tais redes. Neste trabalho, as estruturas de softwares são exploradas à luz de conceitos e características de redes complexas.

7.1.4 O Modelo *Bow-tie*

Broder et al. [2000] identificaram que a estrutura macroscópica da estrutura da Web pode ser modelada por uma figura que eles denominaram *bow-tie*. Esta figura, mostrada na Figura 7.1, sugere que páginas Web podem ser divididas em cinco grupos: *LSCC* (*largest strongly connected component*), *in*, *out*, *tendrils*, *tubes* e *disconnected.* No grafo da Web há um núcleo central no qual todas as páginas estão conectadas umas com as outras diretamente ou não. Esse núcleo é denominado *o componente gigante*

fortemente conectado (LSCC). Um outro grupo de páginas pode alcançar diretamente aquelas de LSCC, mas não podem ser alcançadas diretamente pelas páginas de LSCC. Este grupo é denominado *in*. *Out* consiste de páginas que podem ser alcançadas diretamente por aquelas em *LSCC*, mas não podem alcançar diretamente páginas dos outros componentes. *Tendril* consiste de páginas que não alcançam diretamente as páginas de *LSCC* e também não são alcançadas diretamente pelas páginas de *LSCC*; as páginas em *tendrils* são alcançáveis diretamente a partir de *in* e alcançam diretamente *out*, sem passar por *LSCC*. Há um grupo de páginas em *tendrils* que são alcançadas diretamente a partir de *in*, conectam-se a outra página em *tendril*, que conecta-se a uma página em *out*. Este grupo de páginas é denominado *tubes*. Este modelo tem importantes implicações no estudo da Web, por exemplo na análise de algoritmos para a Web e na definição de modelos sobre a evolução das estruturas da Web [Broder et al., 2000].

No presente estudo, foi investigado se o modelo *bow-tie* ajusta-se a redes de software, e os resultados levaram à identificação de uma figura mais simples do que o *bow-tie* que representa a forma como classes conectam-se umas com as outras em software orientado por objetos.

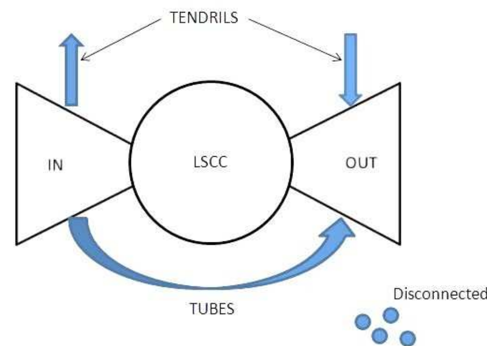


Figura 7.1. O modelo *bow-tie* da Web

7.2 Trabalhos Relacionados

Evolução de software tem sido extensivamente estudada. Um dos trabalhos neste campo resultou nas conhecidas Leis de Lehman, que definem que a evolução de software tem as seguintes propriedades: mudança contínua, complexidade crescente e qualidade decadente [Lehman et al., 1997]. Muitos pesquisadores têm estudado se as Leis de Lehman são aplicáveis a softwares abertos. Um estudo de caso realizado por Godfrey & Tu [2001] com o núcleo do Linux concluiu que ele cresce continuamente a uma taxa muito alta. Lee et al. [2007] analisaram a evolução de JFreeChart, uma

biblioteca de código aberto, com base no tamanho e métricas de acoplamento e coesão. Os resultados do trabalho deles apontam que a evolução desse software segue algumas das Leis de Lehman. Mens et al. [2008] estudaram a evolução do Eclipse usando métricas de software como número de erros, e número de arquivos excluídos, incluídos e alterados. Eles identificaram evidências de crescimento contínuo e complexidade crescente no Eclipse. Israeli & Feitelson [2010] usaram métricas de software para analisar a evolução do núcleo do Linux também. Os resultados do estudo deles mostram que a maior parte das leis de Lehman aplicam-se ao software avaliado. Porém, eles concluíram que a complexidade média das funções tem diminuído. Xie et al. [2009] avaliaram a evolução de sete softwares abertos, tendo concluído que as seguinte leis de Lehman aplicam-se a softwares abertos: mudança contínua, complexidade crescente, auto-regulação e crescrimento contínuo. Além disso, eles observaram que a maior parte das modificações ocorrem em uma parcela pequena de código fonte.

A evolução de software tem sido geralmente estudada sob o aspecto de crescimento. Koch [2007] analisou o crescimento de uma grande amostra de softwares abertos. Os resultados do estudo indicaram que a taxa média de crescimento dos softwares é linear ou tende a diminuir ao longo do tempo. Porém, um percentual alto de softwares exibem crescimento acima de uma taxa linear. Herraiz et al. [2006] realizaram um estudo comparativo entre duas métricas de software geralmente utilizadas para caracterizar evolução de software: número de linhas de código e número de arquivos. Eles analisaram um pacote de Debian GNU/Linux e concluíram que ambas as métricas têm o mesmo comportamento ao longo do tempo. Os resultados do estudo deles confirmam os achado de Godfrey & Tu [2001].

Outras abordagens têm sido utilizadas no estudo da evolução de software. Muitos pesquisadores têm identificado que a distribuição do grau de entrada nas redes de software seguem leis de potência, independente do componente considerado como vértice na rede [Jing et al., 2006; Baxter et al., 2006; Louridas et al., 2008; Potanin et al., 2005; Puppin & Silvestri, 2006; Wheelson & Counsell, 2003]. Olbrich et al. [2009] estudaram a evolução de *code smells* em um software e seus impactos em relação às modificações dos elementos contaminados pelo *code smells*. Eles analisaram dois softwares abertos e consideraram dois *code smells*: *God Class*, que refere-se a classe que possuem muitas responsabilidades, e *Shotgun Surgery*, que ocorre quando uma modificação em uma classe causa modificações em muitas outras classes. Eles identificaram que classes infectadas com esses *code smells* têm uma alta frequência de modificações. Porém, eles não identificaram que tipo de modificações essas classes sofrem. Jenkins & Kirk [2007] avaliaram a evolução de software usando a teoria de Redes Complexas. O estudo deles foi realizado com algumas versões de um dos componentes de Sun Java2 Runtime Environment

(rt.jar). Uma das conclusões do estudo deles é que a distribuição dos graus dos vértices da rede de software segue uma lei de potência. Zimmermann & Nagappan [2008] investigaram a existência de correlação entre algumas métricas de rede e o número de defeitos no Windows Server 2003. Eles concluíram que as métricas utilizadas no estudo podem ser usadas na predição de defeitos do software avaliado.

O presente estudo analisa a evolução de software sob o ponto de vista de redes complexas. A questão principal investigada nesse estudo é como as estruturas de rede de software evoluem ao longo do tempo. Essa análise é realizada em termos de métricas de rede, tais como densidade e diâmetro. Tendo em vista que as classes com alto grau de entrada desempenham papel central no software, é analisado como tais classes evoluem por meio de métricas de rede e de software. Além disso, busca-se identificar se há uma estrutura macroscópica genérica dessas redes de software. Os resultados obtidos trazem novas informações a respeito da formação das estruturas de software. Além desses resultados constituírem embasamento empírico para as premissas assumidas na definição do modelo K3B, eles geram novos conhecimentos sobre a estrutura real de softwares.

7.3 Metodologia

A seleção dos softwares abertos analisados neste estudo foi baseada nos seguintes critérios: idade do software, quantidade de versões e diversidade de domínios de aplicação. Os softwares foram obtidos em Sourceforge[1], que classifica os softwares por domínios de aplicação, tais como *development*, *games* e *communication*. Para cada um dos domínios listados em Sourceforge, foram selecionados até 10 softwares, satisfazendo os seguintes critérios: eles são desenvolvidos em Java, têm pelo menos cinco versões e quatro anos de vida. Outro critério usado foi a disponibilidade de *bytecodes* dos programas porque a ferramenta utilizada na coleta de métricas de software, *Connecta* [Ferreira, 2006], avalia o código compilado e não o código fonte. A amostra inicial resultou em 108 programas. Dentre eles, foram selecionados, por domínio de aplicação, aqueles com os maiores números de versão, com maior idade e mais populares. A popularidade foi avaliada em termos de número de *downloads* por semana. Esta última seleção resultou em 16 softwares cujos dados são mostrados na Tabela 7.1. Os dados foram obtidos de Sourceforge no período de Setembro de 2009 a Abril de 2010.

Com o objetivo de verificar se haveria diferença relevante entre duas versões consecutivas de um programa, inicialmente foram analisados dados de todas as versões

[1] www.sourceforge.net

Tabela 7.1. Softwares analisados no estudo sobre evolução de software

Software	Categoria	downloads/ semana	idade	#classes	#versões	#versões analisadas
JEdit	Text Editor	9.138	2001 a 2009	377 a 1124	13	13
Dr Java	Development	3.837	2002 a 2009	596 a 3692	10	10
Java Groups	Cooperation	465	2003 a 2009	696 a 1137	40	13
KoL Mafia	Game	1.007	2004 a 2009	39 a 1109	13	13
DBUnit	Database	448	2002 a 2009	198 a 369	25	5
FreeCol	Game	7.452	2003 a 2010	112 a 5902	27	5
JasperReports	Development	5.542	2001 a 2010	525 a 5304	50	5
JGNash	Financial	822	2002 a 2010	782 a 3603	40	5
Java msn library	Communication	271	2004 a 2010	494 a 872	10	5
Jsch	Security	2.304	2004 a 2009	202 a 271	29	5
JUnit	Development	1.834	2000 a 2009	78 a 230	18	5
Logisim	Education	1.590	2005 a 2009	908 a 1185	28	5
MeD's Movie Manager	Storage	1.169	2003 a 2010	64 a 517	60	5
Phex	Network	1.084	2001 a 2009	393 a 1352	26	5
Squirrel sql	Database	7.270	2006 a 2010	424 a 1223	26	5
Hibernate	Database	12.906	2004 a 2010	956 a 2446	53	5
Commercial - frontier layer	Commercial application	-	2005 - 2010	1100 a 1246	10	10
Commercial - model layer	Commercial application	-	2005 - 2010	3343 a 4031	10	10

de três dos softwares: JEdit, DrJava e Kolmafia. Para JavaGroups, que possui uma grande quantidade de versões, foram selecionadas 13 delas: a primeira, a última e 11 versões intermediárias, observando-se um período de liberação aproximadamente igual entre duas versões subsequentes. Os resultados desta análise mostraram que os dados obtidos em duas versões consecutivas são muito próximos. Devido a isso, para os demais softwares, foram selecionadas cinco versões: a primeira, a última e três versões intermediárias, observando-se um período de liberação aproximadamente igual entre duas versões subsequentes.

O software proprietário analisado neste estudo é desenvolvido por um laboratório de Engenharia de Software de uma importante universidade brasileira. O laboratório provê soluções em software e consultoria para diferentes segmentos de mercado. A maior parte de seus clientes são órgãos públicos do governo federal. O software analisado é um dos mais antigos e maiores desenvolvidos pelo laboratório. O programa foi

construído usando-se arquitetura de três camadas e tem mais de 6.000 classes, que são divididas em seis pacotes. Neste estudo foram analisados dados de dois desses pacotes separadamente, que correspondem às camadas de fronteira e de modelo do sistema. O motivo pelo qual os dados foram analisados por pacote e não para o sistema como um todo é que, por conveniência do laboratório, a coleta dos dados foi realizada desta forma pela própria equipe do laboratório. Os dados do software proprietário são mostrados também na Tabela 7.1.

As métricas de software foram coletadas com a ferramenta Connecta [Ferreira, 2006], que foi modificada para gerar um arquivo no formato apropriado para a ferramenta Pajek [PAJEK, 2010], a ferramenta para análise de redes que foi utilizada neste estudo.

Tabela 7.2. Dados da evolução dos softwares abertos

Software	Versão	Classes	Conexões	Dens.	Diâm.
DBUnit	2.0	198	429	0,011	9
	2.2.1	289	666	0,008	11
	2.4.0	332	769	0,007	13
	2.4.4	347	780	0,006	17
	2.4.7	369	815	0,006	16
FreeCol	0.1.0	44	112	0,059	5
	0.5.0	416	1899	0,011	12
	0.6.0	611	2609	0,007	11
	0.8.0	927	5150	0,006	13
	0.9.2	1087	5902	0,005	14
Jasper	0.4.0	242	525	0,009	8
Reports	1.0.0	574	1316	0,004	9
	2.0.0	1104	2435	0,002	13
	3.0.0	1233	3038	0,002	13
	3.7.1	1629	5304	0,002	13
JGNash	1.10.0	743	2757	0,005	16
	1.11.1	782	2443	0,004	17
	1.50.0	942	2659	0,003	12
	2.00.0	2716	7374	0,001	24
	2.20.0	3603	12978	0,001	24
Java	10a1	171	494	0,017	10
msn	10a2	186	516	0,015	7
library	10b1	203	615	0,015	7
	10b2	218	662	0,014	9
	10b3	270	872	0,012	9

Software	Versão	Classes	Conexões	Dens.	Diâm.
LogSim	2.0.0	908	3294	0,004	13
	2.1.0	993	3940	0,004	14
	2.1.5	1018	4141	0,004	14
	2.2.0	1054	4439	0,004	14
	2.3.3	1185	4609	0,003	14
MeD's	1.6	64	149	0,037	6
Movie	1.7	73	168	0,032	6
Manager	2.0	517	1067	0,004	10
	2.8	458	1465	0,007	12
	2.9.13	608	1845	0,005	13
Phex	0.6	393	1078	0,007	8
	2.0.0	897	3215	0,004	18
	2.8.0	1205	4352	0,003	16
	3.0.0	1419	6036	0,003	19
	3.4.2	1352	5480	0,003	20
Squirrel	1.0	424	717	0,004	15
-sql	2.0	729	1592	0,003	13
	2.6	940	1765	0,002	14
	3.0	1134	2570	0,002	16
	3.1	1223	2989	0,002	16
JSch	0.1.1.4	80	202	0,032	4
	0.1.20	83	204	0,028	5
	0.1.26	94	210	0,024	5
	0.1.34	109	271	0,023	5
	10.1.42	117	385	0,02	5

Tabela 7.3. Dados da evolução dos softwares abertos - continuação

Software	Versão	Classes	Conexões	Dens.	Diâm.
JUnit	3.4	78	138	0,023	5
	3.8	101	182	0,018	6
	4.0	92	197	0,02	6
	4.5	188	352	0,01	8
	4.8.1	230	421	0,008	10
Java Groups	2.2	696	1935	0,004	10
	2.2.1	849	2880	0,004	10
	2.2.5	829	2059	0,003	10
	2.2.6	832	2074	0,003	10
	2.2.7	857	2201	0,003	10
	2.2.8	810	2621	0,004	8
	2.2.9	922	2621	0,003	8
	2.3	959	2756	0,003	9
	2.4.1	1013	3075	0,003	7
	2.5.1	967	3736	0,004	8
	2.6.1	1012	3639	0,003	8
	2.7.0	1041	3688	0,003	11
	2.8.0	1137	3875	0,003	9
KolMafia	0.2	39	83	0,056	7
	0.4	75	222	0,04	9
	1.0	143	508	0,025	10
	2.0	191	726	0,02	11
	4.0	342	1399	0,012	12
	5.0	334	1780	0,016	11
	6.0	388	2102	0,014	10
	7.0	498	2970	0,012	12
	9.0	616	3410	0,009	11
	10.0	708	4004	0,008	13
	11.0	757	4578	0,008	14
	12.0	772	5357	0,009	12
	13.7	1109	7373	0,006	13
Hibernate	3.0	956	2739	0,003	19
	3.1	1118	3746	0,003	20
	3.2	1302	4102	0,002	23
	3.3.0	1690	5707	0,002	21
	3.5.1	2446	5980	0,001	21
JEdit	2.4	377	1192	0,009	8
	2.5	422	1474	0,008	8
	3.1	426	1595	0,009	8
	3.2	449	1672	0,008	8
	4.0	554	2059	0,007	9
	4.1	618	2393	0,006	12
	4.1-8	646	2550	0,006	12
	4.2	805	3255	0,005	10
	4.3	810	3276	0,005	10
	4.3.4	867	3444	0,005	10
	4.3.9	954	3671	0,004	10
	4.3.13	1008	3885	0,004	13
	4.3.18	1124	4261	0,003	12
DrJava	1011	596	1773	0,005	10
	2148	1064	3393	0,003	14
	1826	1108	3680	0,003	12
	2304	1512	4569	0,002	18
	2332	1622	5259	0,002	19
	1750	2036	8287	0,002	21
	1406	2187	9562	0,002	23
	1942	3003	9732	0,001	17
	r4592	3421	117000	0,001	14
	r4756	3692	13627	0,001	16

7.4 Experimentos e Resultados

Esta seção apresenta e analisa os resultados dos experimentos realizados, de acordo com a seguinte sequência: crescimento dos softwares, evolução da densidade das redes de software, evolução dos graus de entrada das classes, evolução das classe com alto grau de entrada, evolução do diâmetro das redes, a visão macroscópica das redes de software. Os dados sobre a evolução dos softwares abertos são mostrados nas

Tabela 7.4. Dados sobre a evolução do software proprietário

Camada	Versão	Classes	Conexões	Dens.	Diâm.
Frontier	V1	1100	2418	0,002	10
	V10	1162	2698	0,002	10
	V18	1246	1551	0,001	10

Camada	Versão	Classes	Conexões	Dens.	Diâm.
Model	V1	3343	28420	0,003	14
	V10	3796	28812	0,002	14
	V18	4031	32490	0,002	14

Tabelas 7.2 e 7.3, e os dados do software proprietário, na Tabela 7.4.

7.4.1 Crescimento dos Softwares

A métrica usada para avaliar os tamanhos dos softwares foi *número de classes*. Os dados dos softwares avaliados mostram que o número de classes em softwares abertos cresce continuamente. Em 50% dos softwares, a última versão avaliada tem mais do que o dobro de classes da primeira versão. Esse achado está de acordo com os de outros trabalhos que afirmam que o crescimento contínuo é uma característica de software aberto [Godfrey & Tu, 2001; Koch, 2007; Mens et al., 2008; Israeli & Feitelson, 2010]. Esta característica foi também observada no software proprietário, porém em uma escala menor. Uma explicação possível para este fato é que softwares abertos estão em ambientes que são muito mais dinâmicos do que aqueles de muitos softwares proprietários.

7.4.2 Densidade da Rede de Software

Os dados da densidade das redes dos softwares avaliados neste estudo mostram que à medida que o software cresce em número de classes, a sua densidade diminui. Esse resultado mostra que uma nova classe inserida no software, em geral, conecta-se a um número pequeno de outras classes. Embora esse achado possa não ser considerado surpreendente, ele confirma empiricamente que de fato é isso que ocorre na prática. Ele torna evidente também que a métrica de densidade não deve ser tomada isoladamente para avaliar a complexidade e dificuldade de manutenção de software, tendo em vista que se sabe que software tende a se tornar mais complexo e difícil de manter à medida que cresce.

7.4.3 Grau de Entrada

A análise dos dados deste experimento revelam que as classes com os maiores graus de entrada mantêm essa propriedade à medida que o software cresce. Em todos

Tabela 7.5. Evolução do grau de entrada - JEdit 2.4 and 4.3.18

Classe	Grau de entrada
org.gjt.sp.jedit.EditAction	127
org.gjt.sp.jedit.View	124
org.gjt.sp.jedit.textarea.JEditTextArea	120
org.gjt.sp.jedit.jEdit	101
org.gjt.sp.jedit.Buffer	66
org.gjt.sp.jedit.textarea.InputHandler	52
org.gjt.sp.util.Log	46
org.gjt.sp.jedit.GUIUtilities	33
org.gjt.sp.jedit.MiscUtilities	28
org.gjt.sp.jedit.syntax.TokenMarker	24

Classe	Grau de entrada
org.gjt.sp.jedit.jEdit	282
org.gjt.sp.util.Log	159
org.gjt.sp.jedit.GUIUtilities	135
org.gjt.sp.jedit.View	102
org.gjt.sp.jedit.Buffer	73
org.gjt.sp.jedit.MiscUtilities	62
org.gjt.sp.jedit.io.VFSManagerX	51
org.gjt.sp.jedit.textarea.JEditTextArea	49
org.gjt.sp.jedit.buffer.JEditBuffer	45
org.gjt.sp.jedit.bsh.SimpleNode	44

Tabela 7.6. Evolução do grau de entrada - Kolmafia 2.0 and 13.7

Classe	Grau de entrada
net.sourceforge.kolmafia.KoLmafia	69
net.java.dev.spellcast.utilities.Lockable ListModel	34
net.sourceforge.kolmafia.KoLRequest	30
net.sourceforge.kolmafia.AdventureResult	26
net.java.dev.spellcast.utilities.ActionVerify Panel	26
net.sourceforge.kolmafia.KoLFrame	26
net.sourceforge.kolmafia.KoLFrame$KoL Panel	25
net.sourceforge.kolmafia.AdventureFrame	24
net.sourceforge.kolmafia.KoLCharacter	20
net.sourceforge.kolmafia.KoLSettings	16

Classe	Grau de entrada
net.sourceforge.kolmafia.KoLmafia	264
net.sourceforge.kolmafia.KoLConstants	255
net.sourceforge.kolmafia.persistence. Preferences	226
net.sourceforge.kolmafia.utilities. StringUtilities	203
net.sourceforge.kolmafia.RequestThread	193
net.sourceforge.kolmafia.AdventureResult	188
net.sourceforge.kolmafia.KoLCharacter	187
net.sourceforge.kolmafia.RequestLogger	165
net.java.dev.spellcast.utilities.Lockable ListModel	151
net.sourceforge.kolmafia.textui.command. AbstractCommand	149

os softwares do estudo, foram analisados os grupos das 10 classes com maiores graus de entrada ao longo da evolução do software. Observa-se que em todos os softwares, este grupo é constituído praticamente pelas mesmas classes ao longo da evolução do software. Além disso, os graus de entrada dessas classes tendem a crescer. A Tabelas 7.5, 7.6, 7.7 e 7.8 mostram os dados de evolução das classes com os maiores graus de

Tabela 7.7. Evolução do grau de entrada - JUnit 4.0 and 4.8.1

Classe	Grau de entrada
org.junit.runner.Description	12
org.junit.runner.notification.RunListener	9
org.junit.runner.Runner	8
org.junit.runner.notification.RunNotifier	8
junit.framework.TestResult	8
org.junit.runner.notification.Failure	8
org.junit.runner.Request	7
org.junit.runner.manipulation.Filter	6
org.junit.runner.notification. RunNotifier$SafeNotifier	6
junit.framework.TestSuite	5

Classe	Grau de entrada
org.junit.runner.Description	18
org.junit.runners.model.Statement	16
org.hamcrest.BaseMatcher	15
org.junit.runners.model.FrameworkMethod	15
org.junit.runner.notification.Failure	11
org.junit.runner.notification.RunListener	10
org.junit.runner.notification.RunNotifier	10
org.junit.runners.model.RunnerBuilder	9
org.junit.runners.model.TestClass	9
org.junit.runner.manipulation.Filter	9

Tabela 7.8. Evolução do grau de entrada - Software proprietário 1.0 and 1.18

Classe	Grau de entrada
A	808
B	558
C	551
D	314
E	291
F	287
G	283
H	271
I	265
J	248
-	-
-	-

Classe	Grau de entrada
A	912
C	631
B	595
X	385
D	347
E	341
F	317
H	295
G	291
Y	285
I	275
J	258

entradas em quatro softwares, incluindo duas ferramentas, uma biblioteca e o software proprietário.

A associação deste resultado com o fato de a densidade de software diminuir com o seu crescimento mostra que uma nova classe inserida no sistema tende a se conectar a um número pequeno de outras classes e que as classes existentes no sistema que possuem alto grau de entrada continuam tendo os mais altos graus de entrada. Isso indica que o crescimento de software parece ser governado pelo seguinte modelo: uma nova classe inserida no sistema dependerá preferencialmente de uma classe da qual muitas outras classes no sistema já dependem. Esse modelo de crescimento de redes é

Tabela 7.9. Instabilidade da classe net.sourceforge.kolmafia.KoLmafia

Versão	Grau de entrada	COR	atributos públicos	métodos públicos
0.2	7	0,5	0	18
2.0	69	0,33	0	30
5.0	142	0,143	0	74
11.0	145	0,067	8	85
13.7	264	0,05	17	78

Tabela 7.10. Instabilidade das classes com alto grau de entrada do software proprietário

Classe	Versão	Grau de entrada	COR	atributos públicos	método públicos
A	1.0	808	1	0	23
	1.18	912	1	0	25
B	1.0	558	0,071	0	70
	1.18	595	0,067	0	85
C	1.0	551	0,045	0	93
	1.18	631	0,037	1	114
D	1.0	314	0,036	0	105
	1.18	347	0,031	0	116
E	1.0	291	0,05	0	104
	1.18	341	0,048	0	105

usualmente denominado *Preferential Attachment* [Newman, 2003]. Esse modelo explica o surgimento de distribuições do tipo leis de potência. De acordo como esse modelo, distribuições de leis de potência ocorrem em situações nas quais "os ricos tornam-se ainda mais ricos" (*"the rich get richer"*), o que leva uma pequena parcela de ocorrências ter valores altos e a maior parte das ocorrência ter valores baixos. O fato de uma classe nova no sistema vir a depender preferencialmente de uma classe da qual muitas classes já dependem, então, explica o surgimento de leis de potência na distribuição de graus de entrada de classes.

7.4.4 Evolução de Classes com Alto Grau de Entrada

Intuitivamente, pode-se pensar que classes com alto grau de entrada deveriam ser estáveis, já que, como tais classes desempenham papel importante no sistema, deveriam ser bem definidas, contruídas e testadas. O termo *estabilidade* aqui refere-se à baixa

frequência de modificações em uma classe ao longo da vida do software. É razoável esperar que se o software é bem projetado e se lhe foi aplicado apropriadamente o princípio do aberto-fechado (*open-closed principle*)[Meyer, 1997], tais classes sofrerão poucas modificações ou, idealmente, nenhuma modificação. Contudo, os resultados da análise dos dados neste experimento mostram que o oposto ocorre na prática. Classes com alto grau de entrada são extremamente instáveis. As modificações em classes entre duas versões consecutivas foram avaliadas com três métricas: número de atributos públicos, número de métodos públicos e COR, que avalia coesão interna. A Tabela 7.9 mostra dados de uma classe de um software aberto e a Tabela 7.10 mostra dados de uma classe do software proprietário. Neste software, o efeito é mais moderado, mas não inexistente. A cada nova versão, as classes com alto grau de entrada crescem em número de métodos públicos e, em alguns casos, em número de atributos públicos. Além disso, a coesão dessas classes diminui a cada nova versão.

Este resultado mostra que o processo de crescimento de software parece ocorrer da seguinte forma na prática: como as classes com alto grau de entrada já são grande provedoras de serviços, a prática usual é manter essas classe nesta situação, incluindo ainda mais serviços nelas para que elas possam atender às novas classes inseridas no software. Isso propicia a degradação da coesão da classe, o que influencia também a degradação do sistema.

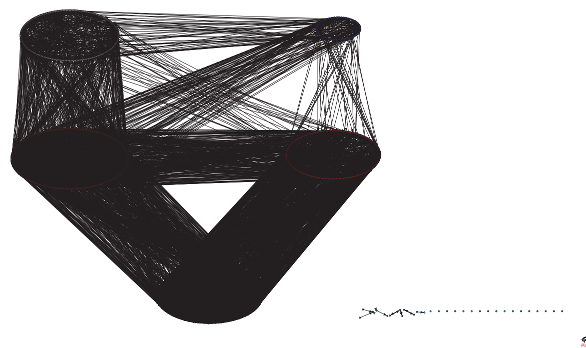
Por esses resultados, é possível inferir que a prática comum é agregar novos serviços às classes já existentes para atender às novas funcionalidades ou modificações no software em vez de refatorá-las. Isso leva ao inchaço das classes que já possuem muitos clientes e, então, elas se tornam menos coesas e têm seus respectivos graus de entrada sempre crescendo. A prática de não refatorar software, então, parece ser a causa base do surgimento do efeito *small-world* e de redes livres de escala em software.

7.4.5 Diâmetro

O efeito *small-world* está relacionado ao fato de a maior parte dos pares de vértices em uma rede serem conectados por um caminho curto. Este efeito tem como consequência a determinação de alguns comportamentos na rede. Por exemplo, em uma rede social, o efeito *small-world* implica que a propagação de informação é rápida. Se o objeto de estudo na rede é a propagação de doenças, o efeito *small-world* determina o tempo que uma doença é disseminada na população [Newman, 2003]. O diâmetro, que é o tamanho do maior caminho mínimo entre os pares de vértices de uma rede, é uma métrica que indica esse efeito. Por exemplo, em uma rede social com diâmetro igual a 4, toda pessoa está no máximo quatro passos distante de qualquer outra pessoa

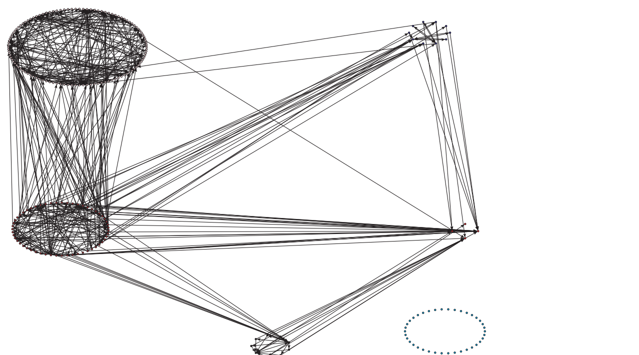
na rede. Se forem consideradas duas redes sociais, uma com diâmetro igual a 4 e outra com diâmetro igual a 100, uma informação pode ser mais rapidamente propagada na primeira rede do que na segunda.

Os dados dos softwares analisados neste estudo mostram que o diâmetro de uma rede de software é curto e cresce pouco à medida que o software cresce. Esse resultado indica que um evento em uma classe, por exemplo uma modificação, pode ser facilmente propagada pelo sistema. Entretanto, é possível que esse efeito de propagação seja controlado ou potencializado por outros fatores da estrutura do software, por exemplo pela forma como as conexões entre as classes são realizadas.



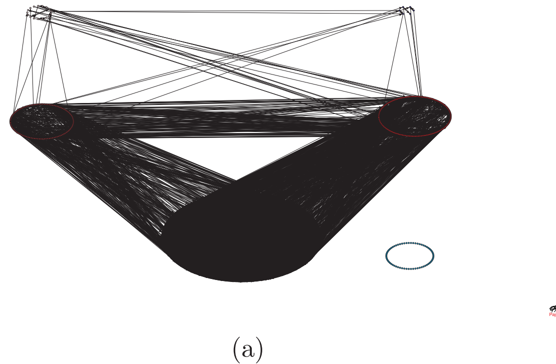
(a)

Figura 7.2. Hibernate (versão 3.5.1) modelado pelo *little house*



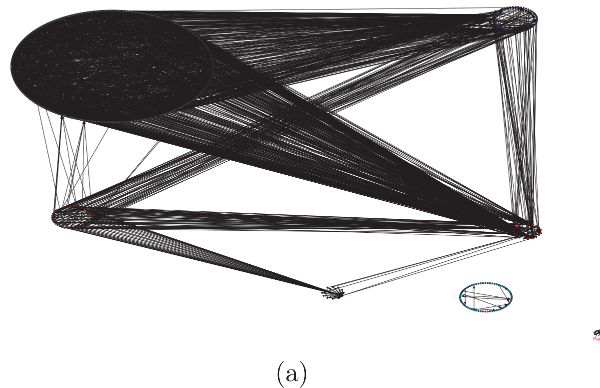
(a)

Figura 7.3. JUnit (versão 4.8.1) modelado pelo *little house*



(a)

Figura 7.4. Kolmafia (versão 13.7) modelado pelo *little house*

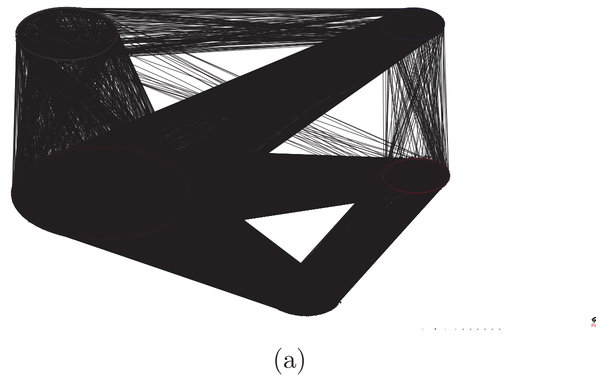


(a)

Figura 7.5. A camada de fronteira do software proprietário (versão 1.18) modelado pelo *little house*

7.4.6 *Little house* – A Estrutura Macroscópica Genérica das Redes de Software

Com o propósito de identificar uma estrutura macroscópica genérica das redes de software, inicialmente o modelo *bow-tie* foi utilizado para modelar algumas versões dos softwares analisados neste experimento. Esta análise foi realizada utilizando-se Pajek, que, dentre outras funcionalidades, gera uma figura da rede e permite sua manipulação. A figura gerada pela ferramenta corresponde ao modelo *bow tie*. Manipulando-se as figuras resultantes, percebeu-se que as conexões entre os componentes das redes formam uma imagem interessante, compatível com um grafo muito conhecido. As Figuras 7.2, 7.3 e 7.4 mostram as imagens resultantes dessa manipulação para três softwares: um *framework*, uma biblioteca e uma ferramenta. As Figuras 7.5 e 7.6 mostram a imagem resultante para a camada de fronteira e para a camada de modelo do software



(a)

Figura 7.6. A camada de modelo do software proprietário (versão 1.18) modelado pelo *little house*

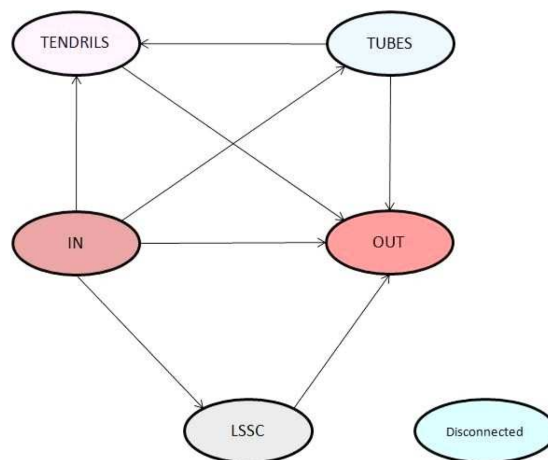


Figura 7.7. *Little house* – A estrutura macroscópica genérica das redes de software

proprietário.

A estrutura macroscópica genérica das redes de software identificada é mostrada na Figura 7.7. Devido à sua semelhança com o grafo informalmente chamado de *casinha*, nós a denominamos *little house*. Foi preservada a mesma terminologia empregada no modelo *bow-tie* para os componentes. O modelo *little house* é um grafo em que cada vértice correponde a um grupo de classes. Dentre de cada um desses grupos, as classes conectam-se livremente umas com as outras. O modelo *little house* é constituído pelos seguintes grupos de classes:

- **LSCC**: é o maior componente fortemente conectado no software. Neste componente, a partir de uma classe é possível alcançar qualquer outra classe dentro de *LSCC*. Com isso, toda classe dentro desse componente é direta ou indiretamente

dependente das demais classes de *LSCC*.

- **In**: classes pertencentes a esse componente podem usar qualquer outra classe do software, mas não são usadas por classes que não pertençam a *in*.
- **Out**: classes pertencentes a esse componente podem ser usadas por qualquer outra classe do software, mas usam somente classes pertencentes a *out*.
- **Tendril**: classes desse componente usam classes do próprio componente ou de *out*. Além disso, uma classe de *tendril* pode ser usada somente por classes do próprio componente ou que pertençam a *tubes* ou a *in*.
- **Tubes**: classes desse componente usam classes do próprio componente, de *out* ou de *tendril*. Além disso, uma classe de *tubes* pode ser usada somente por classes do próprio componente ou que pertençam a *in*.
- **Disconnected**: corresponde a classe que não possui qualquer conexão com outra classe do software.

Este resultado foi observado em todos os softwares analisados neste estudo, o que evidencia sua consistência. Novas questões surgem com esse resultado: a identificação dos tipos de classes que constituem um determinado componente do modelo *little house*; o tamanho de componente gigante fortemente conectado e como ele evolui; as implicações deste modelo na gerência e manutenção de software.

LSCC tem papel central no software, pois suas classes são fortemente conectadas entre si, o que pode tornar difícil entender, testar e manter as classes desse componente. A Tabela 7.11 mostra os dados dos tamanhos de *LSCC* para os softwares abertos em suas primeiras e últimas versões. Esses dados mostram que o tamanho deste componente tende a aumentar. Em sete dos softwares avaliados, o percentual de classes em *LSCC* também cresceu substancialmente.

Pode-se pensar que as conexões entre os componentes identificados possam ter relação com o modelo de arquitetura de camadas. Entretanto, não foram encontradas evidências para suportar esta hipótese na análise dos dados deste estudo. Um contra-exemplo dessa hipótese aparece na análise dos dados do software proprietário. Este software foi desenvolvido segundo o modelo de camadas e foram analisadas duas das camadas separadamente. As duas camadas podem ser modeladas por *little house*.

A estrutura macroscópica identificada neste estudo traz novas informações para os desenvolvedores de software a respeito da natureza de seus objetos de trabalho. A presença de um componente fortemente conectado enfatiza a necessidade de uma abordagem sistemática para as tarefas de manutenção e testes nas classes deste componente,

Tabela 7.11. Evolução dos tamanhos dos componentes LSCC

Software	LSCC - versão inicial (#classes)	LSCC - versão inicial (% do tamanho do software)	LSCC - versão final (#classes)	LSCC - versão final (% do tamanho do software)
Jsch	16	20	16	14
LogSim	289	32	303	25
Jml	17	10	27	10
JavaGroups	9	2	13	1
DBUnit	16	8	24	7
Hibernate	100	10	477	20
Squirel	40	10	579	47
Junit	22	28	9	4
Jedit	69	20	395	35
Phex	165	41	403	29
Jgnash	276	35	322	11
DrJava	43	2	968	26
Jasper	20	8	150	9
MovieManager	50	78	92	18
FreeCol	5	11	758	70
KolMafia	15	38	748	67

pois uma modificação em uma classe desse componente pode afetar amplamente outras classes do software. O conhecimento sobre a maneira como as classes em um software estão conectadas entre si pode contribuir para a melhoria de técnicas de teste, por exemplo.

7.5 Conclusões

Este capítulo apresentou os resultados de um estudo sobre a caracterização da evolução de software com base em conceitos de redes complexas. Foram analisados 16 softwares abertos e um software proprietário, em um total de 129 versões. Os resultados do estudo mostram que: a densidade da rede de software tende a diminuir à medida que o software cresce; o diâmetro desse tipo de rede é curto; classes com alto grau de entrada tendem a manter essa propriedade ao longo da vida do software; tais classes são instáveis, têm coesão degradada ao longo do tempo, crescem em número de método públicos e em número de atributos públicos. Como a densidade da rede tende a diminuir e as classes com alto grau de entrada tendem a ter graus de entrada ainda maiores, é possível inferir que a prática comum é inserir os novos requisitos em tais

classes em vez de refatorar o sistema. A prática de não refatoração pode ser, então, a causa do surgimento do fenômeno *small world* em redes de software.

Foi identificada neste estudo também uma figura da estrutura macroscópica das redes de software, denominada *little house*. Esta estrutura macroscópica revela importantes propriedades dessas redes, por exemplo, a existência de um núcleo de classes que são fortemente conectadas entre si. Os resultados desse estudo podem ser utilizados para aperfeiçoar tarefas no desenvolvimento de software, por exemplo testes e manutenção. Além disso, os resultados do estudo descrito neste capítulo fornecem embasamento para premissas assumidas na definição do modelo K3B. Por esse estudo, pode-se observar empiricamente, por exemplo, a existência de um componente fortemente conectado em software que abrange uma parcela cada vez maior de classes à medida que o software cresce. Essa propriedade torna evidente que o problema de propagação de modificações em software tende a ser cada vez mais crítico com o crescimento do software, e que para que o mecanismo de avaliação de propagação de modificações seja eficiente, ele deve considerar essa propriedade.

Capítulo 8

K3B - Um Modelo de Predição de Amplitude da Propagação de Modificações Contratuais em Software Orientado por Objetos

O objetivo principal desta tese é a definição de um modelo de predição de amplitude de propagação de *modificações contratuais* em software orientado por objetos, denominado K3B. O modelo K3B é definido com base no raciocínio teórico sobre a natureza e implicações das relações entre os módulos de um software e nos conceitos de Probabilidades, em particular, Cadeias de Markov.

Este capítulo apresenta o modelo K3B. A Seção 8.1 enuncia o problema que o modelo visa solucionar. A Seção 8.2 descreve o modelo que Myers [1975] utiliza para ilustrar o problema de propagação de modificações em software. A Seção 8.3 descreve a metodologia utilizada na elaboração de K3B. A Seção 8.4 define o modelo de predição de amplitude da propagação de modificações contratuais em software orientado por objetos. Essa é uma definição preliminar que considera inicialmente que os módulos de um software são totalmente conectados entre si. A Seção 8.5 refina essa definição, introduzindo no modelo o fator que avalia o grau de conectividade do software. A Seção 8.6 apresenta o modelo proposto final, discute as implicações de K3B e indica as perspectivas de seu uso no processo de desenvolvimento e manutenção de software.

8.1 Enunciado do Problema

Um software é constituído por módulos que interagem entre si. Na orientação por objetos, estes módulos são tipicamente as *classes*. Classes definem as características e o comportamento de seus *objetos*. O princípio fundamental no desenvolvimento de um software orientado por objetos é a ocultação de informação, que define que detalhes de implementação de um serviço devem ser transparentes para o usuário dele. Meyer [1997] define um método de construção de software orientado por objetos conhecido como *Design by Contract*, que considera um software como um conjunto de módulos que interagem entre si por meio de regras bem especificadas denominadas *contratos*. Nessa abordagem, a construção de uma classe envolve a definição de seu *contrato*, definido pelo conjunto de serviços exportados pela classe, e as pré e pós condições destes serviços. Uma pré-condição é uma condição que deve ser observada para que o serviço seja realizado corretamente. Uma pré-condição pode determinar, por exemplo, as faixas de valores válidos dos parâmetros a serem passados ao serviço, o estado das variáveis do objeto e das variáveis referenciadas pelo serviço antes da sua execução. Uma pós-condição corresponde ao estado que o sistema terá após a execução do serviço. A comunicação entre objetos se dá, então, pelos contratos definidos em suas classes: um objeto A que utiliza determinado serviço de outro objeto B deve satisfazer as pré-condições para o uso de tal serviço; em contrapartida, o objeto B deve garantir a execução correta do serviço solicitado, gerando o resultado esperado, que corresponde às pós-condições do serviço.

Idealmente, se houver uma modificação na implementação de determinado serviço, mas não houver alteração no *contrato* da classe, o usuário do serviço não deve sofrer impacto. Por outro lado, uma modificação em uma classe que altere a definição de seu contrato pode demandar modificações em outras classes do software. Este tipo de modificação é referenciada neste trabalho como *modificação contratual*.

Em relação à atividade de modificação, os módulos de um software podem assumir dois estados: *em modificação* e *atualizado*. Um módulo assume o estado *em modificação* quando se inicia uma modificação nele que altere seu contrato. O módulo assume o estado *atualizado* quando tal modificação é concluída. Neste trabalho, define-se um *passo de modificação* como uma transição entre esses estados em um dos módulos do software. Um passo de modificação ocorre, então, quando uma modificação é iniciada ou concluída em um módulo do software. Desta forma, um passo de modificação corresponde ao evento que aumenta ou diminui, em uma unidade, o número de módulos *em modificação* durante um processo de modificação. As Figuras 8.1 e 8.2 exemplificam um processo de modificação. Na figura, os retângulos sombreados representam classes

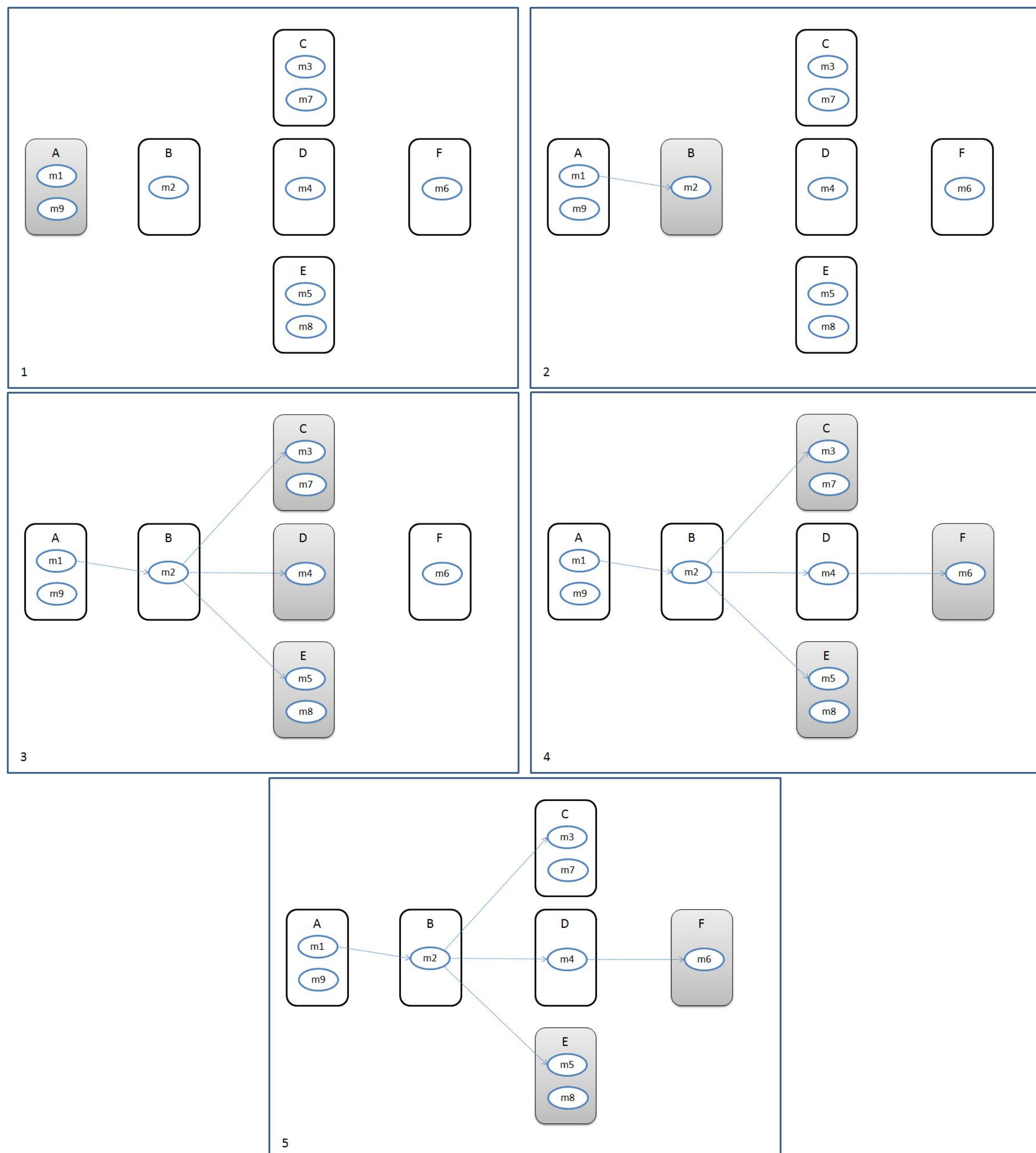


Figura 8.1. Exemplo de propagação de modificações contratuais.

em modificação, as elipses representam métodos e as setas representam propagação das modificações. Inicialmente, o método `m1` da classe `A` sofre uma modificação. Tal modificação demanda que o método `m2` da classe `B` seja também modificado. Esse evento é considerado *um passo de modificação*. Com isso, dois módulos do sistema, `A` e `B`, estão *em modificação*. Por simplificação, a figura não exibe essa situação intermediária. `A`, então, tem sua modificação concluída e passa para o estado *atualizado*, enquanto `B` está

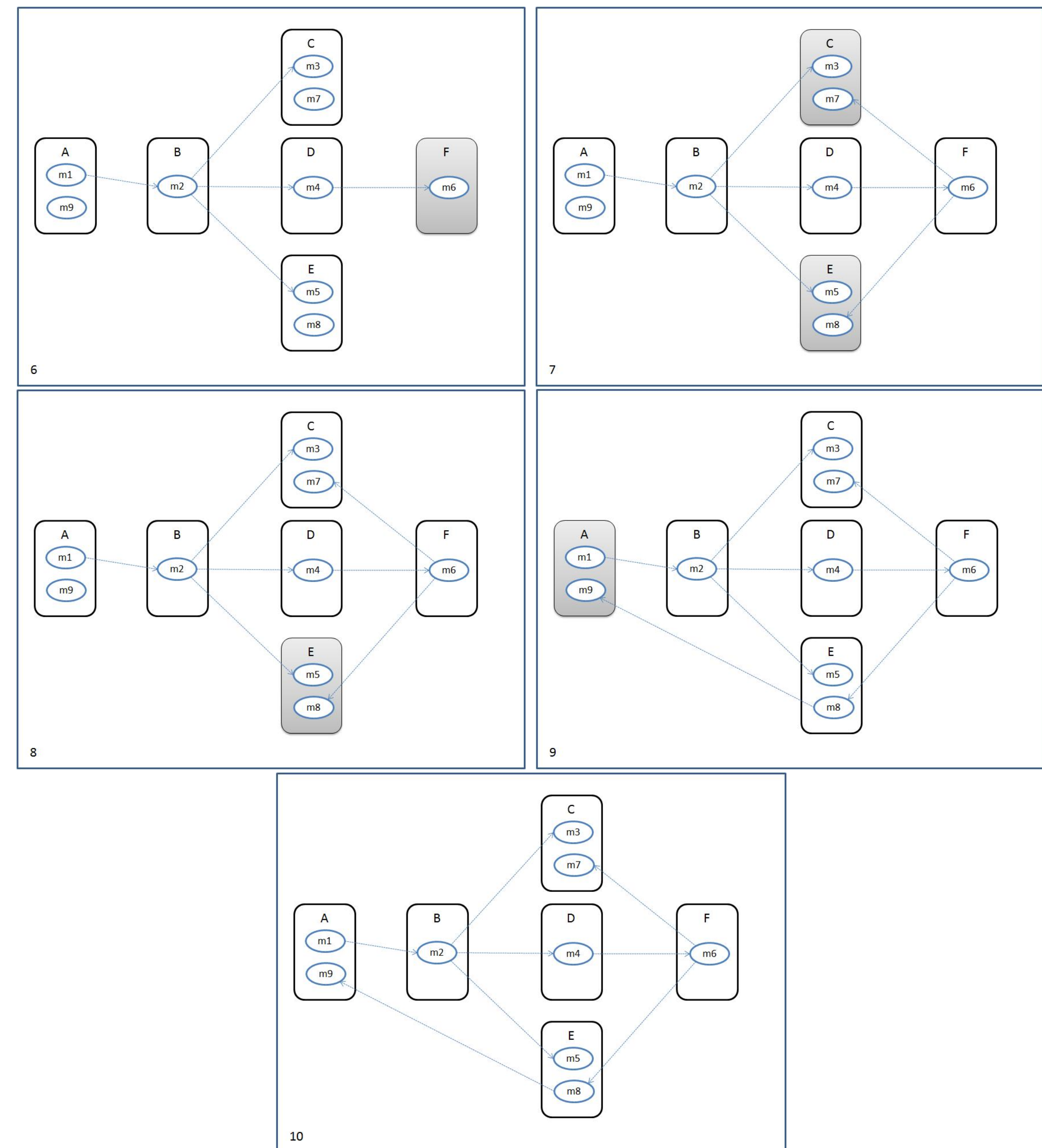


Figura 8.2. Exemplo de propagação de modificações contratuais - continuação

em modificação. A modificação de `m2` é propagada para `m3`, `m4` e `m5`, das classes `C`, `D` e `E`, respectivamente. Neste momento, `C`, `D` e `E` passam para o estado *em modificação*. A modificação em `D` é propagada para `F`. Neste momento, há quatro modulos do software *em modificação*: `C`, `D`, `E` e `F`. `D`, então, passa para o estado *atualizado*. As modificações em `C` e `E` são concluídas. Porém, a modificação em `F` gera novas modificações em `C` e `E`. A nova modificação em `E`, por sua vez, gera outra modificação `A`, que não dispara mais

modificações no sistema.

O modelo proposto neste trabalho lida com o problema de estimar a propagação de modificações contratuais em software orientado por objeto. Dado um software com n módulos, o modelo estima o número médio de passos de modificações a serem realizados quando i desses módulos são inicialmente modificados. A seção seguinte descreve um modelo que Myers [1975] utiliza para ilustrar o problema de propagação de modificações em software.

8.2 Modelo de Myers - Software Visto como um Circuito de Lâmpadas

Seja um software com n módulos. Um módulo pode assumir dois estados: *em modificação* e *atualizado*. Se um módulo está *em modificação* em um dado momento, seja x a probabilidade de ele passar para o estado *atualizado* no próximo intervalo de tempo t; se um módulo está *atualizado*, seja y a probabilidade de ele passar para o estado *em modificação* no próximo intervalo de tempo t se ele estiver conectado a algum outro módulo que esteja *em modificação*. Por outro lado, se um módulo estiver *atualizado*, permanecerá assim enquanto todos os módulos a ele conectados estiverem *atualizados*. Diz-se que o software atingiu o equilíbrio quando todos os módulos estiverem atualizados. Dada uma situação inicial do software na qual i módulos estão *em modificação*, deseja-se saber o tempo de equilíbrio E para que o software alcance o equilíbrio.

Myers [1975] modela de forma simplificada este problema como um circuito de lâmpadas. Neste modelo, as lâmpadas correspondem aos módulos. Uma lâmpada pode assumir dois estados: *acesa* e *apagada*. O estado da lâmpada corresponde à ocorrência de modificação no módulo: *acesa* indica módulo em modificação e *apagada* indica módulo em que não está ocorrendo modificação. Se uma lâmpada está *acesa*, a probabilidade desta passar para o estado *apagada* no próximo segundo é 0,5; se uma lâmpada está *apagada*, a probabilidade desta passar para o estado *acesa* no próximo segundo é 0,5 se estiver conectada a alguma outra lâmpada *acesa*. Por outro lado, se uma lâmpada estiver *apagada*, permanecerá assim enquanto todas as lâmpadas conectadas a ela estiverem *apagadas*. Diz-se que o circuito atingiu o equilíbrio quando todas as lâmpadas estiverem *apagadas*.

Para o caso em que inicialmente todas as lâmpadas do circuito estão acesas, Myers determina o tempo médio de equilíbrio do circuito nas seguintes situações:

- Inexistência de conexões entre as lâmpadas: o tempo médio de equilíbrio do circuito é de 7 segundos.

- Circuito constituído por agrupamentos de 10 lâmpadas, sendo que os agrupamentos são independentes entre si e cada um deles é totalmente conectado: neste caso, o tempo médio 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. Neste caso, o tempo médio para atingir o equilíbrio do circuito é de 10^{22} anos.

O modelo de Myers [1975] é uma metáfora do problema de propagação de modificações em software. Trata-se de um modelo informal e qualitativo, para o qual não há qualquer demonstração. Ele, contudo, enfatiza a importância do grau de conectividade de um software neste problema, mostrando que o controle de propagação de modificações em software cujos módulos sejam muito conectados pode ser um problema intratável. O trabalho realizado nesta tese visa estabelecer formalmente um meio de estimar a propagação de modificações em software. As seções seguintes apresentam a solução proposta para este problema.

8.3 Metodologia

A elaboração de K3B foi realizada de acordo com as seguintes etapas:

1. **Definição matemática do modelo:** o modelo foi elaborado com base nos conceitos de Probabilidades. O processo de modificação de software é modelado como uma Cadeia de Markov, na qual cada estado corresponde ao número de módulos que estão sofrendo modificações em determinado instante. No modelo, as probabilidades de mudança de estados são determinadas pela conectividade do software, por fatores que potencializam a propagação de modificações e por fatores que a amenizam. As transições entre estados são representadas por uma matriz. A aplicação de um teorema que determina o número médio de passos para alcançar estabilidade em uma Cadeia de Markov com as características daquela utilizada no modelo resultou em uma expressão que envolve cálculos com variáveis simbólicas e matrizes, o que dificulta a sua compreensão e uso efetivo. Esta expressão é basicamente um vetor de tamanho n no qual cada posição i representa o número médio de passos para alcançar estabilidade na cadeia a partir do estado i.

2. **Implementação da expressão com variáveis simbólicas e matrizes:** a expressão obtida foi implementada utilizando-se o Matlab. Foram gerados resultados para valores de n de 4 a 25. Não foi possível gerar dados para valores de n superiores a 25 porque o tempo de execução tornou essa tarefa inexequível. Como resultado, obteve-se um vetor no qual cada posição i representa a informação buscada pelo modelo: $t = E(n, i)$, onde t é o número esperado de passos de modificações, n é o número de módulos do software e i é o número de módulos nos quais serão realizadas modificações inicialmente. Não havia expectativa prévia quanto ao formato desse resultado. No vetor resultante dessa implementação, cada posição foi um polinômio. Com isso, foi necessário analisar esses polinômios a fim de identificar uma representação genérica para eles e, então, generalizar o resultado.

3. **Análise preliminar dos polinômios obtidos:** com a análise preliminar dos polinômios obtidos, observou-se a existência de um padrão de formação entre eles de difícil generalização. Optou-se, então, por representar o polinômio que fornece $t = E(n, i)$ por apenas um de seus termos e avaliar o comportamento deste termo. Isso levou a uma expressão preliminar de K3B.

4. **Implementação da expressão preliminar e experimentos:** a implementação e experimentos com a expressão preliminar mostraram que essa expressão não é eficaz, pois em muitos casos gerou números muito próximos de zero, de difícil interpretação. Com isso, fez-se necessário uma nova análise dos polinômios obtidos, a fim de identificar uma representação genérica completa para eles.

5. **Análise geral dos polinômios obtidos e definição de K3B:** a análise geral dos polinômios resultou na identificação de uma expressão genérica para $t = E(n, i)$, que corresponde ao modelo K3B.

K3B foi implementado em uma ferramenta que realiza coleta de métricas em software Java, denominada *Connecta* Ferreira [2006].

8.4 Definição do Modelo K3B

Esta seção define K3B, o modelo de predição de amplitude de propagação de modificações contratuais em software orientado por objetos. O modelo estima o número de passos de modificações em um software constituído por n módulos no qual inicialmente i desses módulos sofrerão modificações.

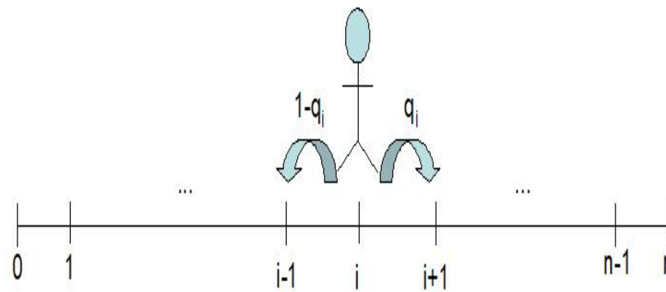


Figura 8.3. Passeio Aleatório do processo de modificação de software: i é o número de módulos em modificação em determinado instante

Seja um sistema totalmente conectado constituído por n módulos, onde $S = \{0, 1, 2, ..., n\}$ denota o conjunto de estados possíveis para o sistema. Cada estado corresponde ao número de módulos em modificação em um dado instante. Quando o número de módulos em modificação for igual a zero, diz-se que o sistema alcançou estabilidade. Deseja-se saber o número médio de passos de modificação necessários para que um sistema chegue à estabilidade dado o seu estado, ou seja, dado o número de módulos em modificação inicialmente.

No sistema, dado que há i módulos em modificação, $i \epsilon \{1, 2, 3, ..., n-1\}$, no próximo instante só é possível ir para o estado em que haverá *i+1* módulos em modificação ou para o estado em que haverá *i-1* módulos em modificação. Este problema pode ser modelado como uma Cadeia de Markov[1] [Petrov & Mordecki., 2003]. Em uma Cadeia de Markov, se i é o estado presente, a probabilidade de passar para os estados futuros *i+1*, ... ,*i+m* depende somente do estado presente i e não dos estados anteriores a i. Em outras palavras, o número de módulos em modificação nos instantes futuros depende somente do número de módulos em modificação no instante presente. A Figura 8.3 ilustra o Passeio Aleatório [Petrov & Mordecki., 2003] para o problema. Esse passeio aleatório modela o processo de modificação de software, no qual i corresponde ao número de módulos em modificação em um dado instante. A cada passo do processo, há duas possibilidades: ou mais um módulo entra em modificação, o que corresponde ao processo ir para o estado *i+1*, ou um módulo tem sua modificação finalizada, o que corresponde ao processo ir para o estado *i-1*. Desta forma, dado um sistema no estado i, a probabilidade de passar para o estado *i+1* é q_i, e a probabilidade de no próximo instante passar para o estado *i-1* é $1 - q_i$.

O Passeio Aleatório é representado por uma matriz de probabilidades $P = (p_{ij})$

[1] A elaboração deste modelo somente foi possível devido à preciosa colaboração do Prof. Bernardo de Lima, do Departamento de Matemática da UFMG

na qual i referencia as linhas e j, as colunas, e cada posição ij na matriz representa a probabilidade de se passar do estado em que há i módulos em modificação para outro seguinte em que há j módulos em modificação. Neste problema, há as seguintes definições:

1. O estado em que não há módulos em modificação é o *estado absorvente*, o estado em que o sistema atingiu estabilidade. A probabilidade de sair deste estado é 0, e a probabilidade de permanecer nele é 1. A matriz de probabilidades, na linha 0, tem valor 1 na coluna 0, e 0 nas demais colunas.

$$p_{0j} = \begin{cases} 1 & \text{se } j = 0 \\ 0 & \text{se } j \neq 0 \end{cases}$$

2. No estado em que todos os módulos estão em modificação, só é possível passar para o estado em que haverá *n-1* módulos em modificação. A probabilidade de ir para o estado *n+1* é 0, e a probabilidade de ir para o estado *n-1* é 1. A matriz de probabilidades, na linha n, tem valor 1 na coluna *n-1*, e valor 0 nas demais colunas.

$$p_{nj} = \begin{cases} 1 & \text{se } j = n - 1 \\ 0 & \text{se } j \neq n - 1 \end{cases}$$

3. Nos estados intermediários i, $i \in \{1, 2, 3, ..., n{-}1\}$, nos quais há de *1* a *n-1* módulos em modificação, não é possível passar para estados diferentes de *i+1* e *i-1*. A probabilidade de passar para o estado *i+1* é q_i, e a probabilidade de passar para *i-1* é $1 - q_i$.

$$p_{ij} = \begin{cases} 0 & \text{se } j \neq i + 1, i - 1 \\ q_i & \text{se } j = i + 1 \\ 1 - q_i & \text{se } j = i - 1 \end{cases}$$

Pelas características do problema de estabilidade de software e *supondo-se que no sistema cada um dos módulos possui conexão com todos os demais*, é razoável concluir que a partir do estado i, a taxa de mudança para o estado *i+1* é:

- proporcional ao número de módulos que não estão em modificação (*n-i*): quanto maior o número de módulos que não estão em modificação, maior a chance de que pelo menos um deles entre em modificação.

- proporcional ao número de módulos em modificação (i): quanto mais módulos em modificação, maior a chance de outros módulos do sistema entrarem em modificação.

Desta forma, a partir de um estado i, a taxa de mudança para *i+1* é dada em função de $\alpha(n-i)i$. Utilizamos o termo "taxa de contaminação" para designar a taxa de módulos nos quais será necessário realizar modificações em decorrência do número de módulos em modificação no instante corrente. Nesta equação, α é um parâmetro que indica a "taxa de contaminação" do sistema. Ele deve ser um fator que quanto maior, maior também a "taxa de contaminação". Um bom candidato a este fator é *grau de acoplamento* [Myers, 1975] entre os módulos do software, pois quanto maior o grau de acoplamento, maior o impacto de uma manutenção de um módulo nos demais.

É possível concluir também que a partir do estado i, a taxa de mudança para o estado *i-1* é proporcional ao número de módulos que estão em modificação. Esta taxa é dada em função de βi. Utilizamos o termo "taxa de melhora" para designar a taxa de módulos nos quais as modificações serão finalizadas sem implicar em modificações de outros módulos do sistema. Nesta equação, o parâmetro β indica a "taxa de melhora" do sistema. Quanto maior for β, maior a chance de as manutenções dos módulos serem finalizadas sem afetar outros módulos. Deve ser um fator que quanto maior, maior também a "taxa de melhora". Um bom candidato a estimar este fator é grau de *coesão interna dos módulos* [Myers, 1975].

Como a soma de $p_{i,i+1}$ e $p_{i,i-1}$, $i \epsilon \{1, 2, 3, ..., n-1\}$, deve ser igual a 1, obtêm-se as Equações 8.1 e 8.2.

$$p_{i,i+1} = \frac{\alpha(n-i)i}{\alpha(n-i)i + \beta i} \tag{8.1}$$

$$p_{i,i-1} = \frac{\beta i}{\alpha(n-i)i + \beta i} \tag{8.2}$$

O problema investigado consiste em saber o número de passos que o sistema levará para sair de um estado i, $i \neq 0$, e chegar ao *estado absorvente*, $i = 0$. Ou seja, busca-se o número médio de passos para o sistema estabilizar dado que há i módulos em modificação, referenciado aqui como $E(t_i)$.

Há um teorema, descrito por Grinstead & Snell. [1991], para resolver a seguinte questão em uma Cadeia de Markov com estados absorventes: dado que a cadeia encontra-se no estado i, qual é o número esperado de passos para que a cadeia alcance o estado absorvente. Para que este teorema, que será definido posteriormente, possa ser aplicado ao presente problema, constrói-se uma matriz de probabilidades

	1	2	3	4	...	n-1	n	0
1	**0**	q_1	0	0	0	0	0	$1- q_1$
2	$1-q_2$	**0**	q_2	0	0	0	0	0
3	0	$1-q_3$	**0**	q_3	0	0	0	0
4	0	0	$1-q_4$	**0**	...	0	0	0
...	...	...	...	...	...	...	...	...
n-1	0	0	0	0	$1-q_{n-1}$	**0**	q_{n-1}	0
n	0	0	0	0	0	1	**0**	0
0	0	0	0	0	0	0	0	1

Figura 8.4. Matriz de probabilidades

$(n+1) \times (n+1)$, onde n é o número total de módulos do sistema. Esta matriz deve ser indexada de forma que o estado absorvente corresponda à última linha e à última coluna da matriz. A matriz de probabilidades $P = (p_{ij})$ é preenchida como mostra a Figura 8.4. Nesta matriz, para cada linha i, somente os elementos correspondentes às colunas *i+1* e *i-1* podem ter valores diferentes de 0. No caso da linha 0, que corresponde ao estado absorvente, a coluna 0 tem valor 1 e as demais têm valor 0.

A forma da matriz de probabilidades é mostrada na Figura 8.5. Nesta matriz: A é uma matriz $n \times n$; *0* é uma matriz linha de n elementos de valor 0; I é a matriz identidade; e B é uma matriz coluna de n elementos, na qual o elemento da linha 1 pode ter valor diferente de 0 e os demais têm valor igual 0.

O teorema, cuja demonstração pode ser vista em Grinstead & Snell. [1991], é definido da seguinte forma:

Teorema: Seja $E(t_i)$ o número esperado de passos antes da cadeia ser absorvida, dado que a cadeia inicia no estado i, e seja E o vetor coluna cuja i-ésima entrada é $E(t_i)$. Então $E = Nc$, onde $N = (I - A)^{-1}$ e c é um vetor coluna no qual todas as entradas são iguais a 1.

Aplicando-se o Teorema ao presente problema, tem-se:

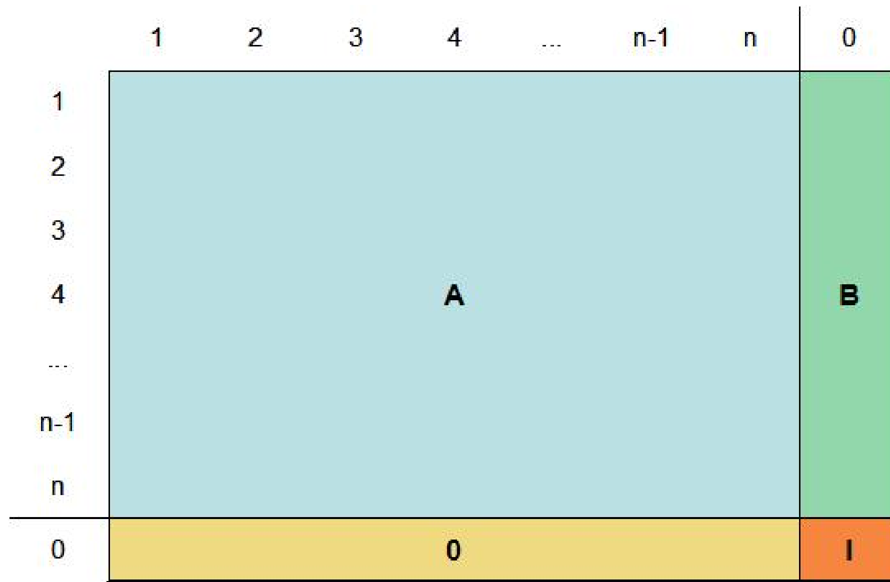


Figura 8.5. Forma da matriz de probabilidades

$$\begin{pmatrix} E(t_1) \\ E(t_2) \\ E(t_3) \\ \vdots \\ E(t_n) \end{pmatrix} = (\mathbf{I} - \mathbf{A})^{-\mathbf{1}} \begin{pmatrix} 1 \\ 1 \\ 1 \\ \vdots \\ 1 \end{pmatrix}$$

Os dois vetores desta igualdade têm n *elementos*. Cada posição do vetor resultante fornece $E(t_i)$, que corresponde ao número de passos esperado para que o sistema chegue ao estado absorvente a partir do estado i.

A fim de saber o número de passos esperado para que o sistema alcance a estabilidade quando todos os módulos estão em modificação, deve-se calcular o valor do vetor resultante na linha n. Isso é dado pela multiplicação entre a última linha da matriz $(I - A)^{-1}$ e o vetor de valores 1, ou seja, é a soma dos elementos da última linha da matriz $(I - A)^{-1}$.

8.4.1 Resultados

Com o auxílio do software Matlab, o vetor E foi calculado para $n \in \{4, 5, 6, ..., 11, 20, 25\}$. Os resultados deste cálculo são apresentados no Apêndice A1. A partir da análise dos dados obtidos, concluiu-se que o número de passos para

alcançar estabilidade em um software cresce fatorialmente em função do tamanho do software. A seguir esta análise é discutida.

O número de passos para alcançar estabilidade em um software com n módulos dentre os quais i módulos passam por modificação, $E(t_i)$, é dado por um polinômio de n termos, na variável α/β, com a formação mostrada na Equação 8.3. Por exemplo, o número de passos esperado para chegar à estabilidade no sistema com 5 módulos no qual há um módulo em modificação é dado pelo polinômio mostrado na Equação 8.4. O fato de esses polinômios serem na variável α/β indica que o número de passos de modificações depende da relação entre os fatores α e β, e não desses fatores isoladamente. As implicações desse achado serão discutidas em maiores detalhes mais adiante, na Seção 8.6.3.

$$Et_i = i + c_1 \frac{\alpha}{\beta} + c_2 \frac{\alpha^2}{\beta^2} + c_3 \frac{\alpha^3}{\beta^3} + ... + c_{n-1} \frac{\alpha^{n-1}}{\beta^{n-1}} \tag{8.3}$$

$$E1 = 1 + 8 \frac{\alpha}{\beta} + 24 \frac{\alpha^2}{\beta^2} + 48 \frac{\alpha^3}{\beta^3} + 48 \frac{\alpha^4}{\beta^4} \tag{8.4}$$

A inspeção dos polinômios gerados mostra que para um determinado valor de n os últimos termos dos polinômios $E(t_i)$ são iguais. Por exemplo, todos os polinômios obtidos para $n = 5$ têm o último termo igual a $48 \frac{\alpha^4}{\beta^4}$. Os valores dos últimos termos dos polinômios para $n \epsilon \{4, 5, 6, ..., 11, 20, 25\}$ são mostrados na Tabela 8.1. Observa-se que, em todos esses polinômios, o último termo é dado pela Equação 8.5.

O parâmetro α indica a "taxa de contaminação" do sistema, e β, a sua "taxa de melhoria". Quanto maior o valor de α, maior também o número de passos necessários para o sistema estabilizar. Quanto menor o valor de β, maior o número de passos necessários para o sistema estabilizar. Se, por exemplo, acoplamento médio entre os módulos do sistema for usado para o parâmetro α e coesão média dos módulos for usada para o parâmetro β, significa dizer que quanto maior o acoplamento e menor a coesão, maior o número de passos de modificações, ou seja, maior a dificuldade de modificação do sistema.

O número de passos de modificações para alcançar estabilização em um sistema é da ordem de $n! \frac{\alpha^{n-1}}{\beta^{n-1}}$. Ou seja, tomando-se apenas o último termo dos polinômios já é possível concluir que a propagação de modificações em software pode ser explosiva. Na prática, isso traduz-se na impossibilidade de garantir a estabilidade de um software fortemente conectado e que apresentam alto valor de α e baixo valor de β.

$$E(n) = (n-2)!(2n-2) \frac{\alpha^{n-1}}{\beta^{n-1}} \tag{8.5}$$

n	*último termo*
4	$12\,\frac{\alpha^3}{\beta^3}$
5	$48\,\frac{\alpha^4}{\beta^4}$
6	$240\,\frac{\alpha^5}{\beta^5}$
7	$1440\,\frac{\alpha^6}{\beta^6}$
8	$10080\,\frac{\alpha^7}{\beta^7}$
9	$80640\,\frac{\alpha^8}{\beta^8}$
10	$725760\,\frac{\alpha^9}{\beta^9}$
11	$7257600\,\frac{\alpha^{10}}{\beta^{10}}$
20	$2432902008176640000\,\frac{\alpha^{19}}{\beta^{19}}$
25	$1240896803466478878720000\,\frac{\alpha^{24}}{\beta^{24}}$

Tabela 8.1. Último termo do polinômio $E(t_i)$

8.5 Aproximação do Modelo Proposto a Software Real

O modelo, tal como está definido até aqui, considera que *no sistema cada um dos módulos possui conexão com todos os demais*. Entretanto, um software real não possui esta configuração necessariamente. Esta seção introduz no modelo o fator ϕ, que representa a probabilidade de existência de conexão entre um par de módulos. Este fator corresponde ao percentual de conexões existentes no software em relação ao número máximo de conexões possíveis nele.

A métrica COF (*Coupling Factor*), proposta por Abreu & Carapuça [1994], indica o grau de conectividade de um software orientado por objetos. Esta métrica é dada pela razão entre o número de conexões existentes no software e o maior número possível de conexões que podem existir no software. O cálculo da métrica considera que um software pode ser modelado como um grafo direcionado sem *self-loops*, no qual os vértices representam as classes do software e as arestas, os relacionamentos de dependência entre elas. Em um software com n classes, o maior número possível de conexões é $n(n-1)$. Um software totalmente conectado possui COF igual a 1, e um software sem conexões possui COF igual a 0. Como os valores do grau de conectividade variam de 0 a 1, COF pode ser tomada como a probabilidade de uma conexão entre duas classes existir. Esta métrica é utilizada em K3B como o fator ϕ. Com isso, o modelo passa a considerar as conexões que de fato existem no software. A revisão das expressões de probabilidades utilizadas na definição de K3B considerando-se ϕ é

descrita a seguir.

Quanto maior o grau de conectividade de um software, maior a chance de uma modificação em um módulo provocar modificações em outros módulos do sistema. Desta forma, a partir de um estado i, a taxa de mudança para o estado *i+1* é dada em função de $\alpha\phi(n-i)i$, enquanto taxa de mudança para o estado *i-1* é dada em função de βi. Como a soma de $p_{i,i+1}$ e $p_{i,i-1}$, $i \epsilon \{1, 2, 3, ..., n-1\}$, deve ser igual a 1, obtêm-se as Equações 8.6 e 8.7.

$$p_{i,i+1} = \frac{\alpha\phi(n-i)i}{\alpha\phi(n-i)i + \beta i} \quad (8.6)$$

$$p_{i,i-1} = \frac{\beta i}{\alpha\phi(n-i)i + \beta i} \quad (8.7)$$

Com o auxílio do software Matlab, o vetor E foi calculado novamente considerando-se a inclusão do fator ϕ nos cálculos das probabilidades utilizadas no modelo. Os resultados obtidos são similares àqueles mostrados no Apêndice A.1, porém são polinômios na variável $\frac{\phi\alpha}{\beta}$. O Apêndice A.2 exemplifica alguns desses resultados, incluindo o fator ϕ. O número de passos de modificação para chegar à estabilidade em um software com n módulos dentre os quais i módulos passam por modificação, $E(t_i)$, é dado por um polinômio de n termos com a formação mostrada na Equação 8.8.

$$Et_i = i + c_1\,\frac{\alpha\phi}{\beta} + c_2\,\frac{\alpha^2\phi^2}{\beta^2} + c_3\,\frac{\alpha^3\phi^3}{\beta^3} + ... + c_{n-1}\,\frac{\alpha^{n-1}\phi^{n-1}}{\beta^{n-1}} \quad (8.8)$$

O número de passos de modificação para alcançar estabilização em um software com n módulos, grau de conectividade ϕ, taxa de contaminação α e taxa de melhora β é da ordem de $n!\,\frac{\alpha^{n-1}\phi^{n-1}}{\beta^{n-1}}$. Esta expressão foi tomada como o modelo K3B preliminar.

8.6 K3B

A expressão obtida na seção anterior foi implementada em *Connecta* e alguns experimentos preliminares foram realizados. Os valores obtidos não foram satisfatórios porque foram, em geral, muito próximos de zero, o que não fornece informação útil e impede o seu uso efetivo. Desta forma, fez-se necessário investir esforço na tentativa de identificar uma fórmula mais apurada que represente os dados obtidos. A dificuldade em se fazer isso é que não há um meio automatizado via software para análise dos dados.

Após uma análise minuciosa dos dados, observou-se a existência de um complexo padrão de formação nos polinômios que fornece o número de passos de modificações

necessários para chegar a estabilização em um software com n módulos dentre os quais i módulos passam por manutenção: há um padrão na formação dos coeficientes dos termos de um polinômio e, além disso, um polinômio é definido recursivamente em relação ao polinômio que define o número de passos quando há *i-1* módulos em modificação. A Figura 8.6 exemplifica o padrão de formação dos polinômios para o caso de software com seis módulos.

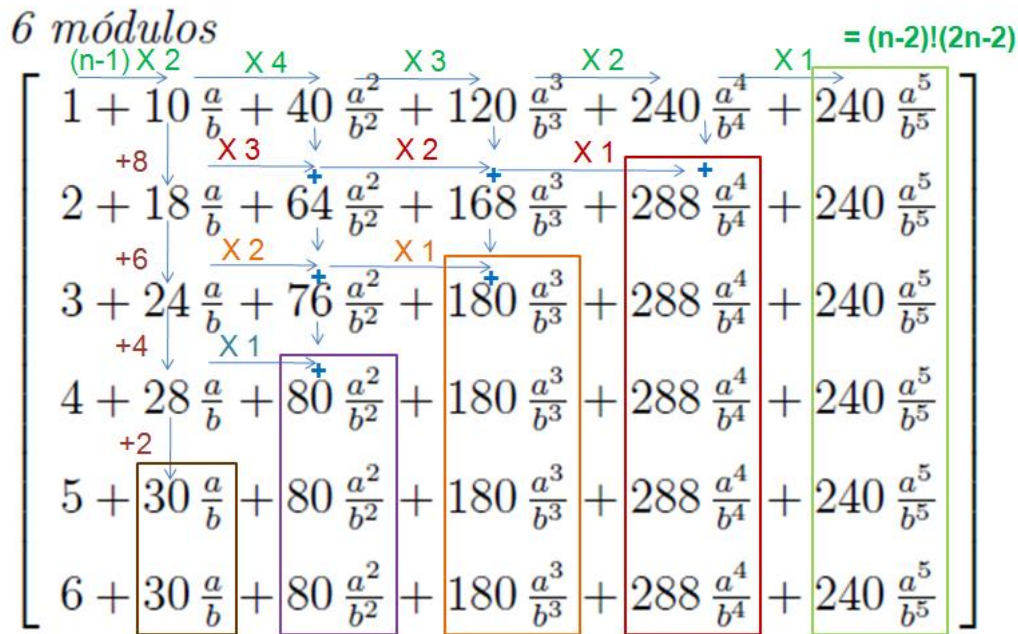


Figura 8.6. Padrão de formação dos polinômios obtidos

$$E(n,1) = 1 + 2(n-1)(n-2)!\,\frac{\alpha^{n-1}\phi^{n-1}}{\beta^{n-1}} + \sum_{k=1}^{k=n-2} \frac{2(n-1)(n-2)!}{k!}\,\frac{\alpha^{n-k-1}\phi^{n-k-1}}{\beta^{n-k-1}} \qquad (8.9)$$

$$E(n,i) = 1 + E(n,i-1) + \sum_{k=i-1}^{k=n-2} 2(n-i)\,\frac{(n-i-1)!}{(k-i+1)!}\,\frac{\alpha^{n-k-1}\phi^{n-k-1}}{\beta^{n-k-1}} \qquad (8.10)$$

A partir desse padrão de formação, foram deduzidas empiricamente as Equações 8.9 e 8.10, que consistem no modelo K3B. A Equação 8.9 fornece o número de passos de modificações quando um módulo do software sofre modificação inicialmente. A Equação 8.10 fornece o número de passos de modificações quando i módulos do software sofrem modificação inicialmente. Nestas equações, α é a taxa de contaminação, β é

a taxa de melhoria e ϕ é o grau de conectividade do software, sendo que $0 \leq \alpha \leq 1$, $0 < \beta \leq 1$ e $0 \leq \phi \leq 1$.

8.6.1 Implementação

A fórmula K3B possui termos que são fatoriais. A implementação de programas para cálculos de fatoriais é problemática devido à ocorrência de *overflow* mesmo para números pequenos. Para solucionar esse problema, foi realizada uma transformação logarítmica em K3B. No caso da função fatorial, ela pode ser generalizada para números reais por uma outra função referenciada na literatura como *função gama* [Wikipedia, 2009]. A relação entre a função fatorial e a função gama é mostrada na Equação 8.11.

$$n! = \Gamma(n+1) = \int_0^\infty t^n e^{-t} dt \tag{8.11}$$

O logaritmo natural de *n!* equivale a $ln(\Gamma(n+1))$. O cálculo deste logaritmo, conhecido como *lngamma*, pode ser realizado de forma direta, sem o cálculo prévio de *n!*. Há implementações em Java disponíveis para o seu cálculo. Uma delas faz parte do projeto *Weka 3: Data Mining Software in Java* [Weka, 2009]. A transformação logarítmica de K3B é demonstrada a seguir.

Sejam A e B_k os termos da expressão *E(n,1)*, da seguinte forma:

$$E(n,1) = 1 + \underbrace{2(n-1)(n-2)!\,\frac{\alpha^{n-1}\phi^{n-1}}{\beta^{n-1}}}_{A} + \sum_{k=1}^{k=n-2} \underbrace{\frac{2(n-1)(n-2)!}{k!}\,\frac{\alpha^{n-k-1}\phi^{n-k-1}}{\beta^{n-k-1}}}_{B_k}$$

$$A = 2(n-1)(n-2)!\,(\frac{\alpha\phi}{\beta})^{n-1}$$

$$B_k = \frac{2(n-1)(n-2)!}{k!}\,(\frac{\alpha\phi}{\beta})^{n-k-1}$$

O logaritmo natural de A é dado por:

$$ln(A) = ln(2) + ln(n-1) + lngamma(n-1) + (n-1)\,(ln(\phi) + ln(\alpha) - ln(\beta))$$

Com essa transformação logarítmica, A é dado por:

$$A = exp(ln(2) + ln(n-1) + lngamma(n-1) + (n-1)\,(ln(\phi) + ln(\alpha) - ln(\beta)))$$

O logaritmo natural de B_k é dado por:

$$ln(B_k) = ln(2) + ln(n-1) + lngamma(n-1) - lngamma(k+1) +$$

$$(n-k-1)\,(ln(\phi) + ln(\alpha) - ln(\beta))$$

Com essa transformação logarítmica, B_k é dado por:

$$B_k = exp(ln(2) + ln(n-1) + lngamma(n-1) - lngamma(k+1) +$$

$$(n-k-1)\,(ln(\phi) + ln(\alpha) - ln(\beta)))$$

Desta forma, utilizando-se os resultados das transformações logarítmicas, o valor de *E(n,1)* é dado por:

$$E(n,1) = 1 + A + \sum_{k=1}^{k=n-2} B_k$$

Seja C_k o termo da expressão *E(n,i)*, definido da seguinte forma:

$$E(n,i) = 1 + E(n,i-1) + \sum_{k=i-1}^{k=n-2} \underbrace{2(n-i)\,\frac{(n-i-1)!}{(k-i+1)!}\frac{\alpha^{n-k-1}\phi^{n-k-1}}{\beta^{n-k-1}}}_{C_k}$$

O logaritmo natural de C_k é dado por:

$$ln(C_k) = ln(2) + ln(n-i) + lngamma(n-i) - lngamma(k-i+2) +$$

$$(n-k-1)\,(ln(\phi) + ln(\alpha) - ln(\beta))$$

Com essa transformação logarítmica, o C_k é dado por:

$$C_k = exp(ln(2) + ln(n-i) + lngamma(n-i) - lngamma(k-i+2) +$$

$$(n-k-1)\,(ln(\phi) + ln(\alpha) - ln(\beta)))$$

Desta forma, utilizando-se os resultados das transformações logarítmicas, o valor de *E(n,i)* é dado por:

$$E(n,i) = 1 + E(i-1,n) \sum_{k=i-1}^{k=n-2} C_k$$

8.6.2 A Escolha de Métricas para α e β

A definição do modelo K3B tem por base dois fatores fundamentais: a capacidade que o software tem de evitar que uma modificação propague-se entre módulos, o que foi denominado "taxa de melhora", e a dificuldade do sistema de evitar essa disseminação, o que foi denominado "taxa de contaminação". Essas taxas são denotadas por β e α, respectivamente. No modelo, elas são usadas para definir a probabilidade de uma modificação em um módulo contaminar outros, bem como a probabilidade de uma modificação em um módulo ser realizada sem contaminar outros.

O modelo K3B não define quais indicadores devem ser utilizados para representar α e β. A definição do modelo indica apenas que α deve ser um fator que contribua para o efeito de propagação de modificações no software, enquanto β deve ser um fator que tenha efeito contrário, ou seja, que contribua para que uma modificação fique contida no módulo afetado.

A modularidade é a característica de um software constituído por módulos que interegem entre si, mas que são o mais independente possível uns dos outros. É um consenso que a modularidade é uma característica essencial na construção de software de alta qualidade e que há dois fatores principais que contribuem com a modularidade: *coesão*, que é o grau de relacionamento entre os elementos internos de um módulo, e *acoplamento*, que é o grau de dependência entre módulos. Idealmente, a coesão interna de um módulo deve ser alta, enquanto o acoplamento entre módulos deve ser baixo. Recomenda-se, então, que o grau de acoplamento e a coesão sejam tomados como fatores para representar α e β, respectivamente. O modelo, contudo, não impossibilita que outros fatores ou combinações de outros fatores sejam utilizados para representar α e β. A seguir, são apresentadas orientações de como obter α e β a partir de métricas de software.

8.6.2.1 Métrica de Coesão Interna

A métrica sugerida para representar β é COR, Coesão de Responsabilidade. Esta métrica, definida no Capítulo 6, resulta em valores entre 0 e 1. Valores próximos de 0 indicam baixa coesão interna da classe, enquanto o valor 1 indica alta coesão da classe. Entretanto, outras métricas de coesão interna podem ser perfeitamente empregadas no modelo. Para facilitar o uso de uma métrica de coesão em K3B, o ideal é que valores baixos da métrica representem baixa coesão, enquanto valores altos correspondam a alta coesão.

Como o fator β representa o software como um todo e não uma classe particular, sugere-se tomar a média dos valores de COR obtidos no software para representar β.

Esta foi a abordagem utilizada na implementação de K3B em *Connecta*.

8.6.2.2 Métrica de Acoplamento

O acoplamento é a medida do grau de dependência entre dois módulos. Se um módulo A depende de um módulo B, há uma conexão entre eles. A forma como essa dependência se dá define o grau de acoplamento entre A e B. Por exemplo, na situação em que A usa um atributo de B há um alto grau de acoplamento, já na situação em que A usa apenas métodos de B há um grau de acoplamento menor.

Myers [1975] define uma escala qualitativa para graus de acoplamento no paradigma estruturado que é amplamente conhecida. O menor nível de acoplamento entre dois módulos, denominado acoplamento por informação, se dá quando a comunicação entre eles ocorre via chamada de rotina com passagem de parâmetros por valor, desde que tais parâmetros não sejam utilizados para controlar o fluxo de execução do módulo chamado. Myers [1975] também associa um peso, com valor entre 0 e 1, a cada tipo de acoplamento. Por exemplo, o acoplamento por conteúdo tem peso 0,95, enquanto o acoplamento por informação tem peso 0,2.

Ferreira [2006] adaptou essa escala qualitativa de acoplamentos à orientação por objetos. Entretanto, a automatização da identificação do tipo de acoplamento envolvido no relacionamento entre dois módulos não é simples. Por exemplo, é difícil diferenciar quando o uso de um atributo público ocasiona acoplamento por conteúdo, por dado comum ou por elemento externo. Na implementação da ferramenta *Connecta* utilizou-se uma simplificação desta escala:

- Acoplamento por uso de campo: ocorre quando uma classe utiliza diretamente atributo público da outra. A esse tipo de acoplamento foi atribuído peso 0,2.
- Acoplamento por herança: ocorre quando uma classe é subclasse de outra. A esse tipo de acoplamento foi atribuído peso 0,1.
- Acoplamento por referência: ocorre quando uma classe utiliza métodos de outra passando-lhes objetos como parâmetro. A esse tipo de acoplamento foi atribuído peso 0,1.
- Acoplamento por informação: ocorre quando uma classe utiliza métodos de outra passando-lhes dados primitivos como parâmetro. A esse tipo de acoplamento foi atribuído peso 0,05.

Os valores desses pesos foram calibrados a partir de experimentos preliminares com coleta de valores de K3B em um conjunto pequeno de programas. Em um primeiro

momento, os valores considerados como pesos para os tipos de acoplamento foram: 0,4 para acoplamento por uso de campo, 0,3 para acoplamento por herança, 0,2 para acoplamento por referência, e 0,1 para acoplamento por informação. Os pesos foram definidos com base no grau do acoplamento, ou seja, quanto maior o acoplamento, maior o peso associado a ele. Neste primeiro experimento, os valores gerados por K3B foram muito altos, obviamente muito díspares da realidade. Os valores dos pesos foram, então, gradualmente reduzidos, até que os valores gerados por K3B fossem considerados razoáveis. A definição desses pesos e dos valores considerados razoáveis para os resultados de K3B, contudo, foi meramente intuitiva.

Entre duas classes pode haver mais de uma forma de comunicação. Por exemplo, uma classe B pode usar um atributo público e um método de A. Nestes casos, sugere-se considerar o acoplamento de maior peso. Como o fator α representa o software como um todo e não uma conexão entre duas classes particulares, sugere-se tomar a média dos valores dos pesos obtidos no software para representar α. Esta foi a abordagem utilizada na implementação de K3B em *Connecta*.

8.6.3 A Relação entre α e β

Decorre de K3B que a propagação de modificações em software é dada em função da relação α/β, e não dos fatores α e β isoladamente. Isso significa que se α for aumentado, mas β for aumentado na mesma proporção, a situação do sistema, no aspecto de propagação de modificações, permanecerá a mesma. É importante ressaltar que esta não foi uma premissa inicial na definição do modelo. Este resultado evidenciou-se somente após a obtenção das expressões que estimam o número de passos de modificações, sobre as quais não havia conhecimento prévio algum.

O uso de grau de acoplamento e de grau de coesão para representar α e β, respectivamente, é coerente com esse achado. Acoplamento e coesão são duas das forças mais importantes na qualidade estrutural de um software, que são compensadas entre si. Por exemplo, se em um sistema houver melhoria da coesão interna dos módulos, mas, por alguma razão, houver também aumento do acoplamento entre os módulos, de maneira que a razão entre acoplamento e coesão não se altere, possivelmente não haverá melhora ou piora significativa na qualidade estrutural do software.

É teoricamente sabido e amplamente aceito que o aumento da coesão interna de módulos e a redução de acoplamento entre módulos contribuem para o aumento de modularidade do software. A modularidade, por sua vez, tem papel fundamental na construção de software de alta qualidade e para evitar o efeito cascata do impacto de modificações em software. Desta forma, o aumento da modularidade do software é uma

estratégia para reduzir esse impacto de modificações. K3B confirma matematicamente essa já conhecida relação entre acoplamento, coesão e propagação de modificações em software.

8.6.4 Implicações de K3B

Os fatores sugeridos para α e β são, respectivamente, o grau de acoplamento médio entre os módulos do software e o grau de coesão interna médio deles, e ϕ corresponde ao grau de conectividade do software. Considerando-se isso, quando $\alpha = 0$ e $\phi = 0$, tem-se a situação em que não há conexões entre os módulos do software. Pelo modelo, neste caso, o número de passos será igual a i, ou seja, o número de passos esperado é igual ao número de módulos modificados inicialmente. Este é de fato o resultado esperado, já que não há possibilidade de propagação de modificações entre os módulos.

Em um sistema no qual há conexões entre os módulos, o melhor caso ocorre quando α e ϕ assumem valores muito baixo e β, valores altos, o que corresponde a um software em que os módulos são pouco dependentes uns dos outros e possuem boa coesão interna. Neste caso, o número de passos tende para i. O pior caso ocorre quando α e ϕ têm valores muito altos e β, valores próximos de 0, o que corresponde a um software fortemente conectado, com forte grau de acoplamento entre os módulos e baixo grau de coesão interna de módulos. Neste caso, o número de passos de modificações é explosivo, o que na prática significa que o problema de modificar software nesta situação é intratável.

8.6.5 Aplicações de K3B

A principal aplicação do modelo K3B é a predição da amplitude de propagação de modificações em software orientado por objetos. Todavia, ele pode ser também aplicado a outros paradigmas, bastando para isso a escolha adequada de fatores para representar α, β e ϕ.

O modelo K3B é um indicador do grau de dificuldade de realizar modificações no software avaliado. Ele pode ser, então, utilizado como indicador da necessidade de reestruturação de um software. Diante de um alto valor de K3B é recomendável investir em melhorias na estrutura do software. Deve-se atuar nos fatores que possam contribuir para isso, por exemplo, priorizando alta coesão interna de módulos e baixo acoplamento entre módulos, empregando ocultação de informação e utilizando relações de herança adequadamente. Para avaliar tais aspectos, há métricas de software. No

presente trabalho, foi realizado um estudo que identificou valores referência para um conjunto de métricas relacionadas a esses aspectos. Esses valores referência podem auxiliar na tarefa de identificar possíveis pontos de melhoria estrutural no software.

8.6.6 Limitações de K3B

K3B é um modelo, e, como tal, pretende ser uma representação próxima da realidade, porém, naturalmente, não se pode garantir que ele expresse completamente a realidade. O processo modelado por K3B é o de propagação de modificações em software. Para estabelecer tal modelo, pensou-se o software como um conjunto de módulos conectados entre si. A forma proposta para calcular os fatores ϕ e α consideram apenas as relações explícitas entre módulos. Porém, em um software, é possível que haja relações de dependência não visíveis entre módulos ou de difícil percepção. Por exemplo, há relacionamento entre módulos que utilizam um mesmo arquivo, já que eles compartilham uma mesma área de dados. Neste tipo de situação, é possível que uma modificação em um desses módulos gere impacto nos demais, ainda que um não utilize explicitamente o outro. Este tipo de situação, por exemplo, não é capturada pela forma proposta para os cálculos de ϕ e de α. Essa não é uma limitação do modelo K3B, mas da forma como as relações entre os módulos do software são representadas.

A fórmula K3B, dada pelas Equações 8.9 e 8.10, foi derivada a partir da observação do padrão de formação dos polinômios obtidos conforme descrito anteriormente. Os polinômios foram gerados para valores de n de 4 a 25. Embora a fórmula aplique-se sistematicamente a todos os polinômios encontrados, ainda não há demonstração matemática de que ela seja aplicável a todo valor de n. Contudo, não há razão para acreditar que ela não o seja. O modelo K3B foi implementado na ferramenta *Connecta*. Ele foi avaliado experimentalmente em mais de 30 versões de softwares. O resultado desta avaliação é apresentado no Capítulo 9.

8.6.7 Conclusões

Manutenção é responsável pela maior parte do custo de software, o que sinaliza que softwares estão sempre sendo modificados, seja para corrigir erros, para incluir ou modificar requisitos. A predição da amplitude de propagação de modificação em software é um instrumento de grande importância no processo de manutenção de software, pois permite ao engenheiro de software estimar o impacto da modificação no software como um todo. O presente trabalho definiu um modelo, denominada K3B, que estima o número de passos de modificações contratuais decorrentes de modificações de i

módulos em um software constituído por n módulos.

O modelo K3B é dado por uma expressão matemática em termos de cinco parâmetros: n, a quantidade de módulos do software; i, o número de módulos que serão inicialmente modificados; α, que corresponde à *taxa de contaminação* de modificações no software; β, que corresponde à *taxa de melhora* de modificações no software; e ϕ, que corresponde ao grau de conectividade do software.

O uso principal de K3B é na estimativa de propagação de modificações. Todavia, o modelo pode também ser empregado como indicador de necessidade de reestruturação de software associado a outras métricas de software. O próximo capítulo apresenta os resultados da avaliação de K3B.

Capítulo 9

Avaliação do Modelo K3B

O modelo K3B tem por objetivo estimar o número de passos de modificações a serem realizadas em um software quando um determinado número de módulos desse software sofre modificações que possam impactar nos demais. O modelo é capaz de estimar esta quantidade de passos para valores de 1 e n, onde n é o número total de módulos do software. Este capítulo apresenta uma avaliação do modelo. Esta avaliação é realizada a partir da comparação dos valores gerados por K3B com os valores observados em simulações de modificações contratuais em um conjunto de softwares orientados por objetos. Os resultados dessa avaliação mostram que os os valores gerados por K3B aproximam-se fortemente dos valores observados nas simulações.

Este capítulo está organizado da seguinte forma: a Seção 9.1 descreve a metodologia empregada na avaliação do modelo, Seção 9.2 identifica o tipo de modificação utilizada na avaliação, Seção 9.3 descreve o algoritmo da simulação realizada e a ferramenta de simulação implementada, Seção 9.4 apresenta, em linhas gerais, os conceitos sobre regressão linear utilizados nessa avaliação, Seção 9.5 apresenta os resultados da avaliação, a Seção 9.6 discute as limitações da avaliação e a Seção 9.7 traz as conclusões.

9.1 Metodologia

A avaliação de K3B em cenários reais de modificação de software demanda dados que indiquem, para cada conjunto de modificações contratuais realizadas em um software, o número de passos de modificações resultantes. No caso de software aberto, as informações sobre modificações mantidas nos repositórios em geral não possuem nível de detalhe suficiente que possa fornecer o número de passos de modificações resultante a partir de um conjunto de modificações iniciais. A obtenção de dados dessa natureza a partir de software proprietário é ainda mais difícil, pois, para isso, é necessário iden-

tificar uma organização que possua um processo de manutenção estabelecido e que mantenha dados históricos sobre as modificações realizadas com o nível de informações necessário ao levantamento dos dados para a avaliação de K3B. Além disso, software é um produto de grande valor para organizações, tanto para quem o utiliza quanto para quem o produz, o que gera grandes restrições por parte de seus proprietários em fornecê-los para estudos.

Diante da dificuldade de obtenção de dados de cenários reais de modificações, a abordagem de avaliação do modelo K3B baseia-se em simulação. Para isso, foi implementada uma ferramenta que simula determinado tipo de modificação contratual em software Java e contabiliza o número de passos de modificações decorrentes de um conjunto de modificações iniciais. A ferramenta tem como entrada: o *bytecode* do software, o tipo de modificação a ser simulada e o número de módulos (classes) que sofrerão modificações inicialmente. A ferramenta simula todas as possibilidades de modificações no padrão informado, contabilizando o número de passos de modificações para cada uma delas. Como resultado, a ferramenta informa o número médio de passos de modificações.

O método utilizado para avaliar K3B é realizado de acordo com os seguintes passos:

1. O valor de K3B é coletado para determinado software. K3B gera valores para $1 \leq i \leq n$, onde i é o número de módulos que sofrerão modificações inicialmente e n, o número total de módulos no software.

2. Para cada valor de i, é realizada a simulação de um determinado tipo de modificação contratual no software.

3. Os valores de K3B são comparados com os valores resultantes da simulação.

A hipótese investigada neste experimento é a de que K3B é um modelo que estima o número médio de passos de modificações em software orientado por objetos. Para averiguar essa hipótese, é verificada a existência de correlação linear entre os valores gerados por K3B e os valores obtidos na simulação.

Foram utilizadas 37 versões de 11 softwares abertos neste experimento. Esses programas fazem parte do conjunto utilizado no estudo sobre evolução de software realizado neste trabalho, apresentado no Capítulo 7. O estudo sobre evolução de software envolveu 129 versões provenientes de 16 programas abertos e de um programa proprietário. Para avaliação de K3B, foram selecionadas as versões que viabilizassem a realização da simulação, pois este é um processo demorado. Algumas das simulações realizadas levaram mais de 24 horas para serem concluídas.

9.2 Tipo de Modificação Avaliada

O modelo K3B estima o impacto de modificações em software orientado por objetos. A modificação considerada pelo modelo é aquela que altera o contrato de uma classe e, portanto, implica em modificações nas usuárias de tal classe. O contrato de uma classe é definido pelos serviços exportados por ela, as pré e pós condições de tais serviços. Para avaliar K3B, foi utilizada simulação de modificações contratuais em software Java.

O objetivo desta avaliação é verificar a eficiência do modelo K3B na estimativa de propagação de modificações contratuais em software. A ideia dessa propagação é que uma modificação realizada em um módulo “contamina” módulos conectados a eles, que também podem provocar “contaminação” dos módulos conectados a eles, e assim sucessivamente. A dinâmica desse processo de propagação é a mesma para qualquer tipo de modificação contratual, pois uma modificação contratual em um módulo pode causar modificações nos demais, independente da natureza da modificação. Para efeitos de simulação, então, é necessário uma situação que reflita esse processo: uma modificação em um módulo disparando modificações nos módulos conectados a ele, sucessivamente.

O tipo de modificação contratual escolhido para realizar a simulação foi a *modificação da lista de parâmetro de método*. Embora esta não seja a única modificação contratual possível em um software, ela se aplica satisfatoriamente à representação do efeito de propagação de modificações em software. Modificar a assinatura de um método implica necessariamente na modificação do contrato da classe. Tomando-se esse evento como causa de modificações nos módulos que utilizam o método modificado, obtém-se um cenário que representa a dinâmica do processo de propagação de modificações. A seguir, são discutidos os efeitos de alguns tipos de modificações em software orientado por objetos, evidenciando-se que o tipo de modificação utilizada na simulação é uma boa representante do processo de propagação de modificações.

Fowler [1999, 2010] catalogou vários tipos de refatorações que podem ser realizadas em software. Uma refatoração é uma modificação realizada no software com o objetivo de reestruturá-lo, contudo sem modificar seu comportamento. Embora uma refatoração não implique em modificação do comportamento do software, ela pode resultar em modificação de uma classe e, em alguns casos, em modificação dos contratos de classes. O catálogo de refatorações descrito por Fowler foi utilizado neste trabalho como base para identificar um conjunto de modificações contratuais possíveis em software orientado por objetos. São exemplos dessas modificações e seus respectivos impactos:

- Incluir Parâmetro (*Add Parameter*): quando um método chamado demanda mais informações do método chamador, incluem-se os parâmetros necessários na lista de parâmetros do método chamado. Neste caso, ou o próprio método chamador pode possuir localmente os dados necessários para construir o parâmetro a ser passado, ou ele pode necessitar que esse valor seja passado a ele como parâmetro também. Considerando-se esta última situação, a necessidade de modificação da lista de parâmetros pode ser propagada a partir do método que sofrerá a modificação inicial para os métodos na sequência de chamadas.

- Diminuir Hierarquia (*Collapse Hierarchy*): quando a superclasse e a subclasse não possuem diferenças significativas entre si, elas devem ser unidas em uma só classe. Desta forma, as classes que utilizavam a superclasse ou a subclasse passarão a utilizar a nova classe resultante. Esta modificação pode ter o efeito de *incluir parâmetro* na lista de parâmetros de um método, pois ao unir duas classes é possível que o construtor da nova classe demande mais parâmetros do que os construtores das classes originais.

- Extrair Classe (*Extract Class*): quando uma classe possui mais de uma responsabilidade, deve-se dividir tal classe em quantas forem necessárias de forma que cada classe resultante tenha somente uma responsabilidade. O resultado desse tipo de refatoração é que as classes usuárias da classe original precisarão ser modificadas para referenciar as novas classes. O efeito desse tipo de refatoração é inicialmente restrito às classes usuárias da classe refatorada, tendo um efeito de propagação mais simples do que o de *modificação da lista de parâmetros de métodos*.

- Parametrizar método(*Parameterize Method*): quando há mais de um método realizando os mesmos serviços que diferem entre si apenas porque manipulam valores diferentes contidos em seus corpos, deve-se criar um único método que receba tal valor por parâmetro. Esta modificação pode ter o efeito de *incluir parâmetro* na lista de parâmetros de um método, e, nesse caso, os usuários dos métodos originais precisarão elaborar o valor a ser passado como parâmetro ao novo método criado.

Outro tipo de modificação possível em uma classe é a inclusão de atributos. No caso de atributos privados, pode ser necessário que a classe passe a ter métodos *get* e *set* para manipulação dos atributos novos. No caso de atributos públicos, a única diferença é que o meio de manipular os valores dos atributos é direto. Em ambos os casos, as classes usuárias da classe modificada podem necessitar ser modificadas também a fim

de produzirem dados para os atributos novos. Nesta situação, há duas possibilidades: ou a própria classe é capaz de produzir os novos dados, ou ela pode demandar que esses dados sejam produzidos externamente e passados a ela por meio de parâmetros de seus métodos. O efeito de propagação desse tipo de modificação é, então, similar ao de *modificação da lista de parâmetros de método*.

Algumas modificações têm impacto restrito às classes usuárias das classes refatoradas. Essas modificações têm efeito similar ao de *renomear método*. Todavia, outras modificações, como *incluir parâmetro*, na pior das hipóteses, podem gerar modificações em cascata no sistema. O modelo K3B lida com o problema de estimar o impacto de propagação de modificações em software. Para fins de simulação, um único tipo de modificação é suficiente. A *modificação da lista de paraêmtros de método* é representativa neste problema e atende à avaliação do modelo. A ferramenta e o algoritmo aplicado para realizar simulação de propagação de modificações são apresentados na próxima seção.

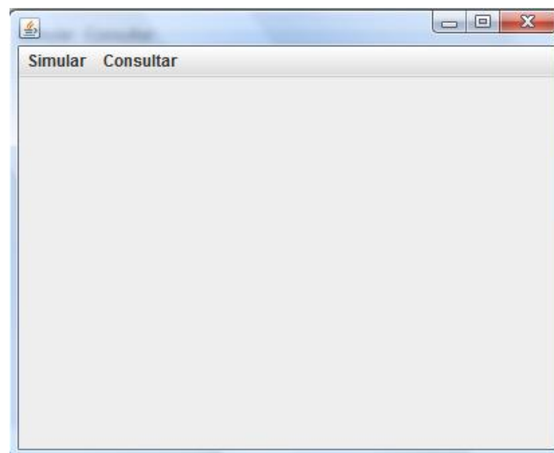


Figura 9.1. Tela Principal – ferramenta de simulação

9.3 Ferramenta de Simulação de Propagação de Modificações Contratuais

Foi construída uma ferramenta para a realização da simulação de modificações contratuais em software orientado por objetos. A ferramenta realiza a simulação de dois tipos de modificações em software Java: *renomear método* e *lista de parâmetros*. A ferramenta tem como entradas o *bytecode* do software a ser analisado e o tipo de

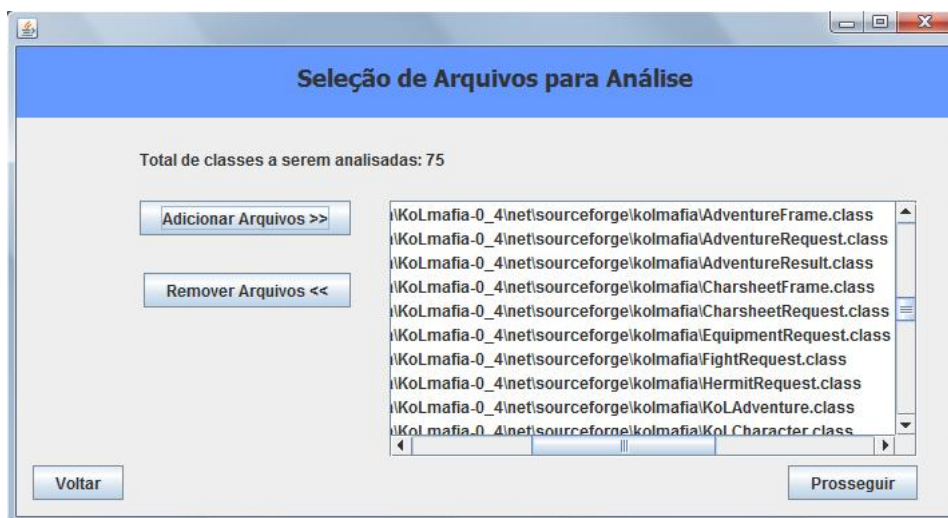


Figura 9.2. Tela Seleção de Arquivos para Análise – ferramenta de simulação

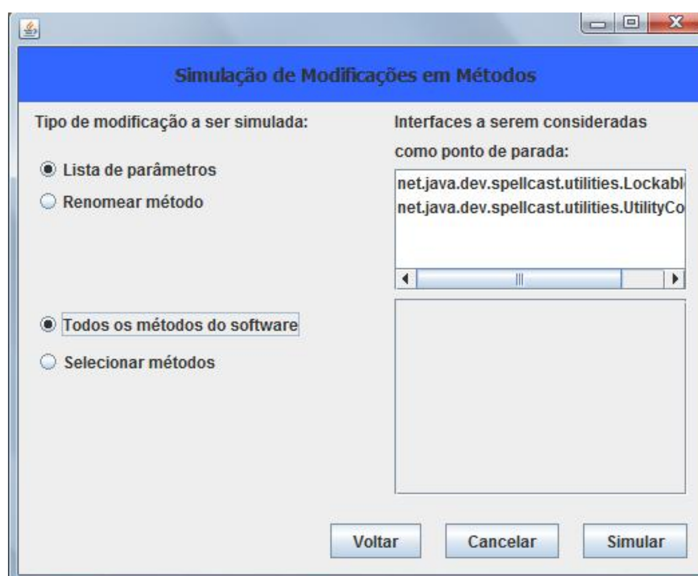


Figura 9.3. Tela Simulação de Modificações em Métodos – ferramenta de simulação

modificação a ser considerada, e gera como resultado o número médio de passos de modificações do software. Ela possui cinco interfaces de usuário principais:

1. Tela Principal: permite selecionar a funcionalidade desejada. A Figura 9.1 mostra esta interface de usuário.

2. Seleção de Arquivos para Análise: permite a seleção dos arquivos a serem analisados na simulação. O usuário deve selecionar a pasta ou diretório que contém

Figura 9.4. Tela Simulação de Modificações em Métodos - Resultado – ferramenta de simulação

os arquivos a serem analisados. A Figura 9.2 mostra esta interface de usuário.

3. Simulação de Modificações em Métodos: A Figura 9.3 mostra a interface de usuário. Nesta tela, o usuário seleciona o tipo de modificação a ser realizada (*lista de parâmetro* ou *renomear método*), e indica o conjunto de métodos que deverão ser analisados (*todos os métodos do software* ou *selecionar métodos*). Caso seja selecionada a opção *selecionar métodos*, os métodos são listados no lado direito dessa tela para que o usuário possa efetuar a seleção. Além disso, nesta tela, o usuário pode indicar as interfaces a serem consideradas como *ponto de parada*. Uma interface ponto de parada é aquela que contém métodos que o usuário assume que não poderão sofrer modificações. Este recurso é utilizado para controlar o processo de simulação. Quando um método em uma classe é a implementação de um método de uma interface de parada, um processo

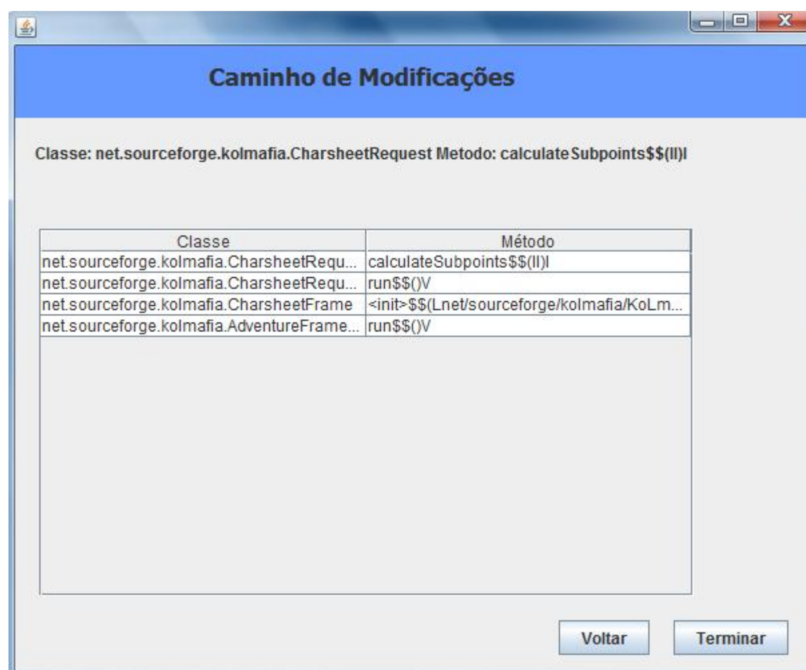


Figura 9.5. Tela Caminho de Modificações – ferramenta de simulação

de propagação de modificação é interrompido nele, ou seja, como o método foi previamente indicado como imutável pelo usuário, considera-se que a partir dele não será disparada modificação alguma.

4. Simulação de Modificações em Métodos - Resultado: nesta tela é exibido o resultado da simulação. A Figura 9.4 mostra a interface de usuário. A simulação realizada é para $i = 1$, ou seja, para o caso em que um módulo do sistema sofrerá modificação. A ferramenta simula todas as modificações possíveis no software que sejam do tipo informado. Desta forma, para cada método do sistema é calculado e exibido o respectivo número de passos de modificações. O programa exibe a média desses resultados. Caso o usuário deseje visualizar os passos de uma modificação, ele deve selecionar o método correspondente e acionar o comando *Detalhar*, que exibe a tela *Caminho de Simulação*.

5. Caminho de Modificações: esta tela exibe os passos de uma modificação, indicando os métodos para as quais a modificação foi propagada. A Figura 9.5 mostra esta interface de usuário.

9.3.1 Algoritmo de Simulação

Esta seção descreve o algoritmo da simulação de propagação de modificações contratuais do tipo *lista de parâmetro* implementado pela ferramenta.

Se um método $M2$ chama um método $M1$, e $M1$ teve sua lista de parâmetros modificada, é possível que $M2$ também tenha sua lista de parâmetros modificada. Isso ocorre porque há duas possibilidades principais de $M2$ passar a fornecer o parâmetro necessário para $M1$:

- o próprio M2 produzirá o parâmetro, por exemplo, com uma variável ou combinação de variáveis no escopo de M2;
- ou $M2$ necessitará receber como parâmetro um dado para, então, passar para $M1$.

Como não é possível definir automaticamente quando $M2$ é capaz de produzir ele mesmo o parâmetro para passar $M1$, para fins de simulação da propagação desse tipo de modificação, foi considerado o pior caso: a lista de parâmetros de $M2$ é também modificada. Considerando-se isso, a propagação desse tipo de modificação a partir de um método é contada até que se chegue a um método que não é chamado por nenhum outro no programa. Em um programa Java, a propagação é contada até que se chegue ao método *main* ou a algum método que não seja chamado por nenhum outro. Essa é a abordagem utilizada na realização dos experimentos relatados neste capítulo. Porém, para possibilitar a suavização dessa premissa pessimista em outras situações em que a ferramenta possa vir a ser utilizada, introduz-se o conceito de *ponto de parada*, que corresponde a um método, indicado pelo usuário da ferramenta, no qual o processo de uma propagação deve parar. Ou seja, considera-se que métodos indicados como *ponto de parada* pelo usuário não são modificáveis. A seleção dos métodos *ponto de parada* se dá por meio da seleção de interfaces. A ferramenta exibe a lista de interfaces que fazem parte do software e o usuário seleciona aquelas cujos métodos devem ser considerados como *ponto de parada*. Desta forma, durante o processo de simulação, quando uma modificação chegar a um método que esteja declarado em uma interface implementada pela sua classe e quando tal interface tiver sido indicada pelo usuário como *ponto de parada*, a propagação da modificação inicial é encerrada. O cálculo do número de passos de modificações na simulação é realizado de acordo com o Algoritmo 9.3.1.

Na simulação do caso em que um módulo sofre modificação inicialmente, é gerado como resultado o número de passos decorrentes de modificação para cada método do software, conforme mostrado na Figura 9.4. Para simular a propagação de modificações

```
1      M é o método que sofrerá modificação
2      visitadas é um conjunto de classes
3      pontoParada é um conjunto de interfaces selecionadas pelo usuário
4
5      visitadas = M.obterClasse();
6      passos = contapassos(M);
7
8      int contaPassos(Metodo M)
9      início
10       int p = 0; // contador de passos
11       para cada classe C que usa M faça
12          se C não é a própria classe de M então
13             se houver método X em C que usa M então
14                 p++;
15                 se C não estiver em visitadas então
16                    Insere C em visitadas
17                    para cada método X em C que usa M e que não está
18                         nas interfaces pertencentes a pontoParada
19                         implementadas por C faça
20                                p += contaPassos(X)
21                    fim para
22                  fim se
23             fim se
24           senão // se C é a própria classe de M
25             para cada método X em C que usa M e que não está
26                  nas interfaces pertencentes a pontoParada
27                  implementadas por C faça
28                         p += contaPassos(X)
29             fim para
30           fim se
31       fim para
32       retorne p
33     fim
```

Algoritmo 9.3.1: Algoritmo de simulação de propagação de modificações contratuais para i=1

quando mais de um módulo sofre modificação inicialmente, esse resultado é utilizado de acordo com o Algoritmo 9.3.2.

9.4 Coeficiente de Correlação

Esta seção apresenta em linhas gerais os conceitos envolvidos no estudo de correlação entre variáveis. Uma descrição mais detalhada sobre o assunto bem como a forma como o coeficiente de correlação é calculado são dados, por exemplo, por Freund & Simon [2000].

```
1         i é o número de módulos que sofrerão modificações inicialmente
2         P é a lista de número de passos resultante da simulação de i=1
3         somaPassosComb é a soma de passos correspondentes
4                        a um conjunto de métodos
5         somaPassos é a soma de passos das várias combinações de métodos
6         totComb é a quantidade de combinações dos métodos
7         mediaPassos é o número médio de passos
8
9         somaPassosComb = 0
10        totComb = 0
11
12        para cada combinação Comb de i métodos do software,
13                  sendo os métodos não pertencentes à mesma classe, faça
14           para cada método M em Comb faça
15                     somaPassosComb += número de passos correspondente a M
16           fim para
17           totComb++
18           somaPassos += somaPassosComb
19           somaPassosComb = 0
20        fim para
21
22        mediaPassos = somaPassos / totComb
```

Algoritmo 9.3.2: Algoritmo de simulação de propagação de modificações contratuais para i>1

O coeficiente de correlação, denotado por r, é uma medida estatística da relação linear entre duas variáveis x e y. O cálculo de r baseia-se no ajuste de uma reta de regressão $y' = mx + b$ a pares de dados (x, y). O símbolo y' é usado para distinguir entre um valor observado y e o valor correspondente na reta de regressão. Os valores de r variam de -1 a 1. Quando r resulta em valor nulo, o que corresponde a uma reta de regressão horizontal, significa que não há correlação entre x e y. Se todos os pontos (x, y) estão sobre a reta de regressão, r resulta em -1 ou 1, dependendo do coeficiente angular da reta. Quando $|r|$ é um valor próximo de 1, significa que há forte correlação entre x e y. Esta situação indica ajuste perfeito dos dados. Valores de r próximos de -1 indicam correlação inversa entre x e y, ou seja, valores altos de y correspondem a valores baixos de x. Valores de r próximos de 1 indicam correlação positiva entre essas duas variáveis. A Figura 9.6 exemplifica conjuntos de dados e suas respectivas retas de regressão [Freund & Simon, 2000].

Para avaliar K3B, é verificada a existência de correlação entre os valores gerados por K3B e os valores obtidos na simulação. A hipótese investigada nesta avaliação é que K3B contribui para estimar o valor real, ou seja, espera-se encontrar correlação

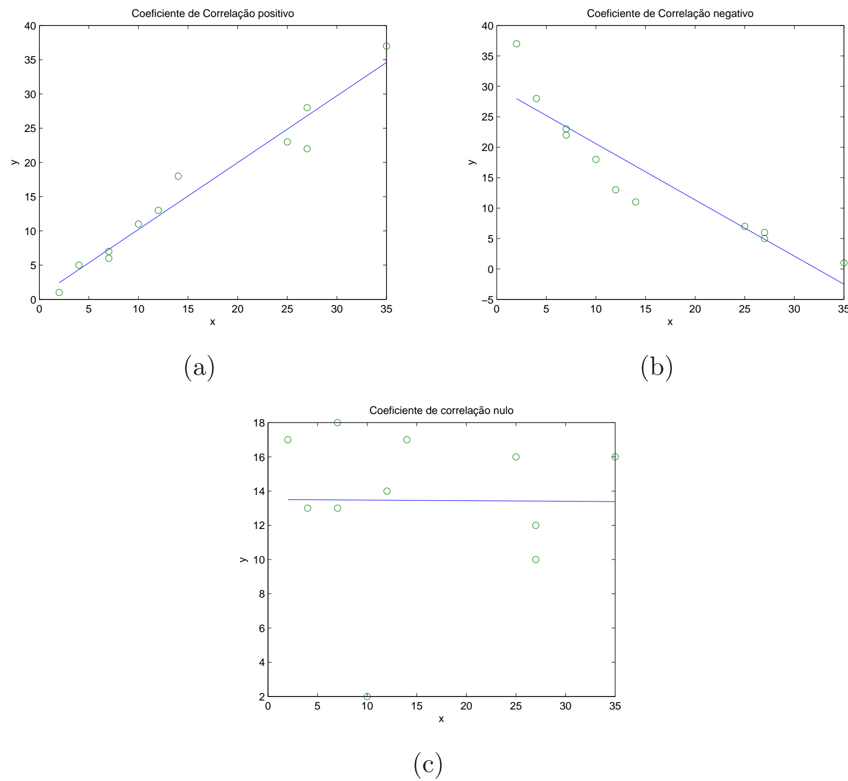


Figura 9.6. Correlação entre dados

positiva entre os valores de K3B e os valores obtidos na simulação. A análise foi realizada para cada uma das 37 versões dos softwares utilizados nesta avaliação. Para cada versão, foram coletados os valores de K3B para i e os valores de simulação correspondentes para comparação. Cada uma dessas avaliações resulta em uma reta de regressão $Simulado = m * K3B + b$.

No caso de existência de correlação positiva entre K3B e o valor simulado, e caso os valores de m e de b mostrem-se constantes ou pouco variáveis dentre essas retas, conclui-se que, independente do software, além de K3B estimar o valor observado na simulação, esse valor observado pode ser obtido diretamente a partir da multiplicação do valor de K3B por m e somado com b. Por outro lado, caso exista grande variabilidade dos valores dos coeficiente m das retas, conclui-se que, embora K3B possa estimar o valor real, a reta de ajuste entre K3B e o valor real é sensível ao software avaliado, ou seja, não é possível identificar um coeficiente m geral para obter o valor real a partir de K3B.

Com isso, além de verificar a correlação entre K3B e os valores obtidos na simulação, é preciso verificar o comportamento dos valores de m e b das retas de regressão. A outra hipótese investigada com isso é que K3B estima o valor a que se propõe e, inde-

pendente do software avaliado, o ajuste desse valor de acordo com uma única constante conhecida resulta no valor real.

9.5 Resultados

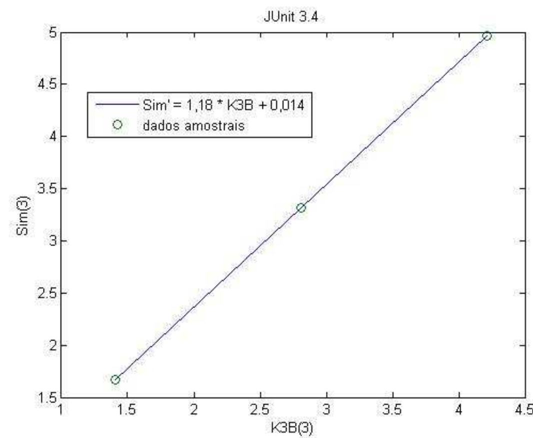
Esta seção apresenta e discute os resultados da avaliação de K3B. O modelo estima o número de passos de modificações em um software de n módulos quando i desses módulos sofrem modificações inicialmente. Por simplicidade, nesta seção é utilizado o termo $K3B(i)$ para designar o valor dado por K3B para um software quando i de seus módulos inicialmente sofrem modificação. Da mesma forma, é utilizado o termo $Sim(i)$ para designar o valor obtido na simulação de modificações em i módulos do software.

A avaliação de K3B foi limitada ao intervalo $1 \leq i \leq 3$ devido ao fato de a simulação ser um processo de alto custo quando i assume valores maiores do que 3. Algumas das simulações com $i = 3$ levaram mais de 24 horas para ser concluídas. Foram analisados 37 programas.

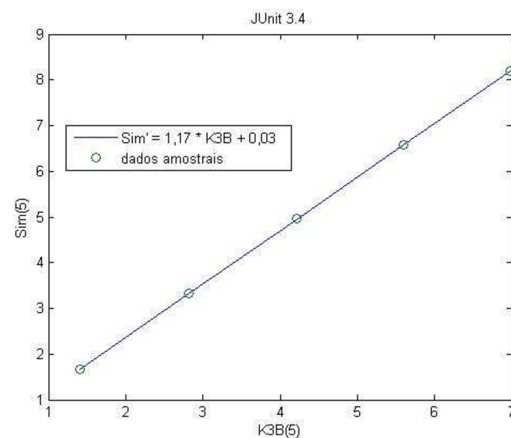
As Tabelas 9.1 e 9.2 mostram os dados da análise para $i = 3$. As colunas α e β correspondem, respectivamente, aos valores de acoplamento médio das conexões entre classes e coesão média das classes do software. A coluna ϕ é o valor do grau de conectividade do software. As colunas K3B(i) e Sim(i) são os valores de K3B e os valores obtidos na simulação, respectivamente. A coluna r fornece o valor do coeficiente de correlação entre K3B e os valores da simulação. As colunas m e b mostram os coeficientes da reta de regressão linear, $Sim' = m * K3B + b$, entre os valores de K3B e os valores da simulação.

Em três dos softwares, foi realizada análise também para $1 \leq i \leq 5$. Tal análise foi realizada somente nos casos em que o processo de simulação leva um tempo razoável para ser concluído. O mesmo foi feito para $1 \leq i \leq 4$. Os resultados dessas análises são reportados na Tabela 9.3. Observa-se que os resultados dessas análises são muito próximos daquela realizada para $1 \leq i \leq 3$. Não há diferença significativa entre os valores de r, m e b nesta análise e os respectivos valores resultantes da análise de $1 \leq i \leq 3$. Um exemplo dessa situação é ilustrado na Figura 9.7, que mostra os resultados do software JUnit, versão 3.4, para $i = 3$ e $i = 5$. Com isso, ainda que não seja possível realizar a análise para $i = 4$ e $i = 5$ no conjunto completo de softwares, é possível considerar os resultados obtidos na análise de $1 \leq i \leq 3$ como bastante representativos.

A análise dos resultados da avaliação de K3B reportados nas Tabelas 9.1 e 9.2 leva às seguintes conclusões:



(a)



(b)

Figura 9.7. Correlação entre dados da simulação e K3B, para $i = 3$ e $i = 5$ - JUnit 3.4

- Em todos os casos analisados, r tem valor praticamente igual a 1: a média dos valores de r é 0,99992, com desvio padrão 0,0005. Esse resultado mostra correlação linear entre os valores gerados por K3B e aqueles observados na simulação. Apesar do número pequeno de pontos utilizados na análise de correlação, isso indica que K3B estima os valores simulados.

- Os coeficientes m das retas de regressão têm valores próximos. O mesmo ocorre com os valores de b. As médias desses coeficientes e seus respectivos intervalos de confiança (IC) a 95% são:

  – m: possui média igual a 1,7, com desvio padrão de 0,82. O IC de m é [1,46; 1,92]. Isso significa que, com 95% de confiança, é possível afirmar que a

média de m está dentro desse intervalo, ou seja, o valor da simulação é no máximo 1,92 vezes maior do que o valor de K3B em média.

  - b: possui média igual a 0,02, com desvio padrão de 0,14. O IC de b é [-0,02; 0,06]. Isso significa que, com 95% de confiança, é possível afirmar que a média de b está dentro desse intervalo. Esse resultado mostra que os valores de b são muito próximos de zero. Com isso, o valor da simulação pode ser obtido pelo ajuste do valor dado por K3B utilizando-se apenas o fator dado por m.

- A reta média de regressão entre os valores de K3B e os valores simulados é dada pela Equação 9.1.

$$Simulado = 1,7 * K3B \tag{9.1}$$

  O significado desse resultado é que o valor de K3B pode ser usado para estimar o valor simulado, a partir da multiplicação por uma constante conhecida. Como essa constante tem um valor pequeno, indica que o valor de K3B é da mesma ordem de grandeza do valor simulado. Com isso, considerando-se que o valor simulado expressa a realidade, o modelo K3B tem como resultados valores muito próximos dos reais.

Os resultados dessa análise mostram que K3B não só estima o valor simulado, como também há uma transformação genérica direta que mapeia K3B para o valor simulado, independente do software avaliado. O processo de simulação tem custo muito alto. Por exemplo, o tempo para processar a simulação para modificação de 4 módulos no software JSCH, que possui somente 80 classes e 671 métodos, foi de 34 minutos. A simulação de modificação de 4 módulos no software Squirrel, que possui 424 classes e 1589 métodos, levou cerca de 24 horas para ser concluída. K3B é uma expressão matemática de implementação simples, capaz de fornecer a estimativa instantaneamente. A aplicação de K3B, em substituição à simulação, provê um meio eficiente para estimar o impacto de modificações em software.

9.6 Limitações da Avaliação

A avaliação de K3B limitou-se a software aberto, devido à dificuldade de obtenção de dados de softwares proprietários. Embora não haja motivos comprovados para acreditar que os resultados obtidos nesta análise não sejam aplicáveis a software proprietário, também não é possível afirmar que eles sejam.

O modelo K3B calcula o número de passos de modificações em um software de n módulos quando i desses módulos, $1 \leq i \leq n$, serão inicialmente modificados. A avaliação de K3B limitou-se a $1 \leq i \leq 5$ devido à inviabilidade de realização de simulações para valores maiores. Com isso, a análise realizada não fornece embasamento para afirmar que os resultados obtidos possam ser generalizados para valores de i superiores a cinco.

O modelo calcula a estimativa a que se propõe com base na probabilidade de uma modificação em um módulo ser propagada para os vizinhos ou não. Tais probabilidades foram calculadas em função de fatores estruturais do software, tais como acoplamento, coesão e conectividade. O processo de simulação utilizado na avaliação de K3B considera apenas o fator conectividade, uma vez que define a propagação de uma modificação com base no fato de uma classe utilizar a classe na qual uma modificação ocorre. Com isso, o processo de simulação é mais pessimista do que K3B. É esperado, então, que os valores simulados sejam um pouco mais altos do que aqueles resultantes de K3B.

9.7 Conclusões

K3B é um modelo que estima o número de passos de modificações contratuais necessários para concluir um processo de modificação iniciado em i módulos de um sistema. A avaliação de K3B foi realizada a partir da simulação de propagação de modificações contratuais em software orientado por objetos. Para isso, foi implementada um ferramenta que simula esse processo em software Java. A ferramenta simula todas as possíveis modificações de lista de parâmetro no software, gerando como resultado o número médio de passos de modificações.

A comparação dos resultados de K3B com aqueles obtidos na simulação mostrou que há forte correlação entre seus valores. Além disso, a análise dos resultados mostrou que, independente do software avaliado, é possível obter o valor simulado a partir da multiplicação do valor K3B por uma constante conhecida, que é relativamente baixa, entre 1,5 e 2. Isso evidencia que o valor que K3B gera como resultado é muito próximo do valor real.

Alta propagação de modificações traduz-se em maior dificuldade, mais tempo e mais custo para realizar manutenções no software. O ideal é que um software seja estruturado de tal forma que uma modificação qualquer nele atinja o menor número possível de módulos. Conhecer o impacto de modificações em software é importante em vários aspectos do desenvolvimento de software, por exemplo na gerência das atividades de modificações e na avaliação de software. A simulação é uma alternativa para avaliar esse impacto. Porém, é um processo caro. O modelo K3B vem contribuir nesse aspecto.

Ele estima o número de passos de modificações contratuais em software orientado por objetos. K3B é uma solução eficiente para o problema, pois realiza a estimativa a partir de uma expressão matemática cuja implementação é simples. Além disso, a expressão utilizada no cálculo de K3B é dada em função de três parâmetros, α, β e ϕ. Como isso, a obtenção de um valor indesejável de propagação de modificações é um indicador de que se deve atuar nos fatores tomados para esses parâmetros: melhorar a estrutura do software para diminuir α e ϕ ou para aumentar β.

Tabela 9.1. Comparação entre K3B e dados de simulação - Valores para $1 \leq i \leq 3$

Software	Versão	α	β	ϕ	# Classes	# Métodos	K3B(1)	K3B(2)	K3B(3)	Sim(1)	Sim(2)	Sim(3)	r	m	b
Hibernate	3.0	0,083	0,584	0,003	956	10358	2,33	4,66	6,99	2,67	5,34	8,01	0,99999997	1,15	0
	3.1	0,085	0,584	0,003	1118	13683	2,38	4,76	7,14	2,73	5,47	8,01	0,99978741	1,11	0,12
Jasper Reports	0.4.0	0,095	0,720	0,009	242	1790	1,75	3,49	5,23	2,73	5,47	8,20	0,99999998	1,57	-0,02
	1.0.0	0,083	0,639	0,004	574	6094	1,88	3,77	5,65	2,79	5,58	8,38	1,00000000	1,48	0,00
	2.0.0	0,081	0,617	0,002	1104	13196	1,97	3,94	5,90	2,82	5,65	8,79	0,99953185	1,52	-0,23
Java Groups	1.0	0,109	0,660	0,007	415	3331	3,01	6,01	8,99	3,73	7,44	11,14	0,99999996	1,24	0,00
	2.1.1	0,105	0,699	0,004	696	8638	2,46	4,92	7,37	4,51	9,02	12,37	0,99633703	1,60	0,76
JML	10a1	0,085	0,535	0,017	171	1167	2,61	5,20	7,77	2,33	4,66	6,98	0,99999954	0,90	-0,02
	10a2	0,079	0,580	0,015	186	1652	2,18	4,35	6,51	2,01	4,01	6,02	0,99999818	0,93	-0,01
	10b1	0,081	0,582	0,015	203	1808	2,42	4,82	7,21	2,13	4,27	6,41	0,99999960	0,89	-0,03
	10b3	0,081	0,576	0,014	218	1956	2,50	4,99	7,47	2,16	4,33	6,49	0,99999949	0,87	-0,02
	10b4	0,083	0,592	0,012	270	2319	2,74	5,48	8,20	2,59	5,18	7,77	0,99999889	0,95	-0,01
JSCH	0.1.14	0,111	0,747	0,032	80	671	2,17	4,32	6,44	2,30	4,60	6,89	0,99999908	1,08	-0,04
	0.1.20	0,108	0,668	0,028	83	740	2,17	4,31	6,43	2,26	4,51	6,75	0,99999839	1,05	-0,03
	0.1.26	0,104	0,681	0,024	94	823	2,03	4,05	6,04	2,15	4,29	6,43	0,99999829	1,07	-0,02
	0.1.34	0,105	0,690	0,023	109	941	2,17	4,32	6,45	2,26	4,53	6,80	0,99999901	1,06	-0,05
	0.1.42	0,103	0,659	0,020	117	1012	2,11	4,21	6,29	2,25	4,50	6,75	0,99999734	1,08	-0,02
JUnit	3.4	0,078	0,815	0,023	78	315	1,41	2,81	4,21	1,67	3,32	4,96	0,99999968	1,18	0,01
	3.8	0,078	0,803	0,018	101	595	1,41	2,81	4,21	2,35	4,70	7,04	0,99999996	1,67	-0,01
	4.0	0,073	0,763	0,020	92	924	1,42	2,82	4,23	2,27	4,53	6,80	0,99999944	1,61	-0,02
	4.5	0,075	0,742	0,010	188	1516	1,47	2,94	4,40	2,58	5,15	7,73	0,99999946	1,76	-0,01
	4.8.1	0,075	0,742	0,008	230	1694	1,46	2,92	4,38	2,68	5,35	8,02	0,99999968	1,83	0,01

Tabela 9.2. Comparação entre K3B e dados de simulação - Valores para $1 \leq i \leq 3$

Software	**Versão**	α	β	ϕ	**# Classes**	**# Métodos**	**K3B(1)**	**K3B(2)**	**K3B(3)**	**Sim(1)**	**Sim(2)**	**Sim(3)**	**r**	**m**	**b**
KolMafia	0.2	0,091	0,785	0,056	39	239	1,65	3,27	4,88	2,83	5,69	8,58	0,99997862	1,78	-0,12
	1.0	0,093	0,760	0,025	143	634	2,47	4,93	7,37	5,53	11,02	16,46	0,99999996	2,23	0,01
Logsim	2.0.0	0,092	0,632	0,004	908	6738	3,17	6,34	9,51	3,86	7,72	11,58	0,99999997	1,22	0,00
	2.1.0	0,092	0,630	0,004	993	7508	3,32	6,64	9,95	3,90	7,79	11,68	0,99999980	1,17	0,00
Movie Manager	1.6beta	0,080	0,767	0,037	64	364	1,64	3,26	4,87	2,93	5,75	8,46	0,99996704	1,71	0,13
	1.7	0,080	0,832	0,032	73	418	1,56	3,12	4,66	2,88	5,64	8,30	0,99996424	1,75	0,16
	2.0	0,088	0,687	0,004	517	8097	1,66	3,31	4,96	6,53	13,04	19,53	0,99999959	3,94	0,00
	2.8	0,095	0,830	0,007	458	3495	2,23	4,46	6,69	8,43	16,77	25,02	0,99999604	3,72	0,14
	2.9.13	0,097	0,834	0,005	608	4655	2,27	4,54	6,81	7,26	14,48	21,64	0,99999818	3,17	0,07
Phex	0.6	0,098	0,648	0,007	393	2232	2,49	4,98	7,45	8,71	17,41	26,10	0,99999996	3,50	-0,02
	2.8.0	0,087	0,705	0,003	1205	9872	2,42	4,83	7,24	4,74	9,48	14,21	1,00000000	1,97	-0,01
	3.4.2	0,087	0,703	0,003	1352	14078	2,75	5,50	8,25	4,16	8,33	12,49	1,00000000	1,52	-0,01
Squirrel	1.0	0,072	0,785	0,004	424	1589	1,37	2,73	4,10	4,06	8,11	12,16	1,00000000	2,97	-0,01
	2.0	0,075	0,746	0,003	729	5103	1,46	2,92	4,37	3,33	6,67	10,00	0,99999997	2,29	-0,01
	2.6	0,079	0,799	0,002	940	6335	1,50	3,00	4,50	3,23	6,46	9,69	1,00000000	2,16	0,00

Tabela 9.3. Comparação entre K3B e dados de simulação para $i = 4$ e $i = 5$

Software	Versão	K3B(1)	K3B(2)	K3B(3)	K3B(4)	Sim(1)	Sim(2)	Sim(3)	Sim(4)	r	m	b
JML	10a1	2,61	5,20	7,77	10,33	2,33	4,66	6,98	9,29	0,999998535	0,90	-0,03
JSCH	0.1.14	2,17	4,32	6,44	8,54	2,30	4,60	6,89	9,17	0,999997453	1,08	-0,05
Kolmafia	1.0	2,47	4,93	7,37	9,79	5,53	11,02	16,46	21,85	0,999999882	2,23	0,02
Squirrel	1.0	1,37	2,73	4,10	5,46	4,06	8,11	12,16	16,20	0,999999996	2,97	-0,004

Software	Versão	K3B(1)	K3B(2)	K3B(3)	K3B(4)	K3B(5)	Sim(1)	Sim(2)	Sim(3)	Sim(4)	Sim(5)	r	m	b
JUnit	3.4	1,41	2,81	4,21	5,60	6,99	1,67	3,32	4,96	6,58	8,19	0,999997486	1,17	0,03
Kolmafia	0.2	1,65	3,27	4,88	6,46	8,02	2,83	5,69	8,58	11,49	14,41	0,999905934	1,82	-0,23
MovieManager	1.6beta	1,64	3,26	4,87	6,46	8,05	2,93	5,75	8,46	11,09	13,66	0,999905934	1,67	0,25

Capítulo 10

Conclusão

A atividade de manutenção é responsável pela maior parte do custo total de um sistema. A fim de reduzir esse custo, é essencial que durante o desenvolvimento do software ocorra o planejamento e o preparo para a sua manutenção. Um dos recursos que podem contribuir diretamente para isso é a medição da manutenibilidade de software. A medição, a avaliação e o controle da manutenibilidade do software são importantes durante as fases de desenvolvimento e operação para que problemas futuros sejam evitados. Além disso, a avaliação da dificuldade de manutenção de software pode auxiliar gerentes na alocação de recursos apropriados para a realização das tarefas de modificações de software. Muitos trabalhos têm essa questão como objeto de estudo. Apesar do grande investimento em estudos para identificar uma maneira de medir a manutenibilidade de software, ainda não há uma solução satisfatória para esse problema.

O trabalho apresentado nesta tese foi realizado como o objetivo de contribuir com uma solução para o problema de predição de impacto de modificações em software. Nesta tese, foi definido um modelo de predição de amplitude da propagação de modificações contratuais em software orientado por objetos. Modificação contratual refere-se à modificação ocorrida dentro de uma classe que altere o seu contrato e que possa gerar impacto em outras classes do software. O modelo, denominado K3B, estima o número de médio de passos *E(n,i)* de modificações para que um software com n módulos alcance estabilidade quando for necessário modificar i de seus módulos. O modelo foi definido com base em conceitos de probabilidades e em fatores que refletem a natureza das relações de classes em software orientado por objetos.

O modelo K3B é dado por uma expressão matemática em termos de cinco parâmetros: n, i, α, β e ϕ. O parâmetro n é o número de módulos do software, e i é o número de módulos que serão inicialmente modificados nele. O parâmetro ϕ representa o percentual de conexões entre classes do software. O parâmetro α representa um fator que contribui para a propagação de modificações em software. Sugere-se adotar

o grau de acoplamento médio entre as classes para ser utilizado como esse fator. O parâmetro β representa um fator que contribui para que modificações em software não sejam propagadas. Sugere-se adotar o grau de coesão médio das classes do software para ser utilizado como esse fator. Decorre da expressão identificada para K3B que o número de passos de modificações em software muito conectado, com forte acoplamento e baixa coesão é explosivo. Isso indica que realizar modificações em software com essa característica é um problema intratável. Por outro lado, mesmo em softwares de grande porte, quando o acoplamento e a coesão têm bom nível, a propagação de modificações é controlável. Essas são premissas que intuitivamente já se podia aceitar como sendo verdadeiras. O modelo K3B, porém, expressa matematicamente as relações entre esses fatores, fornecendo uma estimativa do impacto de modificações em software.

O modelo K3B foi avaliado por meio de simulações de modificações realizadas em vários softwares abertos Java. A comparação dos resultados de K3B com aqueles obtidos na simulação mostra que há forte correlação entre seus valores. A análise dos resultados mostrou também que é possível obter o valor simulado a partir da multiplicação do valor K3B por uma constante conhecida, que é relativamente baixa, entre 1,5 e 2. Isso evidencia que o valor que K3B gera como resultado é muito próximo do valor real.

O modelo pode ser aplicado para auxiliar gerentes na alocação de recursos apropriados nas atividades de modificações, bem como para avaliar ou comparar softwares sob o ponto de vista de manutenibilidade. Se aplicado na fase de projeto, pode auxiliar na predição de dificuldades de modificação do software antes que o software seja implementado.

Alta propagação de modificações reflete-se em maior dificuldade para realizar modificações e, consequentemente, em maior custo de manutenção. O ideal é que um software seja construído de forma que modificações nele gerem o menor impacto possível. O modelo K3B fornece um indicador do grau de dificuldade de se realizar modificações no software, pois quanto maior a quantidade de passos de modificações, possivelmente mais difícil será concluir a tarefa de modificação do software. O valor de K3B pode ser usado como orientação para a necessidade de reestruturar o software. Diante de um valor indesejável de K3B, é preciso, então, atuar nos possíveis aspectos que contribuem para isso. Os fatores principais são aqueles usados para α e β, pois diminuindo-se α e aumentando-se β, diminui-se o valor resultante de K3B.

A identificação de necessidades de reestruturação pode ser auxiliada com o uso de métricas de software. Nesta tese, foi realizado um estudo que identificou os valores referência de seis métricas de software que medem aspectos correlatos à manutenibilidade, tais como coesão e profundidade da árvore de herança. Foi definida também

uma métrica de coesão de classe denominada Coesão de Responsabilidade (COR). Esta métrica resulta em valores ideais para serem aplicados ao parâmetro β de K3B: valores entre 0 e 1, sendo que valores próximos de 0 representam baixa coesão e valores próximos de 1 representam alta coesão. Os valores gerados por COR fornecem uma informação mais simples e direta sobre a estrutura da classe do que os valores gerados por outras métricas similares já propostas. Por exemplo, se a métrica resulta em 1, significa que há um único conjunto de métodos similares na classe; o valor 0,5, indica que há dois conjuntos, 0,25 indica que há quatro conjuntos, e assim por diante.

Nesta tese também foi conduzido um estudo sobre a estrutura real dos software orientado por objetos e sobre como essas estruturas evoluem. Os resultado desse estudo mostram que: o diâmetro desse tipo de rede é curto; a densidade da rede de software tende a diminuir à medida que o software cresce; classes com alto grau de entrada tendem a manter essa propriedade ao longo da vida do software; tais classes são instáveis, têm coesão degradada ao longo do tempo, crescem em número de métodos públicos e em número de atributos públicos. Como a densidade da rede tende a diminuir e as classes com alto grau de entrada tendem a ter graus de entrada ainda maiores, é possível inferir que a prática comum é inserir os novos requisitos em tais classes em vez de refatorar o sistema. Esse estudo identificou a figura da estrutura macroscópica das redes de software, que foi denominada *little house*. Esta estrutura macroscópica revela importantes propriedades dessas redes, por exemplo a existência de um núcleo de classes que são fortemente conectadas entre si. A presença de um componente fortemente conectado enfatiza a necessidade de abordagem sistemática para as tarefas de manutenção e testes de software, principalmente nas classes deste componente, pois uma modificação em uma classe desse componente pode afetar amplamente outras classes do software. O estudo mostrou como software é estruturado e evolui na realidade, evidenciando que o problema de realizar e gerenciar modificações em software tende a se tornar cada vez mais crítico à medida que o software evolui. Os resultados desse estudo empírico fornecem embasamento e motivação para a definição de modelos como K3B, que permitem estimar e planejar as tarefas de modificações em software.

Os resultados obtidos nesta tese geraram as seguintes contribuições principais:

1. Definição e avaliação de K3B
2. Definição da métrica COR para avaliação de coesão interna de classes
3. Identificação de valores referência para métricas de software orientado por objetos
4. Identificação de propriedades da estrutura de softwares e sua evolução

5. Identificação da figura macroscópica da estrutura de software, denominada *Little house*

6. Evolução da ferramenta *Connecta* [Ferreira, 2006]

7. Implementação de uma ferramenta de simulação de propagação de modificações em software

Além dessas contribuições, os resultados das pesquisas realizadas nesta tese geraram os seguintes artigos:

- *Valores Referência de Métricas de Software Orientado por Objetos.* Este artigo foi apresentado no Simpósio Brasileiro de Engenharia de Software em 2009. Ele contém parte dos resultados do estudo descrito no Capítulo 5 e foi premiado como um dos cinco melhores artigos do simpósio.

- *Identifying Thresholds for Object-Oriented Software Metrics.* Este artigo contém o estudo descrito no Capítulo 5 na íntegra. Ele foi submetido a um periódico internacional da área e está em fase final de revisão.

- *Métrica de Coesão de Responsabilidade.* Este artigo contém os resultados do estudo descrito no Capítulo 6. Ele será submetido a um congresso da área.

- *Software Evolution Characterization.* Este artigo foi submetido a um periódico da área. Ele contém o estudo descrito no Capítulo 7.

- *K3B - A Predicting Model for Propagation of Contractual Modifications in Software.* Este artigo abrange o conteúdo dos Capítulos 8 e 9. Ele será submetido a um periódico da área.

Como trabalhos futuros, identificam-se principalmente:

- A criação de um ambiente integrado que, dentre outros aspectos: permita o cadastro de requisitos do software e as respectivas classes que implementam tais requisitos; permita que o desenvolvedor indique os requisitos a serem modificados e, a partir daí, a ferramenta identifica quais classes devem ser modificadas e calcule o valor correspondente de K3B; e forneça ferramentas de avaliação do software por meio de métricas, com indicação dos seus respectivos valores referência.

- Neste trabalho, foi identificada uma figura macroscópica das redes de software, denominada *little house*. A figura fornece uma visão de como classes em um software organizam-se. É preciso explorar as implicações dessa organização nos seguintes aspectos principais: identificar a existência de características comuns das classes de um componente de *little house*; investigar a correlação entre tendência à modificação e o fato de uma classe pertencer a um determinado componente; observar como a figura evolui ao longo do tempo; aplicar a figura como um recurso de visualização de software.

- Aplicar o modelo K3B para predição de amplitude de modificações a partir de diagramas da UML. Isso poderia viabilizar a predição de manutenibilidade de software ainda em fases iniciais do ciclo de vida do software.

- Implementar o modelo K3B para avaliar softwares desenvolvidos em outras linguagens de programação.

- Demonstrar formalmente a fórmula identificada para K3B. A fórmula foi obtida a partir da observação do padrão de formação dos resultados obtidos para softwares com 4 a 25 módulos. Entretanto, a demonstração formal de que a fórmula seja válida para qualquer número de módulos não foi obtida neste trabalho.

- Avaliar o comportamento de K3B em cenários reais de modificações de software. O modelo K3B foi avaliado neste trabalho a partir da comparação de seus valores com aqueles obtidos por meio de simulação de modificações em software Java. Embora os resultados dessa avaliação tenha mostrado que K3B estima satisfatoriamente o valor simulado, é relevante a realização de experimentos que avaliem K3B em cenários reais.

Referências Bibliográficas

Abreu, F. B. & Carapuça, R. (1994). Object-oriented software engineering: Measuring and controlling the development process. In *Proceedings of 4th Int. Conf. of Software Quality*, McLean, Estados Unidos.

Abreu, F. B.; Ochoa, L. & Goulo, M. (1997). The goodly design language for mood metrics collection. In *ISEG/INESC ECOOP Workshop*, Portugal.

Aho, A.; Lam, M. S.; Sethi, R. & Ullman, J. D. (2008). *Compiladores - Princípios, técnicas e ferramentas*. Addison Wesley, São Paulo, 2 edição.

Al-Dallal, J. (2009). Software similarity-based functional cohesion metric. *IET Software*, 3(1):46–57.

Ash, D.; Alderete, J.; Oman, P. W. & Lowther, B. (1994). Using software maintainability models to track code health. In *Proceedings of the International Conference on Software Maintenance*, ICSM '94, pp. 154–160, Washington, DC, Estados Unidos. IEEE Computer Society.

Baxter, G.; Frean, M.; Noble, J.; Rickerby, M.; Smith, H.; Visser, M.; Melton, H. & Tempero, E. (2006). Understanding the shape of java software. In *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, pp. 397–412, New York, NY, Estados Unidos. ACM.

Bertrán, I. M. (2009). *Avaliação da Qualidade de Software com Base em Modelos UML*. Dissertação de Mestrado. Pontifícia Universidade Católica do Rio de Janeiro., Rio de Janeiro, Brasil.

Boehm, B. W. (1981). *Software Engineering Economics*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edição.

Briand, L. C.; Daly, J. W. & Wüst, J. (1998). A unified framework for cohesion measurement in object-oriented systems. *Empirical Softw. Eng.*, 3(1):65–117.

Broder, A.; Kumar, R.; Maghoul, F.; Raghavan, P.; Rajagopalan, S.; Stata, R.; Tomkins, A. & Wiener, J. (2000). Graph structure in the web. In *WWW9 Conference*, pp. 309–320.

Chaumun, M. A.; Kabaili, H.; Keller, R. K. & Lustman, F. (1999). A change impact model for changeability assessment in object-oriented software systems. In *Proceedings of the Third European Conference on Software Maintenance and Reengineering*, pp. 130–139, Washington, DC, Estados Unidos. IEEE Computer Society.

Chhabra, J. K. & Aggarwal, K. K. (2006). Measurement of intra-class & inter-class weakness for object-oriented software. In *Proceedings of the Third International Conference on Information Technology: New Generations*, pp. 155–160, Washington, DC, Estados Unidos. IEEE Computer Society.

Chidamber, S. R. & Kemerer, C. F. (1994). A metrics suite for object oriented design. *IEEE Trans. Softw. Eng.*, 20:476–493.

Counsell, S.; Swift, S. & Crampton, J. (2006). The interpretation and utility of three cohesion metrics for object-oriented design. *ACM Trans. Softw. Eng. Methodol.*, 15:123–149.

Counsell, S.; Swift, S. & Tucker, A. (2005). Object-oriented cohesion as a surrogate of software comprehension: an empirical study. In *Proceedings of the Fifth IEEE International Workshop on Source Code Analysis and Manipulation*, pp. 161–172, Washington, DC, Estados Unidos. IEEE Computer Society.

Czibula, I. G. & Czibula, G. (2008). Clustering based automatic refactorings identification. In *Proceedings of the 2008 10th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing*, pp. 253–256, Washington, DC, Estados Unidos. IEEE Computer Society.

Daly, J.; Brooks, A.; Miller, J.; Roper, M. & Wood, M. (1996). An empirical study evaluating depth of inheritance on the maintainability of object-oriented software. *Empirical Software Engineering*, 1:109–132.

Deissenboeck, F.; Wagner, S.; Pizka, M.; Teuchert, S. & Girard, J. (2007). An activity-based quality model for maintainability. In *IEEE International Conference on Software Maintenance, 2007. ICSM 2007.*, pp. 184–193.

DependencyFinder (2007). *Dependency Finder*. Disponível em http://depfind.sourceforge.net/. Acesso em Abril de 2007.

Fenton, N. E. & Neil, M. (2000). Software metrics: roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, pp. 357–370, New York, NY, Estados Unidos. ACM.

Ferreira, K. A. M. (2006). *Avaliação de Conectividade em Sistemas Orientados por Objetos*. Dissertação de Mestrado - DCC/UFMG., Belo Horizonte, Brasil.

Ferreira, K. A. M.; Bigonha, M. A. S. & Bigonha, R. S. (2008). Reestruturação de software dirigida por conectividade para redução de custo de manutenção. *Revista de Informática Teórica e Aplicada*, 15(2):155–179.

Filho, W. P. P. (2009). *Engenharia de Software - Fundamentos, Métodos e Padrões*. LTC, Rio de Janeiro, Brasil., 3 edição.

Fowler, M. (1999). *Refactoring - Improving the Design of Existing Code*. Addison Wesley.

Fowler, M. (2010). *Refactorings*. http://refactoring.com/catalog/index.html. Acesso em Dezembro de 2010.

Freund, J. E. & Simon, G. A. (2000). *Estatística Aplicada - Economia, Administração e Contabilidade*. Bookman, Porto Alegre, 9 edição.

Gamma, E.; Helm, R.; Johnson, R. & Vlissides, J. (2000). *Padrões de Projeto - Soluções Reutilizáveis de Software Orientado a Objetos*. Bookman, Porto Alegre.

Gilb, T. (1977). *Software Metrics*. Winthrop Publishers, Estados Unidos.

Godfrey, M. & Tu, Q. (2001). Growth, evolution, and structural change in open source software. In *Proc. of the IWPSE*, pp. 103–106, Viena, Áustria.

Grinstead, C. M. & Snell., J. L. (1991). *Introduction to Probability*. America Mathematical Society.

Griswold, W. G.; Shonle, M.; Sullivan, K.; Song, Y.; ant Yuanfang Cai, N. T. & Rajan, H. (2006). Modular software design with crosscuting interfaces. 23(1):51–60.

Grosser, D.; Sahraoui, H. A. & Valtchev, P. (2003). An analogy-based approach for predicting design stability of java classes. In *Ninth International Software Metrics Symposium*, pp. 252–262.

Gyimothy, T.; Ferenc, R. & Siket, I. (2005). Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 31:897–910.

Heitlager, I.; Kuipers, T. & Visser, J. (2007). A practical model for measuring maintainability. In *Proceedings of the 6th International Conference on Quality of Information and Communications Technology*, pp. 30–39, Washington, DC, Estados Unidos. IEEE Computer Society.

Herraiz, I.; Robles, G. & Gonzalez-Barahon, J. M. (2006). Comparison between slocs and number of files as size metrics for software evolution analysis. In *CSMR '06: Proceedings of the Conference on Software Maintenance and Reengineering*, pp. 206–213, Washington, DC, Estados Unidos. IEEE Computer Society.

Hitz, M. & Montazeri, B. (1995). Measuring coupling and cohesion in object-oriented systems. In *Int. Symposium on Applied Corporate Computing*, Monterrey, México.

Hordijk, W. & Wieringa, R. (2005). Surveying the factors that influence maintainability: research design. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering ESEC/FSE-13*, volume 30, pp. 385–388.

Humphrey, W. S. (1995). *A Discipline for Software Engineering*. Addison Wesley, 6 edição.

Israeli, A. & Feitelson, D. G. (2010). The linux kernel as a case study in software evolution. *The Journal of Systems and Software*, 83(3):485–501.

Jancke, S. (2010). *Smell Detection in Context*. Diploma thesis. University of Bonn. Bonn, Germany., http://dirkriehle.com/computer-science/research/dissertation/.

JDepend (2007). *JDepend*. Disponível em http://www.clarkware.com/software/ JDepend.html. Acesso em Abril de 2007.

Jenkins, S. & Kirk, S. R. (2007). Software architecture graphs as complex networks: A novel partitioning scheme to measure stability and evolution. *Information Sciences: an International Journal*, 177(12):2587–2601.

Jing, L.; Keqing, H.; Yutao, M. & Rong, P. (2006). Scale free in software metrics. In *COMPSAC '06: Proceedings of the 30th Annual International Computer Software and Applications Conference*, pp. 229–235, Washington, DC, Estados Unidos. IEEE Computer Society.

Kabaili, H.; Keller, R. K. & Lustman, F. (2001). Cohesion as changeability indicator in object-oriented systems. In *Proceedings of the Fifth European Conference on*

Software Maintenance and Reengineering, CSMR '01, pp. 39–, Washington, DC, Estados Unidos. IEEE Computer Society.

Kan, S. (2003). *Metrics and Models in Software Quality Engineering.* Addison-Wesley, 2 edição.

Kessentini, M.; Vaucher, S. & Sahraoui, H. (2010). Deviance from perfection is a better criterion than closeness to evil when identifying risky code. In *Proceedings of the IEEE/ACM International conference on Automated Software Engineering*, ASE '10, pp. 113–122, New York, NY, Estados Unidos. ACM.

Kiczales, G. & Mezini, M. (2005). Aspect-oriented programming and modular reasoning. In *ICSE'05*, St. Louis, Missouri, Estados Unidos.

Kitchenham, B. (2009). What's up with software metrics? - a preliminary mapping study. *The Journal of Systems and Software*, 83:37–51.

Koch, S. (2007). Software evolution in open source projects - a large-scale investigation. *J. Softw. Maint. Evol.*, 19(6):361–382.

Krakatau (2006). *Krakatau Essencial Metrics.* Disponível em http://www.powersoftware.com/. Acesso em Maio de 2006.

Kulesza, U.; Sant'Anna, C.; Garcia, A.; Coelho, R.; von Staa, A. & Lucena, C. (2006). Quantifying the effects of aspect-oriented programming: A maintenance study. In *Proceedings of the 22nd IEEE International Conference on Software Maintenance*, pp. 223–233, Washington, DC, Estados Unidos. IEEE Computer Society.

Lanza, M. & Marinescu, R. (2006). *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems.* Springer-Verlag, Alemanha.

Lee, Y.; Yang, J. & Chang, K. H. (2007). Metrics and evolution in open source software. In *QSIC '07: Proceedings of the Seventh International Conference on Quality Software*, pp. 191–197, Washington, DC, Estados Unidos. IEEE Computer Society.

Lehman, M. M.; Ramil, J. F.; Wernick, P. D.; Perry, D. E. & Turski, W. M. (1997). Metrics and laws of software evolution - the nineties view. In *Proc. of the Fourth Intl. Software Metrics Symposium (Metrics'97).*

Leskovec, J.; Kleinberg, J. & Faloutsos, C. (2007). Graph evolution: Densification and shrinking diameters. *ACM Trans. Knowl. Discov. Data*, 1.

Li, L.; Zhang, L.; Lu, L. & Fan, Z. (2010). Assessing object-oriented software systems based on change impact simulation. *International Conference on Computer and Information Technology*, 0:1364–1369.

Li, P. L.; Herbsleb, J.; Shaw, M. & Robinson, B. (2006). Experiences and results from initiating field defect prediction and product test prioritization efforts at abb inc. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pp. 413–422, New York, NY, Estados Unidos. ACM.

Li, W. & Henry, S. (1993). Maintenance metrics for object oriented paradigm. In *Proc. First Int'l Software Metrics Symp.*, pp. 52–60.

Lincke, R.; Lundberg, J. & Löwe, W. (2008). Comparing software metrics tools. In *ISSTA '08: Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pp. 131–142, New York, NY, Estados Unidos. ACM.

Louridas, P.; Spinellis, D. & Vlachos, V. (2008). Power laws in software. *ACM Trans. Softw. Eng. Methodol.*, 18:2:1–2:26.

Marcus, A. & Poshyvanyk, D. (2005). The conceptual cohesion of classes. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pp. 133–142, Washington, DC, Estados Unidos. IEEE Computer Society.

Martin, R. (1994). *OO Design Quality Metrics - An analysis of dependencies.* http://www.objectmentor.com/resources/articles/oodmetrc.pdf. Acesso de Julho de 2009.

Martin, R. (2002). *The Single Responsibility Principle.*

Mathwave (2010). *EasyFit.* http://www.mathwave.com/products/easyfit.html.

Mens, T.; Fernández-Ramil, J. & Degrandsart, S. (2008). The evolution of eclipse. In *Proc. 24th Int'l Conf. on Software Maintenance*, pp. 386–395.

Metrics (2007). *Metrics.* Disponível em http://www.metrics.sourceforge.net. Acesso em Outubro de 2007.

Meyer, B. (1997). *Object-oriented software construction.* Prentice Hall International Series, Estados Unidos, 2 edição.

Mirarab, S.; Hassouna, A. & Tahvildari, L. (2007). Using bayesian belief networks to predict change propagation in software systems. *International Conference on Program Comprehension*, 0:177–188.

Mäkelä, S. & Leppänen, V. (2007). Cliente based object-oriented cohesion metrics. In *IEEE 31st Annual International Computer Software and Applications Conference*, pp. 743–748, Beijing.

Munro, M. J. (2005). Product metrics for automatic identification of "bad smell"design problems in java source-code. In *Proceedings of the 11th IEEE International Software Metrics Symposium*, Washington, DC, Estados Unidos. IEEE Computer Society.

Myers, G. J. (1975). *Reliable software through composite design.* Petrocelli/Charter, Nova York, 2 edição.

Nagappan, N.; Ball, T. & Zeller, A. (2006). Mining metrics to predict component failures. In *ICSE'06*, pp. 452–461.

Newman, M. E. J. (2003). The structure and function of complex networks. In *SIAM Reviews*, volume 45, pp. 167–256.

ObjectDetail (2006). *Object Detail.* Disponível em http://www.obsoft.com/Product/DetailPaper.html. Acesso em Maio de 2006.

Olbrich, S.; Cruzes, D. S.; Basili, V. & Zazworka, N. (2009). The evolution and impact of code smells: A case study of two open source systems. In *ESEM '09: Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, pp. 390–400, Washington, DC, Estados Unidos. IEEE Computer Society.

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

Pearse, T. & Oman, P. (1995). Maintainability measurements on industrial source code maintenance activities. In *Proceedings of the International Conference on Software Maintenance*, ICSM'95, pp. 295–305, Washington, DC, Estados Unidos. IEEE Computer Society.

Petrov, V. & Mordecki., E. (2003). *Teoría de Probabilidades.* Matematnka, Editorial URSS, Moscou.

Pfleeger, S. L. (1998). *Software engineering theory and practice*. Prentice-Hall, Upper Saddle River.

Pizka, M. (2004). Straightening spaghetti-code with refactoring? In *Proceedings of the Int. Conf. on Software Engineering Research and Practice - SERP*, pp. 846–852, Las Vegas, Estados Unidos.

Potanin, A.; Noble, J.; Frean, M. & Biddle, R. (2005). Scale-free geometry in oo programs. *Communications of the ACM*, 48:99–103.

Pressman, R. S. (2006). *Engenharia de Software*. MacGraw Hill, Rio de Janeiro, 6 edição.

Puppin, D. & Silvestri, F. (2006). The social network of java classes. In *Proceedings of the 2006 ACM symposium on Applied computing*, SAC '06, pp. 1409–1413, New York, NY, Estados Unidos. ACM.

Riehle, D. (2000). *Framework Design: A Role Modeling Approach*. Dissertation No. 13509, ETH Zürich., http://dirkriehle.com/computer-science/research/dissertation/.

Santa'Anna, C. N.; Garcia, A. F.; Chaves, C. V. F. G.; Lucena, C. J. P. & Staa, A. V. (2003). On the reuse and maintenance of aspect-oriented software: an assessment framework. In *Proceedings of the XVII Brazilian Symposium on Software Engineering*, Porto Alegre, Brasil.

Sarwar, M. I.; WasifTanveer; Sarwar, I. & Mahmood, W. (2008). A comparative study of mi tools: Defining the roadmap to mi tools standardization. *IEEE*.

Schröter, A.; Zimmermann, T. & Zeller, A. (2006). Faults and failures: Predicting component failures at design time. In *ACM/IEEE International Symposium on Empirical Software Engineering ISESE '06*.

SEI, S. E. I. (2009). *Software Tecnology Roadmap*. Disponível em http://www.sei.cmu.edu/str/7/28/2008. Acesso em Abril de 2009.

Seng, O.; Stammel, J. & Burkhart, D. (2006). Search-based determination of refactorings for improving the class structure of object-oriented systems. In *Proceedings of the 8th annual Conference on Genetic and Evolutionary Computation*, GECCO '06, pp. 1909–1916, New York, NY, Estados Unidos. ACM.

Sommerville, I. (2003). *Engenharia de Software*. Addison Wesley, São Paulo, 6 edição.

Subramanyam, R. & Krishnan, M. S. (2003). Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *IEEE Trans. Softw. Eng.*, 29:297–310.

Tempero, E. (2008). On measuring java software. In *Proceedings of the thirty-first Australasian conference on Computer science - Volume 74*, ACSC '08, pp. 7–7, Darlinghurst, Australia, Australia. Australian Computer Society, Inc.

Tsantalis, N.; Chatzigeorgiou, A. & Stephanides, G. (2005). Predicting the probability of change in object-oriented systems. *IEEE Trans. Softw. Eng.*, 31:601–614.

Understand (2007). *Understand for Java.* Disponível em http://www.scitools.com/uj.html. Acesso em Abril de 2007.

V. Basili, G. C. & Rombach, H. (1994a). Goal question metric approach. *Encyclopedia of Software Engineering*, pp. 528–532.

V. Basili, G. C. & Rombach, H. (1994b). Measurement. *Encyclopedia of Software Engineering*, pp. 646–661.

von Staa, A. (2000). *Programação Modular - Desenvolvendo programas complexos de forma organizada e segura.* Campus, Rio de Janeiro.

Wand, M. (2003). Understanding aspects (extended abstract). In *ICFP'03*.

Weka (2009). *Weka 3: Data Mining Software in Java.* Disponível em http://www.cs.waikato.ac.nz/ml/weka/index.html. Acesso em Julho de 2009.

Weyuker, E. (1988). Evaluating software complexity measures. *IEEE Transactions on Software Engineering*, 14:1357–1365.

Wheelson, R. & Counsell, S. (2003). Power law distributions in class relationships. In *Proceedings of 3rd International Workshop on Source Code Analysis and Manipulation (SCAM)*, Amsterdã, Holanda.

Wikipedia (2009). *Factorial.* Disponível em http://pt.wikipedia.org/wiki/Factorial. Acesso em Julho de 2009.

Xenos, M.; Stavrinoudis, D.; Zikouli, K. & Christodoulakis, D. (2000). Object-oriented metrics - a survey. In *FESMA 2000*, Madrid, Espanha.

Xie, G.; Chen, J. & Neamtiu, I. (2009). Towards a better understanding of software evolution: An empirical study on open source software. In *ICSM*.

Zhou, Y. & Leung, H. (2006). Empirical analysis of object-oriented design metrics for predicting high and low severity faults. *IEEE Transactions on Software Engineering*, 32(10):771–789.

Zimmermann, T. & Nagappan, N. (2008). Predicting defects using network analysis on dependency graphs. In *ICSE '08: Proceedings of the 30th International Conference on Software Engineering*, pp. 531–540, Leipzig, Alemanha. ACM.

Apêndice A

Polinômios para Predição de Propagação de Modificações Contratuais em Software

A.1 Resultados

4 módulos

$$\begin{bmatrix} 1 + 6\,\frac{a}{b} + 12\,\frac{a^2}{b^2} + 12\,\frac{a^3}{b^3} \\ 2 + 10\,\frac{a}{b} + 16\,\frac{a^2}{b^2} + 12\,\frac{a^3}{b^3} \\ 3 + 12\,\frac{a}{b} + 16\,\frac{a^2}{b^2} + 12\,\frac{a^3}{b^3} \\ 4 + 12\,\frac{a}{b} + 16\,\frac{a^2}{b^2} + 12\,\frac{a^3}{b^3} \end{bmatrix}$$

5 módulos

$$\begin{bmatrix} 1 + 8\,\frac{a}{b} + 24\,\frac{a^2}{b^2} + 48\,\frac{a^3}{b^3} + 48\,\frac{a^4}{b^4} \\ 2 + 14\,\frac{a}{b} + 36\,\frac{a^2}{b^2} + 60\,\frac{a^3}{b^3} + 48\,\frac{a^4}{b^4} \\ 3 + 18\,\frac{a}{b} + 40\,\frac{a^2}{b^2} + 60\,\frac{a^3}{b^3} + 48\,\frac{a^4}{b^4} \\ 4 + 20\,\frac{a}{b} + 40\,\frac{a^2}{b^2} + 60\,\frac{a^3}{b^3} + 48\,\frac{a^4}{b^4} \\ 5 + 20\,\frac{a}{b} + 40\,\frac{a^2}{b^2} + 60\,\frac{a^3}{b^3} + 48\,\frac{a^4}{b^4} \end{bmatrix}$$

6 módulos

$$\begin{bmatrix} 1+10\,\frac{a}{b}+40\,\frac{a^2}{b^2}+120\,\frac{a^3}{b^3}+240\,\frac{a^4}{b^4}+240\,\frac{a^5}{b^5} \\ 2+18\,\frac{a}{b}+64\,\frac{a^2}{b^2}+168\,\frac{a^3}{b^3}+288\,\frac{a^4}{b^4}+240\,\frac{a^5}{b^5} \\ 3+24\,\frac{a}{b}+76\,\frac{a^2}{b^2}+180\,\frac{a^3}{b^3}+288\,\frac{a^4}{b^4}+240\,\frac{a^5}{b^5} \\ 4+28\,\frac{a}{b}+80\,\frac{a^2}{b^2}+180\,\frac{a^3}{b^3}+288\,\frac{a^4}{b^4}+240\,\frac{a^5}{b^5} \\ 5+30\,\frac{a}{b}+80\,\frac{a^2}{b^2}+180\,\frac{a^3}{b^3}+288\,\frac{a^4}{b^4}+240\,\frac{a^5}{b^5} \\ 6+30\,\frac{a}{b}+80\,\frac{a^2}{b^2}+180\,\frac{a^3}{b^3}+288\,\frac{a^4}{b^4}+240\,\frac{a^5}{b^5} \end{bmatrix}$$

7 módulos

$$\begin{bmatrix} 1+12\,\frac{a}{b}+60\,\frac{a^2}{b^2}+240\,\frac{a^3}{b^3}+720\,\frac{a^4}{b^4}+1440\,\frac{a^5}{b^5}+1440\,\frac{a^6}{b^6} \\ 2+22\,\frac{a}{b}+100\,\frac{a^2}{b^2}+360\,\frac{a^3}{b^3}+960\,\frac{a^4}{b^4}+1680\,\frac{a^5}{b^5}+1440\,\frac{a^6}{b^6} \\ 3+30\,\frac{a}{b}+124\,\frac{a^2}{b^2}+408\,\frac{a^3}{b^3}+1008\,\frac{a^4}{b^4}+1680\,\frac{a^5}{b^5}+1440\,\frac{a^6}{b^6} \\ 4+36\,\frac{a}{b}+136\,\frac{a^2}{b^2}+420\,\frac{a^3}{b^3}+1008\,\frac{a^4}{b^4}+1680\,\frac{a^5}{b^5}+1440\,\frac{a^6}{b^6} \\ 5+40\,\frac{a}{b}+140\,\frac{a^2}{b^2}+420\,\frac{a^3}{b^3}+1008\,\frac{a^4}{b^4}+1680\,\frac{a^5}{b^5}+1440\,\frac{a^6}{b^6} \\ 6+42\,\frac{a}{b}+140\,\frac{a^2}{b^2}+420\,\frac{a^3}{b^3}+1008\,\frac{a^4}{b^4}+1680\,\frac{a^5}{b^5}+1440\,\frac{a^6}{b^6} \\ 7+42\,\frac{a}{b}+140\,\frac{a^2}{b^2}+420\,\frac{a^3}{b^3}+1008\,\frac{a^4}{b^4}+1680\,\frac{a^5}{b^5}+1440\,\frac{a^6}{b^6} \end{bmatrix}$$

8 módulos

$$\begin{bmatrix} 1+14\,\frac{a}{b}+84\,\frac{a^2}{b^2}+420\,\frac{a^3}{b^3}+1680\,\frac{a^4}{b^4}+5040\,\frac{a^5}{b^5}+10080\,\frac{a^6}{b^6}+10080\,\frac{a^7}{b^7} \\ 2+26\,\frac{a}{b}+144\,\frac{a^2}{b^2}+660\,\frac{a^3}{b^3}+2400\,\frac{a^4}{b^4}+6480\,\frac{a^5}{b^5}+11520\,\frac{a^6}{b^6}+10080\,\frac{a^7}{b^7} \\ 3+36\,\frac{a}{b}+184\,\frac{a^2}{b^2}+780\,\frac{a^3}{b^3}+2640\,\frac{a^4}{b^4}+6720\,\frac{a^5}{b^5}+11520\,\frac{a^6}{b^6}+10080\,\frac{a^7}{b^7} \\ 4+44\,\frac{a}{b}+208\,\frac{a^2}{b^2}+828\,\frac{a^3}{b^3}+2688\,\frac{a^4}{b^4}+6720\,\frac{a^5}{b^5}+11520\,\frac{a^6}{b^6}+10080\,\frac{a^7}{b^7} \\ 5+50\,\frac{a}{b}+220\,\frac{a^2}{b^2}+840\,\frac{a^3}{b^3}+2688\,\frac{a^4}{b^4}+6720\,\frac{a^5}{b^5}+11520\,\frac{a^6}{b^6}+10080\,\frac{a^7}{b^7} \\ 6+54\,\frac{a}{b}+224\,\frac{a^2}{b^2}+840\,\frac{a^3}{b^3}+2688\,\frac{a^4}{b^4}+6720\,\frac{a^5}{b^5}+11520\,\frac{a^6}{b^6}+10080\,\frac{a^7}{b^7} \\ 7+56\,\frac{a}{b}+224\,\frac{a^2}{b^2}+840\,\frac{a^3}{b^3}+2688\,\frac{a^4}{b^4}+6720\,\frac{a^5}{b^5}+11520\,\frac{a^6}{b^6}+10080\,\frac{a^7}{b^7} \\ 8+56\,\frac{a}{b}+224\,\frac{a^2}{b^2}+840\,\frac{a^3}{b^3}+2688\,\frac{a^4}{b^4}+6720\,\frac{a^5}{b^5}+11520\,\frac{a^6}{b^6}+10080\,\frac{a^7}{b^7} \end{bmatrix}$$

9 módulos

$$\begin{bmatrix}
1 + 16\,\frac{a}{b} + 112\,\frac{a^2}{b^2} + 672\,\frac{a^3}{b^3} + 3360\,\frac{a^4}{b^4} + 13440\,\frac{a^5}{b^5} + 40320\,\frac{a^6}{b^6} + 80640\,\frac{a^7}{b^7} + 80640\,\frac{a^8}{b^8} \\
2 + 30\,\frac{a}{b} + 196\,\frac{a^2}{b^2} + 1092\,\frac{a^3}{b^3} + 5040\,\frac{a^4}{b^4} + 18480\,\frac{a^5}{b^5} + 50400\,\frac{a^6}{b^6} + 90720\,\frac{a^7}{b^7} + 80640\,\frac{a^8}{b^8} \\
3 + 42\,\frac{a}{b} + 256\,\frac{a^2}{b^2} + 1332\,\frac{a^3}{b^3} + 5760\,\frac{a^4}{b^4} + 19920\,\frac{a^5}{b^5} + 51840\,\frac{a^6}{b^6} + 90720\,\frac{a^7}{b^7} + 80640\,\frac{a^8}{b^8} \\
4 + 52\,\frac{a}{b} + 296\,\frac{a^2}{b^2} + 1452\,\frac{a^3}{b^3} + 6000\,\frac{a^4}{b^4} + 20160\,\frac{a^5}{b^5} + 51840\,\frac{a^6}{b^6} + 90720\,\frac{a^7}{b^7} + 80640\,\frac{a^8}{b^8} \\
5 + 60\,\frac{a}{b} + 320\,\frac{a^2}{b^2} + 1500\,\frac{a^3}{b^3} + 6048\,\frac{a^4}{b^4} + 20160\,\frac{a^5}{b^5} + 51840\,\frac{a^6}{b^6} + 90720\,\frac{a^7}{b^7} + 80640\,\frac{a^8}{b^8} \\
6 + 66\,\frac{a}{b} + 332\,\frac{a^2}{b^2} + 1512\,\frac{a^3}{b^3} + 6048\,\frac{a^4}{b^4} + 20160\,\frac{a^5}{b^5} + 51840\,\frac{a^6}{b^6} + 90720\,\frac{a^7}{b^7} + 80640\,\frac{a^8}{b^8} \\
7 + 70\,\frac{a}{b} + 336\,\frac{a^2}{b^2} + 1512\,\frac{a^3}{b^3} + 6048\,\frac{a^4}{b^4} + 20160\,\frac{a^5}{b^5} + 51840\,\frac{a^6}{b^6} + 90720\,\frac{a^7}{b^7} + 80640\,\frac{a^8}{b^8} \\
8 + 72\,\frac{a}{b} + 336\,\frac{a^2}{b^2} + 1512\,\frac{a^3}{b^3} + 6048\,\frac{a^4}{b^4} + 20160\,\frac{a^5}{b^5} + 51840\,\frac{a^6}{b^6} + 90720\,\frac{a^7}{b^7} + 80640\,\frac{a^8}{b^8} \\
9 + 72\,\frac{a}{b} + 336\,\frac{a^2}{b^2} + 1512\,\frac{a^3}{b^3} + 6048\,\frac{a^4}{b^4} + 20160\,\frac{a^5}{b^5} + 51840\,\frac{a^6}{b^6} + 90720\,\frac{a^7}{b^7} + 80640\,\frac{a^8}{b^8}
\end{bmatrix}$$

10 módulos

$1+18\,\frac{a}{b}+144\,\frac{a^2}{b^2}+1008\,\frac{a^3}{b^3}+6048\,\frac{a^4}{b^4}+30240\,\frac{a^5}{b^5}+120960\,\frac{a^6}{b^6}+362880\,\frac{a^7}{b^7}+725760\,\frac{a^8}{b^8}+725760\,\frac{a^9}{b^9}$

$2+34\,\frac{a}{b}+256\,\frac{a^2}{b^2}+1680\,\frac{a^3}{b^3}+9408\,\frac{a^4}{b^4}+43680\,\frac{a^5}{b^5}+161280\,\frac{a^6}{b^6}+443520\,\frac{a^7}{b^7}+806400\,\frac{a^8}{b^8}+725760\,\frac{a^9}{b^9}$

$3+48\,\frac{a}{b}+340\,\frac{a^2}{b^2}+2100\,\frac{a^3}{b^3}+11088\,\frac{a^4}{b^4}+48720\,\frac{a^5}{b^5}+171360\,\frac{a^6}{b^6}+453600\,\frac{a^7}{b^7}+806400\,\frac{a^8}{b^8}+725760\,\frac{a^9}{b^9}$

$4+60\,\frac{a}{b}+400\,\frac{a^2}{b^2}+2340\,\frac{a^3}{b^3}+11808\,\frac{a^4}{b^4}+50160\,\frac{a^5}{b^5}+172800\,\frac{a^6}{b^6}+453600\,\frac{a^7}{b^7}+806400\,\frac{a^8}{b^8}+725760\,\frac{a^9}{b^9}$

$5+70\,\frac{a}{b}+440\,\frac{a^2}{b^2}+2460\,\frac{a^3}{b^3}+12048\,\frac{a^4}{b^4}+50400\,\frac{a^5}{b^5}+172800\,\frac{a^6}{b^6}+453600\,\frac{a^7}{b^7}+806400\,\frac{a^8}{b^8}+725760\,\frac{a^9}{b^9}$

$6+78\,\frac{a}{b}+464\,\frac{a^2}{b^2}+2508\,\frac{a^3}{b^3}+12096\,\frac{a^4}{b^4}+50400\,\frac{a^5}{b^5}+172800\,\frac{a^6}{b^6}+453600\,\frac{a^7}{b^7}+806400\,\frac{a^8}{b^8}+725760\,\frac{a^9}{b^9}$

$7+84\,\frac{a}{b}+476\,\frac{a^2}{b^2}+2520\,\frac{a^3}{b^3}+12096\,\frac{a^4}{b^4}+50400\,\frac{a^5}{b^5}+172800\,\frac{a^6}{b^6}+453600\,\frac{a^7}{b^7}+806400\,\frac{a^8}{b^8}+725760\,\frac{a^9}{b^9}$

$8+88\,\frac{a}{b}+480\,\frac{a^2}{b^2}+2520\,\frac{a^3}{b^3}+12096\,\frac{a^4}{b^4}+50400\,\frac{a^5}{b^5}+172800\,\frac{a^6}{b^6}+453600\,\frac{a^7}{b^7}+806400\,\frac{a^8}{b^8}+725760\,\frac{a^9}{b^9}$

$9+90\,\frac{a}{b}+480\,\frac{a^2}{b^2}+2520\,\frac{a^3}{b^3}+12096\,\frac{a^4}{b^4}+50400\,\frac{a^5}{b^5}+172800\,\frac{a^6}{b^6}+453600\,\frac{a^7}{b^7}+806400\,\frac{a^8}{b^8}+725760\,\frac{a^9}{b^9}$

$10 + 90\,\frac{a}{b} + 480\,\frac{a^2}{b^2} + 2520\,\frac{a^3}{b^3} + 12096\,\frac{a^4}{b^4} + 50400\,\frac{a^5}{b^5} + 172800\,\frac{a^6}{b^6} + 453600\,\frac{a^7}{b^7} + 806400\,\frac{a^8}{b^8} + 725760\,\frac{a^9}{b^9}$

11 módulos

$1 + 20\,\frac{a}{b} + 180\,\frac{a^2}{b^2} + 1440\,\frac{a^3}{b^3} + 10080\,\frac{a^4}{b^4} + 60480\,\frac{a^5}{b^5} + 302400\,\frac{a^6}{b^6} + 1209600\,\frac{a^7}{b^7} + 3628800\,\frac{a^8}{b^8} + 7257600\,\frac{a^9}{b^9} + 7257600\,\frac{a^{10}}{b^{10}}$

$1 + 20\,\frac{a}{b} + 180\,\frac{a^2}{b^2} + 1440\,\frac{a^3}{b^3} + 10080\,\frac{a^4}{b^4} + 60480\,\frac{a^5}{b^5} + 302400\,\frac{a^6}{b^6} + 1209600\,\frac{a^7}{b^7} + 3628800\,\frac{a^8}{b^8} + 7257600\,\frac{a^9}{b^9} + 7257600\,\frac{a^{10}}{b^{10}}$

$2 + 38\,\frac{a}{b} + 324\,\frac{a^2}{b^2} + 2448\,\frac{a^3}{b^3} + 16128\,\frac{a^4}{b^4} + 90720\,\frac{a^5}{b^5} + 423360\,\frac{a^6}{b^6} + 1572480\,\frac{a^7}{b^7} + 4354560\,\frac{a^8}{b^8} + 7983360\,\frac{a^9}{b^9} + 7257600\,\frac{a^{10}}{b^{10}}$

$3 + 54\,\frac{a}{b} + 436\,\frac{a^2}{b^2} + 3120\,\frac{a^3}{b^3} + 19488\,\frac{a^4}{b^4} + 104160\,\frac{a^5}{b^5} + 463680\,\frac{a^6}{b^6} + 1653120\,\frac{a^7}{b^7} + 4435200\,\frac{a^8}{b^8} + 7983360\,\frac{a^9}{b^9} + 7257600\,\frac{a^{10}}{b^{10}}$

$4 + 68\,\frac{a}{b} + 520\,\frac{a^2}{b^2} + 3540\,\frac{a^3}{b^3} + 21168\,\frac{a^4}{b^4} + 109200\,\frac{a^5}{b^5} + 473760\,\frac{a^6}{b^6} + 1663200\,\frac{a^7}{b^7} + 4435200\,\frac{a^8}{b^8} + 7983360\,\frac{a^9}{b^9} + 7257600\,\frac{a^{10}}{b^{10}}$

$5 + 80\,\frac{a}{b} + 580\,\frac{a^2}{b^2} + 3780\,\frac{a^3}{b^3} + 21888\,\frac{a^4}{b^4} + 110640\,\frac{a^5}{b^5} + 475200\,\frac{a^6}{b^6} + 1663200\,\frac{a^7}{b^7} + 4435200\,\frac{a^8}{b^8} + 7983360\,\frac{a^9}{b^9} + 7257600\,\frac{a^{10}}{b^{10}}$

$6 + 90\,\frac{a}{b} + 620\,\frac{a^2}{b^2} + 3900\,\frac{a^3}{b^3} + 22128\,\frac{a^4}{b^4} + 110880\,\frac{a^5}{b^5} + 475200\,\frac{a^6}{b^6} + 1663200\,\frac{a^7}{b^7} + 4435200\,\frac{a^8}{b^8} + 7983360\,\frac{a^9}{b^9} + 7257600\,\frac{a^{10}}{b^{10}}$

$7 + 98\,\frac{a}{b} + 644\,\frac{a^2}{b^2} + 3948\,\frac{a^3}{b^3} + 22176\,\frac{a^4}{b^4} + 110880\,\frac{a^5}{b^5} + 475200\,\frac{a^6}{b^6} + 1663200\,\frac{a^7}{b^7} + 4435200\,\frac{a^8}{b^8} + 7983360\,\frac{a^9}{b^9} + 7257600\,\frac{a^{10}}{b^{10}}$

$8 + 104\,\frac{a}{b} + 656\,\frac{a^2}{b^2} + 3960\,\frac{a^3}{b^3} + 22176\,\frac{a^4}{b^4} + 110880\,\frac{a^5}{b^5} + 475200\,\frac{a^6}{b^6} + 1663200\,\frac{a^7}{b^7} + 4435200\,\frac{a^8}{b^8} + 7983360\,\frac{a^9}{b^9} + 7257600\,\frac{a^{10}}{b^{10}}$

$9 + 108\,\frac{a}{b} + 660\,\frac{a^2}{b^2} + 3960\,\frac{a^3}{b^3} + 22176\,\frac{a^4}{b^4} + 110880\,\frac{a^5}{b^5} + 475200\,\frac{a^6}{b^6} + 1663200\,\frac{a^7}{b^7} + 4435200\,\frac{a^8}{b^8} + 7983360\,\frac{a^9}{b^9} + 7257600\,\frac{a^{10}}{b^{10}}$

$10 + 110\,\frac{a}{b} + 660\,\frac{a^2}{b^2} + 3960\,\frac{a^3}{b^3} + 22176\,\frac{a^4}{b^4} + 110880\,\frac{a^5}{b^5} + 475200\,\frac{a^6}{b^6} + 1663200\,\frac{a^7}{b^7} + 4435200\,\frac{a^8}{b^8} + 7983360\,\frac{a^9}{b^9} + 7257600\,\frac{a^{10}}{b^{10}}$

$11 + 110\,\frac{a}{b} + 660\,\frac{a^2}{b^2} + 3960\,\frac{a^3}{b^3} + 22176\,\frac{a^4}{b^4} + 110880\,\frac{a^5}{b^5} + 475200\,\frac{a^6}{b^6} + 1663200\,\frac{a^7}{b^7} + 4435200\,\frac{a^8}{b^8} + 7983360\,\frac{a^9}{b^9} + 7257600\,\frac{a^{10}}{b^{10}}$

20 módulos

$1+38\,\frac{a}{b}+684\,\frac{a^2}{b^2}+11628\,\frac{a^3}{b^3}+186048\,\frac{a^4}{b^4}+2790720\,\frac{a^5}{b^5}+39070080\,\frac{a^6}{b^6}+507911040\,\frac{a^7}{b^7}+ 6094932480\,\frac{a^8}{b^8} + 67044257280\,\frac{a^9}{b^9} + 670442572800\,\frac{a^{10}}{b^{10}} + 6033983155200\,\frac{a^{11}}{b^{11}} + 48271865241600\,\frac{a^{12}}{b^{12}} + 337903056691200\,\frac{a^{13}}{b^{13}} + 2027418340147200\,\frac{a^{14}}{b^{14}} + 10137091700736000\,\frac{a^{15}}{b^{15}} + 40548366802944000\,\frac{a^{16}}{b^{16}} + 121645100408832000\,\frac{a^{17}}{b^{17}} + 243290200817664000\,\frac{a^{18}}{b^{18}} + 243290200817664000\,\frac{a^{19}}{b^{19}}$

$2+74\,\frac{a}{b}+1296\,\frac{a^2}{b^2}+21420\,\frac{a^3}{b^3}+332928\,\frac{a^4}{b^4}+4847040\,\frac{a^5}{b^5}+65802240\,\frac{a^6}{b^6}+828696960\,\frac{a^7}{b^7}+ 9623577600\,\frac{a^8}{b^8} + 102330708480\,\frac{a^9}{b^9} + 988020633600\,\frac{a^{10}}{b^{10}} + 8574607641600\,\frac{a^{11}}{b^{11}} + 66056236646400\,\frac{a^{12}}{b^{12}} + 444609285120000\,\frac{a^{13}}{b^{13}} + 2560949482291200\,\frac{a^{14}}{b^{14}} + 12271216269312000\,\frac{a^{15}}{b^{15}} + 46950740508672000\,\frac{a^{16}}{b^{16}} + 134449847820288000\,\frac{a^{17}}{b^{17}} +$

$256094948229120000\,\frac{a^{18}}{b^{18}} + 243290200817664000\,\frac{a^{19}}{b^{19}}$

$3 + 108\,\frac{a}{b} + 1840\,\frac{a^2}{b^2} + 29580\,\frac{a^3}{b^3} + 447168\,\frac{a^4}{b^4} + 6332160\,\frac{a^5}{b^5} + 83623680\,\frac{a^6}{b^6} + 1024732800\,\frac{a^7}{b^7} + 11583936000\,\frac{a^8}{b^8} + 119973934080\,\frac{a^9}{b^9} + 1129166438400\,\frac{a^{10}}{b^{10}} + 9562628275200\,\frac{a^{11}}{b^{11}} + 71984360448000\,\frac{a^{12}}{b^{12}} + 474249904128000\,\frac{a^{13}}{b^{13}} + 2679511958323200\,\frac{a^{14}}{b^{14}} + 12626903697408000\,\frac{a^{15}}{b^{15}} + 47662115364864000\,\frac{a^{16}}{b^{16}} + 135161222676480000\,\frac{a^{17}}{b^{17}} + 256094948229120000\,\frac{a^{18}}{b^{18}} + 243290200817664000\,\frac{a^{19}}{b^{19}}$

$4 + 140\,\frac{a}{b} + 2320\,\frac{a^2}{b^2} + 36300\,\frac{a^3}{b^3} + 534528\,\frac{a^4}{b^4} + 7380480\,\frac{a^5}{b^5} + 95155200\,\frac{a^6}{b^6} + 1140048000\,\frac{a^7}{b^7} + 12621772800\,\frac{a^8}{b^8} + 128276628480\,\frac{a^9}{b^9} + 1187285299200\,\frac{a^{10}}{b^{10}} + 9911341440000\,\frac{a^{11}}{b^{11}} + 73727926272000\,\frac{a^{12}}{b^{12}} + 481224167424000\,\frac{a^{13}}{b^{13}} + 2700434748211200\,\frac{a^{14}}{b^{14}} + 12668749277184000\,\frac{a^{15}}{b^{15}} + 47703960944640000\,\frac{a^{16}}{b^{16}} + 135161222676480000\,\frac{a^{17}}{b^{17}} + 256094948229120000\,\frac{a^{18}}{b^{18}} + 243290200817664000\,\frac{a^{19}}{b^{19}}$

$5 + 170\,\frac{a}{b} + 2740\,\frac{a^2}{b^2} + 41760\,\frac{a^3}{b^3} + 600048\,\frac{a^4}{b^4} + 8101200\,\frac{a^5}{b^5} + 102362400\,\frac{a^6}{b^6} + 1204912800\,\frac{a^7}{b^7} + 13140691200\,\frac{a^8}{b^8} + 131909057280\,\frac{a^9}{b^9} + 1209079872000\,\frac{a^{10}}{b^{10}} + 10020314304000\,\frac{a^{11}}{b^{11}} + 74163817728000\,\frac{a^{12}}{b^{12}} + 482531841792000\,\frac{a^{13}}{b^{13}} + 2703050096947200\,\frac{a^{14}}{b^{14}} + 12671364625920000\,\frac{a^{15}}{b^{15}} + 47703960944640000\,\frac{a^{16}}{b^{16}} + 135161222676480000\,\frac{a^{17}}{b^{17}} + 256094948229120000\,\frac{a^{18}}{b^{18}} + 243290200817664000\,\frac{a^{19}}{b^{19}}$

$6 + 198\,\frac{a}{b} + 3104\,\frac{a^2}{b^2} + 46128\,\frac{a^3}{b^3} + 648096\,\frac{a^4}{b^4} + 8581680\,\frac{a^5}{b^5} + 106686720\,\frac{a^6}{b^6} + 1239507360\,\frac{a^7}{b^7} + 13382853120\,\frac{a^8}{b^8} + 133362028800\,\frac{a^9}{b^9} + 1216344729600\,\frac{a^{10}}{b^{10}} + 10049373734400\,\frac{a^{11}}{b^{11}} + 74250996019200\,\frac{a^{12}}{b^{12}} + 482706198374400\,\frac{a^{13}}{b^{13}} + 2703224453529600\,\frac{a^{14}}{b^{14}} + 12671364625920000\,\frac{a^{15}}{b^{15}} + 47703960944640000\,\frac{a^{16}}{b^{16}} + 135161222676480000\,\frac{a^{17}}{b^{17}} + 256094948229120000\,\frac{a^{18}}{b^{18}} + 243290200817664000\,\frac{a^{19}}{b^{19}}$

$7 + 224\,\frac{a}{b} + 3416\,\frac{a^2}{b^2} + 49560\,\frac{a^3}{b^3} + 682416\,\frac{a^4}{b^4} + 8890560\,\frac{a^5}{b^5} + 109157760\,\frac{a^6}{b^6} + 1256804640\,\frac{a^7}{b^7} + 13486636800\,\frac{a^8}{b^8} + 133880947200\,\frac{a^9}{b^9} + 1218420403200\,\frac{a^{10}}{b^{10}} + 10055600755200\,\frac{a^{11}}{b^{11}} + 74263450060800\,\frac{a^{12}}{b^{12}} + 482718652416000\,\frac{a^{13}}{b^{13}} + 2703224453529600\,\frac{a^{14}}{b^{14}} + 12671364625920000\,\frac{a^{15}}{b^{15}} + 47703960944640000\,\frac{a^{16}}{b^{16}} + 135161222676480000\,\frac{a^{17}}{b^{17}} + 256094948229120000\,\frac{a^{18}}{b^{18}} + 243290200817664000\,\frac{a^{19}}{b^{19}}$

$8 + 248\,\frac{a}{b} + 3680\,\frac{a^2}{b^2} + 52200\,\frac{a^3}{b^3} + 706176\,\frac{a^4}{b^4} + 9080640\,\frac{a^5}{b^5} + 110488320\,\frac{a^6}{b^6} + 1264788000\,\frac{a^7}{b^7} + 13526553600\,\frac{a^8}{b^8} + 134040614400\,\frac{a^9}{b^9} + 1218899404800\,\frac{a^{10}}{b^{10}} + 10056558758400\,\frac{a^{11}}{b^{11}} + 74264408064000\,\frac{a^{12}}{b^{12}} + 482718652416000\,\frac{a^{13}}{b^{13}} + 2703224453529600\,\frac{a^{14}}{b^{14}} + 12671364625920000\,\frac{a^{15}}{b^{15}} + 47703960944640000\,\frac{a^{16}}{b^{16}} + 135161222676480000\,\frac{a^{17}}{b^{17}} + 256094948229120000\,\frac{a^{18}}{b^{18}} + 243290200817664000\,\frac{a^{19}}{b^{19}}$

$9 + 270\,\frac{a}{b} + 3900\,\frac{a^2}{b^2} + 54180\,\frac{a^3}{b^3} + 722016\,\frac{a^4}{b^4} + 9191520\,\frac{a^5}{b^5} + 111153600\,\frac{a^6}{b^6} + 1268114400\,\frac{a^7}{b^7} + 13539859200\,\frac{a^8}{b^8} + 134080531200\,\frac{a^9}{b^9} + 1218979238400\,\frac{a^{10}}{b^{10}} + 10056638592000\,\frac{a^{11}}{b^{11}} + 74264408064000\,\frac{a^{12}}{b^{12}} + 482718652416000\,\frac{a^{13}}{b^{13}} + 2703224453529600\,\frac{a^{14}}{b^{14}} + 12671364625920000\,\frac{a^{15}}{b^{15}} + 47703960944640000\,\frac{a^{16}}{b^{16}} + 135161222676480000\,\frac{a^{17}}{b^{17}} + 256094948229120000\,\frac{a^{18}}{b^{18}} + 243290200817664000\,\frac{a^{19}}{b^{19}}$

$10 + 290\,\frac{a}{b} + 4080\,\frac{a^2}{b^2} + 55620\,\frac{a^3}{b^3} + 732096\,\frac{a^4}{b^4} + 9252000\,\frac{a^5}{b^5} +$

$111456000\,\frac{a^{6}}{b^{6}} + 1269324000\,\frac{a^{7}}{b^{7}} + 13543488000\,\frac{a^{8}}{b^{8}} + 134087788800\,\frac{a^{9}}{b^{9}} + 1218986496000\,\frac{a^{10}}{b^{10}} + 10056638592000\,\frac{a^{11}}{b^{11}} + 74264408064000\,\frac{a^{12}}{b^{12}} + 482718652416000\,\frac{a^{13}}{b^{13}} + 2703224453529600\,\frac{a^{14}}{b^{14}} + 12671364625920000\,\frac{a^{15}}{b^{15}} + 47703960944640000\,\frac{a^{16}}{b^{16}} + 135161222676480000\,\frac{a^{17}}{b^{17}} + 256094948229120000\,\frac{a^{18}}{b^{18}} + 243290200817664000\,\frac{a^{19}}{b^{19}}$

$11 + 308\,\frac{a}{b} + 4224\,\frac{a^{2}}{b^{2}} + 56628\,\frac{a^{3}}{b^{3}} + 738144\,\frac{a^{4}}{b^{4}} + 9282240\,\frac{a^{5}}{b^{5}} + 111576960\,\frac{a^{6}}{b^{6}} + 1269686880\,\frac{a^{7}}{b^{7}} + 13544213760\,\frac{a^{8}}{b^{8}} + 134088514560\,\frac{a^{9}}{b^{9}} + 1218986496000\,\frac{a^{10}}{b^{10}} + 10056638592000\,\frac{a^{11}}{b^{11}} + 74264408064000\,\frac{a^{12}}{b^{12}} + 482718652416000\,\frac{a^{13}}{b^{13}} + 2703224453529600\,\frac{a^{14}}{b^{14}} + 12671364625920000\,\frac{a^{15}}{b^{15}} + 47703960944640000\,\frac{a^{16}}{b^{16}} + 135161222676480000\,\frac{a^{17}}{b^{17}} + 256094948229120000\,\frac{a^{18}}{b^{18}} + 243290200817664000\,\frac{a^{19}}{b^{19}}$

$12 + 324\,\frac{a}{b} + 4336\,\frac{a^{2}}{b^{2}} + 57300\,\frac{a^{3}}{b^{3}} + 741504\,\frac{a^{4}}{b^{4}} + 9295680\,\frac{a^{5}}{b^{5}} + 111617280\,\frac{a^{6}}{b^{6}} + 1269767520\,\frac{a^{7}}{b^{7}} + 13544294400\,\frac{a^{8}}{b^{8}} + 134088514560\,\frac{a^{9}}{b^{9}} + 1218986496000\,\frac{a^{10}}{b^{10}} + 10056638592000\,\frac{a^{11}}{b^{11}} + 74264408064000\,\frac{a^{12}}{b^{12}} + 482718652416000\,\frac{a^{13}}{b^{13}} + 2703224453529600\,\frac{a^{14}}{b^{14}} + 12671364625920000\,\frac{a^{15}}{b^{15}} + 47703960944640000\,\frac{a^{16}}{b^{16}} + 135161222676480000\,\frac{a^{17}}{b^{17}} + 256094948229120000\,\frac{a^{18}}{b^{18}} + 243290200817664000\,\frac{a^{19}}{b^{19}}$

$13 + 338\,\frac{a}{b} + 4420\,\frac{a^{2}}{b^{2}} + 57720\,\frac{a^{3}}{b^{3}} + 743184\,\frac{a^{4}}{b^{4}} + 9300720\,\frac{a^{5}}{b^{5}} + 111627360\,\frac{a^{6}}{b^{6}} + 1269777600\,\frac{a^{7}}{b^{7}} + 13544294400\,\frac{a^{8}}{b^{8}} + 134088514560\,\frac{a^{9}}{b^{9}} + 1218986496000\,\frac{a^{10}}{b^{10}} + 10056638592000\,\frac{a^{11}}{b^{11}} + 74264408064000\,\frac{a^{12}}{b^{12}} + 482718652416000\,\frac{a^{13}}{b^{13}} + 2703224453529600\,\frac{a^{14}}{b^{14}} + 12671364625920000\,\frac{a^{15}}{b^{15}} + 47703960944640000\,\frac{a^{16}}{b^{16}} + 135161222676480000\,\frac{a^{17}}{b^{17}} + 256094948229120000\,\frac{a^{18}}{b^{18}} + 243290200817664000\,\frac{a^{19}}{b^{19}}$

$14 + 350\,\frac{a}{b} + 4480\,\frac{a^{2}}{b^{2}} + 57960\,\frac{a^{3}}{b^{3}} + 743904\,\frac{a^{4}}{b^{4}} + 9302160\,\frac{a^{5}}{b^{5}} + 111628800\,\frac{a^{6}}{b^{6}} + 1269777600\,\frac{a^{7}}{b^{7}} + 13544294400\,\frac{a^{8}}{b^{8}} + 134088514560\,\frac{a^{9}}{b^{9}} + 1218986496000\,\frac{a^{10}}{b^{10}} + 10056638592000\,\frac{a^{11}}{b^{11}} + 74264408064000\,\frac{a^{12}}{b^{12}} + 482718652416000\,\frac{a^{13}}{b^{13}} + 2703224453529600\,\frac{a^{14}}{b^{14}} + 12671364625920000\,\frac{a^{15}}{b^{15}} + 47703960944640000\,\frac{a^{16}}{b^{16}} + 135161222676480000\,\frac{a^{17}}{b^{17}} + 256094948229120000\,\frac{a^{18}}{b^{18}} + 243290200817664000\,\frac{a^{19}}{b^{19}}$

$15 + 360\,\frac{a}{b} + 4520\,\frac{a^{2}}{b^{2}} + 58080\,\frac{a^{3}}{b^{3}} + 744144\,\frac{a^{4}}{b^{4}} + 9302400\,\frac{a^{5}}{b^{5}} + 111628800\,\frac{a^{6}}{b^{6}} + 1269777600\,\frac{a^{7}}{b^{7}} + 13544294400\,\frac{a^{8}}{b^{8}} + 134088514560\,\frac{a^{9}}{b^{9}} + 1218986496000\,\frac{a^{10}}{b^{10}} + 10056638592000\,\frac{a^{11}}{b^{11}} + 74264408064000\,\frac{a^{12}}{b^{12}} + 482718652416000\,\frac{a^{13}}{b^{13}} + 2703224453529600\,\frac{a^{14}}{b^{14}} + 12671364625920000\,\frac{a^{15}}{b^{15}} + 47703960944640000\,\frac{a^{16}}{b^{16}} + 135161222676480000\,\frac{a^{17}}{b^{17}} + 256094948229120000\,\frac{a^{18}}{b^{18}} + 243290200817664000\,\frac{a^{19}}{b^{19}}$

$16 + 368\,\frac{a}{b} + 4544\,\frac{a^{2}}{b^{2}} + 58128\,\frac{a^{3}}{b^{3}} + 744192\,\frac{a^{4}}{b^{4}} + 9302400\,\frac{a^{5}}{b^{5}} + 111628800\,\frac{a^{6}}{b^{6}} + 1269777600\,\frac{a^{7}}{b^{7}} + 13544294400\,\frac{a^{8}}{b^{8}} + 134088514560\,\frac{a^{9}}{b^{9}} + 1218986496000\,\frac{a^{10}}{b^{10}} + 10056638592000\,\frac{a^{11}}{b^{11}} + 74264408064000\,\frac{a^{12}}{b^{12}} + 482718652416000\,\frac{a^{13}}{b^{13}} + 2703224453529600\,\frac{a^{14}}{b^{14}} + 12671364625920000\,\frac{a^{15}}{b^{15}} + 47703960944640000\,\frac{a^{16}}{b^{16}} + 135161222676480000\,\frac{a^{17}}{b^{17}} + 256094948229120000\,\frac{a^{18}}{b^{18}} + 243290200817664000\,\frac{a^{19}}{b^{19}}$

$17 + 374\,\frac{a}{b} + 4556\,\frac{a^{2}}{b^{2}} + 58140\,\frac{a^{3}}{b^{3}} + 744192\,\frac{a^{4}}{b^{4}} + 9302400\,\frac{a^{5}}{b^{5}} + 111628800\,\frac{a^{6}}{b^{6}} + 1269777600\,\frac{a^{7}}{b^{7}} + 13544294400\,\frac{a^{8}}{b^{8}} + 134088514560\,\frac{a^{9}}{b^{9}} + 1218986496000\,\frac{a^{10}}{b^{10}} + 10056638592000\,\frac{a^{11}}{b^{11}} + 74264408064000\,\frac{a^{12}}{b^{12}} + 482718652416000\,\frac{a^{13}}{b^{13}} +$

$2703224453529600\,\frac{a^{14}}{b^{14}} + 12671364625920000\,\frac{a^{15}}{b^{15}} + 47703960944640000\,\frac{a^{16}}{b^{16}} + 135161222676480000\,\frac{a^{17}}{b^{17}} + 256094948229120000\,\frac{a^{18}}{b^{18}} + 243290200817664000\,\frac{a^{19}}{b^{19}}$

$18 + 378\,\frac{a}{b} + 4560\,\frac{a^{2}}{b^{2}} + 58140\,\frac{a^{3}}{b^{3}} + 744192\,\frac{a^{4}}{b^{4}} + 9302400\,\frac{a^{5}}{b^{5}} + 111628800\,\frac{a^{6}}{b^{6}} + 1269777600\,\frac{a^{7}}{b^{7}} + 13544294400\,\frac{a^{8}}{b^{8}} + 134088514560\,\frac{a^{9}}{b^{9}} + 1218986496000\,\frac{a^{10}}{b^{10}} + 10056638592000\,\frac{a^{11}}{b^{11}} + 74264408064000\,\frac{a^{12}}{b^{12}} + 482718652416000\,\frac{a^{13}}{b^{13}} + 2703224453529600\,\frac{a^{14}}{b^{14}} + 12671364625920000\,\frac{a^{15}}{b^{15}} + 47703960944640000\,\frac{a^{16}}{b^{16}} + 135161222676480000\,\frac{a^{17}}{b^{17}} + 256094948229120000\,\frac{a^{18}}{b^{18}} + 243290200817664000\,\frac{a^{19}}{b^{19}}$

$19 + 380\,\frac{a}{b} + 4560\,\frac{a^{2}}{b^{2}} + 58140\,\frac{a^{3}}{b^{3}} + 744192\,\frac{a^{4}}{b^{4}} + 9302400\,\frac{a^{5}}{b^{5}} + 111628800\,\frac{a^{6}}{b^{6}} + 1269777600\,\frac{a^{7}}{b^{7}} + 13544294400\,\frac{a^{8}}{b^{8}} + 134088514560\,\frac{a^{9}}{b^{9}} + 1218986496000\,\frac{a^{10}}{b^{10}} + 10056638592000\,\frac{a^{11}}{b^{11}} + 74264408064000\,\frac{a^{12}}{b^{12}} + 482718652416000\,\frac{a^{13}}{b^{13}} + 2703224453529600\,\frac{a^{14}}{b^{14}} + 12671364625920000\,\frac{a^{15}}{b^{15}} + 47703960944640000\,\frac{a^{16}}{b^{16}} + 135161222676480000\,\frac{a^{17}}{b^{17}} + 256094948229120000\,\frac{a^{18}}{b^{18}} + 243290200817664000\,\frac{a^{19}}{b^{19}}$

$20 + 380\,\frac{a}{b} + 4560\,\frac{a^{2}}{b^{2}} + 58140\,\frac{a^{3}}{b^{3}} + 744192\,\frac{a^{4}}{b^{4}} + 9302400\,\frac{a^{5}}{b^{5}} + 111628800\,\frac{a^{6}}{b^{6}} + 1269777600\,\frac{a^{7}}{b^{7}} + 13544294400\,\frac{a^{8}}{b^{8}} + 134088514560\,\frac{a^{9}}{b^{9}} + 1218986496000\,\frac{a^{10}}{b^{10}} + 10056638592000\,\frac{a^{11}}{b^{11}} + 74264408064000\,\frac{a^{12}}{b^{12}} + 482718652416000\,\frac{a^{13}}{b^{13}} + 2703224453529600\,\frac{a^{14}}{b^{14}} + 12671364625920000\,\frac{a^{15}}{b^{15}} + 47703960944640000\,\frac{a^{16}}{b^{16}} + 135161222676480000\,\frac{a^{17}}{b^{17}} + 256094948229120000\,\frac{a^{18}}{b^{18}} + 243290200817664000\,\frac{a^{19}}{b^{19}}$

25 módulos

$1 + 48\,\frac{a}{b} + 1104\,\frac{a^{2}}{b^{2}} + 24288\,\frac{a^{3}}{b^{3}} + 510048\,\frac{a^{4}}{b^{4}} + 10200960\,\frac{a^{5}}{b^{5}} + 193818240\,\frac{a^{6}}{b^{6}} + 3488728320\,\frac{a^{7}}{b^{7}} + 59308381440\,\frac{a^{8}}{b^{8}} + 948934103040\,\frac{a^{9}}{b^{9}} + 14234011545600\,\frac{a^{10}}{b^{10}} + 199276161638400\,\frac{a^{11}}{b^{11}} + 2590590101299200\,\frac{a^{12}}{b^{12}} + 31087081215590400\,\frac{a^{13}}{b^{13}} + 341957893371494400\,\frac{a^{14}}{b^{14}} + 3419578933714944000\,\frac{a^{15}}{b^{15}} + 30776210403434496000\,\frac{a^{16}}{b^{16}} + 246209683227475968000\,\frac{a^{17}}{b^{17}} + 1723467782592331776000\,\frac{a^{18}}{b^{18}} + 10340806695553990656000\,\frac{a^{19}}{b^{19}} + 51704033477769953280000\,\frac{a^{20}}{b^{20}} + 206816133911079813120000\,\frac{a^{21}}{b^{21}} + 620448401733239439360000\,\frac{a^{22}}{b^{22}} + 1240896803466478878720000\,\frac{a^{23}}{b^{23}} + 1240896803466478878720000\,\frac{a^{24}}{b^{24}}$

$2 + 94\,\frac{a}{b} + 2116\,\frac{a^{2}}{b^{2}} + 45540\,\frac{a^{3}}{b^{3}} + 935088\,\frac{a^{4}}{b^{4}} + 18276720\,\frac{a^{5}}{b^{5}} + 339181920\,\frac{a^{6}}{b^{6}} + 5959910880\,\frac{a^{7}}{b^{7}} + 98847302400\,\frac{a^{8}}{b^{8}} + 1542017917440\,\frac{a^{9}}{b^{9}} + 22537184947200\,\frac{a^{10}}{b^{10}} + 307217415859200\,\frac{a^{11}}{b^{11}} + 3885885151948800\,\frac{a^{12}}{b^{12}} + 45335326772736000\,\frac{a^{13}}{b^{13}} + 484440348942950400\,\frac{a^{14}}{b^{14}} + 4701921033858048000\,\frac{a^{15}}{b^{15}} + 41034947204579328000\,\frac{a^{16}}{b^{16}} + 318020840835489792000\,\frac{a^{17}}{b^{17}} + 2154334728240414720000\,\frac{a^{18}}{b^{18}} + 12495141423794405376000\,\frac{a^{19}}{b^{19}} + 60321372390731612160000\,\frac{a^{20}}{b^{20}} + 232668150649964789760000\,\frac{a^{21}}{b^{21}} + 672152435211009392640000\,\frac{a^{22}}{b^{22}} + 1292600836944248832000000\,\frac{a^{23}}{b^{23}} + 1240896803466478878720000\,\frac{a^{24}}{b^{24}}$

$3 + 138\,\frac{a}{b} + 3040\,\frac{a^{2}}{b^{2}} + 64020\,\frac{a^{3}}{b^{3}} + 1286208\,\frac{a^{4}}{b^{4}} + 24596880\,\frac{a^{5}}{b^{5}} + 446624640\,\frac{a^{6}}{b^{6}} + 7678994400\,\frac{a^{7}}{b^{7}} + 124633555200\,\frac{a^{8}}{b^{8}} + 1903025456640\,\frac{a^{9}}{b^{9}} + 27230282956800\,\frac{a^{10}}{b^{10}} +$

$363534591974400\,\frac{a^{11}}{b^{11}} + 4505374089216000\,\frac{a^{12}}{b^{12}} + 51530216145408000\,\frac{a^{13}}{b^{13}} + 540194353296998400\,\frac{a^{14}}{b^{14}} + 5147953068690432000\,\frac{a^{15}}{b^{15}} + 44157171448406016000\,\frac{a^{16}}{b^{16}} + 336754186298449920000\,\frac{a^{17}}{b^{17}} + 2248001455555215360000\,\frac{a^{18}}{b^{18}} + 12869808333053607936000\,\frac{a^{19}}{b^{19}} + 61445373118509219840000\,\frac{a^{20}}{b^{20}} + 234916152105520005120000\,\frac{a^{21}}{b^{21}} + 674400436666564608000000\,\frac{a^{22}}{b^{22}} + 1292600836944248832000000\,\frac{a^{23}}{b^{23}} + 1240896803466478878720000\,\frac{a^{24}}{b^{24}}$

$4 + 180\,\frac{a}{b} + 3880\,\frac{a^2}{b^2} + 79980\,\frac{a^3}{b^3} + 1573488\,\frac{a^4}{b^4} + 29480640\,\frac{a^5}{b^5} + 524764800\,\frac{a^6}{b^6} + 8851096800\,\frac{a^7}{b^7} + 141042988800\,\frac{a^8}{b^8} + 2116348093440\,\frac{a^9}{b^9} + 29790154598400\,\frac{a^{10}}{b^{10}} + 391693180032000\,\frac{a^{11}}{b^{11}} + 4786959969792000\,\frac{a^{12}}{b^{12}} + 54064489070592000\,\frac{a^{13}}{b^{13}} + 560468536698470400\,\frac{a^{14}}{b^{14}} + 5289872352500736000\,\frac{a^{15}}{b^{15}} + 45008687151267840000\,\frac{a^{16}}{b^{16}} + 341011764812759040000\,\frac{a^{17}}{b^{17}} + 2265031769612451840000\,\frac{a^{18}}{b^{18}} + 12920899275225317376000\,\frac{a^{19}}{b^{19}} + 61547555002852638720000\,\frac{a^{20}}{b^{20}} + 235018333989863424000000\,\frac{a^{21}}{b^{21}} + 674400436666564608000000\,\frac{a^{22}}{b^{22}} + 1292600836944248832000000\,\frac{a^{23}}{b^{23}} + 1240896803466478878720000\,\frac{a^{24}}{b^{24}}$

$5 + 220\,\frac{a}{b} + 4640\,\frac{a^2}{b^2} + 93660\,\frac{a^3}{b^3} + 1806048\,\frac{a^4}{b^4} + 33201600\,\frac{a^5}{b^5} + 580579200\,\frac{a^6}{b^6} + 9632498400\,\frac{a^7}{b^7} + 151201209600\,\frac{a^8}{b^8} + 2238246743040\,\frac{a^9}{b^9} + 31131039744000\,\frac{a^{10}}{b^{10}} + 405102031488000\,\frac{a^{11}}{b^{11}} + 4907639632896000\,\frac{a^{12}}{b^{12}} + 55029926375424000\,\frac{a^{13}}{b^{13}} + 567226597832294400\,\frac{a^{14}}{b^{14}} + 5330420719303680000\,\frac{a^{15}}{b^{15}} + 45211428985282560000\,\frac{a^{16}}{b^{16}} + 341822732148817920000\,\frac{a^{17}}{b^{17}} + 2267464671620628480000\,\frac{a^{18}}{b^{18}} + 12925765079241670656000\,\frac{a^{19}}{b^{19}} + 61552420806868992000000\,\frac{a^{20}}{b^{20}} + 235018333989863424000000\,\frac{a^{21}}{b^{21}} + 674400436666564608000000\,\frac{a^{22}}{b^{22}} + 1292600836944248832000000\,\frac{a^{23}}{b^{23}} + 1240896803466478878720000\,\frac{a^{24}}{b^{24}}$

$6 + 258\,\frac{a}{b} + 5324\,\frac{a^2}{b^2} + 105288\,\frac{a^3}{b^3} + 1992096\,\frac{a^4}{b^4} + 35992320\,\frac{a^5}{b^5} + 619649280\,\frac{a^6}{b^6} + 10140409440\,\frac{a^7}{b^7} + 157296142080\,\frac{a^8}{b^8} + 2305291000320\,\frac{a^9}{b^9} + 31801482316800\,\frac{a^{10}}{b^{10}} + 411136014643200\,\frac{a^{11}}{b^{11}} + 4955911498137600\,\frac{a^{12}}{b^{12}} + 55367829432115200\,\frac{a^{13}}{b^{13}} + 569254016172441600\,\frac{a^{14}}{b^{14}} + 5340557811004416000\,\frac{a^{15}}{b^{15}} + 45251977352085504000\,\frac{a^{16}}{b^{16}} + 341944377249226752000\,\frac{a^{17}}{b^{17}} + 2267707961821446144000\,\frac{a^{18}}{b^{18}} + 12926008369442488320000\,\frac{a^{19}}{b^{19}} + 61552420806868992000000\,\frac{a^{20}}{b^{20}} + 235018333989863424000000\,\frac{a^{21}}{b^{21}} + 674400436666564608000000\,\frac{a^{22}}{b^{22}} + 1292600836944248832000000\,\frac{a^{23}}{b^{23}} + 1240896803466478878720000\,\frac{a^{24}}{b^{24}}$

$7 + 294\,\frac{a}{b} + 5936\,\frac{a^2}{b^2} + 115080\,\frac{a^3}{b^3} + 2138976\,\frac{a^4}{b^4} + 38048640\,\frac{a^5}{b^5} + 646381440\,\frac{a^6}{b^6} + 10461195360\,\frac{a^7}{b^7} + 160824787200\,\frac{a^8}{b^8} + 2340577451520\,\frac{a^9}{b^9} + 32119060377600\,\frac{a^{10}}{b^{10}} + 413676639129600\,\frac{a^{11}}{b^{11}} + 4973695869542400\,\frac{a^{12}}{b^{12}} + 55474535660544000\,\frac{a^{13}}{b^{13}} + 569787547314585600\,\frac{a^{14}}{b^{14}} + 5342691935572992000\,\frac{a^{15}}{b^{15}} + 45258379725791232000\,\frac{a^{16}}{b^{16}} + 341957181996638208000\,\frac{a^{17}}{b^{17}} + 2267720766568857600000\,\frac{a^{18}}{b^{18}} + 12926008369442488320000\,\frac{a^{19}}{b^{19}} + 61552420806868992000000\,\frac{a^{20}}{b^{20}} + 235018333989863424000000\,\frac{a^{21}}{b^{21}} + 674400436666564608000000\,\frac{a^{22}}{b^{22}} +$

$12926008369442488320000000 \frac{a^{23}}{b^{23}} + 1240896803466478878720000 \frac{a^{24}}{b^{24}}$

$8 + 328 \frac{a}{b} + 6480 \frac{a^{2}}{b^{2}} + 123240 \frac{a^{3}}{b^{3}} + 2253216 \frac{a^{4}}{b^{4}} + 39533760 \frac{a^{5}}{b^{5}} + 664202880 \frac{a^{6}}{b^{6}} +$
$10657231200 \frac{a^{7}}{b^{7}} + 162785145600 \frac{a^{8}}{b^{8}} + 2358220677120 \frac{a^{9}}{b^{9}} + 32260206182400 \frac{a^{10}}{b^{10}} +$
$414664659763200 \frac{a^{11}}{b^{11}} + 4979623993344000 \frac{a^{12}}{b^{12}} + 55504176279552000 \frac{a^{13}}{b^{13}} +$
$569906109790617600 \frac{a^{14}}{b^{14}} + 5343047623001088000 \frac{a^{15}}{b^{15}} + 45259091100647424000 \frac{a^{16}}{b^{16}} +$
$341957893371494400000 \frac{a^{17}}{b^{17}} + 2267720766568857600000 \frac{a^{18}}{b^{18}} +$
$12926008369442488320000 \frac{a^{19}}{b^{19}} + 61552420806868992000000 \frac{a^{20}}{b^{20}} +$
$235018333989863424000000 \frac{a^{21}}{b^{21}} + 674400436666564608000000 \frac{a^{22}}{b^{22}} +$
$12926008369442488320000000 \frac{a^{23}}{b^{23}} + 1240896803466478878720000 \frac{a^{24}}{b^{24}}$

$9 + 360 \frac{a}{b} + 6960 \frac{a^{2}}{b^{2}} + 129960 \frac{a^{3}}{b^{3}} + 2340576 \frac{a^{4}}{b^{4}} + 40582080 \frac{a^{5}}{b^{5}} + 675734400 \frac{a^{6}}{b^{6}} +$
$10772546400 \frac{a^{7}}{b^{7}} + 163822982400 \frac{a^{8}}{b^{8}} + 2366523371520 \frac{a^{9}}{b^{9}} + 32318325043200 \frac{a^{10}}{b^{10}} +$
$415013372928000 \frac{a^{11}}{b^{11}} + 4981367559168000 \frac{a^{12}}{b^{12}} + 55511150542848000 \frac{a^{13}}{b^{13}} +$
$569927032580505600 \frac{a^{14}}{b^{14}} + 5343089468580864000 \frac{a^{15}}{b^{15}} + 45259132946227200000 \frac{a^{16}}{b^{16}} +$
$341957893371494400000 \frac{a^{17}}{b^{17}} + 2267720766568857600000 \frac{a^{18}}{b^{18}} +$
$12926008369442488320000 \frac{a^{19}}{b^{19}} + 61552420806868992000000 \frac{a^{20}}{b^{20}} +$
$235018333989863424000000 \frac{a^{21}}{b^{21}} + 674400436666564608000000 \frac{a^{22}}{b^{22}} +$
$12926008369442488320000000 \frac{a^{23}}{b^{23}} + 1240896803466478878720000 \frac{a^{24}}{b^{24}}$

$10 + 390 \frac{a}{b} + 7380 \frac{a^{2}}{b^{2}} + 135420 \frac{a^{3}}{b^{3}} + 2406096 \frac{a^{4}}{b^{4}} + 41302800 \frac{a^{5}}{b^{5}} + 682941600 \frac{a^{6}}{b^{6}} +$
$10837411200 \frac{a^{7}}{b^{7}} + 164341900800 \frac{a^{8}}{b^{8}} + 2370155800320 \frac{a^{9}}{b^{9}} + 32340119616000 \frac{a^{10}}{b^{10}} +$
$415122345792000 \frac{a^{11}}{b^{11}} + 4981803450624000 \frac{a^{12}}{b^{12}} + 55512458217216000 \frac{a^{13}}{b^{13}} +$
$569929647929241600 \frac{a^{14}}{b^{14}} + 5343092083929600000 \frac{a^{15}}{b^{15}} + 45259132946227200000 \frac{a^{16}}{b^{16}} +$
$341957893371494400000 \frac{a^{17}}{b^{17}} + 2267720766568857600000 \frac{a^{18}}{b^{18}} +$
$12926008369442488320000 \frac{a^{19}}{b^{19}} + 61552420806868992000000 \frac{a^{20}}{b^{20}} +$
$235018333989863424000000 \frac{a^{21}}{b^{21}} + 674400436666564608000000 \frac{a^{22}}{b^{22}} +$
$12926008369442488320000000 \frac{a^{23}}{b^{23}} + 1240896803466478878720000 \frac{a^{24}}{b^{24}}$

$11 + 418 \frac{a}{b} + 7744 \frac{a^{2}}{b^{2}} + 139788 \frac{a^{3}}{b^{3}} + 2454144 \frac{a^{4}}{b^{4}} + 41783280 \frac{a^{5}}{b^{5}} + 687265920 \frac{a^{6}}{b^{6}} +$
$10872005760 \frac{a^{7}}{b^{7}} + 164584062720 \frac{a^{8}}{b^{8}} + 2371608771840 \frac{a^{9}}{b^{9}} + 32347384473600 \frac{a^{10}}{b^{10}} +$
$415151405222400 \frac{a^{11}}{b^{11}} + 4981890628915200 \frac{a^{12}}{b^{12}} + 55512632573798400 \frac{a^{13}}{b^{13}} +$
$569929822285824000 \frac{a^{14}}{b^{14}} + 5343092083929600000 \frac{a^{15}}{b^{15}} + 45259132946227200000 \frac{a^{16}}{b^{16}} +$
$341957893371494400000 \frac{a^{17}}{b^{17}} + 2267720766568857600000 \frac{a^{18}}{b^{18}} +$
$12926008369442488320000 \frac{a^{19}}{b^{19}} + 61552420806868992000000 \frac{a^{20}}{b^{20}} +$
$235018333989863424000000 \frac{a^{21}}{b^{21}} + 674400436666564608000000 \frac{a^{22}}{b^{22}} +$
$12926008369442488320000000 \frac{a^{23}}{b^{23}} + 1240896803466478878720000 \frac{a^{24}}{b^{24}}$

$12 + 444 \frac{a}{b} + 8056 \frac{a^{2}}{b^{2}} + 143220 \frac{a^{3}}{b^{3}} + 2488464 \frac{a^{4}}{b^{4}} + 42092160 \frac{a^{5}}{b^{5}} + 689736960 \frac{a^{6}}{b^{6}} +$
$10889303040 \frac{a^{7}}{b^{7}} + 164687846400 \frac{a^{8}}{b^{8}} + 2372127690240 \frac{a^{9}}{b^{9}} + 32349460147200 \frac{a^{10}}{b^{10}} +$
$415157632243200 \frac{a^{11}}{b^{11}} + 4981903082956800 \frac{a^{12}}{b^{12}} + 55512645027840000 \frac{a^{13}}{b^{13}} +$
$569929822285824000 \frac{a^{14}}{b^{14}} + 5343092083929600000 \frac{a^{15}}{b^{15}} + 45259132946227200000 \frac{a^{16}}{b^{16}} +$

$341957893371494400000\,\frac{a^{17}}{b^{17}} + 226772076656885760000\,\frac{a^{18}}{b^{18}} + 1292600836944248832000\,\frac{a^{19}}{b^{19}} + 6155242080686899200000\,\frac{a^{20}}{b^{20}} + 23501833398986342400000\,\frac{a^{21}}{b^{21}} + 67440043666656460800000\,\frac{a^{22}}{b^{22}} + 129260083694424883200000\,\frac{a^{23}}{b^{23}} + 124089680346647887872000\,\frac{a^{24}}{b^{24}}$

$13 + 468\,\frac{a}{b} + 8320\,\frac{a^2}{b^2} + 145860\,\frac{a^3}{b^3} + 2512224\,\frac{a^4}{b^4} + 42282240\,\frac{a^5}{b^5} + 691067520\,\frac{a^6}{b^6} + 10897286400\,\frac{a^7}{b^7} + 164727763200\,\frac{a^8}{b^8} + 2372287357440\,\frac{a^9}{b^9} + 32349939148800\,\frac{a^{10}}{b^{10}} + 415158590246400\,\frac{a^{11}}{b^{11}} + 4981904040960000\,\frac{a^{12}}{b^{12}} + 55512645027840000\,\frac{a^{13}}{b^{13}} + 569929822285824000\,\frac{a^{14}}{b^{14}} + 5343092083929600000\,\frac{a^{15}}{b^{15}} + 45259132946227200000\,\frac{a^{16}}{b^{16}} + 341957893371494400000\,\frac{a^{17}}{b^{17}} + 226772076656885760000\,\frac{a^{18}}{b^{18}} + 1292600836944248832000\,\frac{a^{19}}{b^{19}} + 6155242080686899200000\,\frac{a^{20}}{b^{20}} + 23501833398986342400000\,\frac{a^{21}}{b^{21}} + 67440043666656460800000\,\frac{a^{22}}{b^{22}} + 129260083694424883200000\,\frac{a^{23}}{b^{23}} + 124089680346647887872000\,\frac{a^{24}}{b^{24}}$

$14 + 490\,\frac{a}{b} + 8540\,\frac{a^2}{b^2} + 147840\,\frac{a^3}{b^3} + 2528064\,\frac{a^4}{b^4} + 42393120\,\frac{a^5}{b^5} + 691732800\,\frac{a^6}{b^6} + 10900612800\,\frac{a^7}{b^7} + 164741068800\,\frac{a^8}{b^8} + 2372327274240\,\frac{a^9}{b^9} + 32350018982400\,\frac{a^{10}}{b^{10}} + 415158670080000\,\frac{a^{11}}{b^{11}} + 4981904040960000\,\frac{a^{12}}{b^{12}} + 55512645027840000\,\frac{a^{13}}{b^{13}} + 569929822285824000\,\frac{a^{14}}{b^{14}} + 5343092083929600000\,\frac{a^{15}}{b^{15}} + 45259132946227200000\,\frac{a^{16}}{b^{16}} + 341957893371494400000\,\frac{a^{17}}{b^{17}} + 226772076656885760000\,\frac{a^{18}}{b^{18}} + 1292600836944248832000\,\frac{a^{19}}{b^{19}} + 6155242080686899200000\,\frac{a^{20}}{b^{20}} + 23501833398986342400000\,\frac{a^{21}}{b^{21}} + 67440043666656460800000\,\frac{a^{22}}{b^{22}} + 129260083694424883200000\,\frac{a^{23}}{b^{23}} + 124089680346647887872000\,\frac{a^{24}}{b^{24}}$

$15 + 510\,\frac{a}{b} + 8720\,\frac{a^2}{b^2} + 149280\,\frac{a^3}{b^3} + 2538144\,\frac{a^4}{b^4} + 42453600\,\frac{a^5}{b^5} + 692035200\,\frac{a^6}{b^6} + 10901822400\,\frac{a^7}{b^7} + 164744697600\,\frac{a^8}{b^8} + 2372334531840\,\frac{a^9}{b^9} + 32350026240000\,\frac{a^{10}}{b^{10}} + 415158670080000\,\frac{a^{11}}{b^{11}} + 4981904040960000\,\frac{a^{12}}{b^{12}} + 55512645027840000\,\frac{a^{13}}{b^{13}} + 569929822285824000\,\frac{a^{14}}{b^{14}} + 5343092083929600000\,\frac{a^{15}}{b^{15}} + 45259132946227200000\,\frac{a^{16}}{b^{16}} + 341957893371494400000\,\frac{a^{17}}{b^{17}} + 226772076656885760000\,\frac{a^{18}}{b^{18}} + 1292600836944248832000\,\frac{a^{19}}{b^{19}} + 6155242080686899200000\,\frac{a^{20}}{b^{20}} + 23501833398986342400000\,\frac{a^{21}}{b^{21}} + 67440043666656460800000\,\frac{a^{22}}{b^{22}} + 129260083694424883200000\,\frac{a^{23}}{b^{23}} + 124089680346647887872000\,\frac{a^{24}}{b^{24}}$

$16 + 528\,\frac{a}{b} + 8864\,\frac{a^2}{b^2} + 150288\,\frac{a^3}{b^3} + 2544192\,\frac{a^4}{b^4} + 42483840\,\frac{a^5}{b^5} + 692156160\,\frac{a^6}{b^6} + 10902185280\,\frac{a^7}{b^7} + 164745423360\,\frac{a^8}{b^8} + 2372335257600\,\frac{a^9}{b^9} + 32350026240000\,\frac{a^{10}}{b^{10}} + 415158670080000\,\frac{a^{11}}{b^{11}} + 4981904040960000\,\frac{a^{12}}{b^{12}} + 55512645027840000\,\frac{a^{13}}{b^{13}} + 569929822285824000\,\frac{a^{14}}{b^{14}} + 5343092083929600000\,\frac{a^{15}}{b^{15}} + 45259132946227200000\,\frac{a^{16}}{b^{16}} + 341957893371494400000\,\frac{a^{17}}{b^{17}} + 226772076656885760000\,\frac{a^{18}}{b^{18}} + 1292600836944248832000\,\frac{a^{19}}{b^{19}} + 6155242080686899200000\,\frac{a^{20}}{b^{20}} + 23501833398986342400000\,\frac{a^{21}}{b^{21}} + 67440043666656460800000\,\frac{a^{22}}{b^{22}} + 129260083694424883200000\,\frac{a^{23}}{b^{23}} + 124089680346647887872000\,\frac{a^{24}}{b^{24}}$

$17 + 544\,\frac{a}{b} + 8976\,\frac{a^2}{b^2} + 150960\,\frac{a^3}{b^3} + 2547552\,\frac{a^4}{b^4} + 42497280\,\frac{a^5}{b^5} + 692196480\,\frac{a^6}{b^6} +$

$10902265920\,\frac{a^7}{b^7} + 164745504000\,\frac{a^8}{b^8} + 2372335257600\,\frac{a^9}{b^9} + 32350026240000\,\frac{a^{10}}{b^{10}} +$
$415158670080000\,\frac{a^{11}}{b^{11}} + 4981904040960000\,\frac{a^{12}}{b^{12}} + 55512645027840000\,\frac{a^{13}}{b^{13}} +$
$569929822285824000\,\frac{a^{14}}{b^{14}} + 5343092083929600000\,\frac{a^{15}}{b^{15}} + 45259132946227200000\,\frac{a^{16}}{b^{16}} +$
$341957893371494400000\,\frac{a^{17}}{b^{17}} + 2267720766568857600000\,\frac{a^{18}}{b^{18}} +$
$12926008369442488320000\,\frac{a^{19}}{b^{19}} + 61552420806868992000000\,\frac{a^{20}}{b^{20}} +$
$235018333989863424000000\,\frac{a^{21}}{b^{21}} + 674400436666564608000000\,\frac{a^{22}}{b^{22}} +$
$1292600836944248832000000\,\frac{a^{23}}{b^{23}} + 1240896803466478878720000\,\frac{a^{24}}{b^{24}}$

$18 + 558\,\frac{a}{b} + 9060\,\frac{a^2}{b^2} + 151380\,\frac{a^3}{b^3} + 2549232\,\frac{a^4}{b^4} + 42502320\,\frac{a^5}{b^5} + 692206560\,\frac{a^6}{b^6} +$
$10902276000\,\frac{a^7}{b^7} + 164745504000\,\frac{a^8}{b^8} + 2372335257600\,\frac{a^9}{b^9} + 32350026240000\,\frac{a^{10}}{b^{10}} +$
$415158670080000\,\frac{a^{11}}{b^{11}} + 4981904040960000\,\frac{a^{12}}{b^{12}} + 55512645027840000\,\frac{a^{13}}{b^{13}} +$
$569929822285824000\,\frac{a^{14}}{b^{14}} + 5343092083929600000\,\frac{a^{15}}{b^{15}} + 45259132946227200000\,\frac{a^{16}}{b^{16}} +$
$341957893371494400000\,\frac{a^{17}}{b^{17}} + 2267720766568857600000\,\frac{a^{18}}{b^{18}} +$
$12926008369442488320000\,\frac{a^{19}}{b^{19}} + 61552420806868992000000\,\frac{a^{20}}{b^{20}} +$
$235018333989863424000000\,\frac{a^{21}}{b^{21}} + 674400436666564608000000\,\frac{a^{22}}{b^{22}} +$
$1292600836944248832000000\,\frac{a^{23}}{b^{23}} + 1240896803466478878720000\,\frac{a^{24}}{b^{24}}$

$19 + 570\,\frac{a}{b} + 9120\,\frac{a^2}{b^2} + 151620\,\frac{a^3}{b^3} + 2549952\,\frac{a^4}{b^4} + 42503760\,\frac{a^5}{b^5} + 692208000\,\frac{a^6}{b^6} +$
$10902276000\,\frac{a^7}{b^7} + 164745504000\,\frac{a^8}{b^8} + 2372335257600\,\frac{a^9}{b^9} + 32350026240000\,\frac{a^{10}}{b^{10}} +$
$415158670080000\,\frac{a^{11}}{b^{11}} + 4981904040960000\,\frac{a^{12}}{b^{12}} + 55512645027840000\,\frac{a^{13}}{b^{13}} +$
$569929822285824000\,\frac{a^{14}}{b^{14}} + 5343092083929600000\,\frac{a^{15}}{b^{15}} + 45259132946227200000\,\frac{a^{16}}{b^{16}} +$
$341957893371494400000\,\frac{a^{17}}{b^{17}} + 2267720766568857600000\,\frac{a^{18}}{b^{18}} +$
$12926008369442488320000\,\frac{a^{19}}{b^{19}} + 61552420806868992000000\,\frac{a^{20}}{b^{20}} +$
$235018333989863424000000\,\frac{a^{21}}{b^{21}} + 674400436666564608000000\,\frac{a^{22}}{b^{22}} +$
$1292600836944248832000000\,\frac{a^{23}}{b^{23}} + 1240896803466478878720000\,\frac{a^{24}}{b^{24}}$

$20 + 580\,\frac{a}{b} + 9160\,\frac{a^2}{b^2} + 151740\,\frac{a^3}{b^3} + 2550192\,\frac{a^4}{b^4} + 42504000\,\frac{a^5}{b^5} + 692208000\,\frac{a^6}{b^6} +$
$10902276000\,\frac{a^7}{b^7} + 164745504000\,\frac{a^8}{b^8} + 2372335257600\,\frac{a^9}{b^9} + 32350026240000\,\frac{a^{10}}{b^{10}} +$
$415158670080000\,\frac{a^{11}}{b^{11}} + 4981904040960000\,\frac{a^{12}}{b^{12}} + 55512645027840000\,\frac{a^{13}}{b^{13}} +$
$569929822285824000\,\frac{a^{14}}{b^{14}} + 5343092083929600000\,\frac{a^{15}}{b^{15}} + 45259132946227200000\,\frac{a^{16}}{b^{16}} +$
$341957893371494400000\,\frac{a^{17}}{b^{17}} + 2267720766568857600000\,\frac{a^{18}}{b^{18}} +$
$12926008369442488320000\,\frac{a^{19}}{b^{19}} + 61552420806868992000000\,\frac{a^{20}}{b^{20}} +$
$235018333989863424000000\,\frac{a^{21}}{b^{21}} + 674400436666564608000000\,\frac{a^{22}}{b^{22}} +$
$1292600836944248832000000\,\frac{a^{23}}{b^{23}} + 1240896803466478878720000\,\frac{a^{24}}{b^{24}}$

$21 + 588\,\frac{a}{b} + 9184\,\frac{a^2}{b^2} + 151788\,\frac{a^3}{b^3} + 2550240\,\frac{a^4}{b^4} + 42504000\,\frac{a^5}{b^5} + 692208000\,\frac{a^6}{b^6} +$
$10902276000\,\frac{a^7}{b^7} + 164745504000\,\frac{a^8}{b^8} + 2372335257600\,\frac{a^9}{b^9} + 32350026240000\,\frac{a^{10}}{b^{10}} +$
$415158670080000\,\frac{a^{11}}{b^{11}} + 4981904040960000\,\frac{a^{12}}{b^{12}} + 55512645027840000\,\frac{a^{13}}{b^{13}} +$
$569929822285824000\,\frac{a^{14}}{b^{14}} + 5343092083929600000\,\frac{a^{15}}{b^{15}} + 45259132946227200000\,\frac{a^{16}}{b^{16}} +$
$341957893371494400000\,\frac{a^{17}}{b^{17}} + 2267720766568857600000\,\frac{a^{18}}{b^{18}} +$
$12926008369442488320000\,\frac{a^{19}}{b^{19}} + 61552420806868992000000\,\frac{a^{20}}{b^{20}} +$

$2350183339898634240000000\,\frac{a^{21}}{b^{21}} + 6744004366665646080000000\,\frac{a^{22}}{b^{22}} + 12926008369442488320000000\,\frac{a^{23}}{b^{23}} + 12408968034664788787200000\,\frac{a^{24}}{b^{24}}$

$22 + 594\,\frac{a}{b} + 9196\,\frac{a^2}{b^2} + 151800\,\frac{a^3}{b^3} + 2550240\,\frac{a^4}{b^4} + 42504000\,\frac{a^5}{b^5} + 692208000\,\frac{a^6}{b^6} + 10902276000\,\frac{a^7}{b^7} + 164745504000\,\frac{a^8}{b^8} + 2372335257600\,\frac{a^9}{b^9} + 32350026240000\,\frac{a^{10}}{b^{10}} + 415158670080000\,\frac{a^{11}}{b^{11}} + 4981904040960000\,\frac{a^{12}}{b^{12}} + 55512645027840000\,\frac{a^{13}}{b^{13}} + 569929822285824000\,\frac{a^{14}}{b^{14}} + 5343092083929600000\,\frac{a^{15}}{b^{15}} + 45259132946227200000\,\frac{a^{16}}{b^{16}} + 341957893371494400000\,\frac{a^{17}}{b^{17}} + 2267720766568857600000\,\frac{a^{18}}{b^{18}} + 12926008369442488320000\,\frac{a^{19}}{b^{19}} + 61552420806868992000000\,\frac{a^{20}}{b^{20}} + 2350183339898634240000000\,\frac{a^{21}}{b^{21}} + 6744004366665646080000000\,\frac{a^{22}}{b^{22}} + 12926008369442488320000000\,\frac{a^{23}}{b^{23}} + 12408968034664788787200000\,\frac{a^{24}}{b^{24}}$

$23 + 598\,\frac{a}{b} + 9200\,\frac{a^2}{b^2} + 151800\,\frac{a^3}{b^3} + 2550240\,\frac{a^4}{b^4} + 42504000\,\frac{a^5}{b^5} + 692208000\,\frac{a^6}{b^6} + 10902276000\,\frac{a^7}{b^7} + 164745504000\,\frac{a^8}{b^8} + 2372335257600\,\frac{a^9}{b^9} + 32350026240000\,\frac{a^{10}}{b^{10}} + 415158670080000\,\frac{a^{11}}{b^{11}} + 4981904040960000\,\frac{a^{12}}{b^{12}} + 55512645027840000\,\frac{a^{13}}{b^{13}} + 569929822285824000\,\frac{a^{14}}{b^{14}} + 5343092083929600000\,\frac{a^{15}}{b^{15}} + 45259132946227200000\,\frac{a^{16}}{b^{16}} + 341957893371494400000\,\frac{a^{17}}{b^{17}} + 2267720766568857600000\,\frac{a^{18}}{b^{18}} + 12926008369442488320000\,\frac{a^{19}}{b^{19}} + 61552420806868992000000\,\frac{a^{20}}{b^{20}} + 2350183339898634240000000\,\frac{a^{21}}{b^{21}} + 6744004366665646080000000\,\frac{a^{22}}{b^{22}} + 12926008369442488320000000\,\frac{a^{23}}{b^{23}} + 12408968034664788787200000\,\frac{a^{24}}{b^{24}}$

$24 + 600\,\frac{a}{b} + 9200\,\frac{a^2}{b^2} + 151800\,\frac{a^3}{b^3} + 2550240\,\frac{a^4}{b^4} + 42504000\,\frac{a^5}{b^5} + 692208000\,\frac{a^6}{b^6} + 10902276000\,\frac{a^7}{b^7} + 164745504000\,\frac{a^8}{b^8} + 2372335257600\,\frac{a^9}{b^9} + 32350026240000\,\frac{a^{10}}{b^{10}} + 415158670080000\,\frac{a^{11}}{b^{11}} + 4981904040960000\,\frac{a^{12}}{b^{12}} + 55512645027840000\,\frac{a^{13}}{b^{13}} + 569929822285824000\,\frac{a^{14}}{b^{14}} + 5343092083929600000\,\frac{a^{15}}{b^{15}} + 45259132946227200000\,\frac{a^{16}}{b^{16}} + 341957893371494400000\,\frac{a^{17}}{b^{17}} + 2267720766568857600000\,\frac{a^{18}}{b^{18}} + 12926008369442488320000\,\frac{a^{19}}{b^{19}} + 61552420806868992000000\,\frac{a^{20}}{b^{20}} + 2350183339898634240000000\,\frac{a^{21}}{b^{21}} + 6744004366665646080000000\,\frac{a^{22}}{b^{22}} + 12926008369442488320000000\,\frac{a^{23}}{b^{23}} + 12408968034664788787200000\,\frac{a^{24}}{b^{24}}$

$25 + 600\,\frac{a}{b} + 9200\,\frac{a^2}{b^2} + 151800\,\frac{a^3}{b^3} + 2550240\,\frac{a^4}{b^4} + 42504000\,\frac{a^5}{b^5} + 692208000\,\frac{a^6}{b^6} + 10902276000\,\frac{a^7}{b^7} + 164745504000\,\frac{a^8}{b^8} + 2372335257600\,\frac{a^9}{b^9} + 32350026240000\,\frac{a^{10}}{b^{10}} + 415158670080000\,\frac{a^{11}}{b^{11}} + 4981904040960000\,\frac{a^{12}}{b^{12}} + 55512645027840000\,\frac{a^{13}}{b^{13}} + 569929822285824000\,\frac{a^{14}}{b^{14}} + 5343092083929600000\,\frac{a^{15}}{b^{15}} + 45259132946227200000\,\frac{a^{16}}{b^{16}} + 341957893371494400000\,\frac{a^{17}}{b^{17}} + 2267720766568857600000\,\frac{a^{18}}{b^{18}} + 12926008369442488320000\,\frac{a^{19}}{b^{19}} + 61552420806868992000000\,\frac{a^{20}}{b^{20}} + 2350183339898634240000000\,\frac{a^{21}}{b^{21}} + 6744004366665646080000000\,\frac{a^{22}}{b^{22}} + 12926008369442488320000000\,\frac{a^{23}}{b^{23}} + 12408968034664788787200000\,\frac{a^{24}}{b^{24}}$

A.2 Resultados Obtidos Considerando-se a Conectividade

4 módulos

$$\left[\begin{array}{c} 1+12\,\frac{a^3f^3}{b^3}+12\,\frac{a^2f^2}{b^2}+6\,\frac{af}{b} \\ 2+12\,\frac{a^3f^3}{b^3}+16\,\frac{a^2f^2}{b^2}+10\,\frac{af}{b} \\ 3+12\,\frac{a^3f^3}{b^3}+16\,\frac{a^2f^2}{b^2}+12\,\frac{af}{b} \\ 4+12\,\frac{a^3f^3}{b^3}+16\,\frac{a^2f^2}{b^2}+12\,\frac{af}{b} \end{array}\right]$$

5 módulos

$$\left[\begin{array}{c} 1+48\,\frac{a^4f^4}{b^4}+48\,\frac{a^3f^3}{b^3}+24\,\frac{a^2f^2}{b^2}+8\,\frac{af}{b} \\ 2+48\,\frac{a^4f^4}{b^4}+60\,\frac{a^3f^3}{b^3}+36\,\frac{a^2f^2}{b^2}+14\,\frac{af}{b} \\ 3+48\,\frac{a^4f^4}{b^4}+60\,\frac{a^3f^3}{b^3}+40\,\frac{a^2f^2}{b^2}+18\,\frac{af}{b} \\ 4+48\,\frac{a^4f^4}{b^4}+60\,\frac{a^3f^3}{b^3}+40\,\frac{a^2f^2}{b^2}+20\,\frac{af}{b} \\ 5+48\,\frac{a^4f^4}{b^4}+60\,\frac{a^3f^3}{b^3}+40\,\frac{a^2f^2}{b^2}+20\,\frac{af}{b} \end{array}\right]$$

6 módulos

$$\left[\begin{array}{c} 1+240\,\frac{a^5f^5}{b^5}+240\,\frac{a^4f^4}{b^4}+120\,\frac{a^3f^3}{b^3}+40\,\frac{a^2f^2}{b^2}+10\,\frac{af}{b} \\ 2+240\,\frac{a^5f^5}{b^5}+288\,\frac{a^4f^4}{b^4}+168\,\frac{a^3f^3}{b^3}+64\,\frac{a^2f^2}{b^2}+18\,\frac{af}{b} \\ 3+240\,\frac{a^5f^5}{b^5}+288\,\frac{a^4f^4}{b^4}+180\,\frac{a^3f^3}{b^3}+76\,\frac{a^2f^2}{b^2}+24\,\frac{af}{b} \\ 4+240\,\frac{a^5f^5}{b^5}+288\,\frac{a^4f^4}{b^4}+180\,\frac{a^3f^3}{b^3}+80\,\frac{a^2f^2}{b^2}+28\,\frac{af}{b} \\ 5+240\,\frac{a^5f^5}{b^5}+288\,\frac{a^4f^4}{b^4}+180\,\frac{a^3f^3}{b^3}+80\,\frac{a^2f^2}{b^2}+30\,\frac{af}{b} \\ 6+240\,\frac{a^5f^5}{b^5}+288\,\frac{a^4f^4}{b^4}+180\,\frac{a^3f^3}{b^3}+80\,\frac{a^2f^2}{b^2}+30\,\frac{af}{b} \end{array}\right]$$