Monitoramento de métricas de código-fonte em projetos de software livre

Paulo Roberto Miranda Meirelles

Tese apresentada
ao
Instituto de Matemática e Estatística
da
Universidade de São Paulo
para
obtenção do título
de
Doutor em Ciências

Programa: Ciência da Computação
Orientador: Prof. Dr. Fabio Kon
Coorientador: Prof. Dr. Carlos Denner Santos Jr.

Este trabalho foi financiado pelo Conselho Nacional de Pesquisa (CNPq),
fez parte do Projeto Qualipso e do Núcleo NAPSoL-PRP-USP.

São Paulo, maio de 2013.

Monitoramento de métricas de código-fonte em projetos de software livre

Esta tese contém as correções e alterações sugeridas
pela Comissão Julgadora durante a defesa realizada por
Paulo Roberto Miranda Meirelles em 20/05/2013.
O texto original encontra-se disponível no
Instituto de Matemática e Estatística
da Universidade de São Paulo.

Comissão Julgadora:

- Prof. Dr. Fabio Kon (orientador) - IME-USP
- Prof. Dr. Alfredo Goldman - IME-USP
- Prof. Dr. José Carlos Maldonado - ICMC-USP
- Prof. Dr. Marco Chaim - EACH-USP
- Prof. Dr. Eduardo Guerra - INPE

Agradecimentos

Agradeço ao professor Fabio Kon por apostar em mim como seu aluno de doutorado, pela orientação comprometida e pela frutífera (e ainda próspera) relação que estabelecemos ao longo desses anos. Ao meu coorientador, professor Carlos Denner pela a interação produtiva que tivemos, pela ajuda com meu doutorado sanduíche na SIUC e pelo suporte na UnB. Ao Antonio Terceiro, pela parceria fundamental durante este doutorado e, conquentemente, à sua orientadora Christina Chavez pelos trabalhos produzidos em conjunto. Ao professor Alfredo Goldman pelo aprendizado e colaboração dentro da disciplina Laboratório de Programação Extrema. Ao professor José Carlos Maldonado pela experiência durante o projeto QualiPSo e continuidade de nossa rica interação desde então, em especial com seu doutorando Rafael Messias, a quem também agradeço a parceria. Ao professor John Pearson pelo acolhimento na Southern Illinois University Carbondale (SIUC). Ainda, no IME-USP, tive a oportunidade de trabalhar com alunos brilhantes, tanto do ponto de vista técnico quanto humano. Foi uma honra ter colaborado, mesmo que minimamente, com a formação desses alunos, que foram essenciais durante a evolução deste doutorado: Carlos Morais, João Machini, Diego Araújo, João Marco da Silva, Alessandro Palmeira, Lucianna Almeida e Rafael Manzo. Também, agradeço aos amigos de laboratório pelo conhecimento compartilhado ao longo desses anos, em particular, representando todos aqui, ao (super) Nelson Lago. Agradeço especialmente à minha mãe, Nizete, e aos meus 3 irmãos, Amanda, Marina e Marcelo, que entenderam a distância e me apoiaram sempre que possível. Por fim, à minha esposa, Larissa Negreiros, pelo companherismo e carinhoso apoio durante essa jornada. Aproveito o espaço para dedicar este trabalho e registrar o meu eterno agradecimento aos professores (e educadores) Hilário, Verônica e Wanda, da Escola Estadual Sebastião Fernandes de Oliveira, em Natal-RN, pois sem o ensino e dedicação deles, acima da média no contexto de uma escola pública, eu não teria dado os primeiros passos para chegar nesta tese.

Resumo

Nesta tese de doutorado, apresentamos uma abordagem para a observação das métricas de código-fonte, estudando-as através de suas distribuições e associações, além de discutir as relações de causalidade e implicações práticas-gerenciais para monitoramento das mesmas. Em nossos estudos avaliamos a distribuição e correlações dos valores das métricas de 38 projetos de software livre, dentre os com mais contribuidores ativos em seus repositórios. Para tal, coletamos e analisamos os valores para cada métrica em mais de 344.872 classes e módulos dos projetos avaliados. Complementarmente, para mostrarmos a utilidade do monitoramento de métricas, descrevemos uma extensão e adaptação do modelo de causalidade do conceito de atratividade de projetos de software livre, que indica uma relação estatística entre os valores das métricas de código-fonte e a quantidade de downloads, contribuidores e atualizações (*commits*) nos repositórios dos projetos. Para isso, realizamos estudos empíricos com milhares de projetos de software livre. Do ponto de vista prático, também contribuímos com um conjunto de ferramentas inovador para a automação da avaliação de projetos de software livre, com ênfase nos estudos e na seleção de métricas, o que permite a análise de código-fonte de acordo com a percepção de qualidade das comunidades de software livre. Entre as principais contribuições desta tese está uma análise detalhada, em relação ao comportamento, valores e estudos de caso, de 15 métricas de código-fonte, o que representa um avanço em comparação a literatura relacionada ao ampliar o número de métricas avaliadas e propor uma abordagem que visa diminuir as contradições das análises das métricas.
Palavras-chave: métricas de código-fonte, software livre, engenharia de software experimental.

Abstract

In this Ph.D dissertation we present an approach about source code metrics tracking. We have researched source code metrics distributions and associations to discuss their causality and management-practices implications. Our studies have assessed distributions and correlations of source code metric values on 38 free software projects, which have a lot of activated contributors in their repositories. We have collected and analyzed metrics from 344,872 classes and modules of about 38 free software projects. Additionally, to show how it is useful to track source code metrics, we have extended the model of free software attractiveness to include source code metrics. Our technical attractiveness model indicates a statistical relationship between source code metrics and number of downloads, contributors, and commits in the analyzed free software repositories. For that, we have conducted empirical studies with 8,450 free software projects. From a practical point of view, we have contributed with a set of innovative tools for automated evaluation of free software projects. Our tool allow the analyses of source code metrics that mirror quality perceptions from the free software communities point of view.
Keywords: source code metrics, free and open source software, experimental software engineering.

Sumário

Resumo iii

Abstract v

Lista de Abreviaturas xi

Lista de Figuras xiii

Lista de Tabelas xv

1 Introdução 1
- 1.1 Objetivos . . . 3
  - 1.1.1 Questões de pesquisa . . . 3
- 1.2 Organização do Trabalho . . . 4

2 Software Livre 5
- 2.1 Desenvolvimento de Software Livre . . . 6
- 2.2 Atratividade de Projetos de Software Livre . . . 8

3 Métricas de Software 13
- 3.1 Métricas de código-fonte . . . 14
  - 3.1.1 Por que métricas de código-fonte? . . . 15
  - 3.1.2 Métricas selecionadas . . . 16

4 Estudos 21
- 4.1 Trabalhos relacionados . . . 22
- 4.2 Relacionamento entre métricas de código-fonte e a atratividade de projetos de software livre . . . 25
  - 4.2.1 Amostra de projetos e coleta dos dados . . . 25
  - 4.2.2 Variáveis . . . 26
  - 4.2.3 Resultados . . . 27
- 4.3 Distribuição e valores frequentes das métricas em projetos de software livre . . . 29
  - 4.3.1 Planejamento dos Estudos . . . 31
  - 4.3.2 Resultados . . . 33
  - 4.3.3 Efeito recíproco entre métricas do código e atividades de desenvolvimento . . . 71

5 Ferramentas **75**
5.1 Contexto do desenvolvimento das ferramentas . . . 75
5.2 Analizo . . . 77
5.3 Kalibro Metrics . . . 79
5.4 Mezuro . . . 81
5.4.1 Funcionalidade . . . 83
5.5 Continuidade do Projeto Mezuro . . . 93

6 Conclusão **95**
6.1 Contribuições . . . 95
6.2 Limitações . . . 96
6.3 Outras contribuições . . . 97
6.3.1 Publicações . . . 97
6.3.2 Projetos . . . 98
6.3.3 Orientações . . . 99
6.4 Trabalhos futuros . . . 99

A Levantamento das métricas de código-fonte **101**
A.1 Métricas tradicionais . . . 101
A.1.1 Métricas de tamanho . . . 102
A.1.2 Métricas de complexidade . . . 102
A.1.3 Métricas de manutenibilidade . . . 103
A.1.4 Outras métricas tradicionais . . . 103
A.2 Métricas de orientação a objetos . . . 104
A.2.1 Métricas de classes . . . 104
A.2.2 Métricas de métodos . . . 106
A.2.3 Métricas de acoplamento . . . 106
A.2.4 Métricas de herança . . . 106
A.2.5 Métricas do sistema . . . 107

B Arquitetura do Projeto Mezuro **109**
B.1 Interface de Programação . . . 109
B.1.1 Métricas e resultados . . . 110
B.1.2 Interpretação e configuração . . . 112
B.1.3 Processamento e acompanhamento . . . 115
B.1.4 Preferências . . . 117
B.2 Interface do Serviço . . . 118
B.2.1 Ponto de acesso de ferramentas base . . . 119
B.2.2 Pontos de acesso de cadastros . . . 120
B.2.3 Pontos de acesso de resultados . . . 121
B.3 Organização Interna . . . 122
B.3.1 Acesso e transferência de dados . . . 123
B.3.2 Abstração das entidades . . . 125
B.3.3 Concorrência . . . 127

B.3.4 Processamento 129

Referências Bibliográficas **133**

Lista de Abreviaturas

ACC	Conexões aferentes (*Afferent Connections per Class*).
ACCM	Média da complexidade ciclomática dos métodos (*Average Cyclomatic Complexity per Method*).
ACC	Conexões aferentes (*Afferent Connections per Class*).
ACCM	Média da complexidade ciclomática dos métodos (*Average Cyclomatic Complexity per Method*).
AMLOC	Média de linhas de código por método (*Average Method Lines Of Code*).
ANPM	Média do número de parâmetros por método (*Average Number of Parameters per Method*).
API	Interface de programação de aplicativos (*Application Programming Interface*).
CBO	Acoplamento entre objetos (*Coupling Between Objects*).
COF	Fator de acoplamento (*COupling Factor*).
CSS	Linguagem de estilo em cascata (*Cascading Style Sheets*).
CSV	Valores separados por vírgula, formato de arquivo (*Comma-Separated Values*).
DAO	Objeto de acesso a dados (*Data Access Object*).
DIT	Profundidade na árvore de herança (*Depth of Inheritance Tree*).
DTO	Objeto de transferência de dados (*Data Transfer Object*).
FP	Pontos de função (*Function Points*).
GQM	Meta-pergunta-métrica, abordagem para definição de métricas (*Goal Question Metric*).
HTTP	Protocolo de transferência de hipertexto (*HyperText Transfer Protocol*).
IDE	Ambiente integrado para desenvolvimento de software (*Integrated Development Environment*).
JAX-WS	Java API for XML Web Services.
JDBC	Java Database Connectivity.
JPA	Java Persistence API.
LCOM4	Falta de coesão entre métodos, versão 4 (*Lack of COhesion between Methods*).
LOC	Linhas de código (*Lines Of Code*).
NOM	Número de métodos (*Number Of Methods*).
NOP	Número de parâmetros (*Number Of Parameters*).
NOC	Número de subclasses (*Number Of Children*).
NPA	Número de atributos públicos (*Number of Public Attributes*).
OCP	Princípio aberto/fechado (*Open/Closed Principle*).
RFC	Resposta para uma classe (*Response For a Class*).
RGB	Vermelho-verde-azul, um padrão para descrição de cores (*Red-Green-Blue*).
SC	Complexidade estrutural (*Structural Complexity*).
SOAP	Protocolo simples de acesso a objetos (*Simple Object Access Protocol*).
SRP	Princípio da responsabilidade única (*Single Responsibility Principle*).
TDD	Desenvolvimento dirigido por testes (*Test Driven Development*).

UML	Linguagem de modelagem unificada (*Unified Modeling Language*).
URL	Localizador-padrão de recursos (*Uniform Resource Location*).
WS	Serviço Web (*Web Service*).
WSDL	Linguagem de descrição de serviços Web (*Web Service Description Language*).
XML	Linguagem de marcação extensível (*Extensible Markup Language*).
YAML	*YAML Ain'T Markup Language*.

Lista de Figuras

2.1 Modelo de atratividade – adaptado de Santos Jr. *et al.* (2010). 9
2.2 Modelo de atratividade: inserção de atributos de código-fonte. 10
2.3 Atributos de código-fonte e indicadores de atratividade. 10

4.1 Distribuição da métrica ACC nos códigos do Chrome e do Firefox: Pareto Type II e Percentis. 34
4.2 Distribuição da métrica ACCM no código do Linux e no Free BSD: Weibull e Percentis 37
4.3 Distribuição da métrica AMLOC no código do Firefox e Chrome: Pareto Tipo 2 e percentis 40
4.4 Distribuição da métrica ANPM: Percentis 43
4.5 Distribuição da métrica CBO no código do Eclipse e do Open JDK: Pareto Type II e percentis 46
4.6 Distribuição da métrica SC no código Eclipse e Open JDK: Pareto Type II e Percentis. 68
4.7 Modelo do estudo sobre o efeitos recíproco entre métricas e atratividade. 71
4.8 Seleção de métricas: dendrograma das correlações. 72
4.9 Seleção de métricas: dendrograma das correlações. 72
4.10 Seleção de métricas: dendrograma das correlações. 73

5.1 Interação e integração das ferramentas do projeto Mezuro. 76
5.2 Painel de controle de uma comunidade na rede Mezuro. 83
5.3 Cadastrando um novo projeto no Mezuro. 84
5.4 Adicionando um repositório ao projeto no Mezuro. 84
5.5 Cadastrando as informações do repositório no projeto do Mezuro. 85
5.6 Visualizando a coleta das métricas no Mezuro. 85
5.7 Visualizando os dados do processo de coleta das métricas 86
5.8 Visualizando os resultados de cada uma das métricas. 86
5.9 Visualizando o gráfico com o histórico das métricas. 87
5.10 Painel de controle de um usuário na rede Mezuro. 88
5.11 Criando uma nova configuração do Mezuro. 88
5.12 Adicionando métricas à configuração do Mezuro. 89
5.13 Selecionando uma ferramenta base integrada ao Mezuro. 89
5.14 Selecionando métricas da Analizo numa configuração do Mezuro. 90
5.15 Selecionando métricas do Checkstyle numa configuração do Mezuro. 90
5.16 Configurando uma métrica. 91
5.17 Definindo um intervalo para uma métrica. 91

5.18 Métrica configurada com um intervalo cadastrado. 92
5.19 Configurando um grupo de leitura e interpretação de intervalos. 92
5.20 Definindo um tipo de leitura. 93

B.1 Módulo e granularidade 110
B.2 Módulo e granularidade 111
B.3 Métricas 112
B.4 Ferramenta base - coletor de métricas 112
B.5 Grupo de leituras 113
B.6 configuração de métricas 114
B.7 Projetos e repositórios 115
B.8 Processamento e resultados 116
B.9 Hierarquia de resultados 117
B.10 Preferências 118
B.11 Ponto de acesso principal 119
B.12 Ponto de acesso de ferramentas base 120
B.13 Pontos de acesso de grupos de leituras 120
B.14 Pontos de acesso de leituras 120
B.15 Transferência de erros por XML 121
B.16 Fotografia de configuração de métricas 122
B.17 Fábricas de objetos de acesso a dados 123
B.18 Fluxo de um acesso remoto 124
B.19 Objeto abstrato de transferência 124
B.20 Abstração de entidade e auxiliares 126
B.21 Classes que gerenciam concorrência 128
B.22 Hierarquia de carregadores 130
B.23 Tarefas de processamento 131

Lista de Tabelas

4.1 Estatísticas descritivas das variáveis. 26
4.2 Resultados da regressão múltipla do primeiro estudo. 27
4.3 Projetos analisados: linguagem predominante e número de módulos ou classes. 29
4.4 Distribuição da métrica ACC nos códigos analisados. 35
4.5 Percentis para a métrica `acc`. 36
4.6 Distribuição da métrica ACCM nos códigos analisados. 38
4.7 Percentis para a métrica `accm`. 39
4.8 Distribuição da métrica AMLOC nos códigos analisados. 41
4.9 Percentis para a métrica `amloc`. 42
4.10 Distribuição da métrica ANPM nos códigos analisados. 44
4.11 Percentis para a métrica `anpm`. 45
4.12 Distribuição da métrica CBO nos códigos analisados. 47
4.13 Percentis para a métrica `cbo`. 48
4.14 Distribuição da métrica DIT nos códigos analisados. 49
4.15 Percentis para a métrica `dit`. 50
4.16 Distribuição da métrica LCOM4 nos códigos analisados. 51
4.17 Percentis para a métrica `lcom4`. 52
4.18 Distribuição da métrica LOC nos códigos analisados. 54
4.19 Percentis para a métrica `loc`. 55
4.20 Distribuição da métrica noa nos códigos analisados. 56
4.21 Percentis para a métrica `noa`. 57
4.22 Distribuição da métrica NOC nos códigos analisados. 58
4.23 Percentis para a métrica `noc`. 59
4.24 Distribuição da métrica NOM nos códigos analisados. 60
4.25 Percentis para a métrica `nom`. 61
4.26 Distribuição da métrica NPA nos códigos analisados. 62
4.27 Percentis para a métrica `npa`. 63
4.28 Distribuição da métrica NPM nos códigos analisados. 64
4.29 Percentis para a métrica `npm`. 65
4.30 Distribuição da métrica RFC nos códigos análisados. 66
4.31 Percentis para a métrica `rfc`. 67
4.32 Distribuição da métrica SC nos códigos análisados. 69
4.33 Percentis para a métrica `SC`. 70
4.34 Evolução métricas Firefox. 73

5.1 Ferramentas existentes versus requisitos definidos 80

B.1 Granularidade pai inferida 111
B.2 Exemplos de configurações de métricas 114

Capítulo 1

Introdução

O impacto e utilização do software livre na indústria de software tem aumentando gradativamente nas últimas décadas e influenciado significativamente a economia global (Benkler, 2006). No entanto, há empresas e governos (ou setores dos governos) ainda relutantes em adotar software livre devido a dúvidas legais, incertezas comerciais, questões culturais ou falta de confiança na qualidade do produto (código produzido) e do suporte desse tipo de software. Por outro lado, por suas características, o software livre apresenta um enorme potencial para melhorar a produtividade, competitividade e agilidade da indústria nacional de software, hoje ainda muito aquém das possibilidades do país.

Inicialmente, parte do desenvolvimento de software livre seguia um tipo de abordagem denominada "bazar" (Raymond, 1999), assim sugerindo a não existência de qualquer processo de desenvolvimento com um nível maior de Engenharia de Software, em oposição a um estilo de mercado mais formal denominado "catedral" (Raymond, 1999). Porém, principalmente desde a virada para o século XXI, concomitante ao crescimento da Internet, é notório um crescente avanço e movimentação em torno do desenvolvimento de software livre de forma mais estruturada (Gousios *et al.*, 2007). Além disso, a crescente demanda de mercado por software livre de uso comercial e em setores estratégicos da economia exige que sejam satisfeitas restrições de custo, prazo e qualidade (Gousios *et al.*, 2007).

A Engenharia de Software, desde os anos 1970, procura contornar a chamada "crise do software" (Arthur, 1985) e dar um tratamento de engenharia, sistematizando e controlando o desenvolvimento de sistemas de software complexos. Na última década, começaram a aparecer as primeiras pesquisas e projetos, no âmbito da Engenharia de Software, sobre projetos de software livre, como por exemplo: Ohloh, SQO-OSS, FLOSSMetrics, Qualoss, QSOS, FOSSology e QualiPSo. As discussões a respeito de um processo de desenvolvimento e uso de metodologia de engenharia de software em projetos de software livre avançaram, mas ao mesmo tempo o número de sistemas de software livre criados, utilizados e sua repercussão aumentaram (Benkler, 2006).

Embora haja centenas de empresas e governos interessados em utilizar mais amplamente componentes de software livre em suas infraestruturas de tecnologia da informação, muitas vezes eles podem se perder diante da enorme quantidade de opções disponíveis e a falta de critérios objetivos para avaliar a confiabilidade das opções existentes. Nesse contexto, como motivação inicial desta tese, questionamos

1. *Como é possível medir a qualidade do software para ajudar na escolha de um desses programas ou bibliotecas livres disponíveis?*

2. *Quais são as métricas de qualidade?*

3. *Como auferir essas métricas?*

4. *Como comparar projetos de forma a embasar uma escolha entre várias alternativas semelhantes?*

5. *Como automatizar e avaliar a aplicação dessas métricas?*

Partindo dessas questões iniciais, selecionamos métricas e trabalhamos para implementá-las em uma ferramenta de avaliação semi-automática da qualidade de código-fonte de qualquer projeto de software. O foco desta tese, no entanto, foi dirigido para a análise de projetos de software livre.

Métricas de código-fonte permitem a análise de dados relacionados aos erros, flexibilidade, complexidade, legibilidade, manutenibilidade, segurança e portabilidade do código, que são fatores para a aceitação de um produto de software (Henderson-Sellers, 1996; Sato *et al.*, 2007). No entanto, obter esses dados e extrair deles informações úteis é uma tarefa não trivial que requer a ajuda de ferramentas automatizadas. Ao contrário do software restrito, cujo código-fonte normalmente não é disponível para seus clientes e usuários, o software livre permite a análise de seu código e mesmo a execução de testes do tipo "caixa branca" onde os módulos internos do software são avaliados. Assim, tendo-se acesso ao código-fonte é possível realizar medições sobre a sua estrutura e organização interna de forma a avaliar sua qualidade.

Além dessa análise do código-fonte, em boa parte dos projetos de software livre também é possível explorar seu sistema de controle de versões, lista de participantes e desenvolvedores e base de dados de erros e correções (Gousios *et al.*, 2007). Um estudo sistematizado sobre esses dados poderia ajudar empresas e governos na decisão sobre qual alternativa adotar uma vez que eles poderiam ter uma ideia sobre a qualidade do suporte oferecido pela comunidade por trás de um determinado software livre.

Especificamente no desenvolvimento de software, qualidade é dita como um conjunto de características atendidas, em um determinado nível, para que o produto de software contemple as necessidades explícitas e implícitas de seus usuários (Rocha *et al.*, 2001). O padrão internacional *Software engineering – Product quality*, ISO/IEC 9126 ISO/IEC9126-1 (2001), prevê um conjunto de parâmetros para a avaliação da qualidade de software, com o objetivo de diminuir a subjetividade do conceito de qualidade. São definidos seis atributos, muitas vezes referidos como características de qualidade de software:

- **Funcionalidade** – se o software executa as funções requeridas;
- **Confiança** – refere-se à maturidade, tolerância a falhas e capacidade de recuperação;
- **Usabilidade** – refere-se ao esforço necessário para compreender, aprender e operar o software;
- **Eficiência** – refere-se ao desempenho e utilização dos recursos;
- **Manutenibilidade** – refere-se ao esforço necessário para modificar o software;
- **Portabilidade** – refere-se ao esforço necessário para transferir o software de um ambiente para outro.

Dos seis atributos citados, dois (usabilidade e portabilidade) podem ser considerados totalmente não-funcionais, ou seja, não estão relacionados com as tarefas realizadas pelo código em si. Os demais estão parcialmente ou totalmente vinculados com as características funcionais (relacionadas com o código produzido), em que, geralmente, os requisitos e os testes do software são baseados. Por um lado, desafios e dificuldades na avaliação da qualidade dos requisitos não-funcionais existem, uma vez que medi-las tradicionalmente envolve os conhecimentos e as experiências dos usuários desse software. Por outro lado, para requisitos funcionais, há métricas de software objetivas, que em certa medida, podem refletir a qualidade e, portanto, são utilizadas também na avaliação da qualidade de métodos (Bansiya e Davis, 2002) e de modelos (Zuse, 1990) de software.

Nesta tese, apresentamos argumentos para se observar a qualidade do software também através das métricas de código-fonte. No contexto desta pesquisa, associamos qualidade do software à qualidade do código. Medir a qualidade do código-fonte passou a ser tão importante quanto medir a qualidade do processo, principalmente no contexto do software livre. Em muitos casos, o que está disponível é o código-fonte, promovendo-o ao principal artefato para se medir a qualidade do software. Uma maneira objetiva de se observar as características de um código-fonte é analisando os valores de suas métricas. Entretanto, ainda não está disseminada uma análise sistemática da

qualidade do código-fonte produzido. Mesmo com métricas de software e de código-fonte sendo propostas em alguns trabalhos desde a década de 1970 (Perlis *et al.*, 1981; Wolverton, 1974), seu potencial de uso no desenvolvimento de software não tem sido explorado (Tempero, 2008). Um dos motivos da sub-utilização das métricas de código-fonte por parte dos desenvolvedores é a falta de conhecimento de como coletar automaticamente os valores das métricas, interpretar os seus resultados e os associar à qualidade do código-fonte. Somado a isso, temos o fato da maior parte das métricas não possuir valores frequentes conhecidos, o que seria útil como parâmetro nas análises dos valores das métricas de um determinado código-fonte. É nesse cenário teórico e prático que construímos esta tese de doutorado.

1.1 Objetivos

A proposta inicial deste doutorado foi intitulada de "Mangue: Métricas e Ferramenta para Avaliação Automática da Qualidade de Projetos de Software Livre". O projeto foi aprovado pelo CNPq, em julho de 2008, na seleção pública de propostas para a concessão de bolsas de mestrado e doutorado, na área temática de engenharias, software e tecnologias da informação e comunicação. O objetivo descrito nessa proposta foi realizar um estudo sobre a avaliação da qualidade de projetos de software livre, enfatizado no estudo de métricas que permitem a análise automática de código-fonte, realizadas por ferramentas criadas e adaptadas por nós para prover tais informações.

Nesse contexto, complementarmente, foi realizado o doutorado sanduíche no exterior, na *Southern Illinois University Carbondale* (SIUC), visando o amadurecimento dos estudos sobre como analisar dados de projetos de software livre, o que levou a um modelo que explica o "Relacionamento entre Métricas de Código-Fonte e Atratividade de Projeto de Software Livre".

Dito isso, o objetivo geral deste doutorado proposto, em 2008, ao CNPq foi atingido ao investigar o uso de métricas de código-fonte como indicador para o sucesso de projetos de software livre, ao monitorar, automaticamente, os projetos do ponto de vista técnico.

1.1.1 Questões de pesquisa

As questões de pesquisa respondidas nesta pesquisa foram:

- QP1 - Métricas de código-fonte podem influir na atratividade de projetos de software livre?
- QP2 - Quais métricas devem ser controladas ao longo do tempo?
- QP3 - As métricas de código-fonte melhoram com o amadurecimento (maior número de contribuidores e atualizações no repositório) dos projetos?

Para responder essas questões, nossa pesquisa concentrou-se nos seguintes objetivos específicos:

Objetivos tecnológicos

- OT1 – Desenvolvimento e adaptação da ferramenta de coleta de métricas de código-fonte – Analizo (http://analizo.org): um conjunto de ferramentas livres para análise e visualização de métricas de código-fonte que contempla múltiplas linguagens de programação. Entre suas funcionalidades estão: (i) calculo métricas tanto em nível de projeto quanto métricas de nível de classe, num total de 24 métricas até o momento; (ii) processamento em lote de métricas; (iii) geração de matrizes de evolução; (iv) geração de grafo de dependência; (v) análise do histórico de métricas.
- OT2 – Desenvolvimento do *web service* para interpretação de métricas de código-fonte – Kalibro (http://kalibro.org): um *web service* que tem o objetivo de potencializar o uso das métricas de código-fonte através da interpretação e entendimento dos valores das métricas. Entre suas principais funcionalidades está a criação de intervalos de valores de referência das

métricas com possibilidade de recomendação e explicação de como melhorar o código baseado nesses valores.

- OT3 – Desenvolvimento da rede de monitoramento de métricas de código-fonte – Mezuro (http://mezuro.org): uma rede social para avaliação e monitoramento de código-fonte de projetos de software livre (ou de repositório aberto). O Mezuro foi desenvolvido com base na plataforma livre para rede sociais Noosfero. Através de plugins, foram incorporadas as funcionalidades implementadas no Kalibro e as métricas fornecidas pela Analizo. Além disso, uma série de funcionalidades foram desenvolvidas de modo a permitir que o Mezuro monitore a evolução dos projetos de software analisados ao longo do tempo, através de múltiplas coletas de métricas de código-fonte. O Mezuro se conecta diretamente aos repositórios dos códigos que são avaliados, assim realizando análises num intervalo de tempo (dias) definidos pelos usuários.

Objetivos científicos

- OC1 – Estudo inicial da influência de métricas de código-fonte na atratividade de projetos de software livre escritos na linguagem C a partir do código disponível em 6.700 projetos.
- OC2 - Estudo mais detalhado da influência de métricas de código-fonte na atratividade de projetos de software livre, com mais de 100 mil downloads, escritos nas linguagens C, C++ ou Java a partir do código disponível em 756 projetos.
- OC3 - Estudo abrangente da influência de métricas de código-fonte na atratividade de projetos de software livre escritos nas linguagens C, C++ ou Java a partir do código disponível atualmente (obtido dos repositórios de sistemas de gerenciamento de versões) de 8.450 projetos.
- OC4 - Identificação das distribuições estatísticas dos valores de métricas em 38 projetos de software livre (35 com mais contribuidores no intervalo de 12 meses e 3 de avaliação de métricas), a fim de compreender qual a abordagem estatística mais informativa para o monitoramento dessas métricas, bem como observar os valores frequentes para essas métricas, de forma a servirem de referência para projetos futuros.
- OC5 - Análise estatística da correlação entre métricas a fim de definir um subconjunto reduzido de métricas que podem ser monitoradas ao longo do tempo e ainda oferecer uma boa visão do projeto.
- OC6 - Estudo longitudinal da evolução dos valores das métricas de código-fonte e do número de contribuidores e atualizações nos repositórios dos projetos de software livre Mozilla Firefox e Chrome.

1.2 Organização do Trabalho

Após a leitura desta introdução, o leitor desta tese encontrará no Capítulo 2, a definição e uma discussão sobre software livre, no contexto deste trabalho; no Capítulo 3, uma revisão bibliográfica sobre métricas de software e de código-fonte, bem como os conceitos em detalhes das métricas selecionadas para os estudos e implementadas em nossas ferramentas; no Capítulo 4, a apresentação e discussão dos resultados dos estudos realizados; no Capítulo 5, a descrição detalha das ferramentas que desenvolvemos e com que colaboramos no escopo deste doutorado; por fim, no Capítulo 6, as conclusões, uma síntese de todas as contribuições e um mapeamento de trabalhos futuros.

Capítulo 2

Software Livre

Software expressa uma solução abstrata dos problemas computacionais. O software, em um sistema computacional, é o componente que contém o conhecimento relacionado aos problemas a que a computação se aplica. Por isso, o software é algo de interesse geral, uma vez que vários aspectos relacionados a ele ultrapassam as questões técnicas, como por exemplo:

- O processo de desenvolvimento do software;
- Os mecanismos econômicos (gerenciais, competitivos, sociais, cognitivos etc.) que regem esse desenvolvimento e seu uso;
- O relacionamento entre desenvolvedores, fornecedores e usuários do software;
- Os aspectos éticos e legais relacionados ao software.

O que define e diferencia o software livre do que podemos denominar de software restrito passa pelo entendimento desses quatro pontos dentro do que é conhecido como o *ecossistema do software livre*. O princípio básico desse ecossistema é promover a liberdade do usuário, sem discriminar quem tem permissão para usar um software e seus limites de uso, baseado na colaboração e num processo de desenvolvimento aberto. Software livre é aquele que permite aos usuários usá-lo, estudá-lo, modificá-lo e redistribui-lo, em geral, sem restrições para tal e prevenindo que não sejam impostas restrições aos futuros usuários.

Normalmente, esse software existe por meio de projetos de desenvolvimento que estão centradas em torno de algum código-fonte acessível ao público, geralmente em um repositório na Internet, onde desenvolvedores e usuários podem interagir. O código é necessariamente licenciado sob termos legais formais que estão de acordo com as definições da *Free Software Foundation* [1] ou da *Open Source Initiative* [2].

Uma vantagem oferecida pelo software livre em comparação ao software restrito vem do fato de que o código-fonte pode ser livremente compartilhado. Esse compartilhamento pode simplificar o desenvolvimento de aplicações personalizadas, que não precisam ser programadas a partir do zero, mas podem basear-se em soluções já existentes. Na medida em que o desenvolvimento de aplicações personalizadas é um dos focos do desenvolvimento de software em geral, essa vantagem tem impacto significativo na redução de custos e na diminuição na duplicação de esforços, tirando proveito da característica abstrata do software.

Outra vantagem resultante do compartilhamento do código se refere à possível melhoria na qualidade (Raymond, 1999), em particular frente aos problemas inerentes à sua complexidade, que é o que estamos interessados nesta tese. Isso se deve ao maior número de desenvolvedores e usuários envolvidos com o software. Em outras palavras, um número maior de desenvolvedores, com diferentes perspectivas e necessidades, é capaz de identificar melhorias e corrigir mais *bugs* em menos tempo e, consequentemente, promover refatorações que, geralmente, levam à melhoria do código.

[1] http://www.gnu.org/philosophy/free-sw.html
[2] http://www.opensource.org/docs/definition.html

Além disso, um número maior de usuários gera situações de uso e necessidades mais variadas, o que se traduz em um maior número de *bugs* identificados e mais sugestões de melhorias.

2.1 Desenvolvimento de Software Livre

O desenvolvimento de software livre possui características distintas do modelo restrito. Portanto, a relação com o mercado, o processo de desenvolvimento e o produto a ser oferecido devem ser abordados de maneiras distintas.

No artigo *A Catedral e o Bazar* (Raymond, 1999), Eric Raymond levanta os aspectos que contribuem para que um projeto de software livre tenha sucesso. Suas observações formaram a base do movimento pelo código aberto por ele iniciado na virada do século. Ao observar o modelo de desenvolvimento do Linux, Raymond vislumbrou semelhanças com um barulhento Bazar, onde centenas ou milhares de desenvolvedores davam a sua contribuição que era, então, gerenciada por um pequeno grupo ou por um "ditador benevolente" que dava orientações sobre a qualidade das modificações propostas e as aceitava ou não. Ao identificar essas características, Raymond aplicou essas práticas de desenvolvimento do modelo Bazar de forma consciente em um projeto sob sua coordenação, o Fetchmail. Ele elencou as principais características desse modelo, típicas em boa parte dos projetos de software livre da atualidade, como sendo:

- *Bons programas nascem de necessidades pessoais.* Um projeto tem maiores chances de sucesso quando o desenvolvedor principal ou grupo de desenvolvedores principais tem interesse e sentem a necessidade pessoal de utilizar aquele software.

- *Bons programadores sabem escrever bom código, mas excelentes programadores sabem reescrever e reutilizar código.* Raymond menciona a "preguiça construtiva" como a ideia de que não se deve reinventar a roda, mas sim reaproveitar o que já existe e, se for o caso, modificar o que já existe para melhorá-lo.

- *Esteja preparado para jogar fora código-fonte se necessário e começar de novo.* Ou seja, dificilmente acerta-se na primeira vez.

- *Os usuários devem ser tratados como co-desenvolvedores.* Esse é o melhor caminho para o aprimoramento do código e depuração eficaz.

- *Libere código cedo e libere frequentemente e ouça seus usuários.* Um erro comum de pessoas e grupos que se iniciam no mundo do software livre é achar que seu software ainda não está pronto para ser liberado, que agora ainda não é o momento certo para se fazer isso. Segundo Raymond, o quanto antes o código for liberado e quanto maior a frequência de liberação de novas versões, melhor será o retorno obtido dos usuários e a possibilidade de angariar contribuidores para o projeto.

- *Dados olhos suficientes, todos erros são triviais.* Raymond chamou essa frase de Lei de Linus. Com milhares de pessoas lendo o código-fonte do Linux, os eventuais erros eram localizados e reportados muito rapidamente. Da mesma forma, com centenas de pessoas com conhecimento técnico para resolver aqueles erros, rapidamente aparecia um voluntário com a solução do problema.

- *Trate seus testadores das versões Beta como um recurso valioso e eles logo se tornarão um recurso valioso.* Não há nada mais eficaz para encontrar problemas num programa e sugerir melhorias em suas funcionalidades do que um grupo de usuários ativos e motivados querendo utilizar esse programa e testar as novas funcionalidades o quanto antes.

- *A perfeição (em projetar) é alcançada não quando não há mais nada a adicionar, mas quando não há nada para jogar fora.* Essa é uma ideia quase que de consenso entre os grandes cientistas e engenheiros. Deve-se buscar sempre as soluções mais simples.

- *A melhor coisa depois de ter boas ideias é reconhecer as boas ideias de seus contribuidores.* Um bom líder de um projeto de software livre não é necessariamente aquele que tem ótimas ideias, mas sim aquele que é capaz de criar o ecossistema de colaboração que permita que as boas ideias emerjam e sejam valorizadas e adotadas.

Baseado nessas nove características, queremos atribuir a Eric Raymond, e discutir nesta tese, uma extensão do que ele chamou de Lei Linus: *Dados olhos e desenvolvedores suficientes, todos os bugs serão triviais e o código estará organizado para receber contribuições* – que vamos chamar aqui de "Lei de Raymond".

Devido às atividades de produção de código, documentação, relatos de *bugs* entre outras, as comunidades de software livre vêm construindo coletivamente sistemas de software reconhecidamente de qualidade, em um ambiente de colaboração constante para atualização e evolução desses sistemas, organizados na forma de um rossio (Simon e Vieira, 2008). Nesse contexto, os usuários não necessariamente restringem-se a ser apenas agentes passivos, mas podem atuar como colaboradores ou produtores do software que usam.

Esse fenômeno de produção coletiva extrapolou o movimento do software livre, especialmente a partir da primeira década do Século 21, com o surgimento de serviços criados e mantidos pelos próprios usuários na Internet, como a Wikipédia, blogs pessoais, canais de TV e rádios online (Tapscott e Williams, 2006). Tais serviços, somados às redes sociais, fazem com que as pessoas realmente acreditem que podem influenciar outras através de seus próprios meios de comunicação (Castells, 2006). Esse cenário, em que não fica clara uma diferenciação entre consumidor e produtor de informação e, no caso do software, usuário e desenvolvedor, pode ser chamado de *cultura livre*.

No caso do software livre, essa cultura tipicamente se pauta nas questões técnicas e se organiza como uma meritocracia, onde valoriza-se fortemente as colaborações feitas para o projeto. No entanto, as questões éticas, quando se apresentam, em geral são consideradas tão importantes quanto as técnicas, já que o envolvimento com um projeto é também determinado pelo interesse pessoal. Pessoas e organizações que permanecem no longo prazo colaborando de alguma maneira com o desenvolvimento de um software livre realmente acreditam que estão fazendo a diferença e ajudando o mundo de alguma forma, e essa motivação faz com que sua dedicação seja diferenciada. Mesmo no caso (cada vez mais comum) de desenvolvedores pagos para trabalhar em um projeto livre, os aspectos éticos e o relacionamento com a comunidade norteiam sua participação.

Na prática, o desenvolvimento de software livre envolve administrar ou participar de uma equipe de desenvolvimento onde não há necessariamente uma hierarquia formal, mecanismos de pressão para o cumprimento de prazos e/ou grande formalismo em processos. Uma das estratégias para atrair contribuidores e evitar que eles abandonem o projeto é garantir a qualidade do código, o que favorece a criação de um círculo virtuoso em que o código promove o crescimento da comunidade e a comunidade ativa promove melhorias no código (esse é ponto central discutido no estudos apresentados no Capítulo 4 desta tese).

Na busca por uma formalização sobre qual é a metodologia das comunidades de software livre, estudos mostram que métodos ágeis e software livre têm formas de trabalhos semelhantes. Por exemplo, o desenvolvimento de software livre é considerado um método ágil por Martin Fowler (Fowler, 2000). Alguns autores afirmam que o desenvolvimento de software livre é um método ágil (Abrahamsson *et al.*, 2002) e apontam fortes semelhanças entre métodos ágeis e software livre (Warsta e Abrahamsson, 2003). Por exemplo, há um mapeamento entre as práticas comuns usadas pelas comunidades de software livre e equipes ágeis (Corbucci, 2011). Conceitualmente, os valores semelhantes são:

- Indivíduos e interações são mais importantes que processos e ferramentas.
- Software em funcionamento é mais importante que documentação abrangente.
- Colaboração com o cliente (usuários) é mais importante que negociação de contratos.

- Responder às mudanças é mais importante que seguir um plano.

Além disso, várias práticas disseminadas pelas metodologias ágeis são usadas no dia-a-dia dos desenvolvedores e equipes das comunidades de software livre (Corbucci, 2011): (i) Código compartilhado (coletivo); (ii) Projeto simples; (iii) Repositório único de código; (iv) Integração contínua; (v) Código e teste; (vi) Desenvolvimento dirigido por testes, e (vii) Refatoração.

Observar e entender esses aspectos nos projetos de software livre tornam-se relevantes à medida que muitos projetos de software livre não vão além dos estágios iniciais e muitos acabam sendo abandonados antes de produzir resultados razoáveis. Isso sugere que, mesmo com o sucesso de alguns projetos de software livre, as comunidades, com ou sem a participação de empresas, podem avançar no acompanhamento do desenvolvimento dos projetos de software livre que participam.

Para ilustrar esse cenário, podemos observar alguns dados extraídos do SourceForge.net, um dos mais populares repositórios de projetos de software livre. Entre os seus 201.494 projetos cadastrados, 60.642 lançaram mais de uma versão, 40.228 foram baixados mais de uma vez, 23.754 têm mais de um membro, e apenas 12.141 projetos satisfazem esses três critérios de seleção juntos. Isso sugere que não mais que 6% dos projetos no SourceForge.net foram capazes de constituir uma comunidade de usuários e desenvolvedores que se beneficiem do estilo de desenvolvimento Bazar (Raymond, 1999).

De maneira geral, pode-se afirmar que há uma grande disposição para a criação de projetos de software livre, mesmo que muitas das iniciativas "falhem". Olhar o processo de desenvolvimento de software livre do ponto de vista da Engenharia de Software e as possíveis sinergias com os métodos ágeis podem contribuir para um melhor rendimento dessa disposição na criação e colaboração em torno de projetos de software livre.

Na prática, dentro do processo de desenvolvimento de software livre, após lançar uma versão inicial e divulgar o projeto, os usuários interessados começam a usar o software livre em questão. Lembrando o que foi salientado por Eric Raymond sobre "bons programas nascerem de necessidades pessoais", esses usuários podem também ser desenvolvedores, que irão colaborar com o projeto a fim de atenderem às suas próprias necessidades. Destacando a colaboração no código-fonte, essas melhorias são enviadas aos mantenedores do projeto como *patches*, ou seja, arquivos que contém as modificações no código e que serão analisados pelos mantenedores que, caso concordem com a mudança e com a sua implementação em si, irão aplicá-las ao repositório oficial do projeto.

Mesmo que em projetos maiores outros aspectos sejam levados em consideração ou sigam processos mais burocrático de colaboração, a essência da colaboração técnica está no envio e análise de trechos de código-fonte. Neste contexto, nesta tese de doutorado, argumentamos e discutimos, através de estudos baseados em análises estatísticas, que o monitoramento de métricas de código-fonte, como sendo uma das atividades de acompanhamento (do mesmo conceito de "tracking" dos métodos ágeis), pode ser um recurso valioso de apoio ao gerenciamento da evolução dos projetos de software livre com o objetivo de atrair e facilitar a colaboração dos desenvolvedores.

2.2 Atratividade de Projetos de Software Livre

A satisfação e a percepção de qualidade dos usuários e empresas que utilizam software livre é geralmente assegurada pela colaboração de uma comunidade de usuários, que reportam falhas, e desenvolvedores, que corrigem as falhas e adicionam novas funcionalidades. Dessa forma, o modelo de desenvolvimento de software livre é conhecido por duas principais vantagens: (i) o potencial de revisão por diferentes pares e (ii) a possibilidade de atrair desenvolvedores ao redor do mundo (Michlmayr *et al.*, 2005). Por isso, é importante que um projeto de software livre atraia usuários e desenvolvedores (Stewart e Gosain, 2006).

Entretanto, nem todo projeto de software livre consegue atrair usuários e desenvolvedores, sendo assim considerado bem sucedido e de alta qualidade (Michlmayr *et al.*, 2005). O significado de sucesso de um projeto de software livre é visto de diferentes formas nos trabalhos de diferentes pesquisadores. Em um desses trabalhos, Santos Jr. *et al.* (2010) definiu um modelo teórico para

atratividade de projetos de software livre. Baseado nesse modelo, definimos a hipótese de que a atratividade de projetos de software livre também é influenciada por atributos de código-fonte, como tamanho e complexidade estrutural observadas através de métricas de código-fonte.

Em geral, apesar da grande importância do código-fonte nas comunidades de software livre, o que denominamos de cultura do "show me the code", métricas de código-fonte normalmente não são usadas como um indicador de qualidade. Tendo em vista essa aparente contradição, argumentamos que, alguns atributos do código-fonte, que podem ser analisados pelas métricas, estão relacionados com a atratividade de projetos de software livre. Para testarmos inicialmente nossas hipóteses, analisamos 6.773 projetos escritos em C, cadastrados no repositório SourceForge.net. Posteriormente, avaliamos 756 projetos C, C++ e Java com mais de 100 mil downloads no SourceForge.net. Por fim, analisamos 8.450 projetos em C, C++ e Java para revalidarmos os estudos com uma amostra mais ampla. Descobrimos que, considerando nossas amostras, uma métrica de complexidade estrutural e duas métricas de tamanho possuem um importante papel na explicação da atratividade, nos estudos representada pelo número de *downloads* e número de membros dos projetos de software livre, de acordo como obtidos nas bases de dados do SourceForge.net.

Atratividade é a capacidade de trazer usuários e desenvolvedores para um projeto. Um projeto de software livre é tão atraente quanto a sua capacidade em atrair potenciais usuários e programadores que, depois de um período como usuários passivos, passam a participar das tarefas de melhoria do projeto (Santos Jr. *et al.*, 2010).

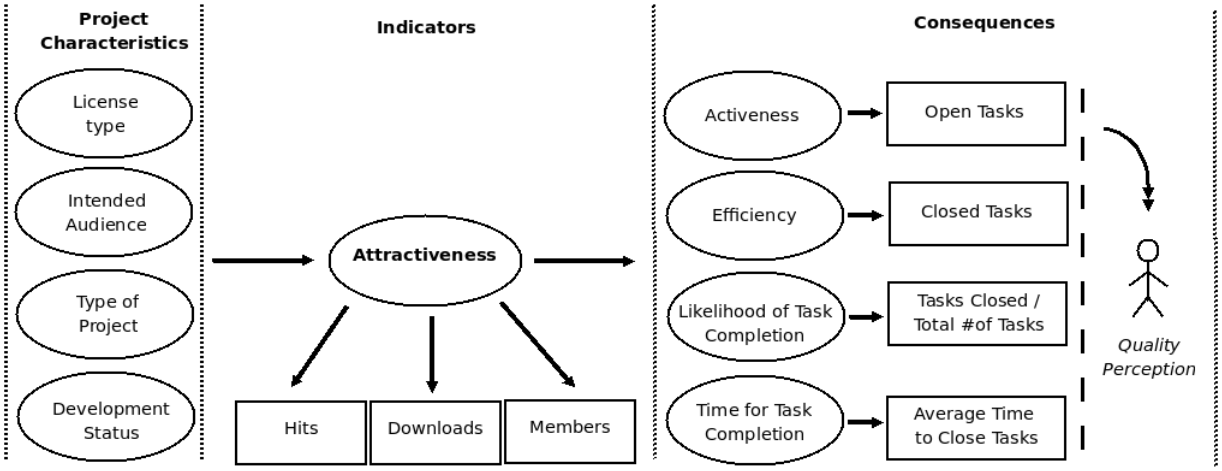


Figura 2.1: *Modelo de atratividade – adaptado de Santos Jr. et al. (2010).*

Nesta tese de doutorado, o conceito de atratividade e suas medidas são baseados no modelo de atratividade definido por Santos Jr. *et al.* (2010), conforme ilustrado na Figura 2.1. Esse modelo especifica características de um projeto de software livre que influenciam em sua atratividade e nas consequências dessa atratividade, tais como níveis de atividade, eficiência, probabilidade de conclusão de tarefas, tempo para conclusão de tarefas e percepção da qualidade (Santos Jr. *et al.*, 2010).

Na proposta original, o modelo de atratividade trata as seguintes características de um projeto:

- *Tipo de Licença* sob a qual um software é distribuído, por exemplo GPL, LGPL, MPL e BSD. A licença determina as possíveis formas de uso e distribuição de um software. Também define as regras para a criação de trabalhos derivados, regulando o que pode e não pode ser feito com o código-fonte. Essas restrições influenciam as pessoas quanto ao uso e a colaboração no desenvolvimento de um software livre (Santos Jr. *et al.*, 2010).

- *Público-Alvo* que é o tipo de usuário, por exemplo iniciante ou avançado, e tipo de membro do projeto, por exemplo administrador de sistemas ou desenvolvedor, que um projeto busca atrair. Isso pode influenciar no número de possíveis desenvolvedores e usuários do projeto, pois diferentes públicos-alvos podem atrair diferentes tipos de membros(Johnson, 2002).

- *Tipo de projeto* que se refere à área a que o projeto está relacionado, como navegadores, sistemas operacionais, jogos, aplicações científicas, etc. (Crowston e Scozzi, 2002). Isso representa o domínio da aplicação e influencia na atratividade de acordo com os diferentes nichos de interesse das pessoas. Alguns nichos possuem mais possibilidades de trabalho voluntário e uma maior base de usuários em relação a outros projetos. Além disso, alguns domínios de aplicação têm mais projetos concorrentes que outros, o que torna mais difícil para um projeto destacar-se como uma opção mais viável de utilização (Johnson, 2002).

- *Estado de desenvolvimento* que diz respeito à atual versão do software disponível, que pode ser, dependendo da terminologia usada pelo projeto, por exemplo, teste, alfa, beta, estável, produção, etc. O estado de desenvolvimento pode influenciar nas decisões dos desenvolvedores em fazer parte e contribuir com um projeto, bem como nas decisões dos usuários em adotar um determinado software (Santos Jr. *et al.*, 2010). Também pode afetar a motivação dos membros em trabalhar para lançar uma nova versão, o que interfere nos índices de produtividade (Raja e Tretter, 2006; Stewart e Gosain, 2006).

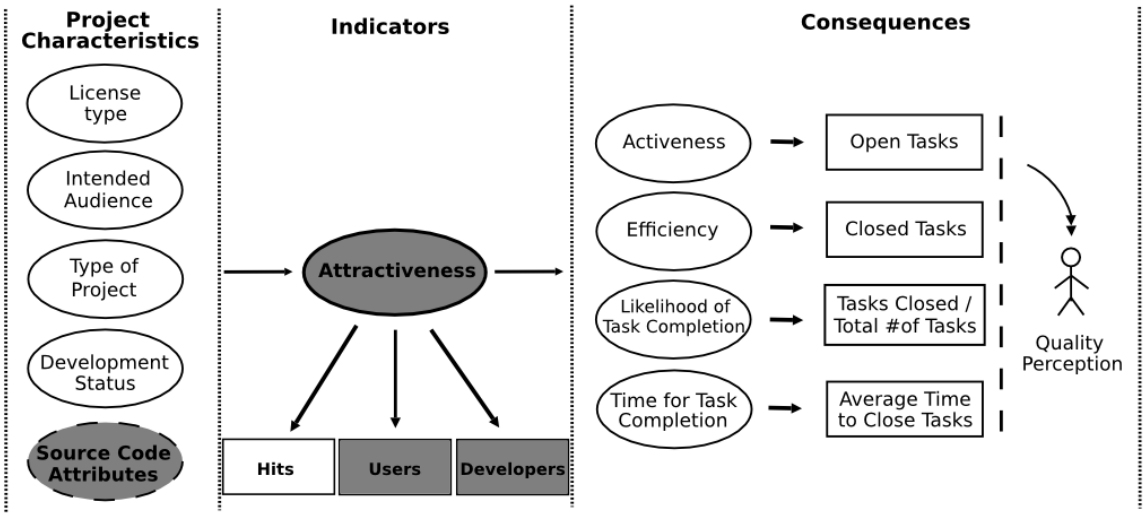


Figura 2.2: *Modelo de atratividade: inserção de atributos de código-fonte.*

Inserimos os atributos de código-fonte como uma das características do projeto nesse modelo teórico de causa-efeito na atratividade de projetos de software livre, conforme ilustrado na Figura 2.2. Para esses estudos, definimos um modelo, representado na Figura 2.3 para observar a influência individual dos atributos de código-fonte na atratividade de projetos de software livre. Em suma, propomos um novo elemento que pode explicar parcialmente a atratividade.

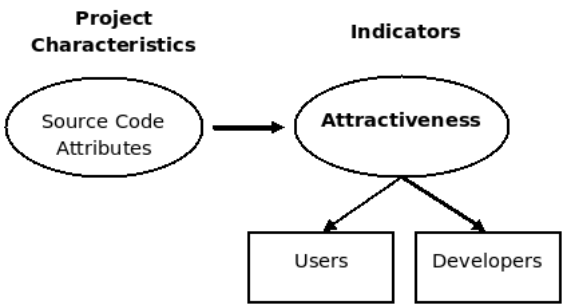


Figura 2.3: *Atributos de código-fonte e indicadores de atratividade.*

Antes de receber relatos de erros, correções de falhas e novas funcionalidades, um projeto de software livre deve ser atraente para possíveis voluntários, que normalmente primeiro usam o software e depois passam a colaborar com o projeto. Ao longo do tempo, essas contribuições afetam o

número de *downloads* e trazem mais membros, criando um ciclo positivo de atratividade do projeto. Assim, nos nossos estudos, definimos atratividade baseado em dois de seus indicadores empíricos:

1. *número de downloads*, de acordo com os dados dos projetos no SourceForge.net, representa o número de pessoas interessadas em usar o software.

2. *número de membros* do projeto, registrados pelo SourceForge.net, representa o número de contribuidores do projeto.

Isso significa que utilizamos o número de *downloads* e de membros fornecidos pelo SourceForge.net como uma estimativa aproximada do número de usuários e de desenvolvedores do projeto, respectivamente. Nossos estudos exploram um grande número de projetos e aplicamos a mesma definição de número de usuários e desenvolvedores para todos eles. Novamente, os dados foram providos pela mesma fonte, o SourceForge.net. Embora reconhecemos que a utilização desses mapeamentos aproximados implicam em uma limitação, não encontramos em trabalhos anteriores melhores definições que poderiam representar mais fielmente o número de usuários e de desenvolvedores, quando se tem uma grande amostra de projetos analisados.

O significado de "sucesso" de um projeto de software livre foi discutido em diferentes perspectivas em trabalhos anteriores:

- modularidade do código-fonte (Shaikh e Cornford, 2003);
- número de linhas de código-fonte geradas (Mockus *et al.*, 2000);
- velocidade de correção de erros (Stewart e Gosain, 2006);
- capacidade de um projeto avançar suas fases de desenvolvimento (por exemplo, de alfa para beta para estável) (Crowston e Scozzi, 2002; Raja e Tretter, 2006);
- número de *downloads* (Balijepally *et al.*, 2009);
- número de membros (Crowston *et al.*, 2006).

Em nosso entendimento, essas medidas de "sucesso", quando usadas individualmente, não indicam completamente se um projeto é bem sucedido. Entretanto, quando analisadas em conjunto, elas podem oferecer indícios para ajudar a alcançar ou manter o "sucesso" (Santos Jr. *et al.*, 2010). Além disso, a maior parte das colaborações para um projeto de software livre são em torno do seu código-fonte, que é o principal artefato gerado e gerenciado pela sua comunidade. Por isso, inserimos os atributos de código-fonte, através de métricas, no modelo de atratividade como uma das causas da atratividade. Em resumo, algumas características do código-fonte podem maximizar as contribuições para um projeto, o que pode atrair mais usuários e desenvolvedores. Assim, elaboramos algumas hipóteses baseadas nessa ideia e fizemos 3 estudos para testá-las, conforme apresentaremos no Capítulo 4 desta tese.

Capítulo 3

Métricas de Software

Uma métrica, segundo definição da ISO/IEC9126-1 (2001), é a composição de procedimentos para a definição de escalas e métodos para medidas. O SEI (*Software Engineering Institute*) definiu o termo "métricas de software" como um conjunto de medidas de um processo ou produto de software, onde um produto de software pode ser visto como um objeto abstrato que evolui de uma instrução inicial para o sistema de software finalizado, incluindo o código-fonte e variadas formas de documentação produzidas durante o desenvolvimento (Mills, 1988). Então, métricas são utilizadas para estimar um cronograma e custos de desenvolvimento do software e medir a produtividade e qualidade do produto.

O objetivo das métricas de software é a identificação e medição dos principais parâmetros que afetam o desenvolvimento de software (Mills, 1988). A necessidade das métricas de software teve maior atenção quando se constatou a chamada "crise do software", como argumentado em (Arthur, 1985). Outro fator que corroborou essa necessidade foi a constatação de uma gerência ineficaz durante o desenvolvimento da maior parte das soluções de software, uma vez que o desenvolvimento de software é complexo e não se tinha medidas bem definidas e viáveis para a avaliação do software e seu desenvolvimento (Brooks, 1975). Entretanto, estimativas precisas e eficazes, planejamento e controle são aspectos difíceis de se concretizar em conjunto (Mills, 1988). Assim, com as métricas de software, propõe-se uma melhoria do processo de gestão com a identificação, medição e controle dos parâmetros essenciais do software.

São consideradas boas métricas aquelas que facilitam a medição dos parâmetros de qualidade definidos para um determinado software (Mills, 1988). Na tentativa de medir tais parâmetros, há o risco de se perder na enorme quantidade de métricas propostas pela literatura (Zuse, 1990). O tratamento de um grande volume de dados pode ser humanamente impossível. Portanto, é preciso definir e justificar os critérios adotados na escolha das métricas. É desejável que as métricas possuam as seguintes características (Fenton e Pfleeger, 1998; Mills, 1988):

- o que a métrica se propõe a medir deve ser claro;
- a métrica deve ser formalmente definida, seu valor deve estar atrelado ao objeto medido, independente de quem (ou qual ferramenta) o obtenha;
- deve ser possível obter seu valor rapidamente e a baixo custo;
- a métrica deve medir efetivamente o proposto por ela;
- pequenas mudanças no software, por exemplo, não podem causar grandes mudanças no valor obtido;
- deve haver formas de mapeamento das métricas para o entendimento do comportamento das entidades analisadas através da manipulação dos números obtidos e
- o resultado da métrica deve ser independente da linguagem de programação utilizada.

Além disso, as métricas de software podem ser classificadas quanto ao âmbito da sua aplicação, quanto ao critério utilizado na sua definição e quanto ao método de obtenção da medida. A maneira mais ampla de classificá-las é quanto ao objeto da métrica. Por exemplo, *métricas de produtos* medem a complexidade e tamanho final do programa. Já as *métricas de processo* referem-se ao processo de concepção e desenvolvimento do software, medindo, por exemplo, o processo de desenvolvimento, tipo de metodologia usada e tempo de desenvolvimento (Mills, 1988).

As métricas também são classificadas quanto ao método de obtenção, dividindo-se em *primitivas* e *compostas* (Grady e Caswell, 1987). As métricas primitivas são aquelas que podem ser diretamente observadas em uma única métrica (p.ex., número de linhas de código). As métricas compostas são as combinações de duas ou mais métricas (p.ex., complexidade estrutural, que vem da multiplicação entre métricas de coesão e acoplamento).

Outra maneira de classificar as métricas de software é quanto aos critérios utilizados para determiná-las. Nesse aspecto, as métricas são diferenciadas em *métricas objetivas* e *métricas subjetivas*. As objetivas são obtidas através de regras bem definidas, sendo a melhor forma de possibilitar comparações posteriores consistentes. As subjetivas dependem do sujeito que está realizando a medida e, portanto, dificultam as comparações e a reprodutibilidade das medidas. Para possibilitar uma avaliação de software de forma sistemática, eficiente, eficaz e de baixo custo, os valores obtidos através das métricas deveriam ser sempre os mesmos, independentemente do momento, condições ou indivíduo que realiza as medidas.

Por fim, as métricas devem estar associadas a uma escala de medição que proporcione significado ao valor obtido no seu cálculo. Elas precisam ser coletadas em um modelo de dados específico que pode envolver cálculos ou análise estatística. Para isso, é importante considerar o tipo de informação obtida. Assim, quatro tipos de dados de medidas foram reconhecidos por estatísticos para as métricas de software (Conte *et al.*, 1986; Fenton e Pfleeger, 1998):

- Nominal: na interpretação dos valores de cada atributo, a ordem não possui significado.
- Ordinal: os valores possuem ordem, mas a distância entre eles não possui significado. Por exemplo, nível de experiência dos programadores.
- Intervalo: a ordem dos resultados é importante, assim como o tamanho dos intervalos que separam os pontos, mas as proporções não são necessariamente válidas. Por exemplo, um programa com "complexidade" de valor 6 é mais complexo em 4 unidades do que um programa com a complexidade de valor 2, mas isso não é muito significativo para dizer que o primeiro programa é 3 vezes mais complexo do que o segundo.
- Racional: semelhante à medida por intervalo, mas preservando as proporções.

Nesta tese de doutorado, concentramos-nos apenas em métricas objetivas, que podem ser calculadas a partir de uma análise estática do código-fonte de um software. Essas métricas podem ser primitivas ou compostas. Seus resultados podem ser mapeados em intervalos para serem interpretados quando analisados.

3.1 Métricas de código-fonte

Parte das métricas classificadas como objetivas tratam de características do código-fonte. Uma série de trabalhos deu início a abordagens para medir o tamanho do programa e a complexidade do software (Henry e Kafura, 1984; Troy e Zweben, 1981; Yau e Collofello, 1985). Existe uma grande quantidade de métricas desses tipos. Por exemplo, Li e Cheung (1987) referencia e compara 31 diferentes métricas de complexidade. Os trabalhos de Card e Agresti (1988); Harrison e Cook (1987) também propuseram novas métricas de complexidade. Além disso, existe um conjunto de métricas usadas para quantificar a complexidade do software com o objetivo de analisar, além da complexidade, a capacidade de compreendê-lo e modificá-lo, sua testabilidade, manutenibilidade e a sua evolução (Zuse, 1990).

Outro conjunto importante de métricas é aquele específico para sistemas orientados a objetos; as métricas mais comumente usadas incluem as propostas por (i) Henry e Kafura (1981); (ii) Chidamber e Kemerer (1991, 1994); (iii) Li e Henry (1993); (iii) Lorenz e Kidd (1994); (v) Briand *et al.* (1997, 1999). Ainda, uma avaliação comparativa dessas métricas e outras em diferentes contextos de desenvolvimento é apresentada por Xenos *et al.* (2000). Somado a isso, há estudos que tentam correlacionar métricas de software com qualidade de software (Subramanyam e Krishnan, 2003). Também, foi apresentado por Basili *et al.* (1996) que cinco das seis métricas de sistemas orientados a objetos proposta por Chidamber e Kemerer são úteis na previsão de falhas em classes, já na fase inicial de desenvolvimento.

Outra relação existente é a entre as métricas de orientação a objetos e o esforço de manutenção de um sistema em caso de mudança de requisitos (Li e Henry, 1993). Esse estudo indica que uma combinação de métricas pode ser usada para prever o esforço para manutenção de um sistema orientado a objetos, sendo um indicador da sua qualidade. Outra tentativa de validar as diversas métricas propostas na literatura é apresentada por Briand *et al.* (2000) que explora as relações entre métricas de projeto orientado a objetos e a qualidade do software.

Em suma, há uma variedade de métricas baseadas análise estática do código-fonte que permitem a avaliação de produtos de software (Gousios *et al.*, 2007). Para uma melhor visão e entendimento, os tipos e métricas de código-fonte mais referenciados e conhecidas do estado da arte de métricas de software são brevemente apresentados no Apêndice A desta tese. Nesta seção manteremos a discussão nos argumentos do porquê monitorar métricas de código-fonte, bem como apresentaremos as métricas selecionadas (e implementadas) para nossas ferramentas estudos.

3.1.1 Por que métricas de código-fonte?

Qualquer que seja a metodologia de desenvolvimento, monitorar a qualidade do software é fundamental. Dentre as inúmeras características que fazem um bom software, várias delas podem ser percebidas no código-fonte, e algumas são exclusivas dele. Por exemplo, código compilado pode ser analisado, mas características como organização e legibilidade são perdidas; mesmo uma bateria de testes com ótima cobertura só apresenta informação sobre o funcionamento atual, não refletindo manutenibilidade, modularidade, flexibilidade e simplicidade. Nesse contexto, as métricas de código-fonte complementam as outras abordagens de monitoramento da qualidade do software.

Engenheiros de software e pesquisadores precisam analisar códigos-fonte para entender melhor projetos de software. Por exemplo, com o advento e reconhecimento dos Métodos Ágeis e do Software Livre, o código-fonte é destacado como tendo papel central nas atividades de desenvolvimento de software. As funcionalidades são "entregues" constantemente aos usuários em pequenos incrementos, de forma mensal, semanal ou, até mesmo, diária. Portanto o código-fonte é escrito gradualmente e diferentes desenvolvedores fazem atualizações e melhorias continuamente (Martin, 2008). Assim, novas funcionalidades são inseridas e erros corrigidos durante as iterações de desenvolvimento e manutenção.

Desenvolvedores tomam decisões sobre o código enquanto estão programando. Em conjunto, essas decisões influenciam fortemente a qualidade do código (Beck, 2007). Observar os atributos do código que está desenvolvendo pode auxiliar o programador na tomada dessas decisões. No contexto dos projetos de software livre, há estudos que mostram que a motivação para se envolver com um projeto de software livre é criticamente afetada pela modularidade do seu código (Baldwin e Clark, 2006) e sua organização e complexidade influenciam o número de *downloads* e membros do projeto (Meirelles *et al.*, 2010).

De modo geral, existe uma diferença significativa tanto em tempo gasto como em quantidade de linhas que um desenvolvedor lê e escreve. Como é preciso entender uma implementação lendo código (inclusive escrito no passado pelo próprio leitor/programador) para poder melhorará-la ou incrementá-la, código-fonte deve ser escrito para ser entendido principalmente por pessoas, e não pela máquina (Martin, 2008). Nesse cenário, métricas de código-fonte podem auxiliar no desenvolvimento de código claro, simples e flexível. A partir de uma coleção de métricas coletadas automaticamente e de uma forma objetiva de interpretar seus valores, engenheiros de software podem

monitorar características específicas do seu código – assim como implementações problemáticas – para tomar melhores decisões ao (re)escrevê-las.

Métricas de código-fonte foram propostas desde que os primeiros conceitos da engenharia de software surgiram. As métricas pioneiras foram rapidamente absorvidas pela indústria, que as usa com frequência. Métricas de complexidade e tamanho são as mais usuais, por exemplo LOC (linhas de código) e seus derivados (p.ex., erros/LOC, LOC/tempo). Essas métricas são reconhecidamente limitadas quando usadas isoladamente. Mas mesmo com o avanço da pesquisa em métricas desde então, a indústria continua usando as mesmas métricas simplistas e de forma isolada (Fenton e Neil, 1999).

Especialistas em métricas podem ser capazes de entender e usar métricas, mas todos desenvolvedores deveriam saber como usá-las para monitorar e melhorar seu código. Facilitar o monitoramento pelos desenvolvedores em geral é uma das contribuições desta tese de doutorado ao mostrar (no Capítulo 4) como é possível monitorar e estudar um conjunto de métricas de código-fonte de projetos de software livre e acompanhar a sua evolução. Também, complementando isso, apresentamos neste contexto (no Capítulo 5) um conjunto de ferramentas (i.e., uma plataforma) que sistematiza a avaliação do código-fonte. Antes de chegarmos nesses pontos, apresentamos em detalhes o conjunto de métricas que nortearam os estudos nesta tese.

3.1.2 Métricas selecionadas

Durante os 3 primeiros anos que envolveram as pesquisas e esforços de implementação relacionados a esta tese, colaboramos com o projeto QualiPSo. Parte das métricas que implementamos em nossas ferramentas e, consequentemente conduzimos estudos, foram baseados nos resultados obtidos em conjunto com os grupos de trabalhos do projeto QualiPSo para podermos trabalhar com número de métricas em comum entre os grupos de pesquisa (del Bianco *et al.*, 2007, 2008a,b,c).

Além dessa seleção baseada na sinergia e necessidades do projeto QualiPSo, selecionamos algumas, entre as inúmeras métricas propostas pela literatura, tendo em vista medir os aspectos mais relevantes à manutenibilidade do software, como sua complexidade interna, modularidade e grau de dependência entre os módulos. Outro critério utilizado na escolha das métricas que detalharemos nesta seção foi a existência de trabalhos que, para elas, sugerem valores teóricos de referência para compararmos nossos estudos posteriores.

Por exemplo, um estudo analisou diferentes métricas para 40 programas desenvolvidos em Java e disponíveis no SourceForge.net (Ferreira *et al.*, 2009). Esses programas pertenciam a 11 diferentes domínios de aplicação, entre eles: *clustering*, bancos de dados, arcabouços, jogos, multimídia, rede e segurança. O trabalho sugere então intervalos qualitativos para as métricas analisadas.

Em um dos trabalhos mais reconhecidos da área de métricas de código-fonte, (Lanza e Marinescu, 2006) coletaram diversas métricas de tamanho, complexidade e herança de 45 sistemas feitos em Java e 37 em C++. Seus valores são apresentados no livro de sua autoria, que sugere valores considerados como baixo, médio, alto e muito alto para cada métrica, obtidos através da média e desvio padrão dos dados coletados dos sistemas estudados.

A seguir, a descrição dos conceitos de cada métrica selecionada, bem como, quando encontrados, os valores teóricos de referência de acordo com a literatura (mas que questionamos e discutimos em um dos estudos apresentados do Capítulo 4). Começando pelas métricas de tamanho:

- **LOC** (*Lines of Code* — Número de linhas de código) é a medida mais comum para o tamanho de um software. São contadas apenas as linhas executáveis, ou seja, são excluídas linhas em branco e comentários. Para efetuar comparações entre sistemas usando LOC, é necessário que ambos tenham sido feitos na mesma linguagem de programação e que o estilo esteja normalizado (Jones, 1991). Os intervalos sugeridos para o LOC de uma classe (Java e C++) são: até 70 (bom); entre 70 e 130 (regular); de 130 em diante (ruim).

- **AMLOC** (*Average Method LOC* — Média do número de linhas de código por método). Essa medida indica se o código está bem distribuído entre os métodos. Quanto maior, mais

"pesados" são os métodos. É preferível ter muitas operações pequenas e de fácil entendimento que poucas operações grandes e complexas. Os intervalos sugeridos são: até 10 (bom); entre 10 e 13 (regular); de 13 em diante (ruim). Uma outra métrica derivada de LOC, ao invés do número médio por método, é a **MMLOC** (*Max Method LOC*), número de linhas de código do maior método da classe.

- *Total Number of Modules or Classes* — Número total de módulos ou classes é outro indicador de tamanho, que é menos influenciada por linguagens de programação, nível dos desenvolvedores e estilos de codificação. Assim, essa métrica pode ser usada para comparar projetos escritos em diferentes linguagens de programação, por exemplo (Tempero, 2008).

Ao considerar características como esforço de manutenibilidade, flexibilidade, compreensão e qualidade do código fonte, em geral, deve-se levar em consideração não só as métricas de tamanho acima descritas, mas também indicadores estruturais, como descritos a seguir:

- **NOA** (*Number of Attributes* – Número de atributos) calcula o número de atributos de uma classe. Seu valor mínimo é zero e não existe um limite máximo para o seu resultado. Uma classe com muitos atributos pode indicar que ela tem muitas responsabilidades e apresentar pouca coesão, i.e., deve estar tratando de vários assuntos diferentes.

- **NOM** (*Number of Methods* — Número de métodos) é usado para medir o tamanho das classes em termos das suas operações implementadas. Essa métrica é usada para ajudar a identificar o potencial de reúso de uma classe. Em geral, as classes com um grande número de métodos são mais difíceis de serem reutilizadas, pois elas são propensas a serem menos coesas (Lorenz e Kidd, 1994). Assim, é recomendado que uma classe não tenha um número excessivo de métodos (Beck, 1997).

- **NPA** (*Number of Public Attributes* — Número de atributos públicos) mede o encapsulamento. Os atributos de uma classe devem servir apenas às funcionalidades da própria classe. Portanto, boas práticas de programação recomendam que os atributos de uma classe devem ser manipulados através dos métodos de acesso (Beck, 1997), os atributos de uma classe devem ser privados, indicando que o número ideal para essa métrica é zero. Os intervalos sugeridos para Java e C++ são: até 1 (bom); entre 1 e 9 (regular); de 9 em diante (ruim).

- **NPM** (*Number of Public Methods* — Número de métodos públicos) representa o tamanho da "interface" da classe. Métodos estão diretamente relacionados às operações previstas na respectiva classe. Altos valores para essa métrica indicam que uma classe tem muitos métodos e, provavelmente, muitas responsabilidades, o que conflita com boas práticas de programação (Beck, 1997). Os intervalos sugeridos para Java e C++ são: até 10 (bom); entre 10 e 40 (regular); de 40 em diante (ruim).

- **ANPM** *Average Number of Parameters per Method — Média do Número de Parâmetros por Método)*: calcula a média de parâmetros dos métodos da classe. Seu valor mínimo é zero e não existe um limite máximo para o seu resultado, mas um número alto de parâmetros pode indicar que um método pode ter mais uma responsabilidade, ou seja, mais de uma função (Bansiya e Davi, 1997).

- **DIT** (*Depth of Inheritance Tree* — Profundidade da árvore de herança) é o número de superclasses ou classes ancestrais da classe sendo analisada. São contabilizadas apenas as superclasses do sistema, ou seja, as classes de bibliotecas não são contabilizadas. Nos casos onde herança múltipla é permitida, considera-se o maior caminho da classe até uma das raízes da hierarquia. Quanto maior for o valor DIT, maior é o número de atributos e métodos herdados, e portanto maior é a complexidade. Os intervalos sugeridos são: até 2 (bom); entre 2 e 4 (regular); de 4 em diante (ruim). Entretanto, valores baixos indicam pouco reúso de código via herança.

- **NOC** (*Number of Children — Número de filhos*): número total de filhos de uma classe (Rosenberg e Hyatt, 1997).

- **RFC** (*Response For a Class — Respostas para uma classe*: número de métodos dentre todos os métodos que podem ser invocados em resposta a uma mensagem enviada por um objeto de uma classe (Sharble e Cohen, 1993).

- **ACCM**(*Average Cyclomatic Complexity per Method* — Média da Complexidade Ciclomática por método) mede a complexidade do programa (McCabe, 1976). Essa métrica pode ser representada através de um grafo de fluxo de controle, onde os nós representam uma ou mais instruções sequenciais e os arcos orientados indicam o sentido do fluxo de controle entre várias instruções. A complexidade ciclomática de um determinado grafo pode ser calculada através de uma fórmula da teoria dos grafos:

  $$v(G) = e - n + 2$$

  onde e é o número de arestas e n é o número de nós do grafo.

  Surgiram propostas de extensões e alterações no cálculo dessa métrica para melhorar a sua validade (Myers, 1977; Stetter, 1984). Myers observou que complexidade ciclomática mede a complexidade do software, mas não consegue diferenciar a complexidade de alguns casos simples, em especial que envolvam uma única condição. Para melhorar a fórmula original, foi sugerido:

  $$v(G)' = [l : u]$$

  onde l e u são limites inferiores e superiores, respectivamente, para a complexidade, sendo mais satisfatórios para os casos verificados por Myers. Por outro lado, Stetter (1984) propôs que o grafo fosse expandido para incluir declarações e referências de dados, mostrando uma complexidade mais completa. Nesse grafo do fluxo do programa (H), geralmente, há múltiplos nós de entradas e saídas. Assim, as irregularidades verificadas por Myers são eliminadas por uma função $f(h)$.

  O interesse na métrica de complexidade ciclomática levou à sua normalização (McCabe, 1982). Posteriormente, McCabe definiu outras cinco métricas: *complexidade da unidade real, complexidade essencial, complexidade do projeto de módulos, complexidade total do projeto e complexidade de integração* (McCabe e Butler, 1989).

Acoplamento é uma medida de como uma classe está ligada a outras classes no software. Altos valores de acoplamento indicam uma maior dificuldade para alterar uma classe do sistema, pois uma mudança em uma classe pode ter um impacto em todas as outras classes que são acopladas a ela. Em outras palavras, se o acoplamento é alto, o software tende a ser menos flexível, mais difícil de se adaptar e modificar e mais difícil de entender. Métricas que medem o acoplamento são explicadas abaixo:

- **ACC** (*Afferent Connections per Class* — Conexões aferentes de uma classe). Mede a conectividade de uma classe. Se uma classe C_c acessa um método ou atributo da classe C_s, dizemos que C_c é cliente da classe fornecedora C_s e denotamos por $C_c \Rightarrow C_s$. Consideremos a seguinte função:

  $$cliente(C_i, C_j) = \begin{cases} 1 & \text{sse } C_i \Rightarrow C_j \wedge C_i \neq C_j \\ 0 & \text{caso contrário} \end{cases}$$

  Então $ACC(C) = \sum_{i=1}^{n} cliente(C_i, C)$, onde n é o número total de classes do sistema. Se o valor dessa métrica for grande, uma mudança na classe tem potencialmente mais efeitos colaterais, tornando mais difícil a manutenção. Os intervalos sugeridos são: até 2 (bom); entre 2 e 20 (regular); de 20 em diante (ruim).

- **CBO** (*Coupling Between Objects* — Acoplamento entre objetos) é a recíproca da métrica ACC. Mede quantas classes são utilizadas pela classe analisada, ou seja,

  $CBO(C) = \sum_{i=1}^{n} cliente(C, C_i)$.

- **COF** (*Coupling Factor* — Fator de acoplamento) indica o quão interconectado é o software. Seu valor é dado por:

  $$COF = \frac{\sum_{i=1}^{n} ACC(C_i)}{n^2 - n}$$

  O numerador é o total de ligações entre as classes e o denominador é o total possível de ligações. Portanto, COF é a razão de ligações. Um software fortemente conectado apresenta maior COF, indicando um baixo grau de independência entre os módulos, alta complexidade e difíceis entendimento e manutenção. Os intervalos sugeridos são: até 0,02 (bom); entre 0,02 e 0,14 (regular); de 0,14 em diante (ruim).

Coesão mede a diversidade de "assuntos" que uma classe implementa. Altos valores de coesão indicam se o "foco" de uma classe está em um único aspecto do sistema. Enquanto uma baixa coesão indica que a classe trata de diferentes aspectos. Assim, uma classe deve ser coesa. Uma métrica comumente usada para análise do grau de coesão é:

- **LCOM** (*Lack of Cohesion in Methods* — Ausência de coesão em métodos), foi originalmente proposta por Chidamber e Kemerer (1994). Valores grandes para LCOM indicam uma baixa coesão, enquanto valores baixos indicam uma alta coesão. A primeira definição de LCOM, chamado LCOM1, corresponde ao número de pares de métodos de uma classe que manipulam as variáveis da mesma classe. Após ter recebido críticas e revisões, foi definida uma versão da LCOM revista por Hitz e Montazeri, conhecida como LCOM4 (Hitz e Montazeri, 1995). Para calcular LCOM4 de um módulo, é necessário construir um gráfico não-orientado em que os nós são os métodos e atributos de uma classe. Para cada método, deve haver uma aresta entre ele e um outro método ou variável que ele usa. O valor da LCOM4 é o número de componentes fracamente conectados nesse gráfico. Seja $M = \{M_1, \ldots, M_n\}$ o conjunto dos métodos da classe analisada. Dois métodos M_i e M_j estão relacionados se ambos acessam pelo menos um mesmo atributo da classe ou se M_i chama ou é chamado por M_j. LCOM4 é a quantidade de partições de M formadas após separar os métodos em conjuntos de métodos relacionados. Coesão entre os métodos de uma classe é uma propriedade desejável, portanto o valor ideal dessa métrica é 1. Se uma classe tem diferentes conjuntos de métodos não relacionados entre si, é um indício de que a classe deveria ser quebrada em classes menores e mais coesas. Os intervalos sugeridos para código C++ e Java são: até 2 (bom); entre 2 e 5 (regular); de 5 em diante (ruim).

- **SC** (Structural Complexity — Complexidade estrutural). Quanto mais complexo for um software, mais difícil será alterá-lo e evoluí-lo. Acoplamento e coesão foram descritos e discutidos em outros trabalhos como indicadores essenciais de complexidade estrutural (Darcy *et al.*, 2005). É aconselhável manter um baixo acoplamento e uma alta coesão das classes (Richter, 1999). Darcy *et al.* (2005) mostrou que, individualmente, tanto acoplamento quanto coesão não estão relacionados ao esforço de manutenção de software. Ambos devem ser considerados em conjunto. Quando combinado como uma métrica, o produto de acoplamento e coesão é positivamente correlacionado ao esforço de manutenção. Portanto, nós usamos nesta pesquisa de doutorado o produto de acoplamento (CBO) e coesão (LCOM4) como nossa métrica de complexidade estrutural (Darcy *et al.*, 2005).

Capítulo 4

Estudos

Neste capítulo apresentamos os estudos empíricos realizados durante nossa pesquisa de doutorado. Respondemos explicitamente e discutimos as questões de pesquisa desta tese:

- QP1 - Métricas de código-fonte podem influir na atratividade de projetos de software livre?
- QP2 - Quais métricas devem ser controladas ao longo do tempo?
- QP3 - As métricas de código-fonte melhoram com o amadurecimento (maior número de contribuidores e atualizações no repositório) dos projetos?

Para melhor responder tais questões, detalharemos os seguintes estudos e resultados:

Objetivos científicos

- OC1 – Estudo inicial da influência de métricas de código-fonte na atratividade de projetos de software livre escritos na linguagem C a partir do código disponível em 6.700 projetos.
- OC2 - Estudo mais detalhado da influência de métricas de código-fonte na atratividade de projetos de software livre, com mais de 100 mil downloads, escritos nas linguagens C, C++ ou Java a partir do código disponível em 756 projetos.
- OC3 - Estudo abrangente da influência de métricas de código-fonte na atratividade de projetos de software livre escritos nas linguagens C, C++ ou Java a partir do código disponível atualmente (obtido dos repositórios de sistemas de gerenciamento de versões) de 8.450 projetos.
- OC4 - Identificação das distribuições estatísticas dos valores de métricas em 38 projetos de software livre (35 com mais contribuidores no intervalo de 12 meses e 3 de avaliação de métricas), a fim de compreender qual a abordagem estatística mais informativa para o monitoramento dessas métricas, bem como observar os valores frequentes para essas métricas, de forma a servirem de referência para projetos futuros.
- OC5 - Análise estatística da correlação entre métricas a fim de definir um subconjunto reduzido de métricas que podem ser monitoradas ao longo do tempo e ainda oferecer uma boa visão do projeto.
- OC6 - Estudo longitudinal da evolução dos valores das métricas de código-fonte e do número de contribuidores e atualizações nos repositórios dos projetos de software livre Mozilla Firefox e Chrome.

4.1 Trabalhos relacionados

Uma revisão sistemática de 63 estudos empíricos mostra que existem poucas pesquisas que abordam as características ou propriedades de projetos de Software Livre, tais como a qualidade, crescimento e evolução (Stol *et al.*, 2009). Por exemplo, nossos estudos contribuem com uma análise sem precedentes de métricas de código-fonte de milhares de projetos de software livre, associando os atributos do código-fonte com a atratividade dos projetos. Ao fazer isso, chamamos a atenção dos desenvolvedores e pesquisadores para um tema ainda negligenciado em nossa opinião: projetos de software livre não tem sucesso quando não são atrativos. Portanto, compreender o que influencia a capacidade de atração e fornecer esse conhecimento gerencial aos líderes de projeto, apontando uma direção para priorizar os seus recursos, é uma das contribuições desses estudos iniciais.

Grandes projetos de software livre, como Debian, Gnome e KDE investem na criação de equipes de garantia da qualidade. Esses esforços envolvem a correção de erros, remoção de componentes obsoletos e definição de normas e estratégias para prevenir erros e melhorar a qualidade (Michlmayr *et al.*, 2005). Entretanto, a maioria dos projetos de software livre não possuem essa infraestrutura e não tem recursos nem grandes comunidades que possibilite que eles tenham um time dedicado à qualidade.

Michlmayr *et al.* (2005) realizaram um estudo sobre problemas e a garantia de qualidade em software livre, como código sem suporte, gerenciamento de configuração, atualizações de segurança, usuários que não sabem como relatar erros, dificuldade em atrair voluntários, falta de documentação e problemas com a coordenação e comunicação com a comunidade. Nenhum desses problemas, no entanto, está relacionados às características do código-fonte.

Barkmann *et al.* (2009) analisaram 146 projetos de software livre escrito em Java, identificando a correlação entre um conjunto de métricas de orientação a objeto e os seus valores teoricamente ideais. No entanto, em seu trabalho os valores de métricas de código-fonte, não foram associados com problemas ou com a atratividade dos projetos de software livre.

Stamelos *et al.* (2002) apresentaram resultados empíricos sobre a relação entre o tamanho dos componentes do software e aspectos de qualidade como a satisfação dos usuários. Nesse trabalho, as características de qualidade de 100 aplicativos escritos para GNU/Linux foram comparados com os padrões industriais. Os resultados indicaram que as chamadas estruturas de qualidade (por exemplo, o tamanho dos componentes) de um software estão relacionado com a satisfação do usuário.

Midha (2008) analisou 450 projetos do SourceForge.net e verificou que altos valores de complexidade ciclomática e métricas de Haltead (que são métricas de complexidade) são correlacionados positivamente com o número de erros e com o tempo necessário para corrigi-los. Essas métricas também foram correlacionadas negativamente com as contribuições de novos desenvolvedores, ou seja, quanto mais complexo o código, menos novos desenvolvedores são atraídos. No entanto, esse estudo usou métricas de complexidade medido ao nível de sub-rotina (métodos), enquanto em nosso estudo utilizamos métricas de complexidade no nível de classes (e módulos).

Capra *et al.* (2008) mostram que o gerenciamento aberto está associado a uma maior qualidade do projeto (*design*) do software, em um estudo com 75 projetos de software livre. Eles definiram qualidade de software em termos de 5 métricas de orientação a objetos, dos quais apenas CBO é utilizada em nossos estudos. Uma estrutura de gerenciamento aberto, juntamente com a falta de gestão formal e prazos rigorosos permitem aos desenvolvedores melhorar a concepção do software para ter um produto de alta qualidade, uma vez que não sofrem pressão para lançamento de versões em prazos curtos (Capra *et al.*, 2008). Além disso, um melhor *design* do software promove um gerenciamento mais aberto, permitindo que os desenvolvedores trabalhem em módulos independentes, sem a necessidade de coordenação explícita de atividades. No entanto, esse estudo não abordou a questão da atratividade.

Barbagallo *et al.* (2008) analisaram 56 projetos de software livre, para estudar a relação entre a qualidade do projeto (*design*) do software e sucesso do software. Eles definiram o sucesso em termos de *downloads*, visualização de páginas e atividades de desenvolvimento. A qualidade do *design* do software foi definida em termos de métricas de orientação a objeto: CBO (*Coupling Between Objects*), DIT (*Depth of Inheritance tree*), MIF (*Method Inheritance Factor*) e NOC (*Number of*

Children). Os resultados indicaram que os projetos mais bem sucedidos apresentaram menor qualidade no *design* do software. Eles argumentam que, talvez, os principais desenvolvedores dos projetos de sucesso tendem a transferir sua atenção para atividades laterais, como responder aos usuários em fóruns, em vez de se concentrar em melhorar a qualidade do código. Nossos resultados parecem contradizer os deles, mas não é o caso. Em primeiro lugar, a sua conceituação de sucesso é diferente de nossa conceituação de atratividade. Além disso, levamos em consideração a complexidade estrutural em termos de CBO e LCOM4, enquanto eles utilizaram um conjunto diferente de métricas para representar a percepção de qualidade do *design* do software, assim, temos apenas CBO em comum com esse estudo. Portanto, uma comparação direta entre esse estudo e os nossos não é factível.

Os estudos citados, assim como os nossos, têm uma limitação do ponto de vista de não avaliarem se a média das métricas é uma medida representativa, informativa estatisticamente, para tais análises. Mesmo estudo mais recentes baseados nas análises de métricas também não deixam claro a representatividade da média das métricas. Por exemplo, Williams e Carver (2010) determinaram os tipos de mudanças na arquitetura de software e propuseram uma caracterização dessas mudanças, o que também envolveu a análise de métricas de complexidade de código-fonte. Terceiro e seu grupo (Terceiro, 2010; Terceiro e Chavez, 2009b; Terceiro *et al.*, 2010b, 2012) realizaram estudos para definir uma caracterização da complexidade estrutural em sistemas de software livre. Eles analisaram a complexidade estrutural média entre todos os módulos de 13 projetos de software livre, de diferentes domínios de aplicação e linguagens de programação, observando o histórico de mudanças desses sistemas. Na próxima seção, verificaremos em detalhes a distribuição e os valores frequentes da métrica de complexidade estrutural em projetos de software livre, assim proveremos, como uma de nossas contribuições, as informações necessárias para estudos como esses (Terceiro, 2010; Terceiro e Chavez, 2009b; Terceiro *et al.*, 2010b, 2012; Williams e Carver, 2010) não terem essa limitação quanto à medida estatística informativa.

Na evolução dos estudos desta tese, que envolveram análises estatísticas, num primeiro momento replicamos o que está descrito na literatura, como foi o caso dos estudos sobre atratividade de projeto de software livre e métricas de código-fonte, apresentados na próxima seção. Entretanto, alguns outros trabalhos relacionados mencionam que métricas de software, incluindo as métricas de código-fonte, têm sido analisados como dados que são leis de potência (Clauset *et al.*, 2007), em especial as métricas de orientação a objetos. Isso significa que, de acordo com esses trabalhos, as métricas de código-fonte possuem distribuições estatísticas que têm como característica geral ter cauda longa e ser de escala livre, o que significa que a média não é um valor informativo (Clauset *et al.*, 2007).

Depois de uma avaliação mais cuidadosa, observarmos que a maioria desses trabalhos estavam analisando apenas códigos em Java e generalizando suas observações para todos os projetos orientados a objetos. Além disso, os critérios de seleção dos projetos avaliados não contemplam, em uma amostra maior, os principais projetos de software livre disponibilizados, em particular os com mais desenvolvedores ativos e atualizações nos repositório (atividade de desenvolvimento).

Por exemplo, Wheeldon e Counsell (2003) avaliaram 4 projetos em Java: JDK (Java Development Kit), Apache, Ant e Tomcat. Eles identificaram leis de potência para as métricas de número de atributos, número de métodos. Um estudo com poucos projetos e poucas métricas, mas que levantou a questão das leis de potências. Potanin *et al.* (2005) analisaram 35 programas Java e observaram que, neles, os relacionamentos entre os objetos constituem um grafo livre de escala, que difere de um grafo com arestas distribuídas aleatoriamente. Em um de grafo livre de escala, uma vez que a distribuição dos graus de seus vértices seguem uma lei de potência, a média não é representativa. Fizeram constatações interessantes, mas limitados a Java e a seleção não clara dos projetos não permite generalizar os resultados. Baxter *et al.* (2006) coletaram métricas de 56 projetos de software livre em Java. Eles demonstram que nem todas as métricas estão sob leis de potência, indicando que algumas métricas, como grau de entrada (*fan-in*) e número de subclasses, seguem leis de potência, mas outras métricas não, como grau de saída (*fan-out*), número de atributos e número de atributos públicos. Louridas *et al.* (2008) avaliaram partes de 11 projetos (10 livres e um restrito) escritos em Java, C, Ruby e Perl. Eles investigaram apenas o grau de entrada (*fan-in*) e de saída (*fan-out*) dos módulos e classes. Seus resultados indicam que essas métricas possuem distribuições na família

de leis de potência, independente do paradigma de programação. Em relação à métrica de grau de saída, o estudo de Baxter *et al.* (2006) apresenta resultados diferentes. Isso nos levou a refletir pela literatura que as métricas que trabalhamos nesta tese não são governadas necessariamente pelas leis de potência; além disso, não podemos generalizar para uma distribuição normal.

Mesmo com esses estudos sobre as leis de potência, observamos que, na área de avaliação de métricas de código-fonte e visualização de software, os trabalhos de Michele Lanza, em particular o seu livro (Lanza e Marinescu, 2006), são os mais reconhecidos. Entretanto, não se verifica cuidadosamente como seus dados foram analisados. Lanza e Marinescu (2006) definiram valores referência para 3 métricas de código-fonte (número de métodos por classe, número de linhas de código linhas de código e complexidade ciclomática por método) baseado em informações estatísticas. Eles avaliaram essas 3 métricas em 37 programas desenvolvidos em C++ e 45 em Java, entre projetos de software livre e software restrito. (Lanza e Marinescu, 2006) generalizaram as análises e consideraram os valores das métricas como uma distribuição Normal. Dessa forma, um cálculo simples com a média e o desvio padrão foi feito para definir o que eles denominaram de intervalos de referência: o valor de uma métrica é considerando fora desse padrão quando ele é 50% maior que o valor do intervalo determinado. Em resumo, eles assumem a média como um valor informativo para suas análises.

Na mesma linha do trabalho de Lanza e Marinescu (2006), na tentativa de definir valores de referência, Ferreira *et al.* (2009) conduziram um estudo que analisou diferentes métricas para 40 programas, de 11 diferentes domínios de aplicação, desenvolvidos em Java. Esse trabalho sugere intervalos qualitativos para 6 métricas: COF (conectividade), LCOM (coesão), DIT (profundidade da arvore de herança), ACC (conexões aferentes), NPA (atributos públicos) e NPM (número de métodos públicos). Eles argumentam que essas métricas podem ser modeladas pelas distribuições Weibull e Poisson, que são leis de potência, sendo possível identificar valores típicos para tais métricas e usá-los com referência para o desenvolvimento de software de qualidade. Mesmo sendo uma tentativa de definir valores de referência, a limitação pela linguagem Java e por não contemplar os principais projetos de software livre, somado às contradições e generalização dos trabalhos relacionados citados, nos levou a verificar em detalhes as distribuições de cada métrica nesta tese (como apresentado na seção 4.3). Por fim, em particular, na abordagem de Ferreira *et al.* (2009) não ficou clara a definição dos pontos de corte para a indicação dos intervalos, bem como nos testes das distribuições foi usada uma ferramenta restrita (fechada), o que impede a reprodução na íntegra do referido estudo.

Nesse contexto, como visto, as distribuições estatísticas de métricas de código-fonte têm sido estudadas. Por um lado, em particular as métricas para códigos escritos em linguagens orientadas a objetos, no caso Java, parecem obedecer a leis de potência (Concas *et al.*, 2007; Ferreira *et al.*, 2009; Potanin *et al.*, 2005; Wheeldon e Counsell, 2003; Yao *et al.*, 2009). Por outro lado, observamos que há indícios que não necessariamente algumas métricas têm suas distribuições de modo que seu comportamento siga uma lei de potência (Baxter *et al.*, 2006; Herraiz *et al.*, 2012, 2011; Lanza e Marinescu, 2006).

Estamos argumentando e apresentando aqui que não há consenso sobre se as métricas de código-fonte podem ser melhor descritas usando leis de potência, distribuição normal ou lognormal (como foi o caso dos nossos estudos iniciais, por exemplo). Nesta pesquisa, essa informação é relevante para determinarmos qual medida estatística (média, mediana ou algum percentil) devemos observar para monitorarmos as métricas de código-fonte. Também, se há essa dispersão de abordagens sobre as distribuições da métricas em códigos escritos no paradigma de orientação a objetos, que vem sendo o alvo de estudos da maioria dos trabalhos, no contexto de projetos de software livre, em que boa parte dos projetos são escritos em C (Robles *et al.*, 2006), essa questão se apresenta mais aberta ainda.

4.2 Relacionamento entre métricas de código-fonte e a atratividade de projetos de software livre

De acordo com os conceitos de atratividade sobre o relacionamento entre os atributos de código-fonte e a atratividade de projetos de software, investigamos se dois atributos, tamanho e complexidade estrutural, obtidos através de quatro métricas de código-fonte podem ou não influenciar a atratividade. Para tanto, definimos 3 hipóteses:

- **H1** – ***Projetos de software livre com alta complexidade estrutural do seu código-fonte são menos atrativos.*** Quanto maior a complexidade do software, mais difícil é compreender seu código-fonte para manutenção e evolução. Isto leva a um aumento do esforço de manutenção e torna mais difícil atrair novos membros e usuários para o projeto. Ao longo do tempo, com menos membros e usuários, o projeto pode perder sua capacidade de adicionar novas funcionalidades e corrigir erros e, consequentemente, sua capacidade de evoluir e atender às necessidades do usuário.

- **H2** – ***Projetos de software livre com mais linha de código são mais atrativos.*** De certa forma, entendemos que as linhas de código refletem a quantidade de funcionalidades implementadas em um projeto e a quantidade de trabalho que foi colocada nele. Portanto, os projetos com mais linhas de código geralmente atraem mais usuários, uma vez que tem mais funcionalidade, e também atraem mais desenvolvedores, já que um projeto grande oferece mais oportunidades de contribuição e reconhecimento perante a comunidade de software livre.

- **H3** – ***Projetos de software livre com muitos módulos ou classes são mais atrativos.*** O número de módulos ou de classes pode indicar o tamanho do projeto e as possibilidades de trabalho em paralelo, em partes independentes. Mais classes podem indicar uma preocupação com um bom projeto (*design*) e uma melhor modularização, o que facilita as contribuições. Atraindo mais membros que podem escrever mais funcionalidade e corrigir mais erros, naturalmente, o projeto tende também a atrair mais usuários.

4.2.1 Amostra de projetos e coleta dos dados

O SourceForge.net disponibiliza seus dados e meta-dados, o que permite inúmeros pesquisas na área de software livre. Em nosso estudo, utilizamos os dados disponíveis na base de dados gerenciada pela Universidade de Notre de Dame[1] e nos conjuntos de dados disponibilizados pelo projeto FLOSSmole[2]. Acessamos inicialmente essas bases de dados e coletamos informações sobre todos os projetos que atendiam aos seguintes critérios:

- *Projetos com código-fonte escritos na linguagem C, C++ e Java.* Enquanto boa parte dos programas livres são escrito em C (Robles *et al.*, 2006), a maioria das pesquisas concentram suas análises apenas em projetos escritos em Java (ver a Seção 4.1). Dada essa disparidade entre a realidade do ecossistema do software livre e as pesquisas, optamos no primeiro estudo por avaliar os projetos em C. Nos demais estudos, incluímos também os projetos escritos em C++ e Java, uma vez que nossas ferramentas também permite o cálculo de métricas para essas linguagens.

- *Mais de um download.* Provavelmente, projetos sem *downloads* não estão em desenvolvimento, ou iniciaram suas atividade há pouco tempo, ou representam cadastros incorretos no SourceForge.net. No segundo estudo, selecionamos os projetos com mais de 100 mil downloads para observarmos apenas os mais populares, segundo este parâmetro.

Com esses critérios iniciais de filtragem de projetos, dos aproximadamente 200 mil projetos disponíveis na base, obtivemos uma lista de 11.500 projetos para o primeiro estudo com os projetos

[1] nd.edu/~oss/Data/data.html
[2] flossmole.org

em C; para o segundo estudo, uma lista de 1600 projetos; no último estudo, uma lista de 42 mil projetos. Baseado nessas listas, em momento distintos do desenvolvimento deste trabalho, fizemos os seguintes passos, auxiliado por nossas ferramentas:

1. Para os dois primeiros estudos, baixamos o código-fonte disponível na páginas de arquivos de todos os projetos listados. Isso resultou em 10 mil projetos no SourceForge.net para o primeiro estudo e 1600 para o segundo estudo. Isso indicou que alguns projetos não disponibilizavam os fontes nas páginas dos arquivos dos projetos. Dessa forma, para o terceiro estudo, desenvolvemos um outro *script* para baixar os fontes diretos dos repositórios dos projetos, clonando o repositório de cerca de 22 mil projetos.

2. Limpamos os arquivos baixados, deixando apenas aqueles com extensões correspondentes às linguagens C, C++ e Java. Executamos a `Analizo` (que será apresentada na Seção 5.2) para todos os projeto baixados. Assim, computamos todos os valores das métricas e foi gerado um arquivo CSV fornecido pela Analizo. A Analizo conseguiu calcular as métricas para 6.773 projetos, 756 e 8.450, respectivamente na sequência dos estudos, isso porque (i) alguns projetos baixados não tinham o código-fonte disponível entre seus arquivos, por exemplo, apenas os binários; (ii) em alguns casos, o código-fonte não era escrito em C, C++ e ou Java (o projeto foi cadastrado de forma errada no SourceForge.net ou, (iii) em outros casos os arquivos não puderam ser processados pela Analizo por problemas com o código do projeto.

3. Em cada estudo, juntamos os conjuntos de dados. Por fim, as informações de número de *downloads* e membros, disponibilizadas na base da dados da Universidade de Notre Dame e FLOSSMole, e os valores das métricas de código-fonte de cada projeto foram cruzadas, gerando as bases de dados para nossas análises estatísticas.

4.2.2 Variáveis

Nesta seção, discutiremos como foram selecionadas as métricas (variáveis) apresentadas na Tabela 4.1. Essa tabela apresenta, na primeira parte, os valores naturais para mínimo, máximo, média aritmética e desvio padrão de cada variável, indicando as características da amostra do nosso primeiro estudo com projetos em C. Com esses valores, verificamos se a distribuição era normal. Por exemplo, observamos que a distribuição de probabilidade de Skewness e Kurtosis apresentaram valores elevados, o que indica uma não normalidade dos dados (Hair *et al.*, 2006). Devido a essa não normalidade, realizamos uma transformação logarítmica para linearizar os dados, o que reduziu os valores de Skewness e Kurtosis, assim adequando os valores das variáveis para executarmos uma regressão múltipla (Crowston e Scozzi, 2002).

Tabela 4.1: *Estatísticas descritivas das variáveis.*

	Natural				Logaritmo	
Variáveis	Min.	Max.	Média	Desvio padrão	Média	Desvio padrão
CBO	0,0015	711,50	2,26	9,04	0,35	0,98
LCOM	0,0004	262,00	4,77	12,00	1,01	1,09
SC	0	4.940,00	15,79	114,69	1,37	1,57
NM	1	7.177,00	74,98	276,54	3,08	1,39
LOC	11	2.983.103,00	17.722,23	91.614,70	8,28	1,58
Mbrs	1	288,00	2,90	6,19	059	0,79
DLs	6	941.498.760,00	956.674,26	17.760.732,37	8,20	2,66

De acordo com nossas hipóteses, selecionamos as seguintes métricas calculadas pela Analizo: número total de linhas de código (LOC), número total de módulos (NM), média do acoplamento

entre objetos (CBO) e média da falta de coesão entre métodos (LCOM4). Além de código C++ e Java, que seguem o paradigma de orientação a objetos, também avaliamos código em C. Para podermos aplicar ao paradigma procedimental da linguagem C, as métricas CBO e LCOM4, proposta originalmente no contexto da orientação a objetos, assumimos um mapeamento dos conceitos de "classe" e "método" para os conceitos da linguagem C: "arquivo-fonte" e "função".

LOC e NM foram coletadas, uma vez que medem diferentes aspectos do código. Também calculamos CBO e LCOM4 para obtermos o valor da complexidade estrutural (SC) do projeto – métrica apresentada na Seção 3.1.2. Portanto, nosso modelo de regressão múltipla é composto por:

- **Variáveis dependentes (atratividade)**
  - *Número de Downloads (DLs).*
  - *Número de Membros (Mbrs).*
- **Variáveis independentes (métricas de código-fonte)**
  - *Complexidade estrutural (SC)*;
  - *Linhas de código (LOC)*;
  - *Número de Módulos(NM)*;

O modelo usado nestes estudos tem a atratividade como elemento central, assim queremos explicar as suas causas. Dessa forma, as medidas de atratividade (*downloads* e membros) são chamadas de variáveis dependentes. Em outras palavras, são as variáveis explicadas pelos atributos de código-fonte, propostos em nossas hipóteses. Consequentemente, as métricas de código-fonte são as variáveis independentes, ou seja, os influenciadores de atratividade.

4.2.3 Resultados

Após a linearização dos valores das variáveis dependentes e independentes, testamos nossas hipóteses de acordo com nosso modelo de regressão múltipla. A Tabela 4.2 resume os resultados da regressão. Esses resultados estatísticos indicam uma dependência linear entre as métricas de código-fonte selecionadas e cada um dos indicadores de atratividade. Na Tabela 4.2, β é o coeficiente que indica o quão cada métrica influi na atratividade.

Tabela 4.2: *Resultados da regressão múltipla do primeiro estudo.*

	Downloads				Membros			
Métricas	β	Std. β	T-value	P-value	β	Std. β	T-value	P-value
(Constant)	1,551	-	6,12	<0.001	-0,668	-	-8,47	<0,001
SC (log)	-0,286	-0,150	-8,616	<0,001	-0,033	-0,058	-3,238	0,001
LOC (log)	0,856	0,506	18,624	<0,001	0,126	0,249	8,846	<0,001
NM (log)	0,008	0,004	0,186	0,852	0,087	0,148	6,625	<0,001
R	0,425				0,348			
R^2	0,180				0,121			

Como podemos observar na Tabela 4.2, LOC é mais fortemente correlacionada com *downloads* e membros que complexidade estrutural e número de módulos, de acordo com o valor "padronizado" de beta (*std.* β). Os valores padronizados de beta são calculados para realizar comparações entre as variáveis que são medidas por escalas diferentes, como no caso de linhas de código e complexidade estrutural. Não se pode comparar os valores normais desses coeficientes sem antes padronizá-los.

Além disso, a complexidade estrutural apresenta uma correlação negativa com atratividade, conforme o esperado. Nota-se que os valores T (teste) e P (probabilidade) representam se uma

métrica de código-fonte é um preditor significativo ou influenciador dos indicadores de atratividade. Para *downloads*, o número de módulos não é significativo, porque seu P-value é superior a 0,05. Finalmente, na última linha da Tabela 4.2, os valores de R^2 indicam o percentual de atratividade (usuários e desenvolvedores) que esse conjunto de métricas de código-fonte é capaz de explicar. Por exemplo, no primeiro estudo com os projetos em C, para chegarmos a esses valores nos obtemos as seguintes equações:

$$\begin{aligned} downloads = \quad & 1.551 - 0.286 \times log(SC) \\ & +0.856 \times log(LOC) + 0.008 \times log(NM) \end{aligned}$$

$$\begin{aligned} members = \quad & -0.668 - 0.033 \times log(SC) \\ & +0.126 \times log(LOC) + 0.087 \times log(NM) \end{aligned}$$

Cada equação tem um valor R. O coeficiente β de cada variável é o tamanho da influência que uma métrica de código-fonte (variável independente) tem sobre a atratividade (variável dependente). Então, uma unidade alterada na variável independente gera uma influência de tamanho β sobre a variável dependente, em média.

O valor R representa o quanto as variáveis dependentes podem ser explicadas pelo conjunto de variáveis independentes. Em nossa primeira análise, o valor R indicou que as métricas de código-fonte explicam 18% ($R^2 = 0.180$) do número de *downloads* e 12,1% ($R^2 = 0.121$) do número de membros. No nosso segundo estudo, com projetos com mais de 100 mil downloads, o valor R indicou que as métricas de código-fonte explicam 32% ($R^2 = 0.320$) do número de *downloads* e 19,6% ($R^2 = 0.196$) do número de membros. No último estudo, com código-fonte direto dos repositórios, o valor R indicou que as métricas de código-fonte explicam 26,2% ($R^2 = 0.262$) do número de *downloads* e 16,3% ($R^2 = 0.163$) do número de membros.

Esses valores são significativos nesse contexto no qual, de uma forma mais ampla, estamos lidando com aspectos das ciências sociais ao sabermos que inúmeras variáveis podem influenciar uma pessoa a usar e colaborar com um projeto de software livre: desde uma indicação de um site ou amigo, passando pela reputação do projeto, até chegar nas questões técnicas. Por meio desses estudos e suas respectivas análises estatísticas, podemos afirmar que respondemos positivamente a nossa primeira questão de pesquisa:"Métricas de código-fonte podem influenciar na atratividade de projetos de software livre?". Nas próximas seções vamos comparar estes 3 estudos com alguns trabalhos relacionados e avançarmos para discussão e análises dos demais estudos desta tese.

Nosso estudo se diferencia dos anteriores, principalmente, pelo tamanho da amostra de projetos de software livre. Nossa amostra foi concebida obedecendo critérios bem definidos e a quantidade de projetos envolvidos oferece uma confiabilidade estatística aos resultados significativamente melhor do que os resultados anteriores encontrados na literatura.

Nossos resultados indicaram que o tamanho do código-fonte e sua complexidade estrutural explicam um percentual razoável da atratividade de projetos de software livre. Atratividade é baseada em percepções humanas e influenciada pela cognição das pessoas, o que a torna algo complexo, difícil de entender e explicar completamente. No entanto, um de nossos estudos foi capaz de explicar até 32% em relação à atratividade de usuários e 19,6% para os desenvolvedores do projeto, através de quatro métricas de código-fonte.

Nestes estudos, mostramos que as linhas de código (LOC) tem um efeito positivo sobre o número de usuários do projeto. Nossos resultados também indicaram que a complexidade estrutural (SC) tem um impacto negativo na atratividade do projeto. Portanto, um projeto enfrenta maiores dificuldades se crescer sem observar alguns atributos do código-fonte, como a coesão, acoplamento e modularidade. O controle desses atributos pode favorecer as contribuições dos desenvolvedores como, por exemplo, a inclusão de novas funcionalidades e a correções de erros.

Em outras palavras, nossas análises indicam que o crescimento da complexidade estrutural do software pode diminuir os efeitos positivos da inserção de novas funcionalidades. Idealmente, um projeto deve manter constante a sua complexidade quando um novo código é incorporado. Estamos apresentando aos líderes de projetos (nas comunidades, fundações, governos e empresas)

a importância do monitoramento de métricas como LCOM4 e CBO juntamente com a NM e LOC, para aumentarem as chances de formar uma comunidade de colaboradores em torno de seus sistemas, o que potencializa a percepção de qualidade do software. Assim, os projetos deverão crescer e controlar sua complexidade, para facilitar as contribuições de novos membros e a evolução do software.

4.3 Distribuição e valores frequentes das métricas em projetos de software livre

	Linguagem Predominante	Módulos ou classes
Linux	C	26.196
Freebsd	C	22.572
Android	C	33.450
Bash	C	289
Chromium OS	C	19.911
GCC	C	22.757
Gimp	C	1.619
Git	C	348
Gnome	C	11.554
HTTP Server	C	326
KVM	C	120
MPlayer	C	1.936
OpenLDAP	C	1.073
PHP	C	2.056
Postgresql	C	1.226
Python	C	1.223
Subversion	C	1.545
VLC	C	3.248
Chrome	C++	5.660
Firefox	C++	17.697
Inkscape	C++	3.460
Media Player	C++	5.659
Mysql	C++	5.539
OpenOffice	C++	19.562
Eclipse IDE	Java	13.579
Open JDK8	Java	23.566
Ant	Java	1.618
Checkstyle	Java	1.177
Eclipse Metrics	Java	561
Findbugs	Java	3.278
GWT	Java	9.705
Hudson	Java	1.667
JBoss	Java	9.084
Kalibro	Java	499
Log4J	Java	416
Netbeans	Java	61.117
Spring	Java	7.283
Tomcat	Java	2.296
TOTAL	–	**344.872**

Tabela 4.3: *Projetos analisados: linguagem predominante e número de módulos ou classes.*

Depois de (i) verificarmos a relação de causalidade entre métricas de código-fonte e a atratividade de projetos de software livre, através dos estudos quantitativos relatados na Seção 4.2, (ii) constatarmos a contradição nos trabalhos da área em relação às distribuições das métricas (Seção 4.1) e (iii) observarmos o número limitado de métricas estudadas, geralmente apenas em projetos escrito em Java, fizemos uma série de estudos qualitativos para conseguirmos responder nossa segunda pergunta de pesquisa: "Quais métricas devem ser controladas ao longo do tempo?" e para conduzirmos os nossos estudos relacionados às nossas contribuições científicas.

Selecionamos um conjunto de 35 projetos de software livre (18 escritos predominantemente em C, 6 em C++ e 11 em Java) notoriamente reconhecidos e de sucesso, somados a outros 3 projetos do domínio aplicação de avaliação de código-fonte (todos escritos em Java). Esses últimos foram escolhidos também verificarmos se ferramentas do domínio de aplicação de avaliação de código-fonte têm boas métricas e explicitarmos os valores das métricas de uma de nossas ferramentas. Os critérios de seleção e a forma com que automatizamos a coleta de dados serão apresentados na Seção 4.3.1. As métricas coletadas e analisadas, de acordo com o apresentado na Seção 3.1.2 e dentre as reportadas pela ferramenta Analizo (Seção 5.2), foram:

- *Conexões aferentes de uma classe*
- *Média da Complexidade Ciclomática por método*
- *Média do número de linhas de código por método*
- *Média do Número de Parâmetros por Método*
- *Acoplamento entre objetos*
- *Profundidade da árvore de herança*
- *Ausência de coesão em métodos*
- *Número de linhas de código*
- *Número de atributos*
- *Número de filhos*
- *Número de métodos*
- *Número de atributos públicos*
- *Número de métodos públicos*
- *Respostas para uma classe*
- *Complexidade estrutural*

A Tabela 4.3 apresenta a lista de projetos de software livre que foram avaliados, com sua respectiva linguagem prodominante e número de classes e módulos (no caso dos escritos em C): a primeira parte da tabela, encabeçada pelo Linux, contém os projetos predominantemente escritos em C; a segunda parte, iniciada pelo Chrome, lista os projetos majoritariamente em C++; e a terceira parte, liderada pelo Eclipse IDE, apresenta os projetos escritos em Java. Enfatizamos que estamos falando de projetos predominantemente escritos em uma determinada linguagem, porque os projetos podem conter mais de uma linguagem de programação entre os arquivos de seu código-fonte. Por exemplo, o sistema de gerenciamento de banco de dados MySQL é 60% escrito em C++, 34% em C e 6% em outras linguagens. Isso pode ser testado usando uma ferramenta como o Sloccount[3] ou em uma plataforma disponível na Web como o Ohloh[4]. Ainda, na Tabela 4.3, destacamos os projetos

[3] dwheeler.com/sloccount/
[4] ohloh.net

Linux (aqui avaliamos o código do *kernel*), FreeBSD, Chrome, Firefox, Eclipse IDE e Open JDK8, isto é, dois de cada grupo de linguagem de programação, porque serão usados como exemplos nas discussões dos resultados deste estudo (Seção 4.3.2).

4.3.1 Planejamento dos Estudos

Neste estudo qualitativo, fizemos análises estatísticas detalhadas, desde a análise descritiva dos dados, passando pela verificação do tipo de distribuição em cada métrica, até chegarmos nos percentis dos valores das métricas das classes (ou módulos) de cada um dos projetos. Mesmo a análise estatística sendo detalhada, não achamos pertinente discutir a nossa abordagem do ponto de vista da estatística, mas sim mostrar de uma forma objetiva como usamos algumas técnicas como "ferramenta" de análise desses dados. Desse modo, entendemos que um pesquisador da área da computação (ou sistemas de informação e afins), não especialista em estatística, pode melhor entender nossos passos e também, ao reusar nossos *scripts* de análise[5], por exemplo, replicar o tipo de avaliação feito neste estudo em outros projetos de seu interesse.

Por conta da quantidade de dados obtidos e pelo volume de figuras e tabelas produzidos com eles, organizamos esses resultados um conjunto de apêndices disponível na Internet: ccsl.ime.usp.br/mangue. Caso o leitor tenha curiosidade em verificar os dados deste estudo em um projeto específico, ele poderá consultar esse material.

Hipóteses

Este estudo qualitativo com os projeto supracitados foi conduzido levando em consideração as seguintes hipóteses:

- **H1** – ***O tipo da distribuição dos valores das métricas de código-fonte tendem a ser diferentes entre projetos diferentes.*** Baseado na contradição apresentada nos estudos relacionados, esperamos que as distribuições das métricas tenham, em um certo grau, variação entre os tipos de distribuição. Em outras palavras, a mesma métrica pode se ajustar a distribuições diferentes em projetos diferentes. Em cada um dos projetos, testamos as distribuições Weibull, Poisson, Gamma, Exponencial e Pareto Tipo II, de acordo com as distribuições citadas nos trabalhos relacionados (disticutidos na Seção 4.1). Também, dado que muitas vezes os valores das métricas não se ajustavam às métricas citadas, realizamos um teste baseado no critério da informação generalizada de Akaike (Akaike, 1974). Esse critério se baseia na minimização de uma medida da distância entre um modelo verdadeiro e um modelo candidato para obter um modelo final, que represente o mecanismo que gerou os dados em questão. Em outras palavras, do ponto de vista prático, usamos a função *FitDist* do R (software livre de análise estatística) que usa uma medida da distância entre um bom modelo (no nosso caso uma distribuição) e vários modelos candidatos (um conjunto de distribuições), para obtermos uma distribuição que se ajuste aos dados analisados.

- **H2** – ***A média e mediana não são dados informativos na maioria dos casos.*** Esperamos que as métricas não sigam uma distribuição normal na maioria dos casos, o que pode indicar que a média não é um valor informativo. A média é o melhor valor de análise quando não temos informação alguma sobre o comportamento dos dados e assumimos que os dados são normais. Uma vez que, pela nossa Hitótese H1, olhamos a distribuição, ou seja, comportamento de cada métrica em cada projeto, temos a informação necessária para verificar se a média é informativa ou não. Da mesma forma, apostamos que a mediana também não é um valor informativo pelo o fato de as métricas de código-fonte terem a chamada "cauda pesada", isto é, a maioria dos dados está concentrada nos menores valores. Nossa hipótese é que a partir do percentil 75% (também chamado de terceiro quartil) será possível obter dados informativos sobre as métricas.

[5]Nossos scripts estão sendo organizados e serão disponibilizados em ccsl.ime.usp.br/mangue

- **H3** – ***Ao agrupar os projetos por linguagem de programação, será possível observar valores frequentes das métricas nos projetos avaliados.*** Esperamos que nos projetos escritos na mesma linguagem, que podem geralmente seguir o mesmo paradigma de programação, os valores dos percentis das métricas, isto é, os valores frequentes entre as classes e módulos, sejam não muito dispersos ao ponto de podemos sinalizar valores frequentes dentro do grupo da linguagem de programação.

Amostra de projetos e coleta dos dados

Para testarmos nossas hispóteses, coletamos métricas para cada classe e módulo de todos os 38 projetos selecionados, num total de 344.872 classes e módulos, como destacado na na Tabela 4.3. Certamente, pela reputação dos projetos, podemos observar que estamos lidando com uma amostra que contempla os trechos de código-fonte "mais executados" e com mais contribuidores dentre os projetos de software livre. Entretanto, os projetos não foram selecionados seguindo um critério subjetivo a respeito das suas reputações. Diferentemente dos nossos estudos sobre a correlação das métricas com a atratividade de projetos de software livre (Seção 4.2), não poderíamos ficar limitados a uma plataforma única em que os projetos hospedem seus repositórios e arquivos, como é o caso do SourceForge.net. Particularmente, nos últimos 3 anos, plataformas como GitHub [6], Gitorious [7] e Launchpad [8] atraíram muitos projetos de software livre por conta de serem baseados nos mais modernos tipos de sistemas de controle de versão e intregração com funcionalidades que permitem uma melhor colaboração das pessoas em torno do repositório. Por exemplo, em 2011, Linus Torvalds, criador e ainda principal mantenedor do núcle do Linux, colocou o código do Linux do Github[9].

Mesmo tendo o SourceForge.net se modernizado, o que também implicou em mudanças nos nossos *scripts* dos primeiros estudos para baixarmos os códigos dos projetos hospedados nessa plataforma, ele ainda assim perdeu espaço para outras plataformas disponíveis. Isso nos obrigou a repensarmos nossa estratégia de obtenção do código-fonte dos projetos para, dentro de uma amostra real, contemplarmos o máximo possível de projetos de software livre em nossa seleção. Baixar os códigos manualmente, em especial direto dos repositórios, não é algo escalável e reprodutível. Limitarmo-nos a uma plataforma ou desenvolver scripts para cada uma delas também não se mostrou uma boa abordagem, pois teríamos que justificar a escolha de um determinado conjunto de plataformas, o que poderia introduzir mais uma limitação e risco para a validade de nossos estudos. Dessa forma, investigamos mais a fundo alguns projetos que já reunem e coletam informações sobre projetos de software, em especial dos seus repositórios como número de *commits* e *commiters*, como o FLOSSMetrics (*Free/Libre Open Source Software Metrics*[10]), SQO-OSS (*Software Quality Assessment of Open SourceSoftware*[11]) e o Ohloh[12].

O Olhoh tem mais de 58 mil projetos cadastrados com informações sobre os seus repositórios. É uma plataforma que oferece um conjunto de dados sobre e para as comunidades de software livre, com o objetivo de prover uma visão geral da evolução dos projetos em desenvolvimento. Ela fornece uma API para se obter as listas dos projetos cadastrados e alguns dados coletados pela mineração dos repositórios desses projetos. Para os dados não fornecidos por essa API, verificamos alguns padrões nos dados que dão origem aos gráficos mostrados pelo Ohloh. Com essas informações, escrevemos um conjunto de scripts[13] para:

1. Selecionar uma lista desejada de projetos de software livre com mais contribuidores no intervalo dos últimos 12 meses, de acordo com a mineração do repositório oficial de cada projeto,

[6] github.com
[7] gitorious.org
[8] launchpad.net
[9] theregister.co.uk/2011/09/06/linus_torvalds_dumps_kernel_for_github/
[10] flossmetrics.org
[11] sqo-oss.org
[12] ohloh.net
[13] gitorious.org/ohloh-scripts

passando o número de projetos desejados (por exemplo, os 100 projetos com mais colaboradores nos últimos 12 meses);

2. Selecionar os projetos com mais usuários, segundo o *ranking* do Ohloh, também passando a quantidade de projetos desejados (por exemplo, os 100 projetos com mais usuários de acordo com o *ranking* no Ohloh);

3. Obter dados sobre as linguagens predominantes dos projetos;

4. Dada uma lista com os nomes dos projetos, obter o histórico do número de contribuidores no repositórios (*commiters*) e atualizações no código (*commits*).

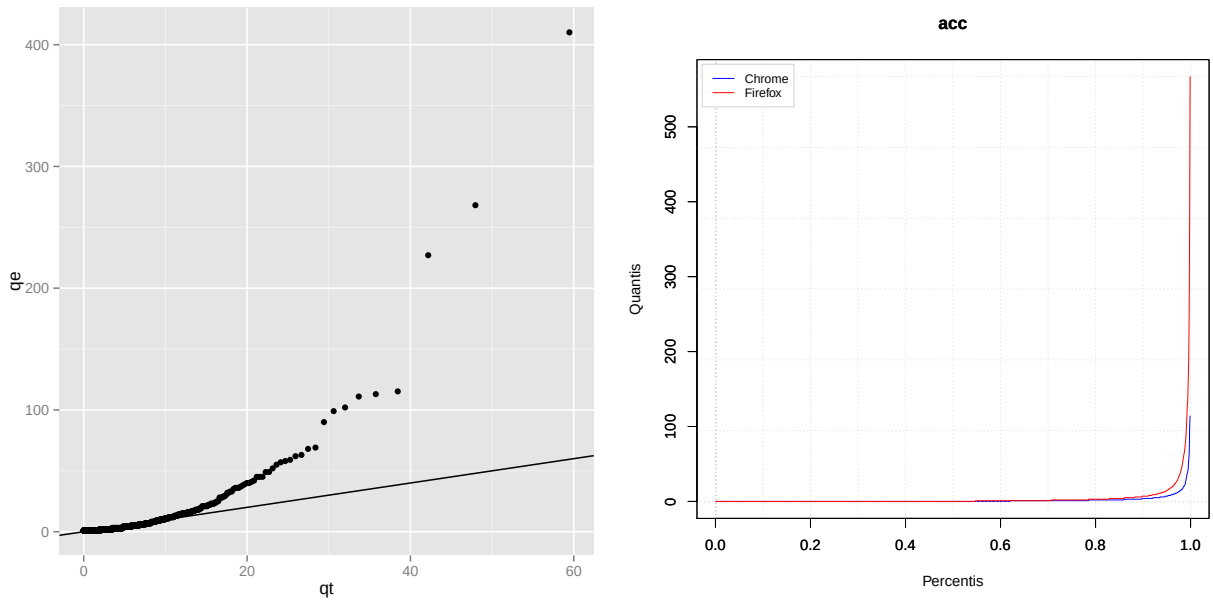
5. Passada uma lista com os nomes dos projetos, obter os repositórios oficiais cadastrados no Ohloh, que é de onde eles obtém seus dados também.

De posse das listas dos projetos e seus respectivos repositórios, baixamos os repositórios dos projetos e executamos a ferramenta Analizo, com processamento em lote (ver Seção 5.2), para gerar uma saída em CSV com os valores das métricas de cada classe e módulo de cada projeto. No caso deste estudo, selecionamos os 100 projetos com mais contribuidores e usuários, de acordo com o Ohloh. Verificamos quais dos projetos predominantemente escritos em C, C++ e Java estavam nas duas listas. No momento, a Analizo provê a análise de código-fonte escrito nessas três linguagens. Assim, obtivemos os repositórios desses projetos e conseguimos obter as métricas de 35 projetos com a Analizo. Depois, somamos os projetos CheckStyle, Eclipse Metrics e Kalibro (esse último faz parte do contexto deste doutorado e será aprensetado na Seção 5.3) entre os projetos que avaliamos sob o ponto de vista dos tipos de distribuição e valores frequentes em suas classes e módulos. Esses três projetos também estão cadastrados no Ohloh.

4.3.2 Resultados

Baseados em nossas hipóteses e tendo conduzido os estudos como relatado na seção anterior, nesta seção apresentaremos os resultados das análises das métricas em cada projeto. Nossas análises estão organizadas por métricas. Nos gráficos estamos sempre exemplificando as observações variando entre os seis projetos, dois de cada linguagem, que selecionamos: Linux, FreeBSD, Chrome, Firefox, Eclipse IDE e Open JDK 8. Para cada análise das métricas, produzimos duas tabelas: uma que resume os tipos de distribuição encontradas para a métrica em questão e outra que apresenta os valores dos percentis, ambas para todos os 38 projetos avaliados.

Conexões aferentes de uma classe (ACC)



(a) Pareto Type II

(b) Percentilis

Figura 4.1: *Distribuição da métrica ACC nos códigos do Chrome e do Firefox: Pareto Type II e Percentis.*

Verificamos que a distribuição da métrica ACC para os projetos analisados não é Weibull, Poisson, Gamma ou Exponencial, em particular devido à sua cauda pesada. A distribuição que melhor se ajusta na maioria dos casos é a Pareto tipo 2, como exemplificado pelo Gráfico 4.1(a) para o código do Chrome e Firefox, e observado na Tabela 4.4.

	Distribuição	média representativa?
Linux	Pareto Type II	Não
FreeBSD	Pareto Type II	Não
Chrome	Pareto Type II	Não
Firefox	Pareto Type II	Não
Eclipse	Pareto Type II	Não
OpenJDK	Pareto Type II	Não
Android	Pareto Type II	Não
Ant	Pareto Type II	Não
HTTP-Server	Pareto Type II	Não
OpenOffice	Pareto Type II	Não
Bash	Pareto Type II	Não
Checkstyle	Generalized t	Sim
ChromiumOS	Pareto Type II	Não
Eclipse Metrics	Pareto Type II	Não
Findbugs	Pareto Type II	Não
GCC	Pareto Type II	Não
Gimp	Pareto Type II	Não
Git	Pareto Type II	Não
Gnome	Pareto Type II	Não
GWT	Pareto Type II	Não
Hudson	Pareto Type II	Não
Inkscape	Pareto Type II	Não
Jboss	Pareto Type II	Não
Kalibro	Pareto Type II	Não
KVM	Sin-Arcsinh	Não
Log4j	Pareto Type II	Não
Media Player	Pareto Type II	Não
Mplayer	Pareto Type II	Não
Mysql	Pareto Type II	Não
Netbeans	Pareto Type II	Não
OpenLDAP	Pareto Type II	Não
PHP	Pareto Type II	Não
Postgresql	Pareto Type II	Não
Python	Pareto Type II	Não
Spring	Pareto Type II	Não
Subversion	Pareto Type II	Não
Tomcat	Pareto Type II	Não
VLC	Pareto Type II	Não

Tabela 4.4: *Distribuição da métrica ACC nos códigos analisados.*

Antes de qualquer teste, verificamos se há similaridades nas distribuição da métrica ACC nas duplas de projetos selecionados por linguagem. Os projetos Chrome e Firefox já apresentavam evidências de similaridades entre suas distribuições para a métrica ACC. Dessa forma, o Gráfico 4.1(a) representa o comportamento da métrica ACC para ambos os projetos. Já o Gráfico 4.1(b) ilustra a evolução dos valores dos percentis da métrica ACC entre as classes do Chrome e Firefox. Como podemos observar, até um certo percentual, certa de 70%, as classes mantêm valores bem próximos. Acima desse percentual de 70%, observamos as variações, o que já sinaliza que a mediana (percentil 50) não será algo informativo.

Como observado na Tabela 4.4, na maioria dos projetos a distribuição que melhor se ajusta é a Pareto tipo 2. Com isso, para a métrica ACC, nossa primeira hipótese sobre tipos de distribuição diferentes entre os projetos não se confirma. Entretanto, em comparação com alguns trabalhos

relacionados citados na Seção 4.1, mostramos que não estamos lidando com uma distribuição normal.

	Mín	1%	5%	10%	25%	50%	75%	90%	95%	99%	Máx
Linux	0,0	0,0	0,0	0,0	0,0	0,0	2,0	7,0	13,0	52,2	743,0
Freebsd	0,0	0,0	0,0	0,0	0,0	0,0	2,0	8,0	15,0	59,3	929,0
Android	0	0	0	0	0	0	1	4	8	32	1631
Bash	0	0	0	0	0	1	5	17	26	67	95
Chromium OS	0	0	0	0	0	0	3	7	13	65	1916
GCC	0	0	0	0	0	0	1	5	12	62	2779
Gimp	0	0	0	0	0	1	4	15	39	200	907
Git	0	0	0	0	0	1	6	26	60	166	248
Gnome	0	0	0	0	0	1	3	9	17	47	461
HTTP Server	0	0	0	0	0	0	3	30	80	185	229
KVM	0	0	0	0	1	3	8	14	19	32	35
MPlayer	0	0	0	0	0	0	2	11	25	127	408
OpenLDAP	0	0	0	0	0	1	6	21	47	135	464
PHP	0	0	0	0	0	0	3	9	27	148	270
Postgresql	0	0	0	0	0	2	8	27	52	131	549
Python	0	0	0	0	0	0	2	14	47	133	386
Subversion	0	0	0	0	0	0	3	13	31	94	424
VLC	0	0	0	0	0	0	2	7	18	95	371
Chrome	0,0	0,0	0,0	0,0	0,0	0,0	1,0	4,0	7,0	27,8	633,0
Firefox	0,0	0,0	0,0	0,0	0,0	0,0	2,0	7,0	15,0	83,5	3519,0
Inkscape	0	0	0	0	0	1	2	9	22	99	581
Media Player	0	0	0	0	0	0	2	5	12	50	622
Mysql	0	0	0	0	0	0	3	11	25	126	1321
OpenOffice	0	0	0	0	0	0	0	2	4	28	2325
Eclipse	0,0	0,0	0,0	0,0	0,0	0,0	2,0	7,0	14,0	61,0	930,0
Open JDK8	0,0	0,0	0,0	0,0	0,0	0,0	1,0	5,0	12,0	63,0	1759,0
Ant	0	0	0	0	0	0	1	5	11	43	806
Checkstyle	0	0	0	0	0	0	0	2	5	41	292
Eclipse Metrics	0	0	0	0	0	0	1	6	11	21	49
Findbugs	0	0	0	0	0	0	1	3	6	36	237
GWT	0	0	0	0	0	0	1	5	11	45	1281
Hudson	0	0	0	0	0	0	1	4	7	26	182
JBoss	0	0	0	0	0	0	1	3	6	25	675
Kalibro	0	0	0	0	0	0	1	4	12	27	197
Log4J	0	0	0	0	0	0	1	4	8	32	105
NetBeans	0	0	0	0	0	0	1	3	6	21	2017
Spring	0	0	0	0	0	0	1	5	11	40	398
Tomcat	0	0	0	0	0	0	1	5	10	37	252

Tabela 4.5: *Percentis para a métrica* `acc`.

Pareto tipo 2 é uma distribuição de probabilidade de leis de potência (Clauset *et al.*, 2007). Tem como característica geral ser de escala livre, o que significa que a média não é um valor informativo. Assim, nossa segunda hipótese (média e mediana não representativas) para a métrica ACC se confirma. A distribuição da métrica ACC neste caso se comporta de acordo com a Lei de Pareto – para cada fenômeno, 80% das conseqüências vem de 20% das causas.

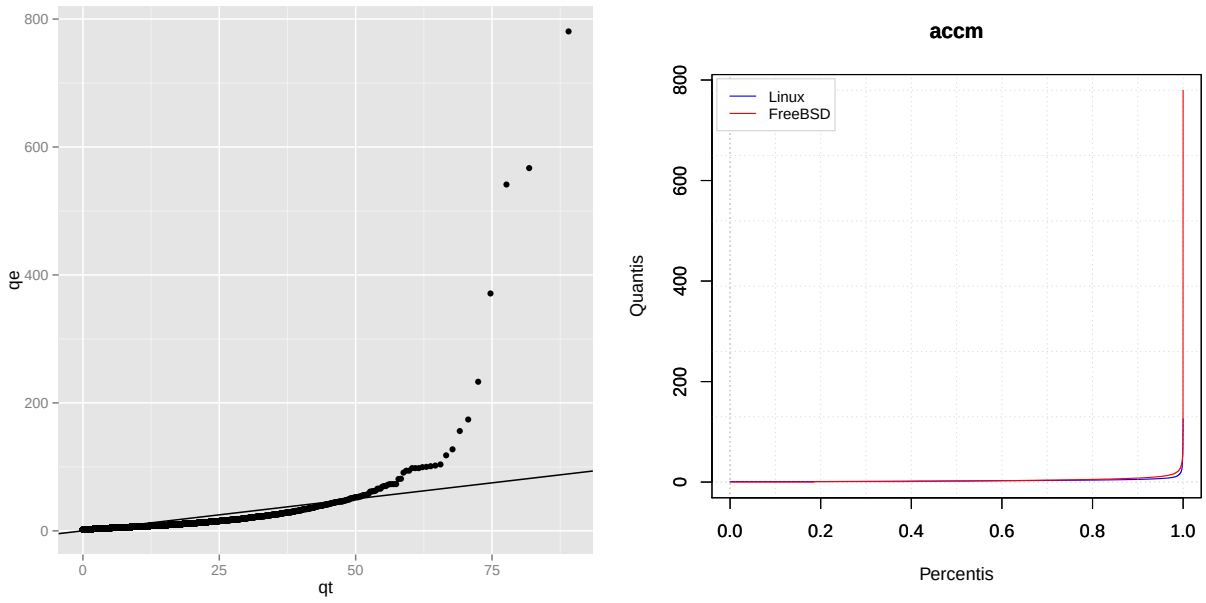
Em relação à nossa terceira hipótese, sobre os valores frequentes entre os projetos de mesma linguagem de programação poderem ser próximos, observamos na Tabela 4.9 que há variação significativa entre esses valores, negando nossa hipótese. Considerando a média e a mediana como medidas não informativas, definimos os pontos de corte nos percentis 75, 90 e 95. Por exempo, no

caso da linguagem C, observamos uma variação no percentil 75 entre Android (ACC igual a 1) e KVM (ACC igual 8). No percentil 90 a variação é maior ainda, em particular nos projetos em C e C++. Para os projetos Java, a variação é menor ao observar os dados na Tabela 4.9.

Esses dados sinalizam que o monitoramento dessa métrica pode variar de acordo com a linguagem de programação, domínio de aplicação ou mesmo de acordo com a maturidade (idade) do projeto. Aqui, para a métrica ACC, vamos sugerir como referência de valores frequentes os projetos com um certo equílibro dentre os demais, não necessariamente os com melhores valores. Para essa escolha subjetiva, estamos considerando o conceito das métricas discutidos na Seção3.1.2.

Para projetos escritos em C, o código do Linux sinaliza ter valores frequentes não tão bons quanto o do Android, mas factíveis de serem usados como uma referência para C: de 0 a 2,0: muito frequente; de 2,1 a 7,0: frequente; de 7,1 a 13,0: pouco frequente; acima de 13,0: não frequente. Para C++, o código do Firefox, mesmo com ACC não tão bom quanto o OpenOffice, indica um equilíbrio para usarmos seus valores frequentes: de 0 a 2,0: muito frequente; de 2,1 a 7,0: frequente; de 7,1 a 15,0: pouco frequente; acima de 15,0: não frequente. Para Java, o Open JDK8 pode ser nossa referência ao observarmos os 14 projetos: de 0 a 1,0: muito frequente; 1,1 a 5,0: frequente; de 5,1 12,0: pouco frequente; acima de 12,0: não frequente.

Média da Complexidade Ciclomática por método (ACCM)



(a) Weibull (b) Percentis

Figura 4.2: *Distribuição da métrica ACCM no código do Linux e no Free BSD: Weibull e Percentis*

Para a métrica de complexidade ciclomática, ilustramos sua distribuição e percentis nos projetos Linux e FreeBSD. Por exemplo, a distribuição que melhor se ajusta para essa métrica é a Weibull, exceto alguns poucos valores discrepantes, de acordo com o Gráfico 4.2(a). Assim como Pareto do tipo 2, Weibull é uma distribuição de probabilidade de leis de potência (Clauset *et al.*, 2007).

	Distribuição	média representativa?
Linux	Weibull	Não
FreeBSD	Weibull	Não
Chrome	Sinh-Ancsinh	Não
Firefox	Sinh-Ancsinh	Não
Eclipse	Skew t	Não
OpenJDK	Skew t	Não
Ant	Skew t	Não
HTTP-Server	Skew Exponential Power	Não
OpenOffice	Skew t	Não
Bash	Sin-Arcsinh	Não
Checkstyle	Skew t	Não
ChromiumOS	Sin-Arcsinh	Não
Eclipse Metrics	Sin-Arcsinh	Não
Findbugs	Sin-Arcsinh	Não
GCC	t Family	Sim
Gimp	Sin-Arcsinh	Não
Git	Skew t	Não
Gnome	Sin-Arcsinh	Não
GWT	Skew t	Não
Hudson	Sin-Arcsinh	Não
Inkscape	Skew t	Não
Jboss	t Family	Sim
Kalibro	t Family	Sim
KVM	Skew Exponential Power	Não
Log4j	Skew t	Não
Media Player	Skew t	Não
Mplayer	Skew Exponential Power	Não
Mysql	t Family	Sim
Netbeans	Skew t	Não
OpenLDAP	Sin-Arcsinh	Não
PHP	Skew t	Não
Postgresql	Sin-Arcsinh	Não
Python	Skew t	Não
Spring	Skew t	Não
Subversion	Sin-Arcsinh	Não
Tomcat	Skew t	Não
VLC	Sin-Arcsinh	Não

Tabela 4.6: *Distribuição da métrica ACCM nos códigos analisados.*

Entretanto, o tipo de distribuição para a métrica de complexidade ciclomática varia entre os projetos, o que confirma nossa primeira hipótese neste caso. Além da Weibull, dependendo do projeto, ela se ajusta ao *Skew Exponential Power* (DiCiccio e Monti, 2004), que também vamos considerar como uma lei de potência, uma vez que as distribuições que seguem as leis de potência são consideradas um caso particular das distribuições Skew (Mahnke *et al.*, 2012). Em especial, a *Skew normal* e *Skew t* (Jones e Faddy, 2003) estão atualmente em intesa investigação por se mostrarem adequadas e flexíveis para as análises comparadas com a normal e as distribuições da família t (Azzalini e Arellano-Valle, 2013). Por fim, também há casos em que a métrica ACCM se ajusta com a Sin-Arcsinh, que é um caso especial das distribuições Beta, e assim também entendemos que a média não é informativa para ela (Jones, 2005). Exposto isso, para a ACCM, nossa segunda hipótese também se confirma na maioria dos casos, uma vez que estamos lidando com as leis de potência e com um caso especial das distribuições Beta. Portanto, não generalizamos nossa análise

assumirmos uma única distribuição para todos os projetos, como em alguns trabalhos que discutimos na Seção 4.1.

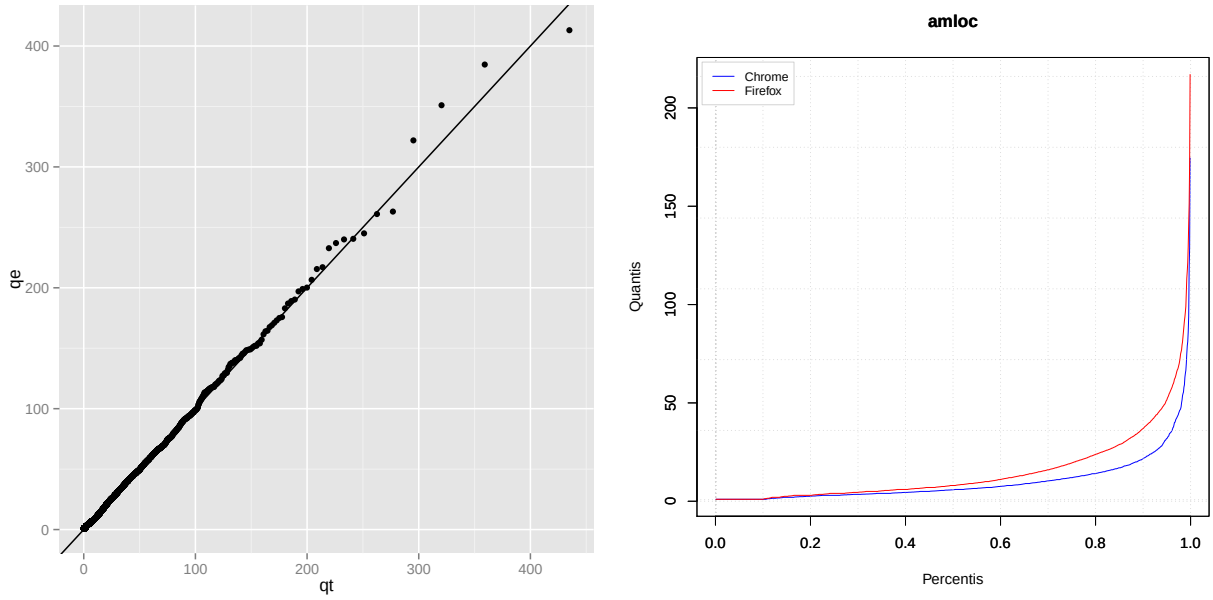
	Mín	1%	5%	10%	25%	50%	75%	90%	95%	99%	Máx
Linux	1,0	1,0	1,0	1,0	1,3	2,3	3,6	5,3	7,0	14,0	173,0
FreeBSD	0,0	0,0	0,0	0,0	1,0	2,0	4,5	8,0	11,1	22,8	830,0
Android	1	1	1	1	1	1	2	4	6	13	110
Bash	1	1	1	1	2	3	6	10	18	30	214
Chromium OS	1	1	1	1	1	2	3	6	8	15	163
GCC	1	1	1	1	1	1	2	4	6	13	214
Gimp	1	1	1	1	1	2	3	5	7	12	226
Git	1	1	1	2	3	4	6	9	10	22	46
Gnome	1	1	1	1	1	2	3	5	6	12	527
HTTP Server	1	1	1	1	2	4	6	9	12	18	26
KVM	1	1	1	1	2	3	6	10	13	20	20
MPlayer	1	1	1	1	2	4	6	9	12	21	91
OpenLDAP	1	1	1	1	2	5	9	17	23	36	89
PHP	1	1	1	1	1	2	5	10	16	37	640
Postgresql	1	1	1	1	1	3	6	10	15	32	106
Python	1	1	1	1	1	3	5	8	11	20	424
Subversion	1	1	1	1	1	2	4	7	9	17	68
VLC	1	1	1	1	1	2	4	7	8	14	36
Chrome	1,0	1,0	1,0	1,0	1,0	1,0	1,7	2,5	3,1	6,0	58,0
Firefox	1,0	1,0	1,0	1,0	1,0	1,4	2,7	4,9	7,0	13,8	102,0
Inkscape	1	1	1	1	1	1	2	4	6	11	28
Media Player	1	1	1	1	1	2	3	7	9	16	69
Mysql	1	1	1	1	1	2	3	6	8	18	58
OpenOffice	1	1	1	1	1	2	3	5	7	18	139
Eclipse	1,0	1,0	1,0	1,0	1,0	1,5	2,4	3,6	4,5	8,0	80,7
Open JDK8	1,0	1,0	1,0	1,0	1,0	1,6	2,8	4,4	6,0	11,0	181,0
Ant	1	1	1	1	1	1	2	3	4	7	17
Checkstyle	1	1	1	1	1	1	2	3	4	8	81
Eclipse Metrics	1	1	1	1	1	2	3	4	5	7	15
Findbugs	1	1	1	1	1	1	2	4	5	9	71
GWT	1	1	1	1	1	1	1	2	3	6	18
Hudson	1	1	1	1	1	1	2	3	3	6	9
JBoss	1	1	1	1	1	1	2	3	4	8	48
Kalibro	1	1	1	1	1	1	1	1	2	2	2
Log4J	1	1	1	1	1	1	2	3	4	7	11
Netbeans	1	1	1	1	1	1	2	3	5	9	305
Spring	1	1	1	1	1	1	2	2	3	5	17
Tomcat	1	1	1	1	1	2	3	4	6	10	57

Tabela 4.7: *Percentis para a métrica* `accm`.

Da mesma forma como apresentado para a métrica ACC, no caso da métrica de complexidade ciclomática, não conseguimos definir um conjunto de valores frequentes único para cada linguagem. Dessa forma, com a mesma abordagem, observamos que os valores para a métrica ACCM para o Linux, Firefox e Open JDK8 demonstram um equilíbrio entre os demais para podermos sugerir seus valores como frequentes. Para C, de 0 a 3,6 muito frequente; de 3,1 a 5,3 frequente; de 5,4 a 7,0 pouco frequente, acima de 7,0 não frequente. Para projetos escritos em C++, de 0 a 2,0 muito frequente; de 2,1 a 4,0 frequente; 4,1 a 6,0 pouco frequente; acima de 6 não frequente. Para os projetos Java, de 0 a 2,8 muito frequente; de 2,9 a 4,4 frequente; de 4,5 a 6,0 pouco frequente;

acima de 6 não frequente.

Média do número de linhas de código por método (AMLOC)



(a) Pareto 2

(b) Percentis

Figura 4.3: *Distribuição da métrica AMLOC no código do Firefox e Chrome: Pareto Tipo 2 e percentis*

Para a métrica do número médio de linhas de código por método, ilustramos nas Figuras 4.3 a comparação dos percentis dos valores das métricas para os projetos Firefox e Chrome, bem como a distribuição Pareto tipo 2 para o projeto Firefox, uma vez que, já entre eles, o tipo de distribuição é diferente. Como podemos observar na Tabela 4.8, também para a métrica AMLOC há um variação entre os tipos de distrição, o que confirma positivamente nossa hipótese H1.

Além da Pareto tipo 2, entre os projetos, a AMLOC ajusta-se com a *Skew Exponential Power* e *Skew t*, que seguem as leis de potência, como argumentamos para a métrica ACCM, bem como a Sin-Arcsinh, também comentada na seção anterior. Nesses casos, confirmando nossa segunda hipótese sobre a média não ser um valor representativo.

	Distribuição	média representativa?
Linux	Skew t	Não
FreeBSD	Skew t	Não
Chrome	Skew Exponential Power	Não
Firefox	Pareto type II	Não
Eclipse	Skew t	Não
OpenJDK	Skew Exponential Power	Não
Android	Skew t	Não
Ant	Skew Exponential Power	Não
HTTP-Server	Box-Cox Power Exponential	Sim
OpenOffice	Skew Exponential Power	Não
Bash	Skew t	Não
Checkstyle	Skew t	Não
ChromiumOS	Skew t	Não
Eclipse Metrics	Skew t	Não
Findbugs	Skew Exponential Power	Não
GCC	Skew t	Não
Gimp	Box-Cox Power Exponential	Sim
Git	Box-Cox Power Exponential	Sim
Gnome	Box-Cox t	Sim
GWT	Generalized t	Sim
Hudson	Skew Exponential Power	Não
Inkscape	Sin-Arcsinh	Não
Jboss	Skew Exponential Power	Não
Kalibro	Skew Exponential Power	Não
KVM	Box-Cox t	Sim
Log4j	Skew t	Não
Media Player	Sin-Arcsinh	Não
Mplayer	Box-Cox Power Exponential	Sim
Mysql	Skew Exponential Power	Não
Netbeans	Skew t	Não
OpenLDAP	Box-Cox t	Sim
PHP	Skew Exponential Power	Não
Postgresql	Box-Cox t	Sim
Python	Box-Cox t	Sim
Spring	Skew Exponential Power	Não
Subversion	Skew Exponential Power	Não
Tomcat	Skew t	Não
VLC	Skew Exponential Power	Não

Tabela 4.8: *Distribuição da métrica AMLOC nos códigos analisados.*

Entretanto, há projeto em que a distribuição da métricas AMLOC se ajusta com *Generalized t*, *Box-Cox Power Exponential* e *Box-Cox t*. A Generalized t (ou Student's t ou simplesmente distribuição t) tem a média como um valor informativo, uma vez que estima a média de uma população distribuída normalmente. As distribuições da família de Box-Cox são conhecidas como uma distribuição de potência normal e são muitas vezes modelas como normal em alguns estudos (Rigby e Stasinopoulos, 2006). Com isso, nesses casos, a média é uma médida informativa, o que nega em parte a nossa hipótese H2. Isso também pode sinalizar que, dependo do conjunto de projetos, a abordagem apresentada por Lanza e Marinescu (2006), discutida na Secão 4.1, pode ser usada. Assim, também nos faz considerar a mediana como um dado informativo para o número médio de linhas de código por métodos, o que neste caso derruba completamente nossa hipótese H2.

	Mín	1%	5%	10%	25%	50%	75%	90%	95%	99%	Máx
Linux	1,0	1,0	1,2	4,0	8,6	15,8	25,5	39,3	52,5	109,8	1573,3
Freebsd	0,0	0,0	0,0	0,0	3,0	16,2	33,4	55,0	75,0	142,3	3545,0
Android	1	1	1	2	4	8	17	32	48	121	946
Bash	1	1	1	1	9	20	35	72	110	214	389
Chromium OS	1	1	1	3	8	15	26	42	56	112	646
GCC	1	1	1	1	4	8	16	32	48	129	1793
Gimp	1	4	8	12	18	26	41	65	84	153	699
Git	4	6	10	12	19	26	35	47	62	103	287
Gnome	1	1	4	6	12	19	28	42	55	99	2214
HTTP Server	1	1	5	11	20	30	46	68	92	122	156
KVM	1	4	9	12	18	27	37	54	78	138	153
MPlayer	1	1	1	7	16	26	40	57	75	134	325
OpenLDAP	1	1	1	4	19	42	73	131	166	505	818
PHP	1	1	1	1	10	24	42	71	97	214	2214
Postgresql	1	1	7	10	20	37	62	92	119	248	783
Python	1	1	1	5	14	23	33	53	74	150	469
Subversion	1	1	1	4	11	28	49	75	100	151	245
VLC	1	1	1	1	5	17	34	53	67	121	262
Chrome	1,0	1,0	1,0	1,0	3,0	5,8	12,0	21,7	32,0	66,4	223,3
Firefox	1,0	1,0	1,0	1,0	4,0	8,0	19,5	37,0	51,5	99,0	832,0
Inkscape	1	1	1	1	2	6	15	28	38	76	452
Media Player	1	1	1	1	3	10	24	46	64	127	442
Mysql	1	1	1	1	4	10	23	44	65	138	435
OpenOffice	1	1	1	1	4	10	21	37	55	121	567
Eclipse	1,0	1,0	1,0	1,0	3,5	6,9	12,7	21,5	30,0	61,3	1004,0
Open JDK8	1,0	1,0	1,0	1,2	3,5	8,3	18,0	34,0	50,0	129,7	5271,5
Ant	1	1	1	3	3	6	10	17	24	43	435
Checkstyle	1	1	1	1	4	8	13	21	26	54	699
Eclipse Metrics	1	1	1	4	6	13	22	33	40	56	120
Findbugs	1	1	1	3	3	6	11	18	25	55	5798
GWT	1	1	1	1	3	4	9	19	32	116	2173
Hudson	1	1	1	2	3	5	8	16	22	42	602
JBoss	1	1	1	1	3	6	11	19	27	48	235
Kalibro	1	1	1	3	3	4	6	7	8	10	12
Log4J	1	1	1	3	4	7	14	22	30	56	71
Netbeans	1	1	1	2	3	6	13	23	34	118	2824
Spring	1	1	1	1	3	4	8	14	19	34	127
Tomcat	1	1	1	3	4	8	16	27	37	76	975

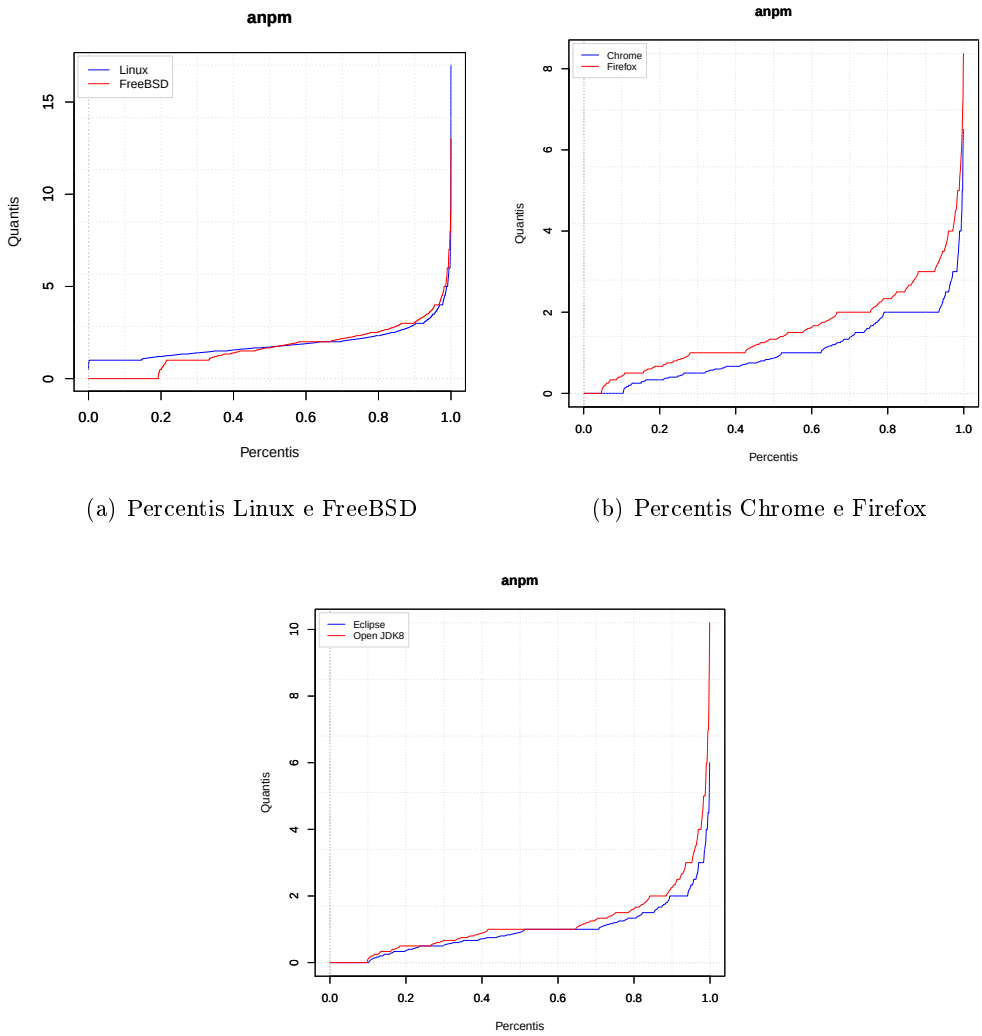
Tabela 4.9: *Percentis para a métrica* `amloc`.

Uma vez considerando a mediana um valor representativo, nossa interpretação para os valores frequentes da métrica AMLOC tem como primeiro ponto de corte o percentil 50. Mais uma vez, vamos sinalizar os valores de AMLOC para o Linux, Firefox e Open JDK8 como os mais equilibrados. Na busca de uma melhor referência, por exemplo, para os códigos em C, os valores do código do GCC para a métrica AMLOC se apresentam mais indicados. Em nossa abordagem, estamos indicando esses três projetos como referência entre os de sua linguagens, o que de qualquer forma nega nossa hipótese H3 em ter valores únicos para um grupo de linguagem de programação. Portanto, estamos sugerindo que esses nossos dados tenham outras interpretações, por outros pesquisadores, e que possam gerar outros valores frequentes de referência, de acordo das diferentes interpretações. Nosso objetivo não é definir valores frequentes em si, mas mostrar como é possível achá-los e indicá-los: uma maneira é interpretar esse nosso conjunto de dados ou reproduzir nossas coletas em outro

projetos para indicar os valores frequentes também baseado neles, preferencialmente somando-se aos avaliados aqui neste trabalho.

Baseado nos valores observados no código do Linux, para um código escrito em C: de 0 a 15,6 muito frequente; de 15,7 a 25,5 frequente; de 25,6 a 39,3 pouco frequente; acima de 39,3 não frequente. Para os projetos em C++, de acordo com valores do Firefox: de 0 a 8,0 muito frequente; de 9 a 19,5 frequente; 19,6 a 37 pouco frequente; acima de 37 não frequente. Por fim, para os projetos escritos em Java, conforme observado no código do Open JDK8, temos para a AMLOC: de 0 a 8,3 muito frequente; de 8,4 a 18 frequente; de 19 a 34 pouco frequente; acima de 34 não frequente.

Média do número de parâmetros por método (ANPM)



(a) Percentis Linux e FreeBSD

(b) Percentis Chrome e Firefox

(c) Percentis Eclipse e OpenJDK

Figura 4.4: *Distribuição da métrica ANPM: Percentis*

Para a métrica de número médio de parâmetros por método, em nenhum dos seis projetos que destacamos para exemplicarmos os resultados, o tipo de distribuição Weibull, Poisson, Gamma, Exponencial e Pareto Tipo 2, ou seja, as que olhamos antes partimos para os testes baseados no critério da informação de Akaike (Akaike, 1974), com a função *FitDist* da ferramenta R. Dessa forma, estamos ilustrando nos gráficos da Figura 4.4 a comparação entre os percentis das duplas de projetos por linguagem de programação para observarmos a variação dos valores da métrica ANPM nesses projetos.

	Distribuição	média representativa?
Linux	Skew Exponential Power	Não
FreeBSD	Skew Exponential Power	Não
Chrome	Skew Exponential Power	Não
Firefox	Johnson's SU	Sim
Eclipse	Skew t	Não
OpenJDK	Skew Exponential Power	Não
Android	Skew t	Não
Ant	Skew t	Não
HTTP-Server	Skew t	Não
OpenOffice	Skew t	Não
Bash	Power Exponential	Não
Checkstyle	Pareto Type II	Não
ChromiumOS	Skew Exponential Power	Não
Eclipse Metrics	Skew Normal	Não
Findbugs	Skew Exponential Power	Não
GCC	Sin-Arcsinh	Não
Gimp	Box-Cox Power Exponential	Sim
Git	ex-Gaussian	Sim
Gnome	Skew t	Não
GWT	Sin-Arcsinh	Não
Hudson	Skew Exponential Power	Não
Inkscape	Skew Exponential Power	Não
Jboss	Skew Exponential Power	Não
Kalibro	Pareto Type II	Não
KVM	Log-Normal	Sim
Log4j	Skew t	Não
Media Player	Skew Exponential Power	Não
Mplayer	Skew Exponential Power	Não
Mysql	Johnson's Su	Sim
Netbeans	Skew t	Não
OpenLDAP	Skew Exponential Power	Não
PHP	Skew Exponential Power	Não
Postgresql	Skew Exponential Power	Não
Python	Skew t	Não
Spring	Sin-Arcsinh	Não
Subversion	Skew Normal	Não
Tomcat	Skew t	Não
VLC	Skew Exponential Power	Não

Tabela 4.10: *Distribuição da métrica ANPM nos códigos analisados.*

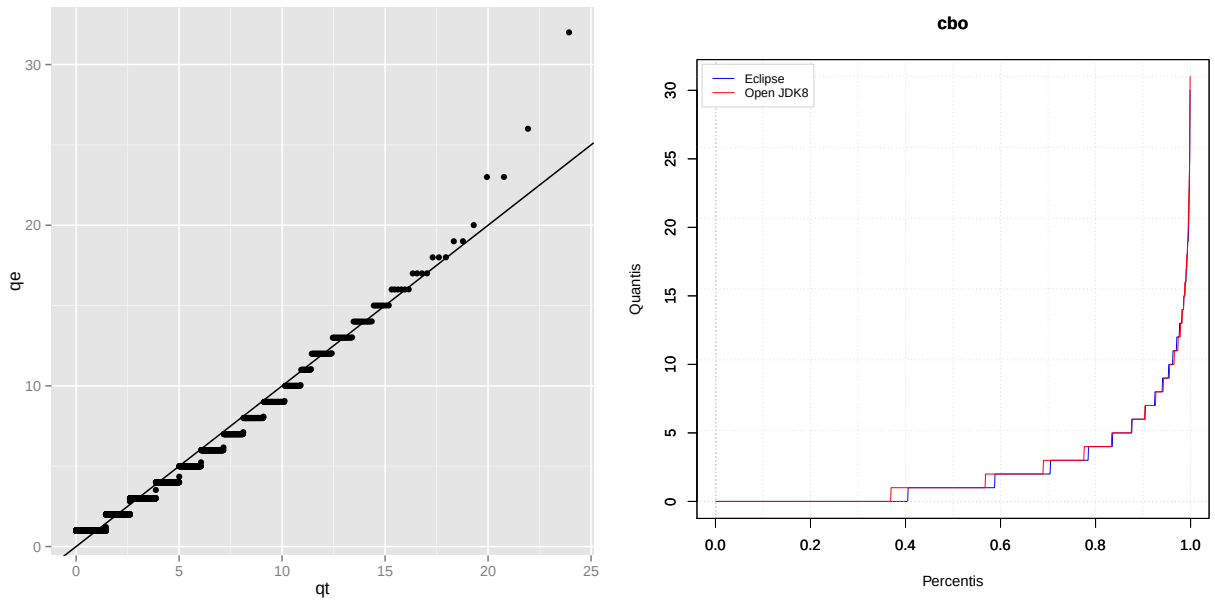
Para a métrica ANPM, nossa hipótese H1 se confirma, uma vez que há uma grande variação entre os tipos de distribuição para os valores dessa métrica entre os projetos analisados. Ela se ajusta na maioria dos casos com Pareto tipo 2, *Skew Exponential Power*, *Skew t*, que seguem as leis de potência, e Sin-Arcsinh – todas já comentadas anteriormente. Nesses casos, confirmando nossa hipótese H2, sobre a média não ser um valor representativo na maioria dos casos. Mesmo considerando projetos em que encontramos os tipos de distribuição *Johnson's Su* (Hill, 1976), *Exponentially modified Gaussian* (ex-Gaussion) (Cousineau *et al.*, 2004) e *Log-normal*, que entendemos ter a média como uma medida informativa.

	Mín	1%	5%	10%	25%	50%	75%	90%	95%	99%	Máx
Linux	0,5	1,0	1,0	1,0	1,3	1,7	2,2	2,9	3,5	5,0	90,1
Freebsd	0,0	0,0	0,0	0,0	1,0	1,7	2,3	3,1	3,8	6,0	19,0
Android	0	0	0	0	1	1	2	3	3	6	19
Bash	0	0	0	1	1	1	2	2	3	4	5
Chromium OS	0	1	1	1	1	2	2	3	4	5	90
GCC	0	0	0	0	1	1	1	2	3	4	19
Gimp	1	1	1	1	2	2	3	4	5	7	13
Git	1	1	1	2	2	2	3	3	4	5	6
Gnome	0	0	1	1	2	2	2	3	3	5	15
HTTP Server	0	1	1	1	2	2	3	3	3	4	6
KVM	1	1	1	1	2	2	3	3	4	5	7
MPlayer	0	0	1	1	2	2	3	4	4	5	8
OpenLDAP	0	0	1	1	2	2	3	4	5	7	10
PHP	0	1	1	1	1	2	3	4	4	6	9
Postgresql	1	1	1	1	1	2	2	3	4	6	13
Python	0	1	1	1	2	2	3	3	4	5	19
Subversion	0	0	1	1	2	3	4	6	7	9	13
VLC	0	0	0	1	1	2	2	3	4	9	27
Chrome	0,0	0,0	0,0	0,0	0,4	0,9	1,6	2,0	2,4	4,0	8,0
Firefox	0,0	0,0	0,1	0,4	0,8	1,3	2,0	3,0	3,6	5,4	18,7
Inkscape	0	0	0	0	0	1	2	3	3	5	12
Media Player	0	0	0	1	1	2	2	4	5	7	12
Mysql	0	0	0	0	1	1	2	3	4	6	14
OpenOffice	0	0	0	0	1	1	2	3	4	5	54
Eclipse	0,0	0,0	0,0	0,0	0,5	0,9	1,2	2,0	2,3	4,0	9,0
Open JDK8	0,0	0,0	0,0	0,1	0,5	1,0	1,5	2,3	3,0	5,9	45,0
Ant	0	0	0	0	0	1	1	1	2	3	6
Checkstyle	0	0	0	0	0	0	1	1	2	2	4
Eclipse Metrics	0	0	0	0	1	1	1	2	2	3	3
Findbugs	0	0	0	0	1	1	1	2	2	3	10
GWT	0	0	0	0	0	1	1	2	2	4	8
Hudson	0	0	0	0	0	1	1	2	2	3	9
JBoss	0	0	0	0	0	1	2	2	3	4	12
Kalibro	0	0	0	0	0	0	1	1	1	2	2
Log4J	0	0	0	0	0	1	1	2	2	3	4
Netbeans	0	0	0	0	0	1	1	2	2	4	10
Spring	0	0	0	0	0	1	1	2	2	3	7
Tomcat	0	0	0	0	1	1	2	2	3	5	8

Tabela 4.11: *Percentis para a métrica* `anpm`.

Na indicação de valores frequentes para a métrica ANPM, conseguimos sinalizar positivamente a nossa hipótese H3 ao agruparmos os valores por linguagem de programação, sem destacar um projeto como referência. Isso porque, observamos na Tabela 4.11 que não há uma grande variação entre os valores dos percentis 75, 90 e 95 – usados uma vez que refutamos a média e a mediana como informação para os valores da métrica ANPM. Dessa forma, para projetos escritos em C sinalizamos: de 0 a 3,0 muito frequente; de 3,1 a 4,0 frequente; 4,1 a 5,0 pouco frequente; acima de 5,0 não frequente. Aos projetos escritos em C++ podemos indicar: de 0 a 2,0 muito frequente; 2,1 a 3,0 frequente; 3,1 a 5,0 pouco frequente; acima de 5,0 não frequente. Para os projetos em Java temos: de 0 a 1,5 muito frequente; de 1,6 a 2,0 frequente; de 2,1 a 4,0 pouco frequente; acima de 4,0 não frequente.

Acoplamento entre objetos (CBO)



(a) Pareto Type II

(b) Percentis

Figura 4.5: *Distribuição da métrica CBO no código do Eclipse e do Open JDK: Pareto Type II e percentis*

O Gráfico 4.5(a) nos indica que a distribuição da métrica CBO para os códigos do Eclipse IDE e Open JDK8, por exemplo, é Pareto do tipo 2. Da mesma, forma podemos observar na Tabela 4.12 que na maioria dos projetos também encontramos esse tipo de distribuição para essa métrica de acoplamento. De toda maneira, nossa primeira hipótese, para a métrica CBO, também é confirmada, uma vez que há uma certa variação nos tipos de distribuição para a CBO entre os projetos avaliados. Além de se ajustar com a Pareto tipo 2, nos diferentes projetos ela se ajusta com *Skew Exponential Power*, *Skew t*, *t family*, Sin-Arcsinh – todas já comentadas anteriormente – e a distribuição Exponencial.

	Distribuição	média representativa?
Linux	Pareto Type II	Não
FreeBSD	Pareto Type II	Não
Chrome	Pareto Type II	Não
Firefox	Pareto Type II	Não
Eclipse	Pareto Type II	Não
OpenJDK	Pareto Type II	Não
Android	Pareto Type II	Não
Ant	Sin-Arcsinh	Não
HTTP-Server	Skew Exponential Power	Não
OpenOffice	Pareto Type II	Não
Bash	Sin-Arcsinh	Não
Checkstyle	Pareto Type II	Não
ChromiumOS	Sin-Arcsinh	Não
Eclipse Metrics	Pareto Type II	Não
Findbugs	Pareto Type II	Não
GCC	Pareto Type II	Não
Gimp	Skew t	Não
Git	Exponential	Não
Gnome	Sin-Arcsinh	Não
GWT	Pareto Type II	Não
Hudson	Pareto Type II	Não
Inkscape	Pareto Type II	Não
Jboss	t Family	Sim
Kalibro	Pareto Type II	Não
KVM	Exponential	Não
Log4j	Pareto Type II	Não
Media Player	Pareto Type II	Não
Mplayer	Skew t	Não
Mysql	Sin-Arcsinh	Não
Netbeans	Pareto Type II	Não
OpenLDAP	Exponential	Não
PHP	Pareto Type II	Não
Postgresql	Pareto Type II	Não
Python	Skew Exponential Power	Não
Spring	Pareto Type II	Não
Subversion	Sin-Arcsinh	Não
Tomcat	Pareto Type II	Não
VLC	Sin-Arcsinh	Não

Tabela 4.12: *Distribuição da métrica CBO nos códigos analisados.*

Mais uma vez, de acordo com a maioria dos tipos de distribuição observados na Tabela 4.12, refutaremos a média e mediana como uma medida informativa e verificaremos os percentis 75, 90 e 95 para interpretarmos os valores frequentes.

	Mín	1%	5%	10%	25%	50%	75%	90%	95%	99%	Máx
Linux	0,0	0,0	0,0	0,0	1,0	2,0	5,0	9,0	12,0	20,0	90,0
Freebsd	0,0	0,0	0,0	0,0	0,0	2,0	5,0	11,0	16,0	30,0	217,0
Android	0	0	0	0	0	1	2	5	8	19	72
Bash	0	0	0	0	1	3	7	14	20	41	44
Chromium OS	0	0	0	0	1	4	7	12	16	26	402
GCC	0	0	0	0	0	1	3	6	9	27	101
Gimp	0	0	1	1	3	7	14	22	27	41	65
Git	0	0	0	1	3	8	14	23	28	39	102
Gnome	0	0	0	0	1	3	6	11	15	26	60
HTTP Server	0	0	0	2	7	11	17	22	24	30	40
KVM	0	0	0	0	2	4	7	13	18	25	33
MPlayer	0	0	0	0	3	7	12	18	21	38	93
OpenLDAP	0	0	0	0	3	9	16	27	35	49	95
PHP	0	0	0	0	2	5	12	20	31	49	150
Postgresql	0	0	0	1	3	8	15	27	37	60	146
Python	0	0	0	0	3	6	13	25	32	48	62
Subversion	0	0	0	0	1	6	15	27	34	59	112
VLC	0	0	0	0	0	4	10	17	24	42	131
Chrome	0,0	0,0	0,0	0,0	0,0	1,0	3,0	5,0	7,0	12,0	39,0
Firefox	0,0	0,0	0,0	0,0	0,0	3,0	7,0	14,0	21,0	43,0	160,0
Inkscape	0	0	0	0	0	2	5	11	18	38	84
Media Player	0	0	0	0	0	2	5	9	13	21	63
Mysql	0	0	0	0	1	4	9	16	25	59	232
OpenOffice	0	0	0	0	0	1	3	6	8	16	49
Eclipse	0,0	0,0	0,0	0,0	0,0	1,0	3,0	6,0	9,0	16,0	62,0
Open JDK8	0,0	0,0	0,0	0,0	0,0	1,0	3,0	6,0	9,0	16,0	90,0
Ant	0	0	0	0	0	1	3	5	7	12	25
Checkstyle	0	0	0	0	0	0	2	3	4	6	11
Eclipse Metrics	0	0	0	0	0	1	2	3	4	5	6
Findbugs	0	0	0	0	0	0	1	4	7	14	33
GWT	0	0	0	0	0	0	1	4	6	12	70
Hudson	0	0	0	0	0	0	1	2	4	7	21
JBoss	0	0	0	0	0	0	2	3	5	8	38
Kalibro	0	0	0	0	0	0	1	3	4	7	10
Log4J	0	0	0	0	0	1	2	3	4	7	9
Netbeans	0	0	0	0	0	0	1	3	4	7	87
Spring	0	0	0	0	0	0	2	4	6	12	39
Tomcat	0	0	0	0	0	1	2	5	7	13	34

Tabela 4.13: *Percentis para a métrica* `cbo`.

Ao verificarmos a Tabela 4.13, podemos observar que, por se tratar de uma métrica de acomplamento entre objetos, um conceito do paradigma de orientação a objetos, há uma discrepância quando se compara os valores dos projetos escritos em C com os em C++ e Java. Mesmo considerando um mapeamento de classes para módulos e de métodos para função entre os paradigmas de orientação a objetos e procedimental (Terceiro e Chavez, 2009a), podemos ver que o paradigma influencia nesse caso. De qualquer forma, para os projetos escritos em C, indicamos os valores de CBO baseado também no código do Linux – argumentamos que também para CBO seus valores têm um equílibrio para servir de referência aos demais escritos em C: de 0 a 5 muito frequente, de 6 a 9 frequente; de 9 a 12 pouco frequente; acima de 12 não frequente.

Para os projetos escritos em C++ sugerimos os valores de CBO do Chrome como uma possível referência dos valores frequentes: de 0 a 3 muito frequente; 4 e 5 frequente; 6 e 7 pouco frequente;

acima de 7 não frequente. No caso dos projetos Java, já observamos no gráfico da Figura 4.5(b) que os valores de CBO das classes do Eclipse e do Open JDK são bem próximos. Olhando a Tabela 4.13, verificamos que os percentis 75, 90 e 95 têm os mesmos valores para esses projetos: de 0 a 3 muito frequente; de 4 a 6 frequente; de 7 a 9 pouco frequente; acima de 9 não frequente. Portando, em relação à nossa hipótese H3, apenas conseguimos agrupar valores "únicos" para a métrica CBO em códigos escritos em Java. Nos casos de C e C++ apenas apresentamos um dos projetos como uma referência, de acordo com nossa interpretação do Linux e do Chrome para com os respectivos projetos da mesma linguagem, ou seja, para C e C++ nossa hipótese H3 é negada.

Profundidade na Hierarquia de Classes (DIT)

	Distribuição	média representativa?
Chrome	Pareto Type II	Não
Firefox	Pareto Type II	Não
Eclipse	Pareto Type II	Não
OpenJDK	Pareto Type II	Não
Ant	Pareto Type II	Não
Checkstyle	Pareto Type II	Não
Eclipse Metrics	Pareto Type II	Não
Findbugs	Pareto Type II	Não
GWT	Pareto Type II	Não
Hudson	Pareto Type II	Não
Inkscape	Pareto Type I	Não
Jboss	Pareto Type II	Não
Kalibro	Pareto Type II	Não
Log4j	Pareto Type II	Não
Media Player	Pareto Type II	Não
Mysql	Pareto Type II	Não
Netbeans	Pareto Type II	Não
OpenOffice	Pareto Type II	Não
Spring	Pareto Type II	Não
Tomcat	Pareto Type II	Não

Tabela 4.14: *Distribuição da métrica DIT nos códigos analisados.*

No caso da métrica de profundidade na hierarquia de classes, estamos lidando coma um métrica de orientação a objeto que um mapeamento para o paradigma procedimental não cabe. Por isso, na Tabela 4.14 não temos os projetos escritos em C, uma vez que não consideramos essa métrica para eles.

Para a métrica DIT, nossa primeira hipótese é negada, uma vez que para todos os projetos escrito em C++ e Java analisados, ela se ajusta à distribuição Pareto tipo 2, ou seja, não variando entre os diferentes projetos. Consequentemente, nossa segunda hipótese é confirmada, porque estamos falando de um tipo de distribuição guiado pelas leis de potência (conforme argumentado ao tratarmos a primeira métrica apresentada – ACC). Portanto, também refutamos a possibilidade de observar os valores da métrica DIT pelo média ou mediana por não serem informativas. Verificamos os valores dos percentis 75, 90 e 95, como estamos procedendo para a maioria das métricas até aqui.

	Mín	1%	5%	10%	25%	50%	75%	90%	95%	99%	Máx
Chrome	0,0	0,0	0,0	0,0	0,0	0,0	1,0	2,0	2,0	3,0	5,0
Firefox	0,0	0,0	0,0	0,0	0,0	0,0	1,0	2,0	4,0	8,7	11,0
Inkscape	0	0	0	0	0	0	1	3	3	5	6
Media Player	0	0	0	0	0	0	1	2	2	3	5
Mysql	0	0	0	0	0	0	2	4	4	6	7
OpenOffice	0	0	0	0	0	0	0	1	1	3	8
Eclipse	0,0	0,0	0,0	0,0	0,0	1,0	2,0	4,0	4,0	6,0	10,0
Open JDK8	0,0	0,0	0,0	0,0	0,0	1,0	2,0	3,0	4,0	6,0	8,0
Ant	0	0	0	0	1	1	2	3	4	5	6
Checkstyle	0	0	0	0	0	1	2	4	5	6	7
Eclipse Metrics	0	0	0	0	1	1	1	2	3	3	3
Findbugs	0	0	0	0	0	0	1	2	3	7	7
Gnome	0	0	0	0	0	0	0	0	0	1	4
GWT	0	0	0	0	0	1	3	4	5	6	9
Hudson	0	0	0	0	0	1	2	3	3	5	7
JBoss	0	0	0	0	0	1	1	2	3	4	8
Kalibro	0	0	0	0	1	1	4	6	6	6	7
Log4J	0	0	0	0	0	1	2	2	3	4	5
Netbeans	0	0	0	0	0	1	1	2	3	4	8
Spring	0	0	0	0	0	1	2	3	4	7	12
Tomcat	0	0	0	0	0	1	1	2	3	4	5

Tabela 4.15: *Percentis para a métrica* `dit`.

Nossa hipótese H3 é positiva para a métrica DIT. Conseguimos sugerir valores frequentes para os projetos escritos em C++ e Java, observamos os valores dos percentis na Tabela 4.15. Para os projetos C++ indicamos: 0 e 1 muito frequente; 2 e 3 frequente; 4 pouco frequente; acima de 4 não frequente. No caso dos projetos escritos em Java, interpretamos: de 0 a 2 muito frequente; 3 e 4 frequente; 5 e 6 pouco frequente; acima de 6 não frequente.

Ausência de coesão em métodos (LCOM4)

	Distribuição	média representativa?
Linux	Skew Exponential Power	Não
FreeBSD	Sinh-Arcsinh	Não
Chrome	Skew Exponential Power	Não
Firefox	Sinh-Arcsinh	Não
Eclipse	Skew t	Não
OpenJDK	Skew t	Não
Android	Skew t	Não
Ant	Skew t	Não
HTTP-Server	Skew Exponential Power	Não
OpenOffice	Sin-Arcsinh	Não
Bash	Sin-Arcsinh	Não
Checkstyle	Skew t	Não
ChromiumOS	Sin-Arcsinh	Não
Eclipse Metrics	Skew t	Não
Findbugs	t Family	Sim
GCC	Skew t	Não
Gimp	Skew Exponential Power	Não
Git	Skew t	Não
Gnome	Sin-Arcsinh	Não
GWT	Skew t	Não
Hudson	Skew t	Não
Inkscape	Sin-Arcsinh	Não
Jboss	Skew t	Não
Kalibro	Sin-Arcsinh	Não
KVM	Skew Exponential Power	Não
Log4j	Skew t	Não
Media Player	Sin-Arcsinh	Não
Mplayer	Sin-Arcsinh	Não
Mysql	Sin-Arcsinh	Não
Netbeans	Skew t	Não
OpenLDAP	Skew t	Não
PHP	Sin-Arcsinh	Não
Postgresql	Skew t	Não
Python	Skew t	Não
Spring	Skew t	Não
Subversion	Skew t	Não
Tomcat	Skew t	Não
VLC	Sin-Arcsinh	Não

Tabela 4.16: *Distribuição da métrica LCOM4 nos códigos analisados.*

Na análise da métrica de ausência de coesão entre métodos de uma classe, também encontramos tipos de distribuição diferentes entre os projetos, corroborando nossa primeira hipótese. Para a LCOM temos as distribuições Skew Exponential Power, *Skew t*, *t family* e Sin-Arcsinh, já comentadas no contexto das outras métricas. Com exceção da *t family*, todas demonstram um comportamento dos valores de modo que a média não é uma informação representativa, confirmando também nossa hipótese H2.

	Mín	1%	5%	10%	25%	50%	75%	90%	95%	99%	Máx
Linux	1,0	1,0	1,0	1,0	2,0	6,0	11,0	19,0	29,0	62,0	357,0
Freebsd	0,0	0,0	0,0	0,0	1,0	2,0	5,0	12,0	20,0	55,0	1349,0
Android	1	1	1	1	1	2	4	8	14	40	437
Bash	1	1	1	1	1	3	7	13	21	52	77
Chromium OS	1	1	1	1	2	4	8	16	23	51	309
GCC	1	1	1	1	1	2	4	8	15	47	3439
Gimp	1	1	1	1	1	3	4	8	14	34	94
Git	1	1	1	1	1	1	3	6	9	46	165
Gnome	1	1	1	1	1	3	6	10	15	39	381
HTTP Server	1	1	1	1	2	4	6	13	17	36	69
KVM	1	1	1	1	1	2	5	9	12	37	77
MPlayer	1	1	1	1	1	3	5	9	13	34	522
OpenLDAP	1	1	1	1	1	2	4	9	19	49	259
PHP	1	1	1	1	1	3	7	19	30	79	241
Postgresql	1	1	1	1	1	3	7	16	27	93	991
Python	1	1	1	1	1	3	10	27	47	87	189
Subversion	1	1	1	1	1	2	5	16	30	86	218
VLC	1	1	1	1	2	3	6	13	21	40	524
Chrome	1,0	1,0	1,0	1,0	2,0	3,0	5,0	10,0	14,0	31,0	165,0
Firefox	1,0	1,0	1,0	1,0	1,0	3,0	5,0	10,0	17,0	45,0	1082,0
Inkscape	1	1	1	1	2	2	4	8	12	28	191
Media Player	1	1	1	1	2	3	7	25	46	138	2943
Mysql	1	1	1	1	2	3	6	11	18	48	895
OpenOffice	1	1	1	1	1	2	4	7	11	26	252
Eclipse	1,0	1,0	1,0	1,0	1,0	2,0	4,0	7,0	10,0	24,0	335,0
Open JDK8	1,0	1,0	1,0	1,0	1,0	2,0	3,0	7,0	12,0	29,0	495,0
Ant	1	1	1	1	1	2	3	7	10	22	53
Checkstyle	1	1	1	1	1	2	3	5	8	17	33
Eclipse Metrics	1	1	1	1	1	2	3	4	6	11	14
Findbugs	1	1	1	1	1	1	3	6	8	23	198
GWT	1	1	1	1	1	2	4	12	24	38	385
Hudson	1	1	1	1	1	2	4	7	11	29	149
JBoss	1	1	1	1	1	2	3	6	9	23	382
Kalibro	1	1	1	1	1	3	6	10	14	26	33
Log4J	1	1	1	1	1	2	3	6	9	21	37
Netbeans	1	1	1	1	1	2	4	7	11	26	503
Spring	1	1	1	1	1	2	3	7	11	27	105
Tomcat	1	1	1	1	1	2	3	7	11	37	217

Tabela 4.17: *Percentis para a métrica* `lcom4`.

Diferentemente dos valores teóricos para a LCOM4, em que o ideal seria ter uma falta de coesão o mais baixo possível, como algo em torno de 2, o que podemos observar nos valores apresentados na Tabela 4.17 é a ausência de coesão dos métodos da classes com valores acima do esperado na maioria dos casos, no que podemos chamar de "estado-da-prática" para essa métrica. Alguns desenvolvedores podem questionar o fato de nesta avaliação também se contar as classes de modelo, que no caso dos projetos escritos em Java teremos os "getters" e "setters", que apresentam altos valores de LCOM4, uma vez que cada "get" e "set" refere-se apenas a um atributo, gerando esses "subconjuntos" dentro da classe. Entretanto, como mostrado na Tabela 4.3, o volume de classes e módulos avaliados gerou uma amostra grande, o que estatisticamente garante uma análise adequada para esses casos. Além disso, em nossos pontos de corte, estamos eliminando os valores máximo, uma vez que o último percentil verificado é o de 95%.

Para a LCOM4 também não conseguimos responder positivamente para a nossa hipótese H3. Portanto, interpretamos que os valores apresentados pelo FreeBSD, Chrome e Open JDK8 como os de referência para observarmos os valores frequentes da LCOM4 nas classes C++ e Java e módulos C. Para os projetos escritos em C temos: de 0 a 5 muito frequente; de 6 a 12 frequente; de 12 a 20 pouco frequente; acima de 20 não frequente. Se compararmos com os valores da LCOM4 no código do Linux, fica evidenciado algo muitas vezes dito por programadores do *Kernel* sobre o FreeBSD ter um código mais organizado e legível que o Linux. Certamente, a maior ausência de coesão nas funções dos módulos do Linux reflete nesse tipo de constatação. Baseado no código do Chrome, os valores da LCOM para C++ são: de 0 a 5 muito frequente; de 6 a 10 frequente; de 10 a 14 pouco frequente; acima de 14 não frequente. Para os projetos escritos em Java sugerimos: de 0 a 3 muito frequente; de 4 a 7 frequente; de 8 a 12 pouco frequente; acima de 12 não frequente.

Número de linhas de código (LOC)

	Distribuição	média representativa?
Linux	Skew Exponential Power	Não
FreeBSD	Skew Exponential Power	Não
Chrome	Johnson's SU	Sim
Firefox	Johnson's SU	Sim
Eclipse	Box-Cox Power Exponential	Sim
OpenJDK	Skew Exponential Power	Não
Android	Skew Exponential Power	Não
Ant	Box-Cox Power Exponential	Sim
HTTP-Server	Weibull	Não
OpenOffice	Skew Exponential Power	Não
Bash	Log-Normal	Sim
Checkstyle	Skew Exponential Power	Não
ChromiumOS	Skew Exponential Power	Não
Eclipse Metrics	Generalized Inverse Gaussian	Sim
Findbugs	Skew t	Não
GCC	Skew t	Não
Gimp	Box-Cox t	Sim
Git	Log-Normal	Sim
Gnome	Skew Exponential Power	Não
GWT	Skew Exponential Power	Não
Hudson	Skew t	Não
Inkscape	Skew Exponential Power	Não
Jboss	Skew Exponential Power	Não
Kalibro	Skew Exponential Power	Não
KVM	Pareto Type II	Não
Log4j	Skew Exponential Power	Não
Media Player	Box-Cox Power Exponential	Sim
Mplayer	Box-Cox Power Exponential	Sim
Mysql	Johnson's Su	Sim
Netbeans	Skew Exponential Power	Não
OpenLDAP	Skew Exponential Power	Não
PHP	Generalized Gamma	Sim
Postgresql	Generalized Gamma	Sim
Python	Skew Exponential Power	Não
Spring	Skew Exponential Power	Não
Subversion	Skew Exponential Power	Não
Tomcat	Johnson's Su	Sim
VLC	Skew Exponential Power	Não

Tabela 4.18: *Distribuição da métrica LOC nos códigos analisados.*

A métrica de número de linhas de código das classes e módulos avaliados é a que apresenta maior diversidade de tipos de distribuição e também de valores, conforme apresentado na Tabela 4.18. Com isso nossa primeira hipótese se confirma, mas a segunda não, ao avaliarmos o equilíbrio entre tipos de distribuições que podemos considerar a média informativa ou não, somado as observações ao verificamos os valores na Tabela 4.19. Para LOC, a abordagem de Lanza e Marinescu (2006) pode ser aplicada, dependendo do conjunto de projetos em questão, por exemplo. Nossa avaliação, também levando em consideração os valores teórico apresentados na Seção 3.1.2, sugere que para o LOC a mediana é uma medida informativa.

	Mín	1%	5%	10%	25%	50%	75%	90%	95%	99%	Máx
Linux	1,0	1,0	6,0	13,0	52,0	172,0	473,0	1057,0	1662,0	3654,0	17019,0
Freebsd	0,0	0,0	0,0	0,0	8,0	79,0	324,0	877,0	1518,0	3838,0	60354,0
Android	1	1	2	4	12	36	113	326	578	1713	17381
Bash	1	1	5	7	24	97	269	772	1264	3363	6643
Chromium OS	1	1	4	9	33	107	302	683	1109	2729	14848
GCC	1	1	2	4	9	24	68	229	492	2745	42368
Gimp	3	8	26	50	106	243	531	920	1325	2527	24873
Git	4	7	16	26	72	209	528	1139	1621	3673	9640
Gnome	1	2	8	20	64	183	452	970	1474	3217	12827
HTTP Server	1	1	17	45	133	304	741	1491	1942	3771	4906
KVM	1	4	30	51	124	246	466	865	1455	2625	3623
MPlayer	1	1	4	20	81	195	421	847	1238	2377	4123
OpenLDAP	1	1	2	20	53	166	495	1149	1817	3211	5930
PHP	1	1	4	7	31	109	372	1018	1771	4630	59083
Postgresql	1	5	17	37	124	317	752	1720	2748	4809	21620
Python	1	1	5	13	49	117	493	1340	2189	4867	11594
Subversion	1	1	3	13	57	184	618	1359	2268	4559	13213
VLC	1	1	2	4	16	106	344	748	1167	2437	5252
Chrome	1,0	1,0	2,0	4,0	10,0	31,0	84,0	207,0	366,1	817,0	5393,0
Firefox	1,0	1,0	2,0	4,0	10,0	41,0	146,0	448,0	869,0	2344,0	55498,0
Inkscape	1	1	2	2	8	29	112	367	629	1359	8800
Media Player	1	1	2	6	17	65	196	503	847	1815	9786
Mysql	1	1	3	6	14	44	162	511	938	3452	17630
OpenOffice	1	1	1	4	11	35	100	242	394	964	7457
Eclipse	1,0	1,0	2,0	4,0	11,0	33,0	87,0	200,0	326,0	800,0	9422,0
Open JDK8	1,0	1,0	2,0	3,0	10,0	32,0	95,0	253,0	498,0	1462,0	32611,0
Ant	1	1	2	3	11	35	92	207	328	727	2267
Checkstyle	1	1	1	3	8	24	65	132	182	362	3496
Eclipse Metrics	1	1	4	5	19	43	105	220	300	413	459
Findbugs	1	1	3	3	7	19	56	151	267	841	8398
GWT	1	1	1	2	6	19	70	182	339	803	10867
Hudson	1	1	2	3	7	18	43	103	170	523	2805
JBoss	1	1	2	3	8	20	49	112	192	483	1571
Kalibro	1	2	3	4	9	20	37	62	77	138	191
Log4J	1	1	3	6	11	30	76	179	284	656	819
Netbeans	1	1	2	4	11	31	87	206	334	823	20510
Spring	1	1	1	3	6	16	47	117	204	530	1805
Tomcat	1	1	3	6	12	33	105	276	441	1121	5039

Tabela 4.19: *Percentis para a métrica* `loc`.

Indicamos para o número de linhas de código das classes e módulos os projetos FreeBSD, Chrome e Eclipse como possíveis referências para suas linhagens, não por terem os melhores valores, mas sim por terem valores equilibrados entre os demais. Claramente, a dispersão de valores do projetos escritos em C é maior comparado com os projeto em C++ e Java. Enfatizamos novamente que essa é apenas uma interpretação possível para se observar os valores frequentes dessa métrica.

Partindo da mediana como primeiro ponto de corte, para os projetos escritos em C temos: de 0 a 79 muito frequente; de 80 a 324 frequente; de 325 a 877 pouco frequente; acima de 877 não frequente. Para os projetos C++ sugerimos: de 0 a 31 muito frequente; de 32 a 84 frequente, de 85 a 207 pouco frequente; acima de 207 não frequente. Para projetos escritos em Java indicamos: de 0 a 33 muito frequente; de 34 a 87 frequente; de 88 a 200 pouco frequente; acima de 200 não frequente.

Número de Atributos (NOA)

	Distribuição	média representativa?
Linux	Pareto Type II	Não
FreeBSD	Pareto Type II	Não
Chrome	Pareto Type II	Não
Firefox	Pareto Type II	Não
Eclipse	Pareto Type II	Não
OpenJDK	Sinh-Arcsinh	Não
Android	Pareto Type II	Não
Ant	Pareto Type II	Não
HTTP-Server	Sin-Arcsinh	Não
OpenOffice	Pareto Type II	Não
Bash	Pareto Type II	Não
Checkstyle	Pareto Type II	Não
ChromiumOS	Pareto Type II	Não
Eclipse Metrics	Pareto Type II	Não
Findbugs	Pareto Type II	Não
GCC	Pareto Type II	Não
Gimp	Pareto Type II	Não
Git	Pareto Type II	Não
Gnome	Pareto Type II	Não
GWT	t Family	Sim
Hudson	Pareto Type II	Não
Inkscape	Pareto Type II	Não
Jboss	Pareto Type II	Não
Kalibro	Pareto Type II	Não
KVM	Pareto Type II	Não
Log4j	Sin-Arcsinh	Não
Media Player	Pareto Type II	Não
Mplayer	Sin-Arcsinh	Não
Mysql	Pareto Type II	Não
Netbeans	Pareto Type II	Não
OpenLDAP	Pareto Type II	Não
PHP	Pareto Type II	Não
Postgresql	Pareto Type II	Não
Python	Pareto Type II	Não
Spring	Pareto Type II	Não
Subversion	Pareto Type II	Não
Tomcat	Pareto Type II	Não
VLC	Pareto Type II	Não

Tabela 4.20: *Distribuição da métrica noa nos códigos analisados.*

Como podemos verificar na Tabela 4.20, na maioria dos projetos a métrica de número de atributos das classes e variáveis dos módulos ajusta-se ao tipo de distribuição Pareto 2, mas não nega totalmente nossa hipótese H1 por observamos projetos ajustando-se melhor com outras distribuições. Dado que, como já comentado, Pareto do tipo 2 é uma distribuição governada pelas leis de potência, o que implica que a média não é informativa, nossa hipótese H2 também se confirma para a métrica NOA.

	Mín	1%	5%	10%	25%	50%	75%	90%	95%	99%	Máx
Linux	0,0	0,0	0,0	0,0	1,0	4,0	15,0	41,0	75,0	228,2	4628,0
Freebsd	0,0	0,0	0,0	0,0	0,0	2,0	9,0	29,0	57,0	207,6	4628,0
Android	0	0	0	0	0	1	4	11	19	55	568
Bash	0	0	0	0	0	2	8	24	43	103	128
Chromium OS	0	0	0	0	0	2	9	25	50	157	955
GCC	0	0	0	0	0	0	2	6	10	32	8201
Gimp	0	0	0	0	0	4	11	21	31	66	205
Git	0	0	0	0	1	4	15	31	49	135	194
Gnome	0	0	0	0	0	4	11	24	38	86	492
HTTP Server	0	0	0	0	3	8	19	37	60	135	268
KVM	0	0	0	0	0	3	9	23	27	70	91
MPlayer	0	0	0	0	1	2	6	16	26	70	121
OpenLDAP	0	0	0	0	0	1	3	10	19	51	236
PHP	0	0	0	0	0	1	4	7	11	28	75
Postgresql	0	0	0	0	0	2	11	30	57	224	345
Python	0	0	0	0	0	1	6	14	20	46	439
Subversion	0	0	0	0	0	0	1	4	8	22	39
VLC	0	0	0	0	0	1	3	8	15	47	379
Chrome	0,0	0,0	0,0	0,0	0,0	1,0	3,0	6,0	10,0	19,0	243,0
Firefox	0,0	0,0	0,0	0,0	0,0	1,0	4,0	8,0	13,0	33,0	607,0
Inkscape	0	0	0	0	0	1	3	7	11	26	271
Media Player	0	0	0	0	0	1	4	10	15	41	319
Mysql	0	0	0	0	0	1	4	9	16	43	693
OpenOffice	0	0	0	0	0	1	3	6	10	27	941
Eclipse	0,0	0,0	0,0	0,0	0,0	1,0	3,0	8,0	12,0	32,7	1073,0
Open JDK8	0,0	0,0	0,0	0,0	0,0	1,0	3,0	8,0	13,0	35,0	353,0
Ant	0	0	0	0	0	1	5	10	16	32	55
Checkstyle	0	0	0	0	0	0	2	4	6	13	164
Eclipse Metrics	0	0	0	0	0	1	4	9	13	63	231
Findbugs	0	0	0	0	0	1	3	6	10	25	154
GWT	0	0	0	0	0	0	2	4	7	17	170
Hudson	0	0	0	0	0	1	2	4	7	12	39
JBoss	0	0	0	0	0	1	3	6	8	18	275
Kalibro	0	0	0	0	0	1	2	5	7	10	12
Log4J	0	0	0	0	0	1	3	8	15	24	37
Netbeans	0	0	0	0	0	1	3	8	12	28	505
Spring	0	0	0	0	0	1	2	4	7	14	81
Tomcat	0	0	0	0	0	2	5	10	16	37	127

Tabela 4.21: *Percentis para a métrica* `noa`.

Seguindo nossa abordagem para a maioria das métricas apresentadas até aqui, sugerimos os valores frequentes para a métrica NOA verificando os percetins 75, 90 e 95. Para os projetos escritos em C, baseado nos valores do número de variáveis dos módulos do FreeBSD, temos: de 0 a 9 muito frequente; de 10 a 29 frequente; de 30 a 57 pouco frequente; acima de 57 não frequente. Baseado nos valores de NOA para o Chrome, sugerimos: de 0 a 4 muito frequente; de 5 a 8 frequente; de 9 a 13 pouco frequente; acima de 13 não frequente. Para os projetos escritos em Java, de acordo com os valores de NOA para o Eclipse, indicamos: de 0 a 3 muito frequente; de 4 a 8 frequente; de 9 a 12 pouco frequente.

Número de filhos (NOC)

	Distribuição	média representativa?
Chrome	Pareto Type II	Não
Firefox	Pareto Type II	Não
Eclipse	Pareto Type II	Não
OpenJDK	Pareto Type II	Não
Ant	Pareto Type II	Não
Checkstyle	Pareto Type II	Não
Eclipse Metrics	Pareto Type II	Não
Findbugs	Pareto Type II	Não
GWT	Pareto Type II	Não
Hudson	Pareto Type II	Não
Inkscape	t Family	Sim
Jboss	Pareto Type II	Não
Kalibro	Pareto Type II	Não
Log4j	Pareto Type II	Não
Media Player	Pareto Type II	Não
Mysql	Pareto Type II	Não
Netbeans	Pareto Type II	Não
OpenOffice	Pareto Type II	Não
Spring	Pareto Type II	Não
Tomcat	Pareto Type II	Não

Tabela 4.22: *Distribuição da métrica NOC nos códigos analisados.*

De forma similar da métrica DIT, no caso da métrica de número de filhos, estamos lidando com uma métrica do paradigma de orientação a objeto em que um mapeamento para o paradigma procedimental não é factível. Também, na Tabela 4.14 não temos os projetos escritos em C, uma vez que não consideramos NOC para eles.

Nossa primeira hipótese é refutada para a métrica NOC, uma vez que para todos os projetos avaliados, os valores de NOC ajustam-se apenas à distribuição Pareto tipo 2. Dado que estamos observando valores governados por leis de potência, nossa segunda hipótese é positiva. Com a média e mediana descartadas por não serem informativas, verificamos os valores dos percentis 75, 90 e 95 para a métrica NOC.

	Mín	1%	5%	10%	25%	50%	75%	90%	95%	99%	Máx
Chrome	0,0	0,0	0,0	0,0	0,0	0,0	0,0	1,0	2,0	5,0	32,0
Firefox	0,0	0,0	0,0	0,0	0,0	0,0	0,0	1,0	2,0	6,0	45,0
Inkscape	0	0	0	0	0	0	0	1	2	12	178
Media Player	0	0	0	0	0	0	0	0	2	5	33
Mysql	0	0	0	0	0	0	0	1	2	6	94
OpenOffice	0	0	0	0	0	0	0	0	0	3	638
Eclipse	0,0	0,0	0,0	0,0	0,0	0,0	0,0	2,0	3,0	10,0	243,0
Open JDK8	0,0	0,0	0,0	0,0	0,0	0,0	0,0	1,0	3,0	9,0	301,0
Ant	0	0	0	0	0	0	0	1	3	12	162
Checkstyle	0	0	0	0	0	0	0	0	2	5	144
Eclipse Metrics	0	0	0	0	0	0	0	0	2	11	24
Findbugs	0	0	0	0	0	0	0	1	2	5	55
GWT	0	0	0	0	0	0	0	1	2	8	398
Hudson	0	0	0	0	0	0	0	1	2	7	23
JBoss	0	0	0	0	0	0	0	1	2	6	256
Kalibro	0	0	0	0	0	0	0	1	2	9	101
Log4J	0	0	0	0	0	0	0	1	3	9	23
Netbeans	0	0	0	0	0	0	0	1	2	6	989
Spring	0	0	0	0	0	0	0	2	3	7	175
Tomcat	0	0	0	0	0	0	0	1	2	8	82

Tabela 4.23: *Percentis para a métrica* `noc`.

Para o NOC, conseguimos sugerir valores frequentes para os projetos escritos em C++ e Java, observamos os valores dos percentis na Tabela 4.15. Assim, confirmando nossa hipótese H3, sobre valores frequente por linguagem de programação. Para os projetos C++ indicamos: 0 muito frequente; 1 frequente; 2 pouco frequente; acima de 2 não frequente. No caso dos projetos escritos em Java, interpretamos da seguinte forma: de 0 muito frequente; 1 e 2 frequente; 3 pouco frequente; acima de 3 não frequente.

Número de métodos (NOM)

	Distribuição	média representativa?
Linux	Skew Exponential Power	Não
FreeBSD	Skew Exponential Power	Não
Chrome	Skew Exponential Power	Não
Firefox	Sinh-Arcsinh	Não
Eclipse	Skew t	Não
OpenJDK	Sinh-Arcsinh	Não
Android	Sin-Arcsinh	Não
Ant	Power Exponential	Não
Bash	Sin-Arcsinh	Não
Checkstyle	Skew Exponential Power	Não
ChromiumOS	Skew t	Sim
Eclipse Metrics	Sin-Arcsinh	Não
Findbugs	Sin-Arcsinh	Não
GCC	Box-Cox Power Exponential	Não
Gimp	Skew t	Sim
Git	Sin-Arcsinh	Não
Gnome	Skew Exponential Power	Não
GWT	Sin-Arcsinh	Não
HTTP-Server	Skew t	Sim
Hudson	Sin-Arcsinh	Não
Inkscape	Skew Exponential Power	Não
Jboss	Sin-Arcsinh	Não
Kalibro	Skew t	Sim
KVM	Skew t	Sim
Log4j	Skew Exponential Power	Não
Media Player	Skew t	Sim
Mplayer	Skew t	Sim
Mysql	Skew t	Sim
Netbeans	Skew Exponential Power	Não
OpenLDAP	Sin-Arcsinh	Não
OpenOffice	Sin-Arcsinh	Não
PHP	Sin-Arcsinh	Não
Postgresql	Skew Exponential Power	Não
Python	Skew Exponential Power	Não
Spring	Sin-Arcsinh	Não
Subversion	Sin-Arcsinh	Não
Tomcat	Sin-Arcsinh	Não
VLC	Skew t	Sim

Tabela 4.24: *Distribuição da métrica NOM nos códigos analisados.*

Para a métrica de número de métodos das classes e funções dos módulos, também confirmamos a nossa hipótese H1 ao observarmos na Tabela 4.24 os diferentes tipos de distribuição que o comportamento dos valores da métrica NOM segue nos projetos analisados. Mais uma vez, de acordo com as distribuições indicadas, confirmamos também a nossa segunda hipótese. Assim, definimos os valores frequentes baseados nos percentis.

	Mín	1%	5%	10%	25%	50%	75%	90%	95%	99%	Máx
Linux	1,0	1,0	1,0	2,0	5,0	11,0	24,0	46,0	67,0	136,0	639,0
Freebsd	0,0	0,0	0,0	0,0	1,0	4,0	13,0	29,0	48,0	110,0	1535,0
Android	1	1	1	1	2	4	9	20	31	78	572
Bash	1	1	1	1	2	4	13	34	56	114	185
Chromium OS	1	1	1	1	3	7	16	32	49	104	434
GCC	1	1	1	1	2	2	7	17	30	89	3439
Gimp	1	1	2	2	5	8	16	28	37	79	179
Git	1	1	1	1	3	8	19	41	57	111	321
Gnome	1	1	1	2	4	10	21	38	56	116	406
HTTP Server	1	1	2	3	5	10	22	35	47	82	137
KVM	1	1	2	2	5	9	14	26	40	101	115
MPlayer	1	1	1	2	4	7	14	23	33	72	522
OpenLDAP	1	1	1	1	2	4	11	24	34	85	259
PHP	1	1	1	1	2	5	13	33	55	117	1508
Postgresql	1	1	1	1	4	9	21	42	68	149	992
Python	1	1	1	2	2	4	20	55	99	176	387
Subversion	1	1	1	1	3	7	19	41	63	138	266
VLC	1	1	1	1	3	7	12	25	38	73	524
Chrome	1,0	1,0	1,0	1,0	3,0	5,0	10,0	17,0	26,0	52,8	205,0
Firefox	1,0	1,0	1,0	1,0	2,0	5,0	10,0	22,0	37,0	86,0	1558,0
Inkscape	1	1	1	1	2	5	10	21	31	55	249
Media Player	1	1	1	1	3	7	15	33	54	141	2943
Mysql	1	1	1	1	3	5	11	23	38	103	895
OpenOffice	1	1	1	1	2	3	7	13	20	45	252
Eclipse	1,0	1,0	1,0	1,0	2,0	5,0	9,0	18,0	26,0	49,7	606,0
Open JDK8	1,0	1,0	1,0	1,0	2,0	3,0	8,0	17,0	27,0	60,0	596,0
Ant	1	1	1	1	2	5	11	20	29	54	126
Checkstyle	1	1	1	1	1	3	6	11	15	26	55
Eclipse Metrics	1	1	1	1	2	4	6	12	15	22	27
Findbugs	1	1	1	1	2	3	7	15	22	48	198
GWT	1	1	1	1	2	4	9	20	29	53	385
Hudson	1	1	1	1	2	3	6	12	19	43	149
JBoss	1	1	1	1	2	3	6	11	17	40	382
Kalibro	1	1	1	1	3	5	8	12	16	26	33
Log4J	1	1	1	1	2	4	8	15	23	53	123
Netbeans	1	1	1	1	2	4	9	16	23	46	1002
Spring	1	1	1	1	2	3	7	14	21	46	122
Tomcat	1	1	1	1	2	4	10	21	35	80	300

Tabela 4.25: *Percentis para a métrica* nom.

A variação dos valores, mesmo nos projetos da mesma linguagem, não nos levou a indicarmos valores frequentes únicos por linguagem de programação, por exemplo. Dessa forma, nossa hipótese H3, como para a maioria das métricas, foi refutada para a métrica NOM. Assim, indicamos os projetos FreeBSD, Chrome e OpenJDK8 como fonte dos valores frequentes, segundo nossa interpretação e abordagem. Para os projetos escritos em C temos: de 0 a 13 muito frequente; de 14 a 29 frequente; de 30 a 48 pouco frequente; acima de 48 não frequente. Para os projetos escritos em C++ indicamos: de 0 a 10 muito frequente; de 11 a 17 frequente; 18 a 26 pouco frequente; acima de 26 não frequente.

Número de atributos públicos (NPA)

	Distribuição	média representativa?
Linux	Skew t	Não
FreeBSD	Skew t	Não
Chrome	Pareto Type II	Não
Firefox	Pareto Type II	Não
Eclipse	Pareto Type II	Não
OpenJDK	Pareto Type II	Não
Android	Pareto Type II	Não
Ant	Pareto Type II	Não
Bash	Pareto Type II	Não
Checkstyle	Pareto Type II	Não
ChromiumOS	Pareto Type II	Não
Eclipse Metrics	Pareto Type II	Não
Findbugs	Pareto Type II	Não
GCC	Pareto Type II	Não
Gimp	Pareto Type II	Não
Git	Pareto Type II	Não
Gnome	Pareto Type II	Não
GWT	Pareto Type II	Não
HTTP-Server	Sin-Arcsinh	Não
Hudson	Pareto Type II	Não
Inkscape	Pareto Type II	Não
Jboss	Pareto Type II	Não
Kalibro	Pareto Type II	Não
KVM	Pareto Type II	Não
Log4j	Pareto Type II	Não
Media Player	Pareto Type II	Não
Mplayer	Skew Exponential Power	Não
Mysql	Pareto Type II	Não
Netbeans	Pareto Type II	Não
OpenLDAP	Pareto Type II	Não
OpenOffice	Pareto Type II	Não
PHP	Pareto Type II	Não
Postgresql	Pareto Type II	Não
Python	Pareto Type II	Não
Spring	Pareto Type II	Não
Subversion	Pareto Type II	Não
Tomcat	Pareto Type II	Não
VLC	Pareto Type II	Não

Tabela 4.26: *Distribuição da métrica NPA nos códigos analisados.*

Para a métrica de número de atributos públicos, predominantemente, a distribuição Pareto tipo 2 é a que melhor se ajustou aos projetos analisados, como observado na Tabela 4.26. Por conta de alguns projetos com distribuições diferentes, confirmamos nossa primeira hipótese também para o NPA. Da mesma forma, por conta das distribuições encontradas, em que a média não é representativa, confirmamos nossa segunda hipótese e nos encaminha para observarmos os percentis dos valores das métricas.

	Mín	1%	5%	10%	25%	50%	75%	90%	95%	99%	Máx
Linux	0,0	0,0	0,0	0,0	1,0	4,0	15,0	41,0	75,0	228,2	4628,0
Freebsd	0,0	0,0	0,0	0,0	0,0	2,0	9,0	29,0	57,0	207,6	4628,0
Android	0	0	0	0	0	0	0	3	9	37	567
Bash	0	0	0	0	0	2	8	24	43	103	128
Chromium OS	0	0	0	0	0	2	8	25	50	157	955
GCC	0	0	0	0	0	0	0	2	5	24	8201
Gimp	0	0	0	0	0	4	11	21	31	66	205
Git	0	0	0	0	1	4	15	31	49	135	194
Gnome	0	0	0	0	0	4	11	24	38	86	492
HTTP Server	0	0	0	0	3	8	19	37	60	135	268
KVM	0	0	0	0	0	3	9	23	27	70	91
MPlayer	0	0	0	0	1	2	6	16	26	70	121
OpenLDAP	0	0	0	0	0	0	3	9	19	51	236
PHP	0	0	0	0	0	1	4	7	11	28	75
Postgresql	0	0	0	0	0	2	11	30	57	224	345
Python	0	0	0	0	0	1	6	13	20	46	439
Subversion	0	0	0	0	0	0	1	2	4	11	39
VLC	0	0	0	0	0	0	1	3	6	24	379
Chrome	0,0	0,0	0,0	0,0	0,0	0,0	0,0	1,0	4,0	10,0	243,0
Firefox	0,0	0,0	0,0	0,0	0,0	0,0	1,0	4,0	7,0	23,0	590,0
Inkscape	0	0	0	0	0	0	1	3	6	15	271
Media Player	0	0	0	0	0	0	1	6	11	32	319
Mysql	0	0	0	0	0	0	2	7	13	40	693
OpenOffice	0	0	0	0	0	0	1	3	5	17	941
Eclipse	0,0	0,0	0,0	0,0	0,0	0,0	0,0	1,0	3,0	16,0	1072,0
Open JDK8	0,0	0,0	0,0	0,0	0,0	0,0	0,0	1,0	3,0	13,0	353,0
Ant	0	0	0	0	0	0	0	1	3	9	34
Checkstyle	0	0	0	0	0	0	0	0	0	2	162
Eclipse Metrics	0	0	0	0	0	0	0	0	4	63	230
Findbugs	0	0	0	0	0	0	0	0	1	6	39
GWT	0	0	0	0	0	0	0	0	1	4	151
Hudson	0	0	0	0	0	0	0	1	2	6	18
JBoss	0	0	0	0	0	0	0	1	2	6	275
Kalibro	0	0	0	0	0	0	0	0	0	1	5
Log4J	0	0	0	0	0	0	0	1	3	9	20
Netbeans	0	0	0	0	0	0	0	1	2	8	415
Spring	0	0	0	0	0	0	0	0	1	4	75
Tomcat	0	0	0	0	0	0	0	1	2	12	121

Tabela 4.27: *Percentis para a métrica* `npa`.

Os valores frequentes observados da métrica de atributos públicos das classes e variáveis públicas dos módulos, de acordo com o apresentado na Tabela 4.27, podem ser agrupadas para as linguagens C++ e Java. Para os projetos escritos em C sugerimos os valores de NPA para o código do GCC, uma vez que não identificamos um projeto com valores equilibrados dentre os demais, segundo nossa interpretação. O GCC apresenta os melhores valores, assim, para C temos: 0 muito frequente; até 2 frequente; de 3 a 5 pouco frequente; acima de 5 não frequente. Para os projetos escritos em C++ sugerimos: 0 muito frequente; 1 frequente; de 2 a 4 pouco frequente; acima de 4 não frequente. Por fim, para os projetos escritos em Java identificamos: 0 muito frequente; 1 frequente; 2 e 3 pouco frequentes; acima de 3 não frequente.

Número de métodos públicos (NPM)

	Distribuição	média representativa?
Linux	Skew Exponential Power	Não
FreeBSD	Skew Exponential Power	Não
Chrome	Skew Exponential Power	Não
Firefox	Skew Exponential Power	Não
Eclipse	Skew Exponential Power	Não
OpenJDK	Skew Exponential Power	Não
Android	Skew Exponential Power	Não
Ant	Skew Exponential Power	Não
Bash	Sin-Arcsinh	Não
Checkstyle	Pareto Type II	Não
ChromiumOS	Skew Exponential Power	Não
Eclipse Metrics	Skew Exponential Power	Não
Findbugs	Skew Exponential Power	Não
GCC	Sin-Arcsinh	Não
Gimp	Skew t	Não
Git	Sin-Arcsinh	Não
Gnome	Skew Exponential Power	Não
GWT	Skew Exponential Power	Não
HTTP-Server	Skew t	Não
Hudson	Skew Exponential Power	Não
Inkscape	Skew Exponential Power	Não
Jboss	Pareto Type II	Não
Kalibro	Skew Exponential Power	Não
KVM	Skew t	Não
Log4j	Skew Exponential Power	Não
Media Player	Skew Exponential Power	Não
Mplayer	Skew t	Não
Mysql	Sin-Arcsinh	Não
Netbeans	Skew Exponential Power	Não
OpenLDAP	Sin-Arcsinh	Não
OpenOffice	Sin-Arcsinh	Não
PHP	Sin-Arcsinh	Não
Postgresql	Skew Exponential Power	Não
Python	Skew Exponential Power	Não
Spring	Skew Exponential Power	Não
Subversion	Skew Exponential Power	Não
Tomcat	Skew Exponential Power	Não
VLC	Skew t	Não

Tabela 4.28: *Distribuição da métrica NPM nos códigos analisados.*

Para a métrica de número de métodos públicos e funções públicas, a distribuição *Skew Exponential Power* é a que melhor se ajustou na maioria dos projetos analisados, como observado na Tabela 4.26. Mesmo assim, confirmamos nossa primeira hipótese também para o NPM, porque há uma certa variação nos tipos de distribuição para os valores de NPM entre os projetos. Independente de serem diferentes, as distribuições encontradas sinalizam que a média não é uma valor representativo, o que mais um vez confirma nossa segunda hipótese.

	Mín	1%	5%	10%	25%	50%	75%	90%	95%	99%	Máx
Linux	1,0	1,0	1,0	2,0	5,0	11,0	24,0	46,0	67,0	136,0	639,0
Freebsd	0,0	0,0	0,0	0,0	1,0	4,0	12,0	29,0	48,0	110,0	1535,0
Android	0	0	0	1	1	3	7	16	26	64	553
Bash	1	1	1	1	2	4	13	34	56	114	185
Chromium OS	0	1	1	1	3	7	16	32	49	104	434
GCC	0	0	1	1	1	2	5	14	26	84	3439
Gimp	1	1	2	2	5	8	16	28	37	79	179
Git	1	1	1	1	3	8	19	41	57	111	321
Gnome	0	1	1	2	4	10	21	38	56	116	406
HTTP Server	1	1	2	3	5	10	22	35	47	82	137
KVM	1	1	2	2	5	9	14	26	40	101	115
MPlayer	1	1	1	2	4	7	14	23	33	72	522
OpenLDAP	1	1	1	1	2	4	11	24	34	85	259
PHP	1	1	1	1	2	5	13	33	55	117	1508
Postgresql	1	1	1	1	4	9	21	42	68	149	992
Python	1	1	1	2	2	4	20	55	99	176	387
Subversion	0	0	1	1	3	6	19	41	63	137	266
VLC	0	1	1	1	3	6	12	24	37	73	524
Chrome	0,0	0,0	0,0	1,0	2,0	3,0	7,0	13,0	19,0	42,8	188,0
Firefox	0,0	0,0	1,0	1,0	2,0	4,0	9,0	19,0	32,0	78,7	1558,0
Inkscape	0	1	1	1	2	4	9	18	28	50	249
Media Player	0	1	1	1	2	5	13	31	51	140	2943
Mysql	0	0	1	1	2	4	10	22	36	102	895
OpenOffice	0	0	1	1	1	3	6	11	17	40	252
Eclipse	0,0	0,0	0,0	1,0	1,0	3,0	6,0	12,0	18,0	36,0	560,0
Open JDK8	0,0	0,0	0,0	0,0	1,0	2,0	5,0	11,0	18,0	45,0	465,0
Ant	0	0	0	1	2	4	9	16	23	40	109
Checkstyle	0	0	0	0	1	2	4	8	11	21	32
Eclipse Metrics	0	0	0	1	1	3	4	8	11	19	24
Findbugs	0	0	0	0	1	2	5	11	18	39	197
GWT	0	0	0	0	1	3	7	16	27	45	384
Hudson	0	0	0	1	1	3	5	10	15	38	110
JBoss	0	0	0	0	1	2	4	9	14	33	382
Kalibro	0	0	0	0	1	3	6	9	12	21	26
Log4J	0	0	0	1	2	3	7	13	19	47	122
Netbeans	0	0	0	1	1	3	6	12	17	35	1002
Spring	0	0	0	1	1	2	6	11	17	35	119
Tomcat	0	0	0	1	2	3	8	17	28	67	278

Tabela 4.29: *Percentis para a métrica npm.*

Nossa terceira hipótese não é correspondida para os valores de NPM, conforme observamos nos valores reportados na Tabela 4.29. Dessa forma, sugerimos verificar os valores de NPM do GCC, Chrome e Open JDK8. Para os projetos escritos em C temos: de 0 a 2 muito frequente; de 3 a 5 frequente; de 6 a 14 pouco frequente; acima de 14 não frequente. No caso dos projetos escritos em C++ indicamos: de 0 a 3 muito frequente; de 4 a 7 frequente; de 8 a 13 pouco frequente; acima de 13 não frequente.

Respostas para uma classe (RFC)

	Distribuição	média representativa?
Linux	Skew Exponential Power	Não
FreeBSD	Skew Exponential Power	Não
Chrome	Pareto Type II	Não
Firefox	Johnson's SU	Sim
Eclipse	Skew Exponential Power	Não
OpenJDK	Skew Exponential Power	Não
Android	Johnson's Su	Sim
Ant	Skew Exponential Power	Não
Bash	Skew Exponential Power	Não
Checkstyle	Sin-Arcsinh	Não
ChromiumOS	Johnson's Su	Sim
Eclipse Metrics	Skew Exponential Power	Não
Findbugs	Sin-Arcsinh	Não
GCC	Sin-Arcsinh	Não
Gimp	Generalized Gamma	Sim
Git	Skew Exponential Power	Não
Gnome	Skew Exponential Power	Não
GWT	Sin-Arcsinh	Não
HTTP-Server	Skew Exponential Power	Não
Hudson	Johnson's Su	Sim
Inkscape	Skew Exponential Power	Não
Jboss	Skew Exponential Power	Não
Kalibro	Skew Exponential Power	Não
KVM	Skew t	Não
Log4j	Skew Exponential Power	Não
Media Player	Skew t	Não
Mplayer	Skew Exponential Power	Não
Mysql	Johnson's Su	Sim
Netbeans	Skew Exponential Power	Não
OpenLDAP	Skew Exponential Power	Não
OpenOffice	Sin-Arcsinh	Não
PHP	Johnson's Su	Sim
Postgresql	Skew Exponential Power	Não
Python	Skew Exponential Power	Não
Spring	Sin-Arcsinh	Não
Subversion	Skew Exponential Power	Não
Tomcat	Skew Exponential Power	Não
VLC	Skew Exponential Power	Não

Tabela 4.30: *Distribuição da métrica RFC nos códigos análisados.*

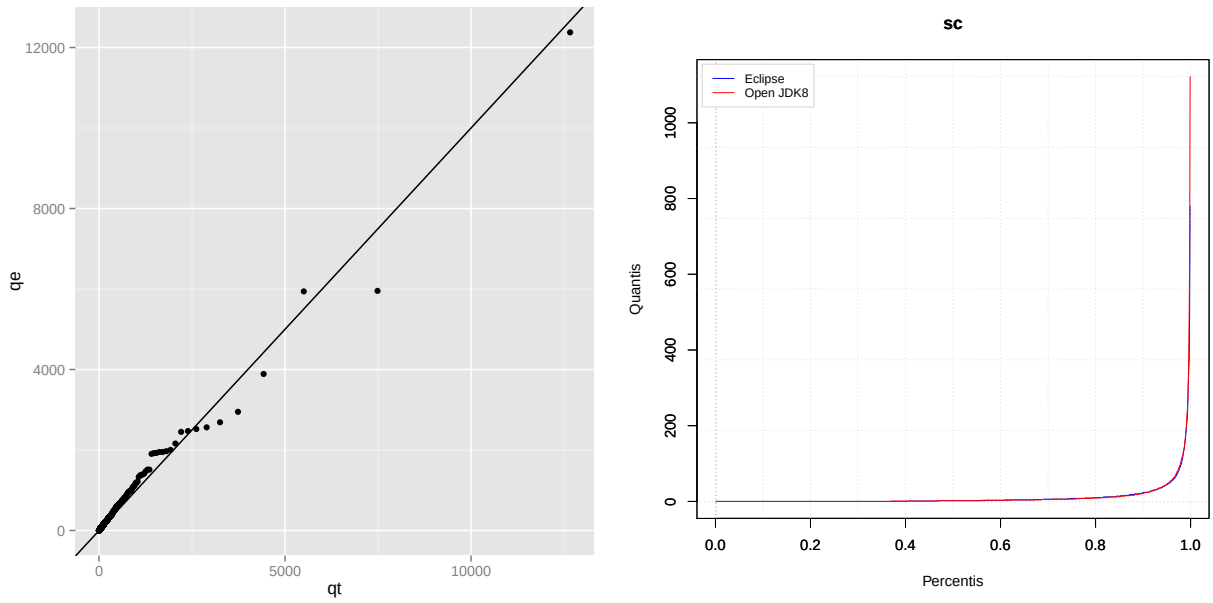
Nossa hipótese H1 também se confirmou para a métrica resposta para uma classe. Conforme observado na Tabela 4.30, os tipos de distribuição para os valores de RFC entre os projetos são diferentes. A distribuição *Skew Exponential Power* é a que melhor se ajustou na maioria dos projetos analisados e, somado as outras distribuições que indicam a média como não representativa, confirma nossa segunda hipótese. De qualquer maneira, podemos observar outros tipos de distribuição que sinalizam a média como informativa. Assim, dependendo do conjunto de projetos, uma abordagem como a de Lanza e Marinescu (2006) pode ser aplicada.

	Mín	1%	5%	10%	25%	50%	75%	90%	95%	99%	Máx
Linux	1,0	1,0	2,0	4,0	11,0	32,0	86,0	197,0	316,0	757,0	4341,0
Freebsd	0,0	0,0	0,0	0,0	2,0	13,0	54,0	152,0	271,5	675,0	14328,0
Android	1	1	1	1	4	10	29	78	138	372	6058
Bash	1	1	1	2	4	12	51	144	298	743	878
Chromium OS	1	1	2	3	8	22	57	132	212	530	2593
GCC	1	1	1	1	3	6	18	54	107	427	4811
Gimp	1	2	6	9	20	44	96	185	250	469	1144
Git	1	1	2	4	17	49	137	315	460	1065	1674
Gnome	1	1	2	4	12	32	84	176	269	612	2436
HTTP Server	1	1	4	8	23	58	132	266	345	620	886
KVM	1	2	3	7	21	43	71	151	299	495	707
MPlayer	1	1	1	4	16	38	86	171	253	452	1080
OpenLDAP	1	1	2	4	13	31	81	178	287	458	1307
PHP	1	1	2	3	6	20	64	160	273	682	12808
Postgresql	1	1	2	5	17	58	146	308	505	1029	3183
Python	1	1	1	4	9	24	91	289	500	963	2035
Subversion	1	1	1	3	11	35	113	274	407	999	2746
VLC	1	1	1	2	7	25	68	155	258	531	1064
Chrome	1,0	1,0	1,0	2,0	5,0	13,0	29,0	64,0	102,0	246,0	1301,0
Firefox	1,0	1,0	1,0	2,0	5,0	14,0	38,0	100,0	189,0	520,0	12487,0
Inkscape	1	1	1	2	4	12	36	92	157	354	1469
Media Player	1	1	2	2	7	19	48	100	149	334	2943
Mysql	1	1	2	3	6	16	47	123	212	833	2488
OpenOffice	1	1	1	1	3	8	18	41	67	174	695
Eclipse	1,0	1,0	1,0	1,0	4,0	11,0	28,0	62,0	99,0	221,0	3024,0
Open JDK8	1,0	1,0	1,0	1,0	3,0	9,0	26,0	59,0	102,0	264,0	1603,0
Ant	1	1	1	2	5	14	34	72	111	209	405
Checkstyle	1	1	1	1	2	6	16	31	42	80	270
Eclipse Metrics	1	1	1	2	4	7	19	37	57	89	112
Findbugs	1	1	1	1	2	6	17	43	74	180	703
GWT	1	1	1	1	2	7	20	39	65	159	1088
Hudson	1	1	1	1	3	6	14	27	45	106	292
JBoss	1	1	1	1	3	7	15	31	49	125	543
Kalibro	1	1	1	2	4	8	15	29	39	58	84
Log4J	1	1	1	2	4	8	23	52	85	193	419
Netbeans	1	1	1	1	3	9	21	46	72	164	2006
Spring	1	1	1	1	2	6	17	41	66	170	644
Tomcat	1	1	1	2	4	11	30	74	130	275	1215

Tabela 4.31: *Percentis para a métrica* `rfc`.

Para a métrica RFC, avaliamos que podemos interpretar os valores frequentes verificando os percentis 75, 90 e 95, apesar da mediana poder ser considerada um ponto de corte ao observamos os dados da Tabela 4.31. Como refutamos nossa hipótese H3 por conta da variação entre os valores dessa métrica entre os projetos, ou seja, não conseguimos indicar valores únicos, mesmo agrupando por linguagem de programação, sugerimos observar os valores da métrica RFC dos projetos FreeBSD, Chrome e Open JDK8. Para os projetos escritos em C, sugerimos: de 0 a 54 muito frequente; de 55 a 152 frequente; de 153 a 271 pouco frequente; acima de 272 não frequente. Para os os projetos escritos em C++, temos: de 0 a 29 muito frequente; de 30 a 64 frequente; de 65 a 102 pouco frequente; acima de 102 não frequente. Ainda, para os projetos escritos em Java, indicamos: de 0 a 9 muito frequente; de 10 a 26 frequente; de 27 a 59 pouco frequente; acima de 59 não frequente.

Complexidade estrutural (SC)



(a) Pareto Type II

(b) Percentis

Figura 4.6: *Distribuição da métrica SC no código Eclipse e Open JDK: Pareto Type II e Percentis.*

Os gráficos da Figura 4.6(a) mostram que, para o código dos projetos Eclipse e Open JDK, a melhor distribuição que se ajusta aos valores da complexidade estrutural é a Pareto tipo 2, como na maioria dos projetos avaliados. Na Figura 4.6(b) observamos que os valores de SC para o código do Eclipse e do Open JDK são bem próximos.

	Distribuição	média representativa?
Linux	Sinh-Arcsinh	Não
FreeBSD	Skew t	Não
Chrome	Pareto Type II	Não
Firefox	Pareto Type II	Não
Eclipse	Pareto Type II	Não
OpenJDK	Pareto Type II	Não
Android	Pareto Type II	Não
Ant	Pareto Type II	Não
Bash	Pareto Type II	Não
Checkstyle	Pareto Type II	Não
ChromiumOS	Sin-Arcsinh	Não
Eclipse Metrics	Pareto Type II	Não
Findbugs	Pareto Type II	Não
GCC	Pareto Type II	Não
Gimp	Pareto Type II	Não
Git	Skew Exponential Power	Não
Gnome	Skew Exponential Power	Não
GWT	Pareto Type II	Não
HTTP-Server	Pareto Type II	Não
Hudson	t Family	Sim
Inkscape	Pareto Type II	Não
Jboss	t Family	Sim
Kalibro	Pareto Type II	Não
KVM	Skew Exponential Power	Não
Log4j	Pareto Type II	Não
Media Player	Pareto Type II	Não
Mplayer	Skew Exponential Power	Não
Mysql	Pareto Type II	Não
Netbeans	Pareto Type II	Não
OpenLDAP	Skew Exponential Power	Não
OpenOffice	Pareto Type II	Não
PHP	Sin-Arcsinh	Não
Postgresql	Skew Exponential Power	Não
Python	Sin-Arcsinh	Não
Spring	Pareto Type II	Não
Subversion	Pareto Type II	Não
Tomcat	Pareto Type II	Não
VLC	Pareto Type II	Não

Tabela 4.32: *Distribuição da métrica SC nos códigos análisados.*

Apesar da predominância da distribuição Pareto tipo 2, há uma variação entre os tipos de distribuição dos projetos (ver Tabela 4.32), confirmando também nossa hipótese H1 para a métrica SC. Além da Pareto tipo 2, que são leis de potência, a maior parte do tipo de distribuição para SC sinalizam a média com uma medina não informativa, o que confirma também nossa segunda hipótese.

	Mín	1%	5%	10%	25%	50%	75%	90%	95%	99%	Máx
Linux	0,0	0,0	0,0	0,0	2,0	12,0	38,0	108,0	210,0	656,6	14805,0
Freebsd	0,0	0,0	0,0	0,0	0,0	4,0	18,0	77,0	168,0	691,7	79794,0
Android	0	0	0	0	0	1	6	18	42	264	28475
Bash	0	0	0	0	1	6	36	107	347	1897	2709
Chromium OS	0	0	0	0	2	12	44	120	224	679	48642
GCC	0	0	0	0	0	2	8	24	51	422	87264
Gimp	0	0	1	3	8	20	50	92	154	565	5076
Git	0	0	0	1	4	11	27	65	98	586	9690
Gnome	0	0	0	0	1	6	20	60	112	405	6948
HTTP Server	0	0	0	5	15	35	79	202	323	695	2208
KVM	0	0	0	0	3	11	24	67	117	215	1540
MPlayer	0	0	0	0	6	16	45	99	160	620	10440
OpenLDAP	0	0	0	0	5	16	52	136	275	1513	3572
PHP	0	0	0	0	2	12	49	256	720	2359	36150
Postgresql	0	0	0	1	6	20	72	224	510	1815	144686
Python	0	0	0	0	4	14	108	618	1274	2846	5600
Subversion	0	0	0	0	1	12	53	230	481	3476	23544
VLC	0	0	0	0	0	10	42	154	336	1042	7860
Chrome	0,0	0,0	0,0	0,0	0,0	3,0	12,0	28,0	51,1	168,1	1054,0
Firefox	0,0	0,0	0,0	0,0	0,0	6,0	24,0	90,0	221,0	1185,4	146265,0
Inkscape	0	0	0	0	0	4	12	45	88	258	2448
Media Player	0	0	0	0	0	4	15	40	70	256	3468
Mysql	0	0	0	0	1	9	33	108	260	1384	20200
OpenOffice	0	0	0	0	0	2	8	27	54	198	4032
Eclipse	0,0	0,0	0,0	0,0	0,0	2,0	8,0	24,0	45,0	178,6	5952,0
Open JDK8	0,0	0,0	0,0	0,0	0,0	2,0	6,0	21,0	45,0	185,5	14824,0
Ant	0	0	0	0	0	3	7	18	30	79	400
Checkstyle	0	0	0	0	0	0	4	8	12	32	112
Eclipse Metrics	0	0	0	0	0	1	4	7	12	20	36
Findbugs	0	0	0	0	0	0	2	10	21	81	2328
GWT	0	0	0	0	0	0	4	12	24	93	3500
Hudson	0	0	0	0	0	0	2	8	16	80	3129
JBoss	0	0	0	0	0	0	3	8	14	44	1080
Kalibro	0	0	0	0	0	0	5	16	27	71	110
Log4J	0	0	0	0	0	1	4	12	18	77	165
Netbeans	0	0	0	0	0	0	4	9	18	60	7917
Spring	0	0	0	0	0	0	3	10	22	120	3045
Tomcat	0	0	0	0	0	1	4	12	24	99	1536

Tabela 4.33: *Percentis para a métrica SC.*

No caso da métrica de complexidade estrutural, mais uma vez refutamos a hipótese H3, como podemos observar nos valores reportados na Tabela 4.33. Novamente, avaliamos os projetos FreeBSD, Chrome e OpenJDK como possíveis referências para C, C++ e Java, respectivamente. Assim, para a métrica SC em projetos escritos em C, temos: de 0 a 18 muito frequente; de 19 a 77 frequente; de 78 a 168 pouco frequente; acima de 168 não frequente. Para os projetos em C++, indicamos: de 0 a 12 muito frequente; de 13 a 28 frequente; de 29 a 51 pouco frequente; acima de 51 não frequente. Por último, para os projetos escritos em Java, sugerimos: de 0 a 6 muito frequente; de 7 a 21 frequente; de 22 a 45 pouco frequente; acima de 45 não frequente.

Na maioria de nossas análises das métricas neste estudo, conseguimos confirmações positivas das nossas hipóteses H1 e H2. Entretanto, para a terceira hipótese não conseguimos. Isso significa que temos que avaliar outras formas possíveis de agrupamento na tentativa de aproximar os valores

frequentes dos projetos, ou seja, nem sempre os projetos com os melhores valores são boas referências para outros. Como já comentado, a junção de critérios como a linguagem de programação, o domínio de aplicação e/ou a idade (maturidade) do projeto pode ser uma alternativa, que não testamos neste estudo.

4.3.3 Efeito recíproco entre métricas do código e atividades de desenvolvimento

A sequência dos dois conjuntos de estudos apresentados nesta tese, será novo estudo com uma abordagem mais completa para explicar a relação entre as métricas de código-fonte, as quais indicam a qualidade do código, e a atratividade de projetos de software livre (Santos *et al.*, 2013), com o foco no monitoramento de código-fonte de projetos de software livre. Essa abordagem é baseada numa proposta de efeito recíproco entre os valores das métricas de código-fonte e atividades de desenvolvimento (*commits* e *commiters*). Nesse caso, não fazendo *proxy* de número de downloads e membros para número de usuários e contribuidores, como ocorreu nos primeiros estudos. Assim, definimos o denominamos de "atividades de desenvolvimento" como sendo o número de desenvolvedores (*commiters*), obtidos pela quantidade de diferentes desenvolvedores que realizaram atualizações nos repositórios dos projetos, e o número de *commits* dos projetos. O modelo teórico desse estudo proposto é ilustrado na Figura 4.7.

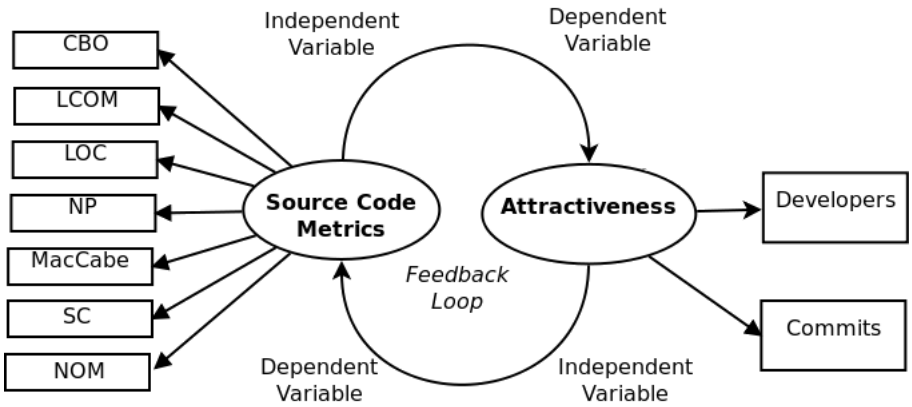
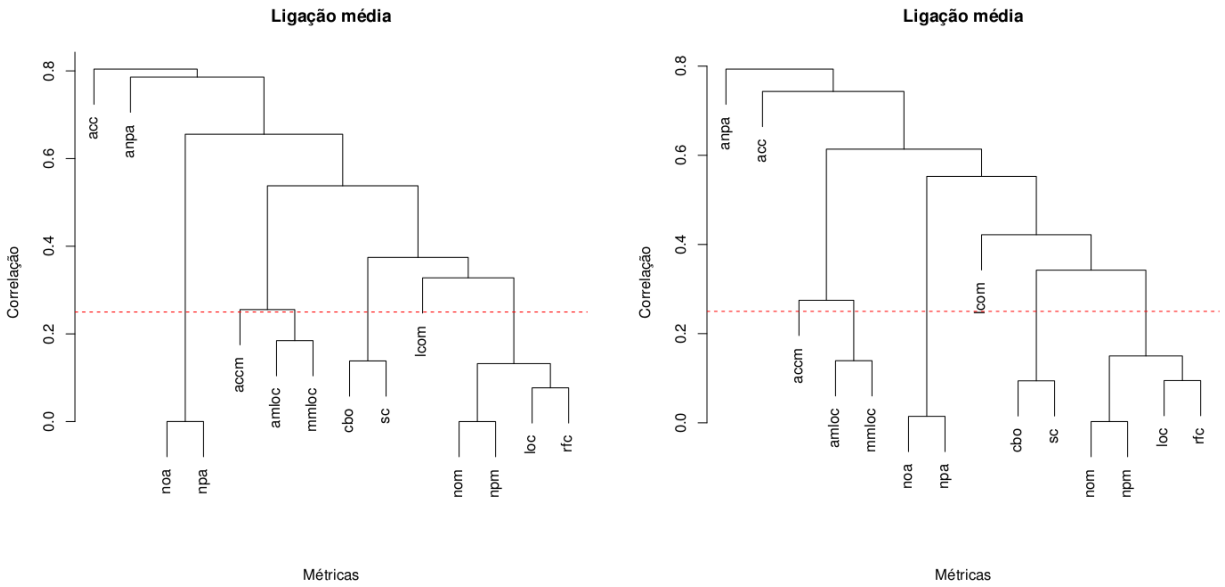


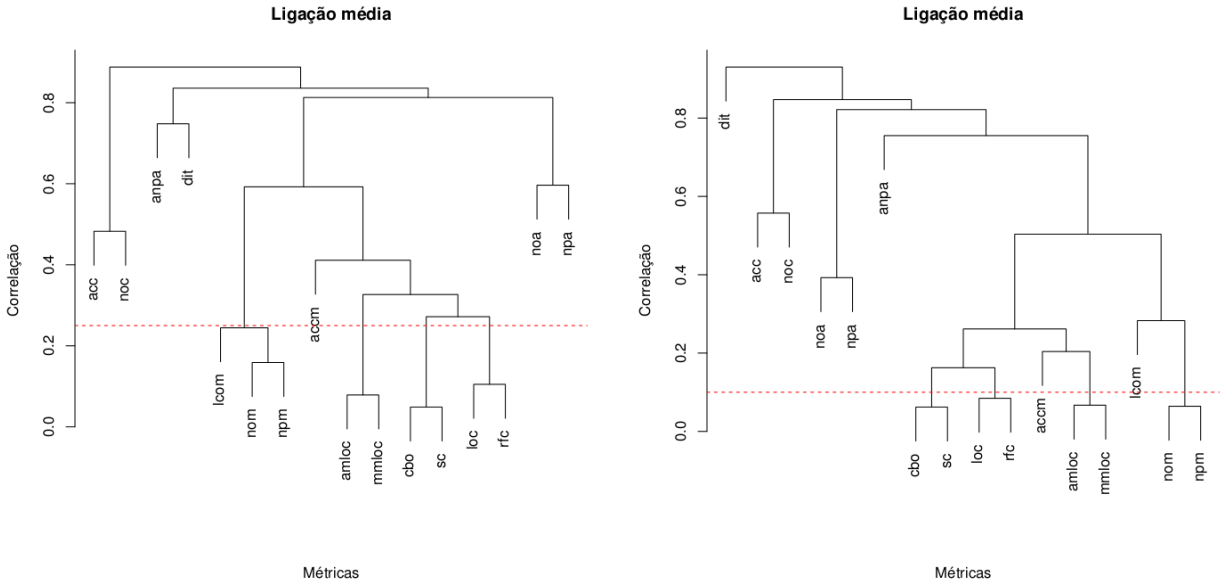
Figura 4.7: *Modelo do estudo sobre o efeitos recíproco entre métricas e atratividade.*

Para esse estudo nos perguntamos se as métricas são influenciadas pelas atividades de desenvolvimento ou se também as atividades de desenvolvimento são influenciadas pela qualidade do código? Nossa hipótese é que ambos os casos são verdadeiros, o que gera o ciclo positivo entre as variáveis, exemplificadas na Figura 4.7, que são dependentes e independentes ao mesmo tempo. Uma vez que estudamos os valores de 15 métricas, o primeiro passo para esse novo estudo foi achar as correlações entre as métricas e, assim, diminuir a quantidade de variáveis do modelo para torná-lo mais menos complexo para o estudo. Dessa forma, também respondendo se podemos encontrar um subconjunto representativo de métricas para o monitoramento de código-fonte.



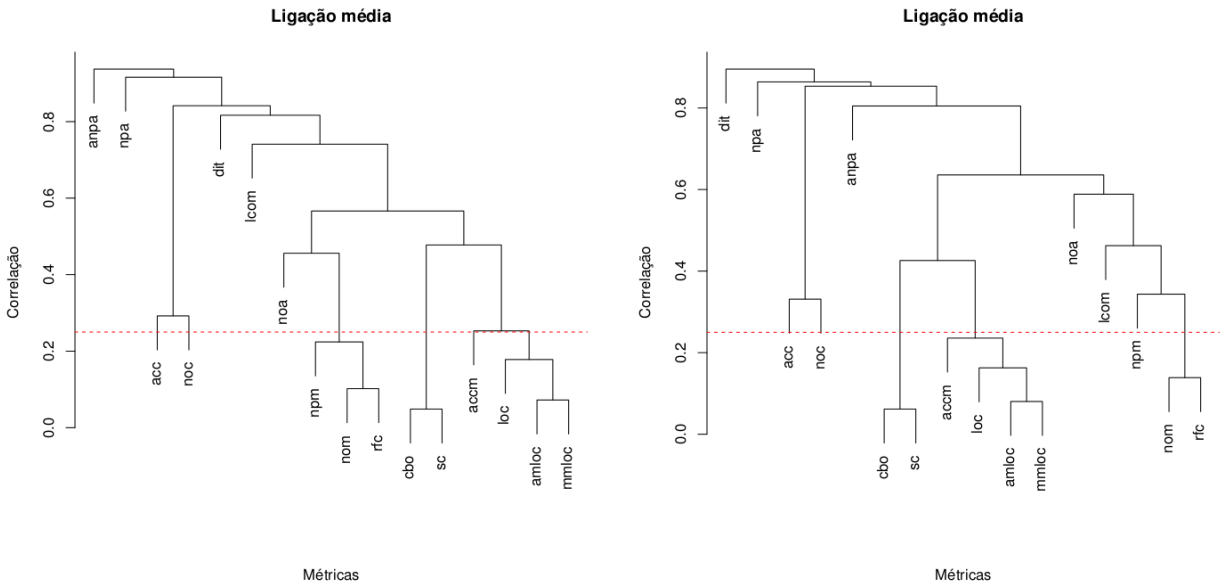
(a) Linux (b) FreeBSD

Figura 4.8: *Seleção de métricas: dendrograma das correlações.*



(a) Chrome (b) Firefox

Figura 4.9: *Seleção de métricas: dendrograma das correlações.*



(a) Eclipse (b) OpenJDK8

Figura 4.10: *Seleção de métricas: dendrograma das correlações.*

Firefox	Data	commits	commiters	acc	accm	amloc	anpm	dit	nom	npa	sc
0.8	02/04	1625	5,6	3,0	3,1	22,3	1,1	1,0	12,0	2,0	33,0
0.9	06/04	1915	12,8	3,0	3,1	22,3	1,1	1,0	12,0	2,0	32,0
0.10	09/04	2096	14,3	3,0	3,1	22,3	1,1	1,0	12,0	2,0	32,0
1.0	12/04	2252	13,0	3,0	3,1	22,3	1,1	1,0	12,0	2,0	32,0
1.5	12/05	3373	18,8	3,0	3,1	22,3	1,1	1,0	12,0	2,0	29,0
2.0	10/06	4715	21,7	3,0	3,1	22,3	1,1	1,0	12,0	2,0	29,0
3.0	06/08	24291	94,9	3,0	2,9	21,6	1,1	1,0	11,0	2,0	30,0
3.5	07/09	41004	153,5	3,0	2,8	21,1	1,1	1,0	11,0	2,0	32,0
3.6	02/10	50067	161,3	3,0	2,8	21,2	1,1	1,0	11,0	3,0	32,0
4.0	03/11	75825	190,8	3,0	2,5	20,2	1,0	1,0	11,0	2,0	18,0
5.0	06/11	81049	197,0	3,0	2,5	20,2	1,0	1,0	11,0	2,0	18,0
6.0	08/11	84768	220,5	3,0	2,5	20,2	1,0	1,0	11,0	2,0	18,0
7.0	09/11	86360	205,0	3,0	2,5	20,2	1,1	1,0	11,0	2,0	18,0
8.0	11/11	90265	217,0	2,0	2,7	20,0	1,1	1,0	10,0	3,0	24,0
9.0	12/11	92375	229,0	2,0	2,7	20,0	1,1	1,0	10,0	3,0	24,0
10.0	01/12	94767	255,0	2,0	2,7	20,0	1,1	1,0	10,0	1,0	24,0
11.0	03/12	100053	267,5	2,0	2,7	19,5	1,1	1,0	10,0	1,0	24,0
12.0	04/12	102186	256,0	2,0	2,7	19,5	1,1	1,0	10,0	1,0	24,0
13.0	06/12	107784	283,0	2,0	2,7	19,5	1,1	1,0	10,0	1,0	24,0
14.0.1	07/12	110843	284,0	2,0	2,7	19,5	1,1	1,0	10,0	1,0	24,0

Tabela 4.34: *Evolução métricas Firefox.*

Para a seleção das métricas, verificamos as correlações estatísticas entre elas. Para isso, foi calculada a matriz de correlações das 15 métricas analisadas em detalhes nesta tese. Assim, foi considerado o coeficiente de Correlação de Spearman, que tem como única hipótese os dados serem medidos em escala mínima ordinal. Para a escolha do tipo de ligação a ser utilizado nos gráficos, foram calculados os coeficientes cofenéticos considerando as ligações completa, simples e média, tendo sido obtidos os valores de 0.82 a 0,91 para ligação média, que apresentou os melhores resultados para os

dendrogramas, ilustrados no conjunto de Figuras 4.8, 4.9 e 4.10. Nos dendrogramas, foram traçadas retas considerando as correlações superiores a 0,75, ou seja, todas as métricas abaixo desse corte possuem uma correlação superior a 0,75, o que indica que estatisticamente essas métricas têm a mesma representatividade, ou seja, tratam informações similares. Após a análises desses dendrogramas, observamos um subconjunto de oito métricas representativas e correlacionadas entre si: ACC, ACCM, ANPM, DIT, AMLOC, NOM, NPA, SC. De posse desse subconjunto e conhecendo o comportamento dele, ou seja, as distribuições das métricas para saber quais medidas são informativas, iniciamos uma análise da evolução do código-fonte do Firefox e Chrome, como primeiro estudo desse efeito recíproco entre as métricas de código-fonte e atividades de desenvolvimento de projetos de software livre. Como exemplo, na Tabela 4.34 é detalhado um monitoramento das métricas selecionadas para 20 versões do Firefox, desde fevereiro de 2004 até julho de 2012. Além das oito métricas selecionadas, de acordo com as análises dos dendrogramas, apresentamos (i) a versão do Firefox; (ii) o mês e ano da respectiva versão; (iii) a variação de *commits* entre um release e outro (ou seja, quantos *commits* a mais foram feitos de um release para o outro); (iv) a variação da média de contribuidores entre um período e outro. Neste estudo, não conseguimos aplicar ainda um modelo estatístico adequado para explicar de maneira mais rigorosa esse tipo de causalidade em *loop*. O "Teorema de Caracterização de Causalidade de Granger" (Hsiao e of Toronto. Institute for Policy Analysis, 1979) mostrou-se como uma alternativa. Entretanto, como primeiro passo, avaliamos empiricamente os dados. Validamos como podemos monitorar as métricas, ao definir um conjunto menor de métricas, selecionando os seus respectivos valores informativos, de acordo com as análises das distribuições e percentis apresentadas neste capítulo. Portanto, certamente, a aplicação desse teorema é um ponto de partida para os trabalhos futuros desta tese.

Capítulo 5

Ferramentas

Neste capítulo, apresentaremos as ferramentas utilizadas nos estudos descritos nesta tese de doutorado e as que foram desenvolvidas para cobrir os conceitos e ideias discutidas neste trabalho. O objetivo dessas ações foi colocar em prática as ideias desenvolvidas na tese e disponibilizá-las como uma contribuição para as comunidades de software livre, de acordo com nossos objetivos tecnológicos:

- OT1 – Desenvolvimento e adaptação da ferramenta de coleta de métricas de código-fonte – Analizo.
- OT2 – Desenvolvimento do *web service* para interpretação de métricas de código-fonte – Kalibro.
- OT3 – Desenvolvimento da rede de monitoramento de métricas de código-fonte – Mezuro.

5.1 Contexto do desenvolvimento das ferramentas

Ao longo deste trabalho, definimos uma plataforma de monitoramento de métricas de código-fonte denominada Mezuro. Essa plataforma é composta por um conjunto de ferramentas com as quais colaboramos em seu desenvolvimento ou implementamos de acordo com as necessidades e critérios que foram definidos aos longo de nossos estudos.

No centro da arquitetura dessa plataforma está o Kalibro Metrics, que é um serviço web para configuração, interpretação e monitoramento de métricas de código-fonte. Antes de ser um serviço web, em 2009, os conceitos e funcionalidades da Kalibro foram prototipados no contexto do projeto QualiPSo, quando trabalhamos na ferramenta JaBUTi - *Java Bytecode Understanding and Testing* (Vincenzi *et al.*, 2003), que estava sendo adaptada para o QualiPSo.

A JaBUTi é uma ferramenta que analisa software em Java através do processamento de código objeto (*bytecode*) para medir cobertura de testes e calcula algumas métricas de complexidade com o objetivo de identificar as partes críticas do código, aquelas que mais precisam ser testadas (porém, isso ainda não foi implementado efetivamente na JaBUTi). Inicialmente, a proposta foi melhorar o módulo de cálculo de métricas de código-fonte da JaBUTi. Entretanto, o foco passou para a visualização de métricas e decidimos fazer um módulo separado, no qual seria possível configurar intervalos para as métricas, tornando seus resultados mais fáceis de apresentar. Posteriormente, decidimos transformar esse módulo em uma ferramenta independente, que chamamos de Crab (Meirelles *et al.*, 2009) e a disponibilizamos sob a licença BSD.

A Crab trazia os conceitos de configuração de métricas, criação de métricas compostas e conexão com a ferramenta base. A JaBUTi calculava as métricas e então chamava a Crab para visualizá-las. Sua independência (i.e., a generalidade da interface de conexão) precisava ser validada através da conexão com outra ferramenta base e a apresentação dos resultados precisava ser melhorada.

Logo após a experiência com a JaBUTi, procuramos uma alternativa de ferramenta base (coletor de métricas de código-fonte) que fosse mais rápida e, em especial, fosse capaz de trabalhar

com outras linguagens além de Java para cobrirmos mais amplamente as análises dos projetos de software livre. Com isso, chegamos à Analizo (Terceiro *et al.*, 2010a), que é um software livre baseado inicialmente em outro software livre chamado Egypt[1]. O Egypt teve o suporte ao cálculo de métricas implementado inicialmente por Antonio Terceiro[2], da Universidade Federal da Bahia. Posteriormente, o projeto começou a receber diversas contribuições do nosso grupo do CCSL-USP, no contexto desta tese de doutorado e do projeto QualiPSo. Dessa forma, a ferramenta passou a ter funcionalidades e sub-ferramentas que fugiram da proposta inicial do Egypt, passando assim a ser outro software, denominado de Analizo ("análise" em esperanto).

Com a Analizo em mãos, começamos a trabalhar na integração da Crab com ela. Ao contrário da JaBUTi, que possuía uma interface Java Swing, a Analizo é uma ferramenta escrita em Perl sem interface gráfica, portanto seria preciso implementar um componente visual para o Analizo, que acionasse a Crab para visualizar os resultados. Ao invés desse caminho, no escopo do trabalho de conclusão do aluno Carlos Morais, integramos a Crab de forma que ficasse acima da ferramenta base, no caso a Analizo, a chamando para coletar os resultados.

Essa inversão de paradigma de integração, junto com outras mudanças de impacto, fez com que a Crab fosse relicenciada sob LGPL[3] versão 3 e, para homogenizar os nomes das ferramentas do projeto, foi rebatizada de Kalibro ("calibrar" em esperanto). Ao final da integração com a Analizo, e como uma primeira versão para ao projeto QualiPSo, o Kalibro possuía base de dados própria, calculava estatísticas para as métricas coletadas e exibia os resultados baseando-se nas configurações cadastradas.

Em 2010, ainda no contexto do QualiPSo, iniciamos a integração da Kalibro/Analizo com a plataforma Spago4Q, para análise de projetos C e C++, pois até então o QualiPSo estava restrito ao Java. Construímos uma interface de serviço Web para um extrator da Spago4Q que pudesse coletar as métricas usando a Analizo através do Kalibro. Baseado nessa experiência, separamos a coleta, configuração e interpretação de métricas da sua visualização.

Dessa separação surgiu o Kalibro Service, um serviço web sem interface gráfica que possui todas as funcionalidades não-visuais do que hoje constitui a plataforma Mezuro, que em resumo tem: (i) acompanhamento de repositórios de código; (ii) cálculo de métricas através de ferramentas base; (iii) criação e compartilhamento de configurações e de interpretações de métricas.

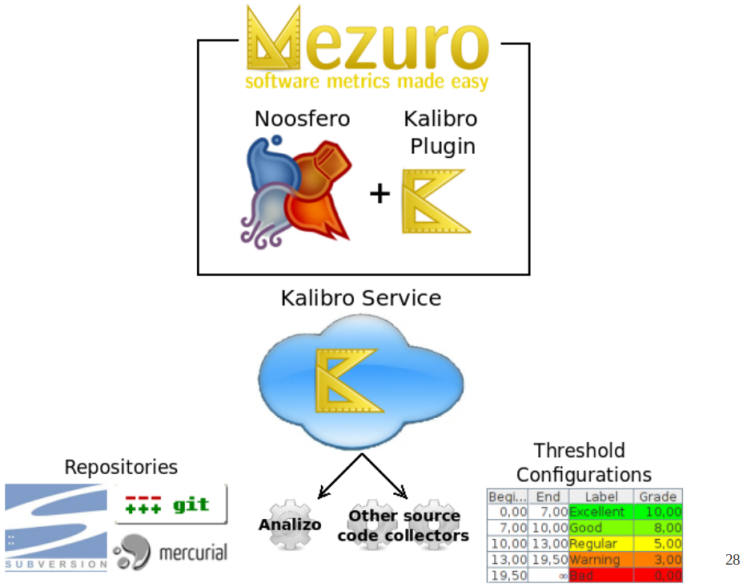


Figura 5.1: *Interação e integração das ferramentas do projeto Mezuro.*

[1] gson.org/egypt
[2] softwarelivre.org/terceiro
[3] http://www.opensource.org/licenses/lgpl-license.php

A rede de monitoramento Mezuro é a principal contribuição do ponto vista prático como parte desta tese de doutorado. Como ilustrado na Figura 5.1, o Mezuro faz uso de uma plataforma de software livre para rede sociais (Noosfero), a estendendo através de um plugin, para se conectar ao Kalibro Metrics Service, que por sua vez é capaz de se integrar a diversos coletores de métricas de código-fonte. Em suma, o Mezuro provê a visualização dos resultados processados pelo Kalibro e possibilita a configuração de métricas de forma colaborativa e aberta, como será explicado na Seção 5.4.

Na primeira parte dessa seção, apresentaremos em detalhes a Analizo, que proporcionou a coleta dos dados de todos dos estudos apresentados no Capítulo 4. Posteriormente, detalheremos as funcionalidades do projeto Mezuro (i.e., Kalibro web service e Mezuro plugin).

5.2 Analizo

A Analizo é um *toolkit* livre, multi-linguagem e extensível para análise e visualização de código-fonte. Ele nasceu da necessidade de analisarmos e visualizarmos projetos de software livre grandes ou de um grande número de projetos individualmente. Nosso grupo de pesquisa trabalha com estudos empíricos que exigem análise de código-fonte em grande escala e, consequentemente, nós recorremos as ferramentas de análise de código-fonte, para realizarmos as nossas pesquisas.

Decidimos concentrar nosso trabalho em torno da Analizo por ela ser mantida constantemente, possuir desenvolvedores ativos que conhecem a arquitetura da ferramenta e tem a capacidade de fornecer atualizações e corrigir problemas. Além disso, a Analizo é capaz de manipular o código-fonte que não pode ser compilado. Por exemplo, o código pode ter erros de sintaxe, as bibliotecas usadas podem não estar mais disponíveis, ou ainda a API das bibliotecas utilizadas podem ter mudado. Isso é importante para sermos capazes de analisar o código-fonte legado em estudos de evolução do software.

A Analizo trabalha com estratégias de extração das informações do código-fonte definidas pelos seus extratores, que obtêm as informações de código-fonte e as armazena. Atualmente, existem dois extratores. Ambos são interfaces para ferramentas externas de análise de código-fonte:

- *Analizo::Extractors::Doxyparse* é uma interface para o Doxyparse uma analisador de código-fonte para C, C++ e Java desenvolvido e mantido pelo grupo de colaboradores da Analizo (Costa, 2009). Doxyparse é baseado no Doxygen[4], uma ferramenta livre e multi-linguagem para a documentação de código-fonte. Ela contém um analisador que atendeu nossas demandas neste trabalho.

- *Analizo::Extractors::Sloccount* é uma interface para o Sloccount[5], uma ferramenta que calcula o número de linhas de código efetivas, ou seja, sem comentários e linhas em branco.

Além da parte de coleta e análise de métricas estáticas de código-fonte, a Analizo possui outras ferramentas. As interações entre elas foram projetadas para manter a fluxo de trabalho do UNIX: elas realizam tarefas especializadas e geram saídas que são entradas para outras ferramentas. Algumas das ferramentas foram implementadas explicitamente para manipular outras estruturas internas da Analizo. Algumas foram projetados para fornecer uma saída para aplicações externas, tais como programas de desenho gráfico, análise de dados ou aplicações de visualização, como é o caso do Kalibro e do Mezuro.

Atualmente, a Analizo realiza a análise de código-fonte escrito em C, C++ e Java. Ela calcula métricas tanto em nível de projeto, que são calculadas a partir de um valor que agrega dados de todos os arquivos do projeto, quanto métricas no nível de classe, que são calculados individualmente para cada classe. No nível do projeto, a Analizo também fornece valores estatísticos (soma, média, mediana, moda, desvio padrão, variância, distribuição de Skewness e Kurtosis, mínimo, máximo e percentis) para cada uma das métricas no nível de classe. As métricas calculadas no momento pela

[4] doxygen.org/
[5] dwheeler.com/sloccount/

Analizo foram selecionadas de acordo com o descrito na Seção 3.1.2. Em particular, foram inseridas métricas selecionadas no contexto do projeto Qualipso para atender as necessidades desse projeto na avaliação de software escrito em C e C++:

- Métricas no nível de projeto:
  - *Total de classes abstratas*
  - *Total do fator de acoplamento*
  - *Total do número de linhas de código efetivas*
  - *Total de número de linhas de código dos métodos*
  - *Total do número de métodos por classe abstrata*
  - *Total do número de classes (ou módulos)*
  - *Total do número de classes (ou módulos) com pelos menos um atributo definido*
  - *Total do número de classes (ou módulos) com pelo menso um método definido*
  - *Total do número de classes (ou módulos)*
- Métricas no nível de classe:
  - *Conexões aferentes de uma classe*
  - *Média da Complexidade Ciclomática por método*
  - *Média do número de linhas de código por método*
  - *Média do Número de Parâmetros por Método*
  - *Acoplamento entre objetos*
  - *Profundidade da árvore de herança*
  - *Ausência de coesão em métodos*
  - *Número de linhas de código*
  - *Número de linhas de código do maior método da classe*
  - *Número de atributos*
  - *Número de filhos*
  - *Número de métodos*
  - *Número de atributos públicos*
  - *Número de métodos públicos*
  - *Respostas para uma classe*
  - *Complexidade estrutural*

O módulo de cálculo de métricas da Analizo possui um comando para o processamento em lote de métricas. Na maioria dos estudos quantitativos de engenharia de software que envolvem métricas de código-fonte de uma grande quantidade de projetos, processar cada projeto, um a um, é impraticável, propenso a erros e difícil de repetir. A Analizo pode processar vários projetos em lote e produzir um arquivo CSV (valores separados por vírgulas) com os dados das métricas para cada projeto, bem como um resumo dos dados em um arquivo CSV com os valores das métricas de nível de projeto para todos os projetos. Esses arquivos podem ser facilmente importados em ferramentas de análise estatística ou em planilhas. Também, podem ser utilizados para analisar diversas versões do mesmo projeto, em estudos de evolução do software. Com esse recurso é que foi possível avaliar a quantidade de projetos analisados nos estudos apresentados no Capítulo 4.

Por fim, em alguns estudos, precisamos processar o histórico dos projetos de software em uma escala mais fina, por exemplo por *commits*. Analizo pode processar um repositório com controle de

versão e fornecer os dados em um arquivo CSV com as métricas de valores para cada revisão em que o código-fonte foi alterado. Os repositórios com as quais a Analizo trabalha atualmente são *Git* e *Subversion*.

Além de usado na obtenção dos valores das métricas de nossos estudos, a Analizo é o principal coletor (ferramenta base) de métricas integrado ao projeto Mezuro por meio do *web service* Kalibro Metrics.

5.3 Kalibro Metrics

O Kalibro Metrics é um serviço web livre, projetado para se conectar a qualquer ferramenta de coleta de métricas de código-fonte. Seu objetivo é fornecer uma avaliação de fácil entendimento da qualidade ("saúde") do software analisado.

O Kalibro Metrics permite que um especialista em métricas (ou qualquer usuário) especifique conjuntos de intervalos de valores para cada métrica fornecida pela ferramenta base, além de possibilitar a criação de novas métricas. Com essa abordagem, pretendemos difundir o uso de métricas de código-fonte, pois com um conjunto de intervalos (ou seja, os valores frequentes) e suas respectivas interpretações, qualquer desenvolvedor pode explorá-las e melhor entendê-las.

Argumentamos que é desejável um software que colete métricas e associe seus resultados a alguma interpretação, tornando-as mais fáceis de entender. Ao constatar dificuldades nas ferramentas de métricas, estabelecemos alguns requisitos para um software que vise a difundir o uso de métricas de código-fonte:

1. **Intervalos de aceitação:** Deve permitir o uso de múltiplos intervalos para fornecer diferentes interpretações sobre os valores das métricas. Por exemplo, usando o Eclipse Metrics (Sauer, 2005) é possível configurar valores mínimo e máximo (um intervalo de aceitação) para cada métrica que ele fornece, e associar uma dica de correção quando o resultado para um método, classe ou pacote está fora desse intervalo. Os intervalos devem ser configuráveis, pois as métricas em geral não possuem valores de referência absolutos ou universalmente aceitos, como discutimos nesta tese. Os valores ideais podem variar de acordo com vários fatores, como a linguagem e o domínio de aplicação.

2. **Extensível:** Não deve se limitar a apenas uma linguagem de programação, nem a um conjunto pré-definido de métricas. Deve fornecer interface clara para adicionar suporte a novas linguagens de programação, pois isso pode atrair uma gama maior de usuários em potencial para sugerir intervalos de acordo com suas experiências em linguagens específicas. Além disso, o usuário deve ser capaz de configurar o conjunto de métricas que deseja analisar. Muitos coletores de métricas calculam e exibem o resultado de todas as métricas de seu repertório, o que consome mais recursos que o necessário (i.e., tempo e memória) e polui a exibição. Deve haver suporte para inclusão de novas métricas, a partir de diferentes coletores de métricas ou a partir das métricas existentes. O índice de manutenibilidade (Vandoren, 1997) é um exemplo de métrica criada como combinação de métricas básicas. Outro exemplo é a complexidade estrutural, explicada na Seção 3.1.2, que é o produto de uma métrica de coesão com uma de acoplamento.

3. **Comparação entre projetos:** A ferramenta deve dar suporte a comparação de características específicas do código-fonte de vários projetos através de métricas, dando suporte aos pesquisadores e ajudando nas decisões de adoção de um determinado software.

4. **Software livre mantido ativamente:** A ferramenta deve ser software livre, disponível sem restrições e mantida ativamente. Assim, qualquer um pode contribuir com seu desenvolvimento e modificá-la de acordo com suas necessidades. Isso também permite que pesquisadores repliquem completamente estudos e resultados – algo que está na concepção das publicações científicas: serem reprodutíveis.

Ferramenta	Linguagens	**Intervalos**	**Extensível**	**Comparação**	**Livre**
Analizo	C, C++, Java	Não	Sim	Não	Sim
Analyst4j	Java	Sim	Não	Não	Não
CCCC	C++, Java	Não	Não	Não	Sim
Checkstyle	Java	Sim	Não	Não	Sim
CK Java Metrics	Java	Não	Não	Não	Sim
CMetrics	C	Não	Sim	Não	Sim
Cscope	C	Não	Não	Não	Sim
CVSAnaly	C, C++, Python	Não	Sim	Não	Sim
Dependency Finder	Java	Não	Não	Não	Sim
Infusion	C, C++, Java	Sim	Não	Sim	Não
JaBUTi	Java	Não	Não	Não	Sim
MacXim	Java	Não	Não	Não	Sim
Metrics	Java	Sim	Não	Não	Sim
OOMeter	Java,C#	Não	Não	Não	Não
Understand for Java	Java	Não	Não	Não	Não
VizzAnalyzer	Java	Não	Não	Não	Não

Tabela 5.1: *Ferramentas existentes versus requisitos definidos*

Ao longo deste trabalho, conhecemos, estudamos e/ou utilizamos 16 ferramentas[6]para análise de código-fonte. Na Tabela 5.1, as ferramentas estudadas são comparadas com os requisitos. Percebe-se que nenhuma delas preenche todos os requisitos. Dessa forma, defendemos que para promover o uso de métricas de código-fonte por desenvolvedores em geral, é necessária uma ferramenta que sistematize a avaliação do código-fonte. Adicionalmente, com exceção do software restrito Infusion, que conhecemos quando já estávamos em um estágio avançado de desenvolvimento de nossa pesquisa, os coletores de métricas de código-fonte que analisamos possuem, em geral, uma ou mais dessas carências:

1. **Associação entre resultados numéricos e forma de interpretá-los:** Ferramentas de métricas frequentemente mostram seus resultados como valores numéricos isolados para cada métrica.

2. **Flexibilidade nessa interpretação:** Algumas ferramentas mostram indicadores binários (bom ou ruim) para os valores das métricas, mas não há forma de configurar os valores de referência.

3. **Flexibilidade para funcionar com diferentes linguagens:** As ferramentas que não pecam nos quesitos anteriores foram desenhadas especificamente para uma só linguagem de programação.

4. **Criação de novas métricas**: capacidade de estender seu uso com novas métricas, derivadas das existentes na ferramenta.

Baseados nesses conceitos desenvolvemos o que é hoje o Kalibro Metrics: um serviço responsável pelo suporte à interpretação das métricas. Do ponto de vista de implementação, foi definida uma interface de conexão simples para delegar a coleta de valores para ferramentas de análise de código-fonte existentes. Dessa forma, o Kalibro pode se conectar às ferramentas mais interessantes para o contexto de cada software analisado.

[6]Ferramentas estudadas: analizo.org, codeswat.com, cccc.sourceforge.net, checkstyle.sourceforge.net, http://metricsgrimoire.github.io/CVSAnalY/, spinellis.gr/sw/ckjm, tools.libresoft.es/cmetrics, cscope.sourceforge.net, depfind.sourceforge.net, intooitus.com/products/infusion, ccsl.icmc.usp.br/pt-br/projects/jabuti, qualipso.dscpi.uninsubria.it/macxim, metrics.sourceforge.net, ccse.kfupm.edu.sa/~oometer/oometer, scitools.com, arisa.se.

Como primeira ferramenta base, foi escolhida a Analizo, pelas características comentadas na Seção 5.2. Posteriormente, o CheckStyle foi integrado para termos uma ferramenta especialista na análise de código escrito em Java no projeto Mezuro, fornecendo métricas mais específica do paradigma de orientação a objetos para código Java.

O Kalibro foi desenvolvido de forma modular e desacoplada, para facilitar modificações, incrementos e incorporações de outras ferramentas e diferentes interfaces de usuário (camadas de visualização). Toda a lógica, processamento e persistência foram isolados em um núcleo independente de interface com o usuário, que chamamos de *KalibroCore*, e o acesso remoto está implementado na interface do serviço, *KalibroService*. Os detalhes da arquitetura da Kalibro Metrics e uma discussão das funcionalidades do projeto Mezuro do ponto de vista da arquitetura do Kalibro estão no Apêndice B.

Por fim, depois do primeiro protótipo do Mezuro, o desenvolvimento do Kalibro Metrics foi associado ao desenvolvimento do plugin Mezuro. Na prática, o plugin Mezuro funciona como uma camada de visualização dos resultados e funcionalidades do Kalibro Metrics.

5.4 Mezuro

Desde a proposta inicial deste doutorado, pensamos em uma ferramenta com a capacidade de reunir periodicamente os dados de projetos de software livre, visitando seu código-fonte diretamente a partir de seus respectivos repositórios, obtendo vários tipos de dados e gerando avaliações sobre a qualidade do código de forma semi-automática. O objetivo era o de tornar essas avaliações públicas de forma a ajudar empresas, governos e indivíduos a decidir sobre a utilização ou não de um software livre e a auxiliar na escolha de uma das soluções dentre as várias possíveis, do ponto vista técnico, pensando em sua manutenabilidade, além de promover a melhoria da qualidade do software livre produzido por desenvolvedores e fomentar a confiança nos produtos.

A ideia central dessa ferramenta passa pelo conceito de configuração de métricas. Uma configuração é um conjunto de métricas em que cada métrica está associada a um grupo de intervalos de valores possíveis para essa métrica e às interpretações sobre o significado desses intervalos. Essa configuração é criada e personalizada pelo usuário (um desenvolvedor ou equipe que irá monitorar um determinado código). Pensamos primeiramente um intervalo como algo que, além de possuir um valor de início e fim, tem uma avaliação (na forma de uma nota) associada, uma cor para exibição e comentários pertinentes a exibir quando o valor da métrica estiver dentro dele. Isso possibilita associar uma avaliação qualitativa ao valor obtido como resultado do cálculo da métrica.

Uma configuração de métricas específica é associada a um dado projeto de software. Quando seu código é analisado, apenas os resultados das métricas contidas na configuração associada a ele são calculados, com seus valores associados aos intervalos correspondentes. Assim, o usuário pode criar várias configurações e utilizá-las para atender às suas necessidades específicas quanto ao código do projeto.

Cada métrica de uma configuração possui um peso. Com a nota dos intervalos e o peso de cada métrica, se calcula uma nota geral do projeto, pacote, classe ou método, com uma média ponderada. Desse modo, além de oferecer uma forma de interpretar resultados de métricas, as configurações também servem como critério específico de avaliação e comparação entre projetos de software.

Em suma, a possibilidade de definir configurações de métricas é a parte mais importante da flexibilidade da plataforma que propusemos e desenvolvemos. A configuração utilizada por um líder de projeto para acompanhar a qualidade do código pode sofrer melhorias ao longo do tempo e, nesse sentido, esse ambiente de monitoramento semi-automático proposto também foi pensado para compartilhar essas configurações, facilitando uma discussão em torno delas em rede e sua subsequente evolução.

Apostando nessas ideias é que investimos boa parte dos esforço deste doutorado no desenvolvimento e na coordenação de um projeto de uma rede sócio-técnica que tem o objetivo de ser um ambiente aberto e colaborativo de avaliações de código-fonte e aprendizado do "estado-da-prática" dos projetos de software livre. Essa rede é o Mezuro, construído sobre a plataforma de redes soci-

ais Noosfero[7], através de um plugin que se comunica com o Kalibro Metrics Service. O Mezuro é norteado pela extração automatizada de métricas de código-fonte e por uma maneira objetiva de interpretar os seus valores, para que os engenheiros de software possam monitorar as características específicas de seu código.

Antes de partirmos para o desenvolvimento do Mezuro, estudamos algumas plataformas correlatas[8]:

- **FLOSSMetrics** (**Free/Libre Open Source Software Metrics**) é um projeto que utiliza metodologias e ferramentas existentes para fornecer um grande banco de dados com informações sobre o desenvolvimento de software livre.
- **Ohloh** é uma plataforma que oferece um conjunto de serviços web e um sistema web para comunidade de software livre, que visa prover uma visão geral da evolução dos projetos de software livre em desenvolvimento.
- **Qualoss** (**Quality in Open Source Software**) é uma metodologia para automatizar a medição da qualidade de projetos de software livre, usando ferramentas para analisar o código-fonte e as informações dos repositórios dos projetos.
- **SQO-OSS** (**Software Quality Assessment of Open Source Software**) fornece um conjunto de ferramentas para análise e avaliação comparativa de projetos de software livre.
- **QSOS** (Qualification and Selection of Open Source Software) é uma metodologia baseada em quatro etapas: definição de referência utilizada; avaliação de software; qualificação dos usuários em contexto específico; seleção e comparação de software.
- **FOSSology** (Advancing open source analysis and development) é um projeto que fornece um banco de dados gratuito com informações sobre licenças de software livre.
- **HackyStat** é um ambiente para visualização, análise e interpretação do processo de desenvolvimento de software e dados do produto de software.

Como já mencionado, nossas ferramentas também foram desenvolvidas no contexto do projeto QualiPSo (*Trust and Quality in Open Source Systems*) [9], que nos ajudou com os estudos para estimular o uso do software livre e adoção das suas práticas de desenvolvimento dentro da indústria de software. Com nossa experiência de pesquisa no projeto QualiPSo e a análise dos outros projetos relacionados citados acima, desenvolvemos a plataforma Mezuro.

Nenhuma das plataformas citadas, quando as avaliamos na concepção do projeto Mezuro, já com o desenvolvimento do primeiro protótipo da parte de configuração de métricas na ferramenta Crab (Meirelles *et al.*, 2009), tem o objetivo de ficar monitorando constantemente a evolução das métricas de código-fonte em si dos projetos de software livre. Como apresentamos, o Mezuro colocar, também, nas mãos dos usuários o poder de definir quais métricas quer monitorar, quando quer monitorar e até quais as interpretações ele quer se basear para sua avaliação sobre as métricas. De qualquer forma, o Mezuro não propõe em ser um "concorrente", mas sim complementar as abordagens das demais plataformas. Por exemplo, com a experiência da nossa interação com o Olhoh, que apenas tem apenas a métrica de número de linhas de código como informação da análise estática do código-fonte, avaliamos ser factível uma integração com essa plataforma, uma vez que podemos integrar a comunicação da API deles com o nosso *web service* para obtermos dados "sociais" das comunidade de software livre; bem como, eles podem usar o Kalibro Metrics (nossa instância ou uma própria deles, uma vez que todas as nossas ferramentas são livres), passando a URL do repositório do projetos em análise, para obter os valores das métricas de código-fonte, e assim relatar mais informações técnicas à respeito dos projetos avaliados no Ohloh.

[7] noosfero.org

[8] Projetos correlatos: flossmetrics.org, ohloh.net, qualoss.org, sqo-oss.org, qsos.org, fossology.org, code.google.com/p/hackystat.

[9] Projeto Qualipso: qualipso.org

5.4.1 Funcionalidade

Uma analogia útil para se ententer a utilidade do Mezuro é lembrar que as pessoas melhoram suas habilidades de escrita à medida em que leem bons livros. Da mesma forma, os engenheiros de software podem aprender e melhorar seus códigos ao encontrar projetos relacionados que possuem bons códigos (ou seja, lendo bons códigos), comparando suas implementações e características. Para que isso seja possível do ponto de vista do monitoramento do código-fonte de projetos de software livre, as funcionalidades implementadas até aqui no projeto Mezuro são:

- Download de código-fonte a partir de repositórios do tipo Subversion, Git, Mercurial, Baazar e CVS.
- Criação de configurações, ou seja, conjuntos pré-definidos de métricas relacionadas para serem usadas na avaliação do código-fonte dos projetos.
- Criação de intervalos qualitativos associados aos valores das métricas.
- Criação de novas métricas (via JavaScript) baseadas em outras fornecidas pelas ferramentas (coletores de métricas) integradas à plataforma.
- Cálculo dos resultados estatísticos para partes do software com granularidade alta (e.g. a média do número de linhas de código das classes por pacote).
- Cálculo de uma nota para o código-fonte analisado, baseado em pesos atribuídos para as métricas e seus intervalos associados, o que permite a comparação entre projetos analisados sob a mesma configuração.
- Interpretação amigável e textual dos valores das métricas com a definição de grupos de leitura e interpretação (*reading groups*) com cores e comentários para cada intervalo definido.
- Visualização do histórico de cada métrica nos projetos avaliados.

Figura 5.2: *Painel de controle de uma comunidade na rede Mezuro.*

O Mezuro tem dois grupos de funcionalidades: em nível de projeto e em nível de configuração (que também contém os grupos de leitura e interpretação dos intervalos para as métricas). O

Figura 5.3: *Cadastrando um novo projeto no Mezuro.*

projeto, ou seja, um tipo de conteúdo na rede que refere-se a um software que terá seu código-fonte monitorado, pertence a um tipo de perfil da rede chamado de "comunidade". Dessa forma, ao se cadastrar na rede Mezuro para monitorar as métricas de código-fonte de um software livre, o usuário deve criar uma comunidade para o projeto que deseja monitorar (ou ainda, procurar uma comunidade correspondente ao software livre de interesse). No painel de controle de uma comunidade está disponível o tipo de conteúdo chamado "Mezuro Project", como ilustrado na Figura 5.2.

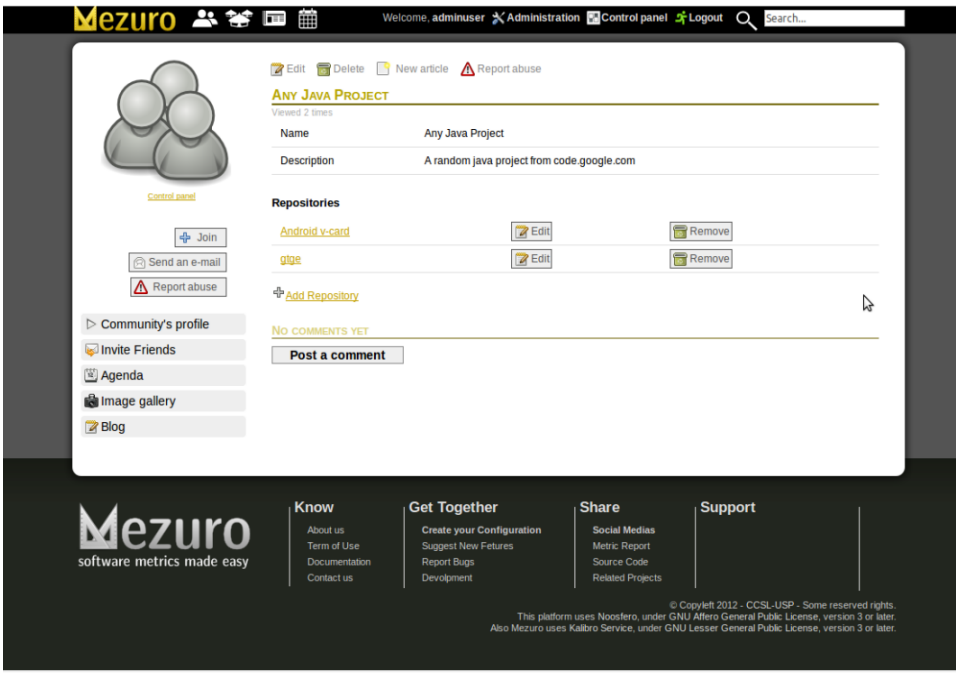


Figura 5.4: *Adicionando um repositório ao projeto no Mezuro.*

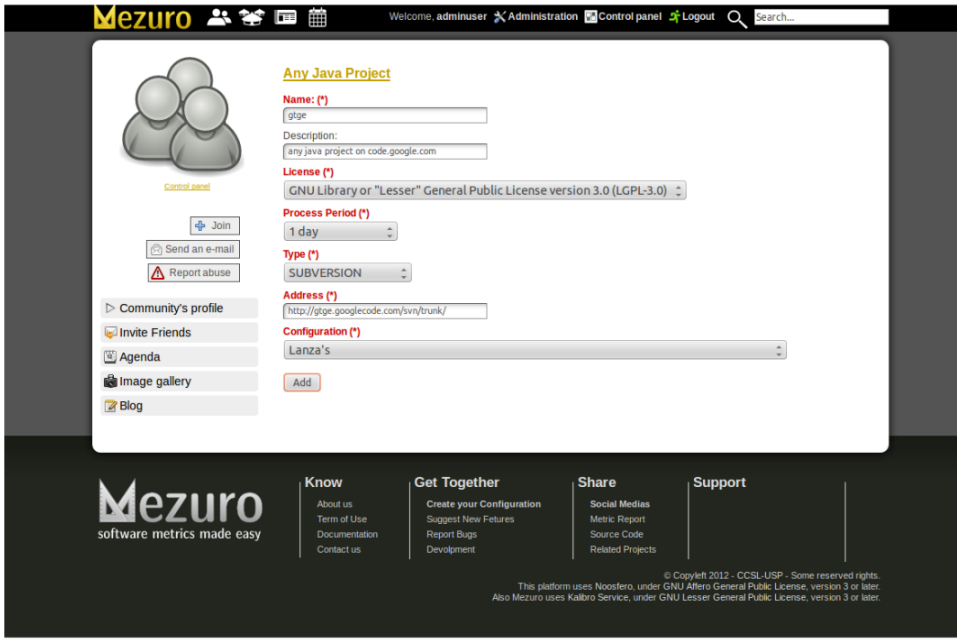


Figura 5.5: *Cadastrando as informações do repositório no projeto do Mezuro.*

Para criar um Projeto Mezuro, basta ao usuário informar um nome e uma descrição para o software, como exemplificado na Figura 5.3. Posteriormente, deverá adicionar pelo menos um repositório de código-fonte correspondente ao projeto, como mostrado na sequência das Figuras 5.4 e 5.5. Vale observar que não é excepcional que projetos de software livre tenham mais de um repositório, muitas vezes correspondentes aos seus subprojetos.

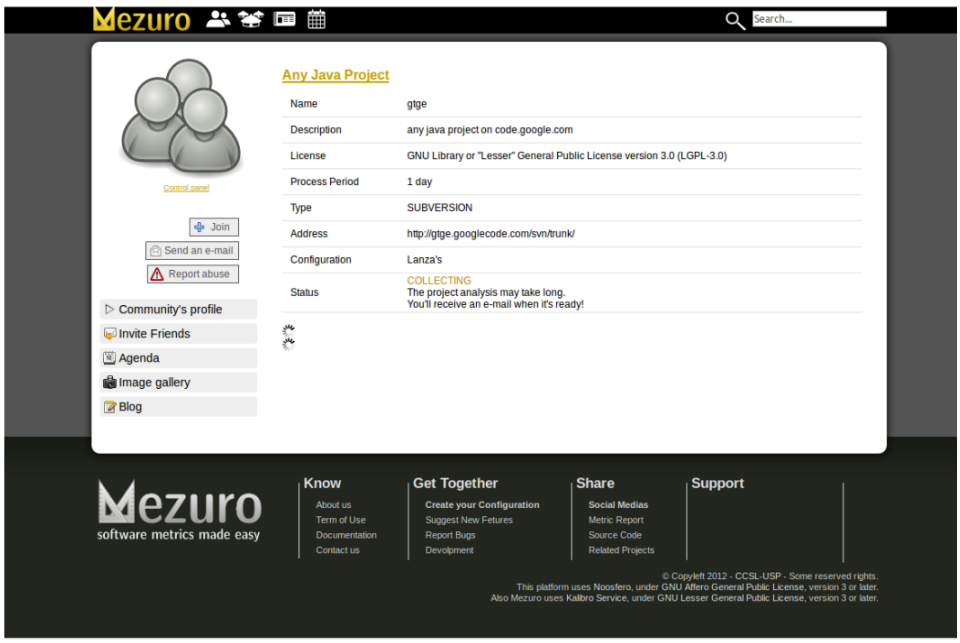


Figura 5.6: *Visualizando a coleta das métricas no Mezuro.*

São cadastrados também (i) a licença do código disponibilizado nos repositórios informados, (ii) a periodicidade (em dias) em que as métricas serão coletadas, (iii) o tipo de repositório, (iv) a

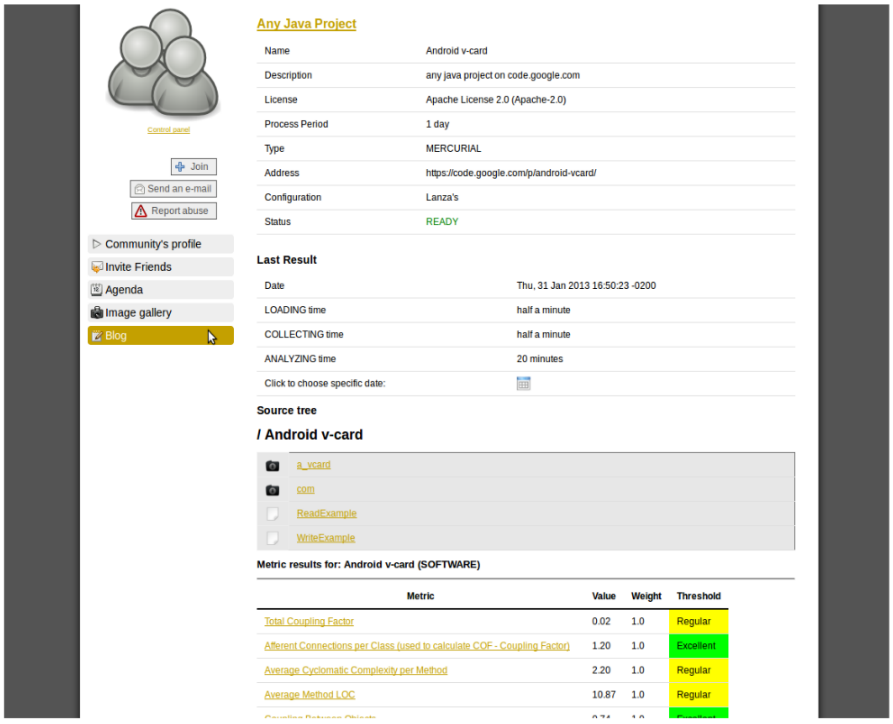


Figura 5.7: *Visualizando os dados do processo de coleta das métricas*

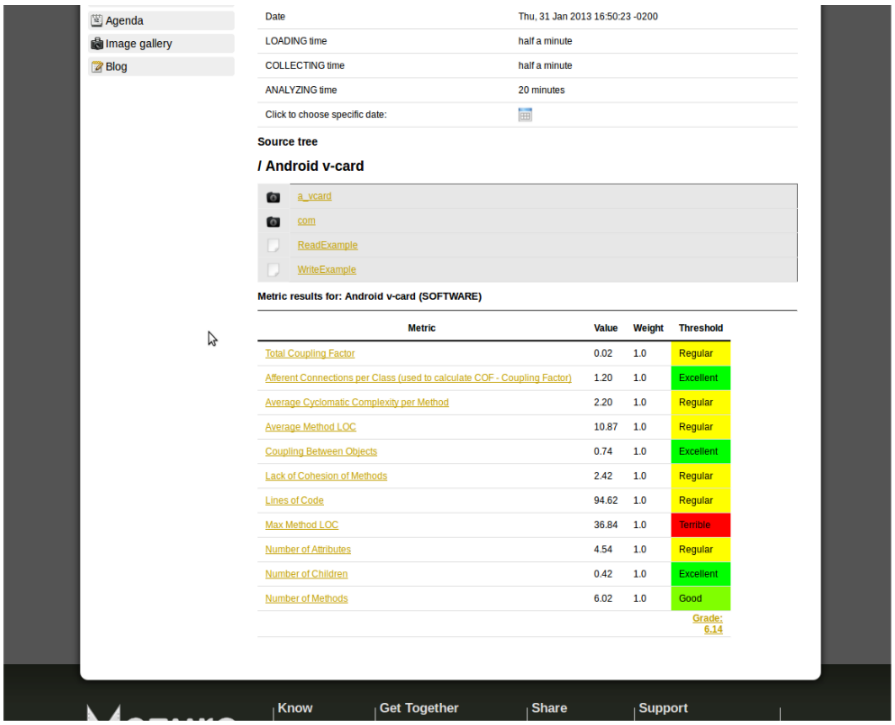
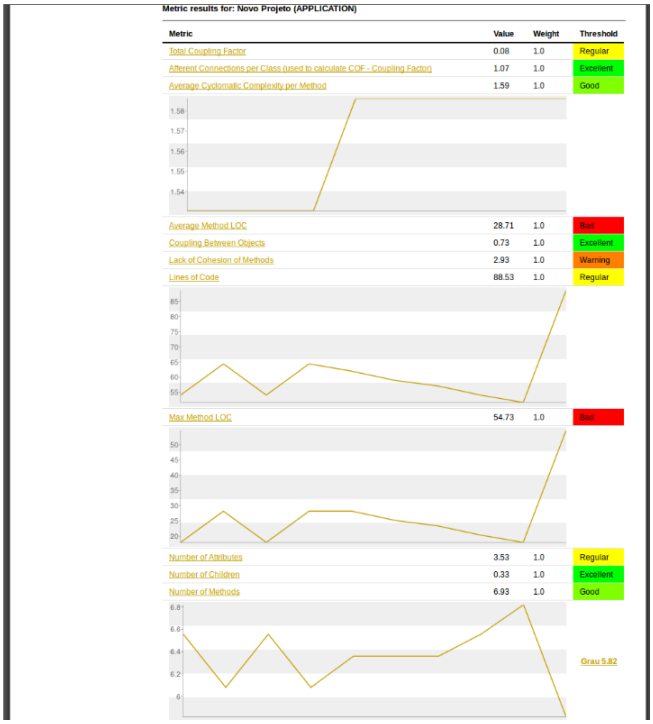


Figura 5.8: *Visualizando os resultados de cada uma das métricas.*

URL do repositório em questão e (v) a configuração de avaliação das métricas. Essas informações são salvas na rede e enviadas ao Kalibro Metrics. O Kalibro posiciona a coleta das métricas dos repositórios cadastrados em uma fila para ser processados; esse processamento pode levar horas dependendo da quantidade classes e módulos do software no repositório. Após a coleta das métricas e processamento de acordo com a configuração selecionada, os resultados podem ser visualizados como exemplicado na sequência das Figuras 5.6, 5.7 e 5.8.



41

Figura 5.9: *Visualizando o gráfico com o histórico das métricas.*

Com as respostas enviadas do Kalibro para o Mezuro plugin, é possível navegar entre os diretórios (pacotes) e arquivos (classes e módulos) do repositório analisado para observar individualmente os valores das métricas para cada parte do software em questão. Também, para cada métrica é possível observar a evolução dos seu valores ao longo do tempo (conforme a quantidade de vezes que um projeto foi analisado, de acordo com a periodicidade cadastrada) através de gráficos, como exemplificado na Figura 5.9. Destacamos que aspectos de visualização de software não fazem parte do escopo desta tese de doutorado, mas ao longo do desenvolvimento do projeto Mezuro identificamos a necessidade de aplicá-los. Assim, o uso de técnicas e abordagens de visualização de software estão sendo estudados para serem aplicados na continuidade deste projeto.

O outro tipo de conteúdo do Mezuro Plugin é o "Mezuro Configuration" (que tem agregado o "Mezuro Reading Group"). Diferentemente de um "Projeto Mezuro", uma "configuração do Mezuro" foi pensada para ser utilizada por especialistas em métricas de código-fonte que desejam cadastrar valores de referência ou frequentes para as métricas, de acordo com suas experiências e necessidades. Nesse ponto é que está a utilização prática e automatizada das análises dos valores frequentes dos projetos discutidos no Capítulo 4. Os valores frequentes de cada métrica em cada projeto, dentro das faixas de percentis encontrados, serão cadastrados no Mezuro para disponibilizarmos um conjunto inicial de configurações de métricas na rede de monitoramento de código-fonte.

Esse tipo de conteúdo é disponibilizado no perfil pessoal de cada usuário da rede Mezuro, como ilustrado na Figura 5.10. Deseja-se que as configurações sejam associadas às pessoas para que os demais usuários conheçam quem definiu cada configuração. Certamente, as configurações disponibilizadas por pessoas reconhecidas dentro de uma comunidade de software livre tenderão a ser mais usadas no monitoramento das métricas dos projetos associados a essa comunidade.

Para cadastrarmos uma nova configuração do Mezuro, informamos um nome e uma descrição para identificá-la na rede, como exemplificado na Figura 5.11. O usuário ainda tem a opção de clonar uma configuração já existente na rede para fazer modificações pontuais, mas sem a necessidade de criar uma configuração do zero.

Os próximos passos, como mostrado na sequência das Figuras 5.12, 5.13, 5.14 e 5.15, consistem em, dentre as ferramentas base, no caso Analizo e CheckStyle, selecionar uma métrica e configurá-la (e assim por diante para cada métrica que o usuário desejar configurar). O Kalibro envia para o

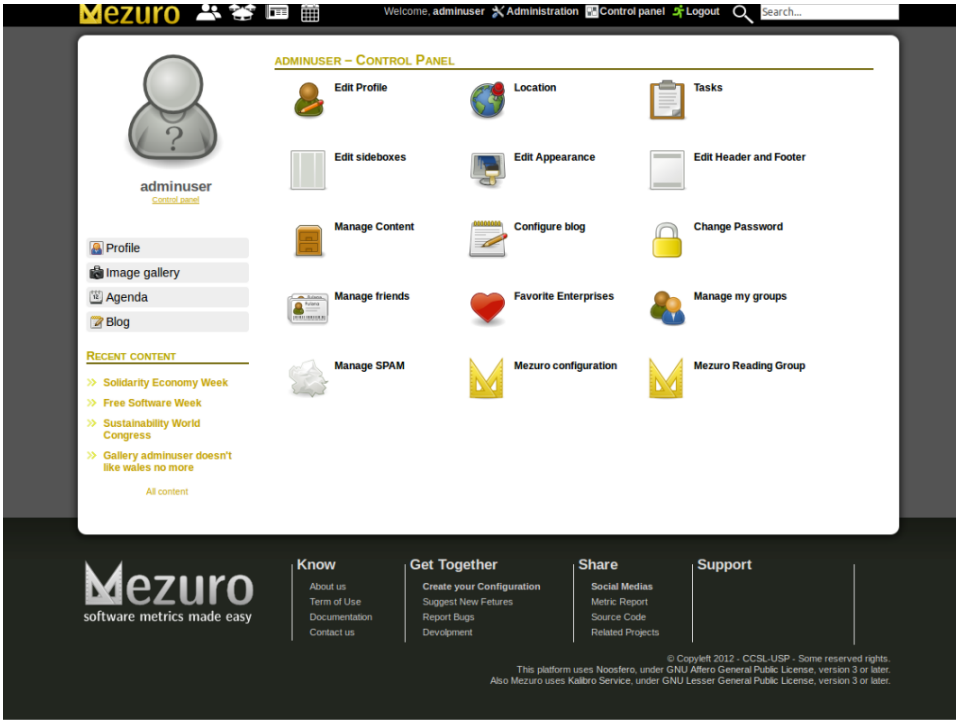


Figura 5.10: *Painel de controle de um usuário na rede Mezuro.*

Figura 5.11: *Criando uma nova configuração do Mezuro.*

Mezuro plugin a lista de métricas que são calculadas em cada ferramenta base. Ao selecionar uma das métricas, o usuário deve definir a configuração da métrica selecionada.

Para cada métrica que deseja adicionar em sua configuração do Mezuro, o usuário especialista em métricas deve informar, como exemplificado na Figura 5.16, (i) um código para a métrica, que é usado como identificador dessa métrica caso ela seja utilizada no cálculo de uma métrica composta; (ii) uma forma de agregação (média, máximo, mínimo etc.) quando tal métrica for contabilizada em uma parte de um software com maior granularidade; (iii) um grupo de leitura e interpretação

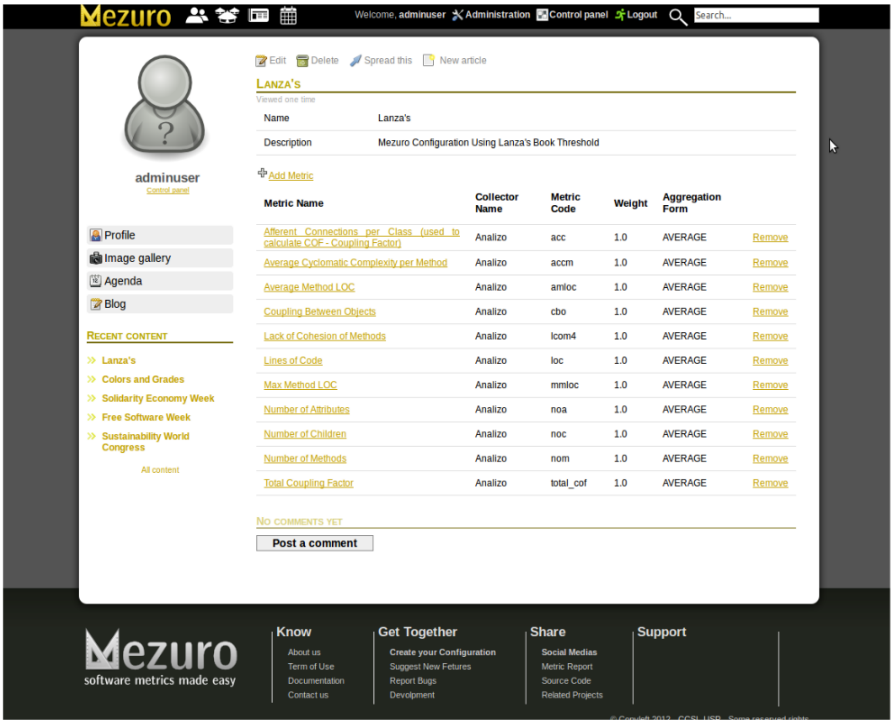


Figura 5.12: *Adicionando métricas à configuração do Mezuro.*

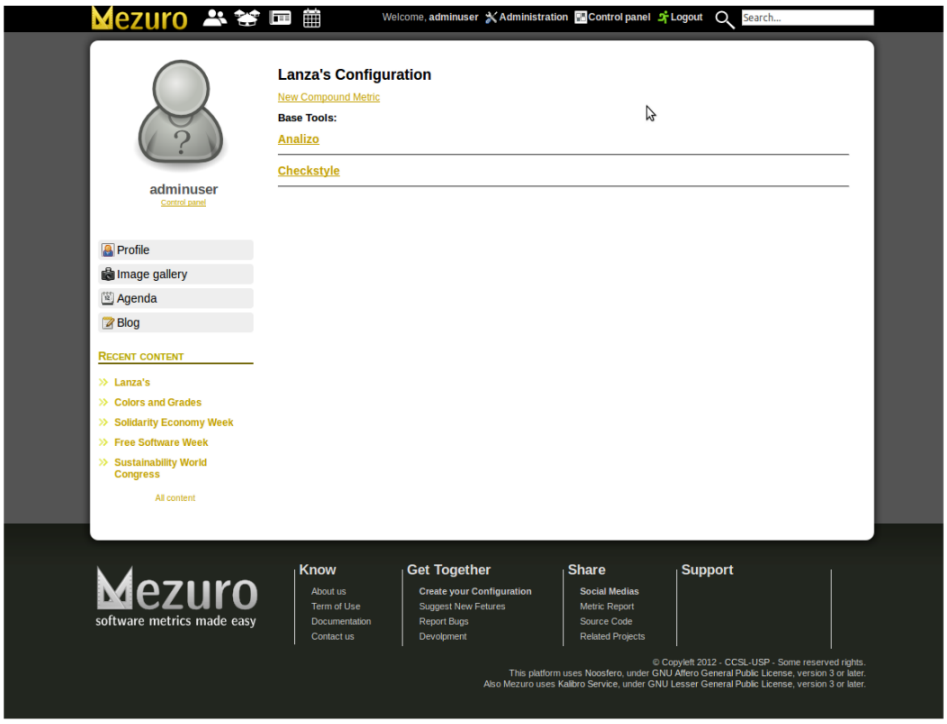


Figura 5.13: *Selecionando uma ferramenta base integrada ao Mezuro.*

de intervalos da métricas (vamos explicar mais adiante essa funcionalidade). Após salvar essas informações, o Mezuro permite que intervalos de valores e a interpretação para cada um desses intervalos sejam adicionados à configuração da métrica em questão, como ilustrado na sequência das Figuras 5.17 e 5.18. Para cada intervalo, o usuário deve escolher um rótulo, que vem pré-definido de acordo com o grupo de leitura e interpretação selecionado. Posteriormente, ele deve atribuir um valor inicial e um final para o intervalo. Por fim, deve inserir um comentário referente à interpretação e possíveis ações que podem ser sugeridas quando o valor da métrica estiver no referido intervalo.

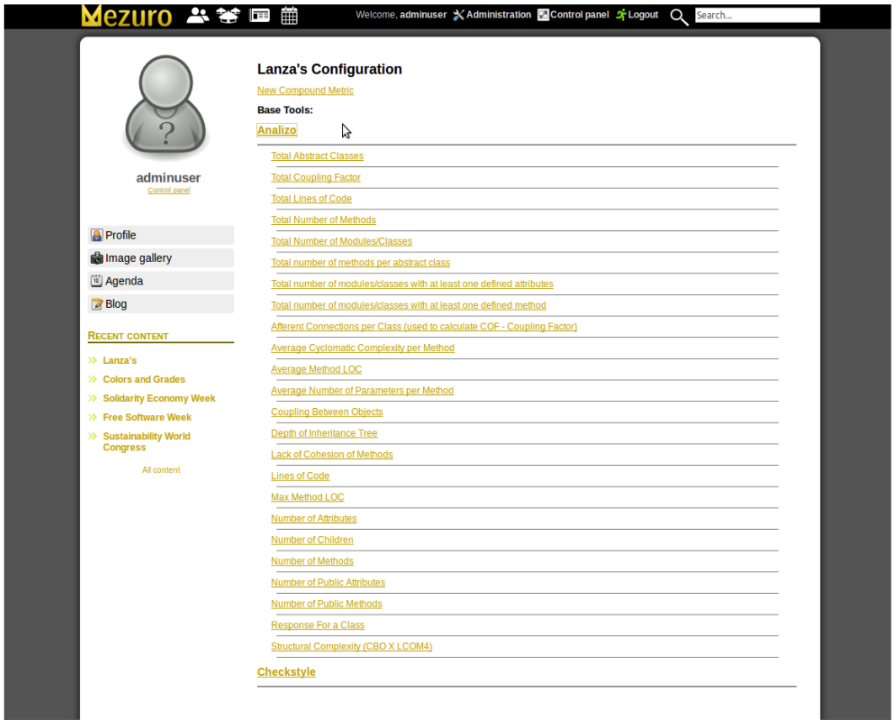


Figura 5.14: *Selecionando métricas da Analizo numa configuração do Mezuro.*

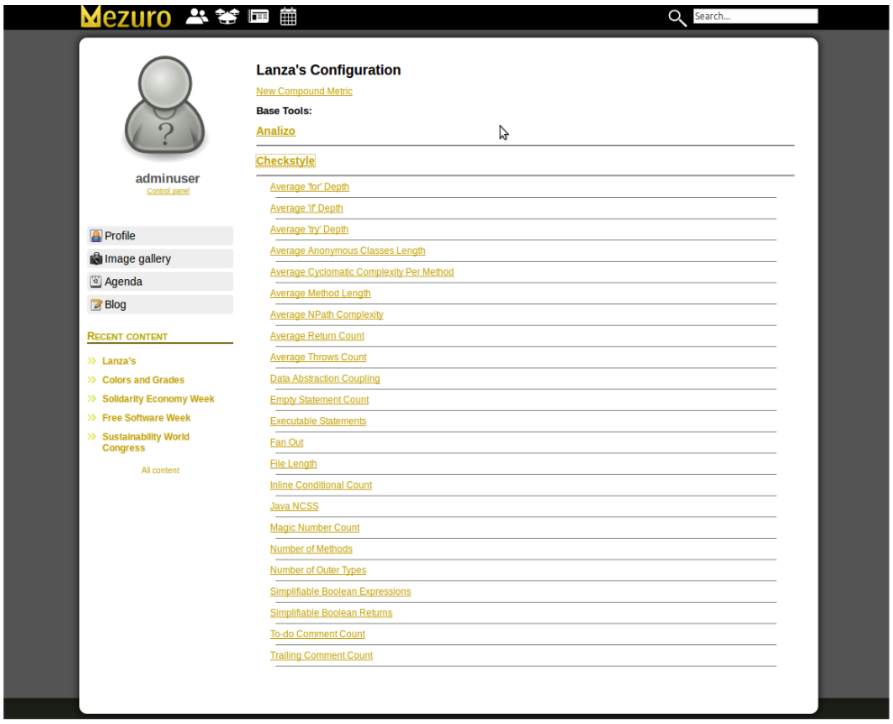


Figura 5.15: *Selecionando métricas do Checkstyle numa configuração do Mezuro.*

Para entendermos melhor a configuração das métricas, podemos fazer uma comparação com um paciente que faz um exame de sangue: ele precisa levar os resultados ao seu médico, que vai usar seus conhecimentos para fazer uma leitura, indicando quais índices estão saudáveis e quais são preocupantes, fazendo recomendações ou prescrevendo algum tratamento. Da mesma forma, resultados de métricas não são fáceis de interpretar por um não especialista sem que sejam associados a uma interpretação que lhes dê significado. No Mezuro, essa interpretação é feita através de intervalos e de leituras desses intervalos.

Figura 5.16: *Configurando uma métrica.*

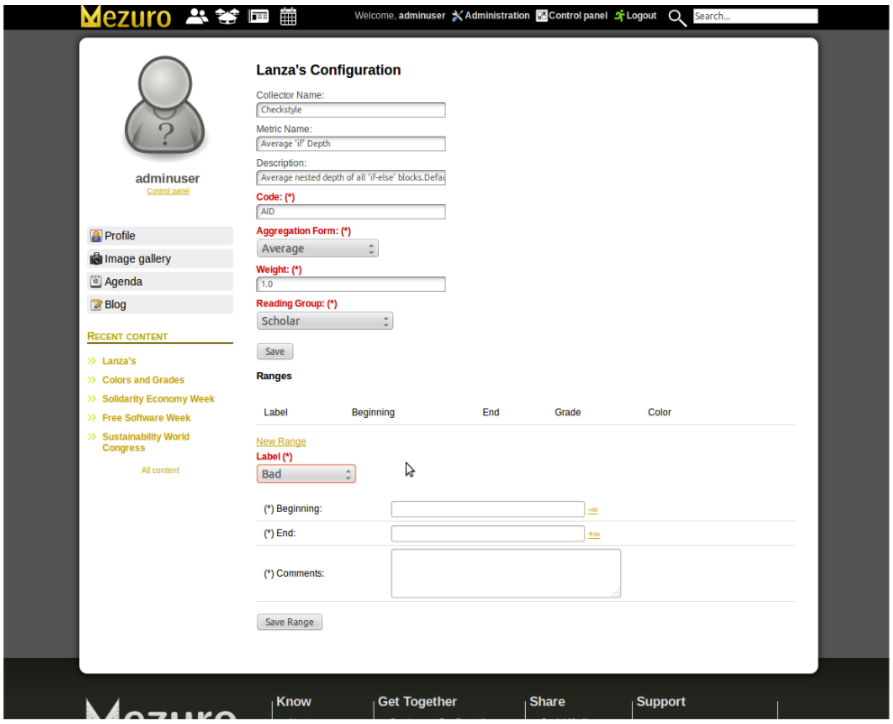


Figura 5.17: *Definindo um intervalo para uma métrica.*

Como ilustrado na sequência das Figuras 5.19 e 5.20, podemos cadastrar tais leituras para facilitar o seu reúso: (i) O rótulo é uma forma verbal concisa de chamar atenção para o conteúdo de uma interpretação. Exemplos: "Saudável", "Ruim", "Complexo", "Belo"; (ii) A cor chama atenção de forma visual, ajudando a identificar rapidamente um resultado que esteja destoando dos outros, quando mostrados em conjunto. A nota é um número, tornando possível comparar resultados e medir a diferença entre esses resultados e seus valores esperados. As leituras se agrupam naturalmente. Os grupos de leitura e interpretação (*Reading Groups*) dão suporte a esses agrupamentos dando a

Figura 5.18: *Métrica configurada com um intervalo cadastrado.*

Figura 5.19: *Configurando um grupo de leitura e interpretação de intervalos.*

eles um nome para facilitar o reúso.

Por fim, o projeto Mezuro foi pensado para o monitoramento de métricas de código-fonte do ponto de vista de sua análise estática. Entretanto, a flexibilidade provida pelo Kalibro Metrics permite que ele também trate outros tipos de métricas, como as de cobertura de testes, fornecidas pela ferramenta JaBUTi, e métricas de atividades do repositório, como o número de contribuidores e *commits*, coletadas pela CVSAnaly – ambas ferramentas estão sendo integradas ao Mezuro. Da mesma forma, a integração com outros coletores permitirão o monitoramento de projetos em outras

Figura 5.20: *Definindo um tipo de leitura.*

linguagens de programação além de C, C++ e Java. Por exemplo, com o CVSAnaly será possível o monitoramente de métricas de código-fonte escrito em Python.

5.5 Continuidade do Projeto Mezuro

O projeto Mezuro e suas ferramentas deixaram de ser apenas uma contribuição para esta tese de doutorado. Além de ter colaborado diretamente com o projeto Qualipso e suas pesquisas na área de qualidade de produto, atualmente o Mezuro é um projeto em si do Núcleo de apoio às pesquisas em software livre da USP (NAPSoL-PRP-USP).

Antes disso, nossas ferramentas fizeram parte e colaboraram com outros trabalhos do IME-USP. Em 2009, colaborações na Kalibro fizeram parte do trabalho de fim de curso do aluno Carlos Morais. A Analizo foi usada como estudo de caso do trabalho de conclusão de curso sobre um mapeamento entre os conceitos de código limpo e métricas de código-fonte dos alunos João Machini e Lucianna Almeida, em 2010. Na dissertação de mestrado de Hugo Corbucci, defendida em 2011 e que realizou um estudo da relação entre métodos ágeis e software livre, a Analizo foi utilizada para coletar algumas métricas também discutidas nesse estudo.

Além disso, o desenvolvimento do Kalibro Metrics e todos os conceitos que foram propostos e implementados nele tornaram-se a base do trabalho de mestrado no IME-USP do mestrando Carlos Morais, previsto para 2013. As questões de escalabilidade e desempenho do Kalibro Metrics, bem como do projeto Mezuro, estão também sendo tratadas como objeto de estudo do trabalho de conclusão de curso do graduando Diego Araújo, em 2013. Ainda, a forma como o Mezuro "orquestra" os acessos aos *end-points* do Kalibro Metrics e outras questões gerais de escalabilidade e desempenho de *web services* deverão ser um estudo de caso real para a tese de doutorado do IME-USP na área de orquestração e coreografias de *web services* do doutorando Paulo Moura (previsto para 2015).

Por fim, as ideias e a experiência no projeto Mezuro estão proporcionando uma interação com o grupo do professor Michelle Lanza, da Universidade de Lugano, através do aluno João Marco Silva, que está como aluno visitante na Suíça, desenvolvendo trabalhos na área de visualização de software para aplicarmos ao projeto Mezuro. No mesmo espírito, os resultados científicos desta tese de doutorado serão apresentados ao grupo do professor Lanza para complementarmos nossas

abordagens sobre as análises de métricas de código-fonte. Também estabelecemos, durante este doutorado, uma boa relação com a Universidade Rey Juan Carlos, o que irá proporcionar um estágio ao aluno Diego Araújo, na Espanha, com o professor Gregório Robles, para integramos nossos projetos do ponto de vista prático.

Capítulo 6

Conclusão

Ao longo das últimas décadas, um novo mercado multi-bilionário tem sido criado com base em produtos e serviços de valor em cima de software livre. Para viabilizar um crescimento mais vigoroso desse fenômeno na indústria de software, os produtos têm de alguma forma que ser comparáveis, inclusive os mesmos sistemas de software de uma versão para outra, assim, podendo ser submetidos a medidas padronizadas.

Neste contexto, um exemplo real no Brasil é a reconhecida falta de uso de métricas nas contratações pelo Governo Federal, que, no momento, apenas recomenda o uso de Pontos de Função (Albrecht e Gaffney, 1983) para estimativas e acompanhamento técnico dos contratos de desenvolvimento de software. Como argumentado no Capítulo 2 desta tese, o uso de uma única métrica isolada pode levar a interpretações errôneas, tanto do lado cliente quanto do fornecedor, nesse cenário. Em termos práticos, isso já vem se refletindo no mercado nacional, ao ponto de boas empresas, em especial aquelas com metodologias modernas de desenvolvimento de software, não considerarem interessante assumirem determinados contratos com o Governo Federal, ou seja, quando elas têm oportunidade, preferem não trabalhar para e com o Governo. Entretanto, também é de conhecimento público que o Tribunal de Contas da União é aberto à novas propostas e estudos no que diz respeito às contratações de software. Também, o Governo Federal tem políticas e normas de incentivo à contrataçao de sistemas de software livre.

Então, pensando na contratação de desenvolvimento de software baseado em uma plataforma livre: como garantir que, além dos requisitos solicitados funcionando, o código gerado realmente será integrado ao repositório oficial do projeto em questão. O fato das funcionalidades serem incorporadas em uma versão de desenvolvimento não garante que será integrado ao projeto oficial, em especial porque questões técnicas, como testes automatizados e qualidade do código, são levadas em consideração.

Esta tese de doutorado colabora para a melhoria desse cenário ao apresentar estudos, com rigor estatístico, que há uma correlação entre métricas de código-fonte e atratividade de projetos de software livre. Apresenta estudos que detalham como analisar e monitorar um conjunto de 15 métricas de código-fonte para obtenção dos seus valores frequentes em um determinado software em avaliação e como esses valores estão distribuídos entre as partes do software. Dessa forma, por exemplo, ao adotar um software livre e abrir um processo licitatório para a contratação de serviços para esse sistema, os técnicos do Governo Federal podem avaliar a "saúde" do código-fonte do projeto em questão para ao final do contrato, além das funcionalidades, poder garantir a qualidade do código-fonte do projeto de software livre. Na sequência desta conclusão, explicitaremos as contribuições e limitações deste trabalho.

6.1 Contribuições

Neste doutorado, apresentamos um modelo de causalidade sobre a influência de métricas de código-fonte na atratividade de projetos de software livre escritos na linguagem C, C++ ou Java, avaliando em nossos estudos até 8.450 projetos. Assim, explicando tal influência em até 32% para o

número de downloads dos projetos e 19,6% para o número de membros, conforme apresentado nos estudos relatados na Seção 4.2.

Entretanto, o mapeamento de downloads para usuários e membros para contribuidores, ou seja, para a variáveis de atratividade, bem como a limitação em avaliar apenas projetos da plataforma Source Forge mostrou uma ameaça à validade desses estudos. Somado a isso, constatamos uma contradição nos trabalhos relacionados (Seção 4.1) quanto a abordagem para se estudar e observar as métricas de código-fonte. Dessa forma, como uma evolução dos primeiros estudos e outra contribuição científica desta tese, discutimos de maneira objetiva sobre o comportamento de 15 métricas selecionadas para este trabalho, em particular, com relação aos tipos de distribuição que as governam e os valores frequentes dessas métricas entre alguns dos principais projetos de software livre escritos em C, C++ e Java, conforme discutido na Seção 4.3.

Em suma, não ficamos limitados a uma única plataforma e fonte de dados. Assim, identificamos as distribuições estatísticas dos valores das métricas em 38 projetos de software livre, dentre eles os com mais contribuidores e usuários, a fim de compreendermos qual a medida mais representativa para o monitoramento de tais métricas. Além disso, mesmo não sendo o objetivo desta tese, observamos os valores frequentes para essas métricas, de forma a servirem de referência para outros projetos. Por fim, realizamos análises estatísticas da correlação entre métricas, a fim de definir um subconjunto de métricas que podem ser monitoradas ao longo do tempo. Com isso, respondemos e discutimos as questões de pesquisa desta tese que nos levaram a uma abordagem sistemática para monitoramento de métricas de código-fonte em projetos de software livre. Do ponto de vista das contribuições práticas, desde a concepção deste trabalho está o desenvolvimento de uma ferramenta para avaliação semi-automática das métricas de código-fonte, possibilitando, em resumo: (i) Obter e publicar informações sobre os sistemas de software livre por uma análise viável e de fácil leitura, percorrendo periodicamente e de forma automatizada os repositórios dos projetos de software livre; (ii) Fornecer um modelo, baseado nas métricas de código-fonte, para avaliar e comparar a qualidade de diferentes projetos de software livre; (iii) Permitir que os usuários (empresas, governos etc.) de software possam usufruir dessa avaliação como uma informação inicial para se decidirem por qual projeto usar; (iv) Prover mecanismos para que as equipes de desenvolvimento de software livre possam se auto-avaliar de forma a melhorar a qualidade do seu próprio produto. Por isso, boa parte dos esforços deste doutorado foram investidos no desenvolvimento do projeto Mezuro (Seção 5.4), que inclui as ferramentas Analizo e Kalibro Metrics. A plataforma construída permitirá colocar em prática as abordagens de monitoramento e avaliação das métricas apresentadas em nossos estudos. Com isso, provendo para a comunidade de software livre um mecanismo prático de aplicação das abordagens propostas.

6.2 Limitações

Há algumas limitações que podem ser apontadas neste trabalho. Por exemplo, os pontos de partida do desenvolvimento desta tese foram problemas práticos sobre a falta de uso de métricas de código-fonte por parte dos desenvolvedores de software e como inserir o monitoramento da análise estática do código sem fugir do que já está estabelecido dentro da cultura e do ecossistema das comunidades de software livre. Portanto, nossas abordagens para o monitoramento de código-fonte foram construídas nas evidências práticas da área de métricas e software livre, o que não contemplou uma (i) revisão sistemática da literatura, (ii) uma metanálise ou ainda (iii) uma abordagem baseada num portifólio de experiências, conforme espera-se para as teorias em engenharia de software, que devem ser construídas a partir de uma dessas três evidências Shull e Feldmann (2008).

Nos primeiros estudos sobre a correlação de métricas de código-fonte e atratividade de projetos de software livre, como já comentado, o mapeamento de downloads para usuários e membros para contribuidores são uma limitação, mas como apresentado na discussão desse estudo, ainda não se tem alternativas para tal associação. Também, a limitação em avaliar apenas projetos da plataforma Source Forge mostrou uma ameaça à validade desses estudos. Nos estudos sobre as distribuições e valores frequentes, resolvemos a questão de podermos obter os dados diretamente dos repositórios,

bem como também extrair dados oriundos da mineração das atividades de desenvolvimento, ou seja, *commits* e autores dos *commits*. Entretanto, nem sempre nossas hipóteses foram confirmadas, em particular a que sugeria a definição de valores frequentes por linguagem de programação. Com isso, a interpretação dos valores frequentes que encontramos ficou em aberto para que possa ser definida por cada "usuário" da abordagens proposta. Também, na se associou uma interpretação aos valores frequentes encontrados. Isto é, o que identificamos é que o comportamento e forma de se observar as métricas de código-fonte pode variar de acordo com o paradigma de programação, linguagem de programação, domínio de aplicação, maturidade (tempo/idade) do projeto ou mesmo por características particulares do sistema que está sendo avaliado e monitorado. Dessa forma, expomos os passos para se estudar as métricas em diferentes projetos, para que essa abordagem possa ser reprodutível em qualquer outro sistema de software que se tenha acesso ao código-fonte afim de realizar uma análise estática do mesmo, atráves das métricas selecionadas para esta tese ou outras métricas que a coleta seja automática.

Por fim, com exceção da Analizo, que vem sendo usada em vários estudos, incluso os desta tese, o Kalibro Metrics e a rede de monitoramento Mezuro ainda não foram amplamente usados. Em especial, não foi realizado um estudo exploratório da ferramenta para que fosse possível cadastrar as configurações baseadas, por exemplo, nos valores frequentes reportados no Capítulo 4. Assim, usuários não especialistas em métricas poderiam monitorar seus projetos baseado em tais conjuntos de métricas, intervalos e as interpretações disponíveis. Esse tipo de estudo permite que usuários reais possam avaliar a ferramenta. Apesar de ter sido colocado em ambiente de homologação em outubro de 2012, por conta de problemas de escalabilidade com o *web service* Kalibro Metric, o estudo não foi realizado e está previsto para o segundo semestre de 2013.

6.3 Outras contribuições

6.3.1 Publicações

Esta tese de doutorado resultou em 10 artigos científicos:

- "Mangue: Metrics and Tools for Automatic Quality Evaluation of Source Code". Simpósio Brasileiro de Qualidade de Software (SBQS), 2009: Contém a motivação deste trabalho conforme apresentado na introdução desta tese no Capítulo 1.

- "Crab: Uma ferramenta de configuração e interpretação de métricas de software para avaliação de qualidade de código". Simpósio Brasileiro de Engenharia de Software (SBES), 2009: Relata o desenvolvimento e as ideias da primeira ferramenta que nos permitiu testar a abordagem de configurações de métricas, discutida na apresentação das ferramentas no Capítulo 5.

- "A Study of the Relationships between Source Code Metrics and Attractiveness". Simpósio Brasileiro de Engenharia de Software (SBES), 2010: Apresenta o primeiro estudo sobre atratividade e métricas de código-fonte, também apresentado no Capítulo 4 desta tese.

- "Analizo: an Extensible Multi-Language Source Code Analysis and Visualization Toolkit". Simpósio Brasileiro de Engenharia de Software (SBES), 2010: Descreve as funcionalidade e arquitetura da ferramenta Analizo. Essa ferramenta foi a responsável por coletar as métricas de código-fonte nos estudos do Capítulo 4 e também é apresentada no Capítulo 5 desta tese.

- "Semi-Automatic Evaluation of Free Software Projects: A Source Code Perspective". International Conference on Open Source Systems (OSS), 2011: Contém os avanços e resultados intermediários das pesquisas realizadas, já incluindo a proposta do modelo de relação em *feedback loop* entre métricas e atividades de desenvolvimento citada no Capítulo 4 deste trabalho.

- "Free and Open Source Software Development and Research: Opportunities for Software Engineering". Simpósio Brasileiro de Engenharia de Software (SBES), 2011: Apresenta um mapeamento sistemático sobre os trabalhos de software livre na principal conferência em engenharia

de software do Brasil. Parte da discussão desse artigo serviu de base para a construção do Capítulo 2 desta tese.

- "Using Free/Libre/Open Source Software for Software Engineering Education: Opportunities and Experiences". Simpósio Brasileiro de Engenharia de Software (SBES), 2011. Apresenta uma discussão da adoção de software livre e uso do mesmo dentro do ensino de disciplina de engenharia de software. Também, serviu como conhecimento para a elaboração do Capítulo 2 deste trabalho.

- "Kalibro Metrics: um serviço para monitoramento e interpretação de métricas de código-fonte". Workshop Internacional de Software Livre (WSL), 2012: Descreve as funcionalidades e arquitetura do web service Kalibro Metrics, apresentado no Capítulo 5 desta tese.

- "Um Estudo de Caso do Mapeamento dos Conceitos de Código Limpo para Métricas de Código-fonte".Workshop Internacional de Software Livre (WSL), 2012. Trata dos conceitos de métricas de código-fonte do ponto de vista da abordagem de boas práticas de código limpo, o que está diretamente relacionado com o conteúdo apresentado no Capítulo 3 desta tese.

- "Mezuro Platform: Source Code Tracking Network". Simpósio Brasileiro de Engenharia de Software (SBES), 2012: Descreve as principais funcionalidades e visão geral da arquitetura da rede sócio-técnica de monitoramento de código-fonte proposta no contexto desta tese, conforme apresentado no Capítulo 5.

Também houve a colaboração na autoria de um capítulo de livro publicado em duas Jornadas de Atualização em informática da Sociedade Brasileira de Computação – "Software Livre e Propriedade Intelectual: Aspectos Jurídicos, Licenças e Modelos de Negócios (V1)". Atualizações em Informática 2011 e 2012.

6.3.2 Projetos

Houve a participação em 3 grandes projetos, representando e conduzindo os trabalhos em nome do Centro de Competência em Software Livre da Universidade de São Paulo, vinculado ao Departamento de Ciência da Computação do Instituto de Matemática e Estatística (IME-USP).

- QualiPSo: atuação no projeto QualiPSo (http://qualipso.org) como (i) líder de desenvolvimento de ferramentas de avaliação de métricas de código-fonte; (ii) responsável pela integração das ferramentas no servidor de homologação adminstrado pelo CCSL-USP; (iii) responsável pela implantação da rede social do projeto QualiPSo (http://community.stoa.usp.br). O projeto QualiPSo foi uma cooperação internacional, financiada pela *Eupean Commission*, entre 8 países, para pesquisar e promover o uso de Software Livre na industria de software da Europa, Brasil e China. A Universidade de São Paulo foi convidada para representar o Brasil neste projeto, o que viabilizou a consolidação do Centro de Competência de Software Livre da USP. Os 3 anos iniciais dos trabalhos realizados no contexto desta tese foram vinculados ao QualiPSo.

- NAPSOL: atuação como líder de projeto do Núcleo da Apoio à Pesquisa em Software Livre (NAPSOL). Atualmente, o Mezuro é um dos principais projetos em desenvolvimento pelo NAPSOL.

- Stoa: atuação como líder técnico da migração e desenvolvimento da rede social do USP, dentro do projeto Stoa, na qualidade de membro do Centro de Competência em Sofware Livre (CCSL) da Universidade de São Paulo (USP). Stoa é uma rede social dos estudantes, professores, funcionários e ex-membros da Universidade de São Paulo (USP). Os objetivos do Stoa são promover uma maior interação entre os membros da USP, criar um espaço onde cada pessoa dentro da Universidade tenha uma identidade digital de fácil acesso, tanto para

quem está dentro da USP quanto para usuários externos, e fornecer um sistema de software que facilite aos professores a administração de seus cursos para os estudantes. O Stoa consiste em um ambiente formado por 3 sistemas de software: (i) uma plataforma para redes sociais (http://social.stoa.usp.br); (ii) um wiki integrado à plataforma de rede social (http://wiki.stoa.usp.br) e (iii) um gerenciador de disciplinas (http://moodle.stoa.usp.br).

6.3.3 Orientações

No contexto deste doutorado, 7 alunos foram orientados, entre trabalhos de conclusão de curso ou iniciação científica no IME-USP.

Iniciação científica

- João Marco Maciel da Silva. Monitoramento de métricas de código-fonte de projetos de software livre – 2011/2012.
- Diego de Araújo Martinez Camarinha. Monitoramento de métricas de código-fonte de projetos de software livre – 2011/2012.
- Alessandro Wagner Palmeira. Monitoramento de métricas de código-fonte de projetos de software livre – 2011/2012.
- Carlos Morais. Initial integration between Analizo and Kalibro tools and the Qualipso tool set – 2010/2012
- João Machini de Miranda. Automatic Quality Evaluation of Source-Code: Extending evaluation to other languages for Qualipso Project – 2009/2010.
- Lucianna Almeida. A Model of Source Code Quality for FLOSS Projects (Qualipso Project) – 2010.
- Vinícius Daros. Support to integration between Analizo and Kalibro tools and Qualipso tool set – 2010.

Trabalhos de conclusão de curso de graduação

- João Machini de Miranda. Código Limpo e seu Mapeamento para Métricas de Código-fonte – 2010.
- Lucianna Thomaz Almeida. Código Limpo e seu Mapeamento para Métricas de Código-fonte – 2010.
- Carlos Morais. Kalibro: uma ferramenta de configuração e interpretação de métricas – 2009.

6.4 Trabalhos futuros

Da perspectiva dos estudos realizados, a correlação entre a atratividade de projetos de software livre e métricas de código-fonte precisam ser revalidados seguindo as medidas representativas de cada métrica, conforme apresentado na Seção 4.3, definindo um arcabouço teórico para estudo e monitoramento de métricas de código-fonte de projetos de software livre. A abordagem sobre as distribuições estatísticas, comportamento e valores frequentes em grandes projetos de software livre pode ser melhor sistematizada e necessita associar as respectivas interpretações aos valores frequentes encontrados. Por fim, aplicar um modelo estatístico adequado para a proposta de estudos sobre o efeito recíproco entre métricas de atratividade.

Pensando na continuidade do desenvolvimento das ferramentas, a colaboração com a ferramenta Analizo, junto ao seus principais desenvolvedores, para integração de novos *parses* que possam

prover o suporte para mais linguagens de programação na coleta de métricas. No contexto do projeto Mezuro, investigar problemas de escalabilidade e desempenho de web services, o que pode resultará em melhorias fundamentais ao Mezuro. Além disso, o uso do Kalibro Metrics como estudo de caso de um projeto real e de grande escala em pesquisas na área de web services. A integração do Kalibro com outros coletores de métricas que coletam métricas de código-fonte em outras linguagens, preferencialmente, Python e Ruby. Finalmente, adaptar a camada de visualização do projeto Mezuro para contemplar uma melhor infraestrutura de gráfico e interações com os dados e resultados de acordo com as técnicas de visualização de software.

Apêndice A

Levantamento das métricas de código-fonte

A.1 Métricas tradicionais

Generalizamos neste trabalho como "métricas de código-fonte tradicionais" as primeiras métricas propostas e as métricas não definidas para paradigma de programação orientado a objetos. As métricas tradicionais existentes, geralmente aplicadas de forma isolada, vêm sendo consideradas insatisfatórias (Mills, 1988). No passado, muitas métricas e uma série de processos foram propostos (Kafura e Canning, 1985; Kemerer, 1987; Mohanty, 1981; Rubin, 1987), mas a maioria das métricas definidas não possuem uma base teórica suficiente e/ou uma significativa validação experimental. No geral, cada uma delas foi definida por um indivíduo e, em seguida, testada e utilizada em apenas um ambiente, provavelmente limitado. Em alguns casos, existem relatos significativos de validação ou de aplicação dessas métricas. No entanto, outros testes ou o seu uso em ambientes diferentes têm produzido resultados não esperados. Essas diferenças não são surpreendentes, uma vez que faltam definições claras e hipóteses de testes bem definidas (Mills, 1988).

Assim, existe um grande número de métricas, mas apenas algumas têm sido amplamente utilizadas ou aceitas. Mesmo nos casos das métricas bem difundidas, como o número de linhas de código, métricas de Halstead e Complexidade Ciclomática, não são universalmente medidas de comum acordo. Estudos relatam experiências que tentam correlacionar as métricas com um número de propriedades de software, incluindo o tamanho, complexidade, confiabilidade, número de erros e manutenibilidade (Curtis *et al.*, 1979a,b; Kafura e Canning, 1985; Li e Cheung, 1987; Potier *et al.*, 1982; Woodfield *et al.*, 1981).

Uma dificuldade no uso de métricas de código-fonte vem do fato de que é difícil interpretar e comparar resultados das métricas, especialmente se elas envolvem diferentes ambientes, linguagens, aplicações e metodologias de desenvolvimento. Como consequência, não existem processos ou modelos de aplicação de métricas com bases teóricas. A maioria é baseada na combinação de intuição, características particulares e análise estatística de dados empíricos (Mills, 1988).

Mais um problema ocorre quando se usa métricas simples, como as que envolvem linhas de código, pois diferenças técnicas, como diferentes linguagens de programação, influem na contagem e podem impossibilitar a comparação dos resultados (Jones, 1986). Entretanto, quando se tem métodos de aplicação de métricas de software em um ambiente controlado, é possível auxiliar, de forma significativa, na melhoria da qualidade do software e da produtividade (Basili e Rombach, 1987; Grady e Caswell, 1987). Em muitos casos, métricas relativamente simples, como linhas de código e complexidade ciclomática foram bons preditores de outras características, como número de erros, esforço total e manutenibilidade (Grady e Caswell, 1987; Li e Cheung, 1987; Rombach, 1987). Embora úteis, essas métricas não podem ainda ser utilizadas indiscriminadamente, mas a aplicação cuidadosa de algumas das métricas e modelos disponíveis podem produzir resultados úteis, se de acordo com o ambiente específico (Mills, 1988). Abaixo apresentaremos as métricas encontradas na literatura durante esta pesquisa de doutorado:

A.1.1 Métricas de tamanho

Algumas métricas de software foram desenvolvidas para tentar quantificar o tamanho e auxiliar as medições na fase de concepção do software. Em sua origem, seguem o paradigma de desenvolvimento de software clássico, por exemplo, não orientado a objetos. Entre métricas de tamanho podemos citar:

- Linhas de Código (LOC) – O conceito mais comum de LOC parte do princípio de contar qualquer linha que não seja linha em branco ou comentário, independentemente do número de declarações por linha (Boehm, 1981; Jones, 1986), só podendo ser comparados valores diferentes dessa métrica caso elas se referirem à mesma linguagem e se o estilo de programação estiver normalizado (Jones, 1991);

- Pontos de Função (FP) – Baseado nos modelos e ciclos de vida clássicos de desenvolvimento de software, FP foi proposta como uma medida de tamanho de software que pode ser estimada no início do desenvolvimento. A métrica calcula o valor total Pontos de Função para o projeto, que tem como parâmetros o número de entradas do usuário, as consultas, as saídas e os principais arquivos (Albrecht e Gaffney, 1983).

- System Bag – É uma medida de dimensão total de um software determinada a partir das funcionalidades descritas em especificações formais. O algoritmo de cálculo se difere por se aplicar aos sistemas orientados por dados ou orientados por processos. Um dos objetivos dessa métrica é maximizar o quociente Bag/Custo total no desenvolvimento do projeto (DeMarco, 1982).

- Métricas de Halstead – É um conjunto de métricas baseadas na teoria da informação, consideradas as primeiras métricas com fundamentação teórica comum (Halstead, 1972, 1977). Essas métricas se aplicam à vários aspectos do software e também são usadas para avaliar o esforço global de desenvolvimento. O Vocabulário (n), comprimento (N) e volume (V) são métricas que aplicam-se especificamente ao software final. Também foram especificadas fórmulas para calcular o esforço total (E) e o tempo de desenvolvimento (T) de software.

A.1.2 Métricas de complexidade

Métricas de complexidade são uma forma objetiva de medir a complexidade de uma parte do software. A alta complexidade deve ser evitada pois prejudica a compreensão do código e o torna mais suscetível a erros. Softwares grandes são mais suscetíveis a ter alta complexidade, portanto recomenda-se usá-las juntamente com as métricas de tamanho. Dentre as principais métricas de complexidade, temos:

- Complexidade Ciclomática – Parte do princípio que a complexidade depende do número de condições (caminhos), correspondendo ao número máximo de percursos linearmente independentes em um software. A proposta é que se possa medir a complexidade do programa, assim orientando o desenvolvimento e os testes do software (McCabe, 1976, 1982; McCabe e Butler, 1989; Myers, 1977; Stetter, 1984). Essa métrica será detalhada na Seção 3.1.2.

- Número de Nós –

  Corresponde ao número de nós em um grafo de fluxo de controle que representa a sequência de controle de execução de instruções num programa. Um nó é definido como uma passagem necessária das linhas direcionais no grafo, sendo assim uma proposta de métricas de complexidade de software (Woodward *et al.*, 1979).

- Fluxo de Informação – Foi proposto que os fluxos de informações na estrutura de um programa são medidas de complexidade de software (Kafura e Henry, 1981). Esse método conta o número de parâmetros de entradas (*fan-in*) e saídas (*fan-out*).

A.1.3 Métricas de manutenibilidade

O SEI recomenda o uso de um índice de manutenibilidade (Maintainability Index — MI) (VanDoren, 1997). A manutenibilidade de um sistema é calculada usando uma combinação de resultados de outras métricas. O índice é um polinômio da seguinte forma:

$$171 - 5,2\ln(aveV) - 0,23aveV(g') - 16,2 * \ln(aveLOC) - 50\sin(\sqrt{2.4 * perCM})$$

Os coeficientes sugeridos são resultado de calibragem usando vários sistemas mantidos pela HP (Hewlett-Packard). Os termos são os seguintes:

- $aveV$ é a média do volume V de Halstead;
- $aveV(G)$ é a média da complexidade ciclomática por módulo;
- $aveLOC$ é a média de linhas de código por módulo;
- $perCM$ (opcional) é a média da porcentagem de linhas de comentário por módulo.

A.1.4 Outras métricas tradicionais

Existem ainda outras métricas. Elas se aproximam conceitualmente de alguma das métricas anteriormente detalhadas, ou ainda são correlacionadas como precursoras de algumas métricas que serão apresentadas dentro do contexto de orientação a objeto na Seção A.2. São elas, em ordem de publicação:

- Dados Compartilhados *(BAM - Binding Among Modules)*: mede o volume de dados compartilhados entre diferentes módulos do programa (Basili e Turner, 1975; Henry e Kafura, 1981).
- Tamanho Médio dos Módulos *(AML - Average Module Length)*: mede o tamanho médio dos módulos que compõem o programa (Boehm *et al.*, 1978).
- Condições e Operações *(COC - Conditions and Operations Count)*: contabiliza os pares de todas condições e laços nas operações (Hansen, 1978).
- Razão de Coesão *(CRM - Cohesion Ratio Metrics)*: mede a relação entre o número de módulos com coesão funcional e o número total de módulos (Bieman e Ott, 1994; Macro e Buxton, 1987; Yourdon e Constantine, 1979).
- Uso de Variável *(LVA - Live Variables)*: mede o período em que cada variável é usada (Dunsmore e Gannon, 1979).
- Menor Número de Caminho *(MNP - Minimum Number of Paths)*: mede o menor número de caminhos em um programa e seu alcance em qualquer nó (Schneidewind e Hoffmann, 1979).
- Métricas Morfológicas *(MOR - Morphology metrics)*: mede características morfológicas de um módulo, como tamanho, profundidade, largura e razão entre aresta e nó. (Yourdon e Constantine, 1979).
- Complexidade de Fluxo de Controle e de Dados *(CDF - Control flow complexity and Data Flow complexity)*: combinação de métricas baseadas na definição de variáveis e de referências cruzadas (Oviedo, 1980).
- Número de Funções *(FCO - Function Count)*: mede o número de funções e as linhas de código das funções (Smith, 1980).
- Métrica de Complexidade Composta *(SSC - composite metric of Software Science and Cyclomatic complexity)*: combina as métricas de Halstead com Complexidade Ciclomática (Baker e Zweben, 1980).

- Instruções no Código *(DSI - Delivered Source Instructions)*: Conta declarações separadas numa mesma linha como distintas e ignora linhas de comentários (Boehm, 1981).
- Equivalência entre Módulos *(ESM - Equivalent Size Measure)*: mede o percentual de modificações num módulo reutilizado (Boehm, 1981).
- Declarações Executáveis *(EST - Executable Statements)*: contabiliza declarações separadas em uma linha de código como distintas, ignorando comentários, declarações de variáveis e cabeçalhos (Boehm, 1981).
- Níveis de Aninhamento *(NLE - Nesting Levels)*: mede a complexidade pela profundidade de aninhamentos (Zolnowski e Simmons, 1981).
- Métricas de Peso Específico *(SWM - Specification Weight Metrics)*: mede funções primitivas num diagrama de fluxo de dados (DeMarco, 1982).
- Distância de Árvore *(TRI - Tree Impurity)*: determina o quanto um grafo se distanciou de uma árvore (Ince e Hekmatpour, 1988).
- Modularidade Global *(GLM - Global Modularity)*: descrição da modularidade global em termos de várias visões específicas de modularidade (Hausen, 1989).
- Relação de Acoplamento *(COR - Coupling Relation):* sinaliza uma relação de todo par de módulos segundo o tipo de acoplamento (Fenton e Melton, 1990).
- Extensão de reúso *(ERE - Extent of Reuse)*: classifica um módulo segundo o grau de reúso (Gaffney *et al.*, 1995).

A.2 Métricas de orientação a objetos

Como o paradigma da orientação a objetos usa entidades e não algoritmos como componentes fundamentais, a abordagem das métricas de código-fonte para programas orientados a objetos deve ser diferente. Além de tamanho e complexidade, em um software orientado a objetos é possível medir o uso da herança e o grau de interdependência entre as entidades. Desse conjunto de métricas, destacam-se as dos grupos CK (Chidamber e Kemerer, 1994) e MOOD (Brito e Abreu e Carapuça, 1994). As métricas apresentadas nesta seção foram selecionadas a partir de uma análise das métricas propostas para orientação a objetos realizada por (Xenos *et al.*, 2000). As métricas de orientação a objetos foram divididas em 5 categorias para um melhor entendimento do conceito das mesmas: métricas de classe, métricas de métodos, métricas de herança, métricas de acoplamento e métricas do sistema (características gerais do software orientado a objetos). Dentro de cada um desses grupos, as métricas são apresentadas em ordem de publicação.

A.2.1 Métricas de classes

- Encapsulamento dos Atributos *(AHF - Attribute Hiding Factor)*: razão entre a soma de todos os atributos herdados de todas as classes do sistema em consideração ao número total de atributos das classes disponíveis (Morris, 1988).
- Métodos Reusados *(PMR - Percent of Potential Method uses actually Reused)*: percentual de reúso dos métodos da classe (Morris, 1988).
- Coesão de Classe *(CCO - Class Cohesion)*: mede a relação entre as classes (Chidamber e Kemerer, 1991).
- Número de Ancestrais (*NOA - Number Of Ancestors*): número total de ancestrais de uma classe (Kolewe, 1993).

- Ponderação do Tamanho da Classe (*WCS - Weighted Class Size*): número de ancestrais mais o tamanho total de métodos da classe (Kolewe, 1993).
- Respostas para uma Classe *(RFC - Response For a Class)*: número de métodos dentre todos os métodos que podem ser invocados em resposta a uma mensagem enviada por um objeto de uma classe (Sharble e Cohen, 1993).
- Linhas de comentário por Método *(CLM - Comment Lines per Method)*: mede o percentual de comentários no método (Lorenz e Kidd, 1994).
- Porcentagem de Métodos Comentados *(PCM - Percentage of Commented Methods)*: percentual de métodos com comentários em uma classe (Lorenz e Kidd, 1994).
- Código Orientado a Função *(FOC - Function Oriented Code)*: mede o percentual de código não orientado a objeto usado no programa (Lorenz e Kidd, 1994).
- Número de Métodos de Classe em uma Classe *(NCM - Number of Class Methods in a class)*: mede os métodos disponíveis em uma classe, mas não em suas instâncias (Lorenz e Kidd, 1994).
- Número de Variáveis de Instância em uma Classe *(NIV - Number of Instance Variables in a class)*: mede a relação de uma classe com outro objeto do programa (Lorenz e Kidd, 1994).
- Falta de Coesão entre Métodos *(LCM - Lack of Cohesion between Methods)*: indica o nível de coesão entre os métodos (Chidamber e Kemerer, 1994).
- Privacidade Interna *(INP - Internal Privacy)*: refere-se ao uso de funções de acesso à classe, incluindo as chamadas da própria classe (Chidamber e Kemerer, 1994) .
- Dados Públicos *(PDA - Public Data)*: contabiliza o número de acessos aos dados protegidos e públicos de uma classe (McCabe *et al.*, 1994).
- Percentual de Dados Públicos *(PPD - Percentage of Public Data)*: é o percentual de dados públicos de uma classe (McCabe *et al.*, 1994).
- Métodos ponderados por Classe *(WMC - Weighted Methods per Class)*: soma ponderada de todos os métodos da classe (McCabe e Associates, 1994).
- Número de Parâmetros por Métodos *(NPM - Number of Parameters per Method)*: número médio de parâmetros por método (Bansiya e Davi, 1997).
- Entropia de Complexidade da Classe *(CEC - Class Entropy Complexity)*: mede a complexidade da classe, baseada no conteúdo de suas informações (Bansiya e Davi, 1997).
- Métrica de Acesso aos Dados *(DAM - Data Access Metric)*: razão do número de atributos privados em relação ao número total de atributos declarados nas classes (Bansiya e Davi, 1997).
- Medida de Abstração dos Atributos *(MAA - Measure of Attribute Abstraction)*: razão do número de atributos herdados por uma classe em relação ao número total de atributos nas classes (Bansiya e Davi, 1997).
- Medida de Abstração das Funções *(MFA - Measure of Functional Abstraction)*: razão entre o número de métodos herdados por uma classe em relação ao número total de métodos acessíveis na classe (Bansiya e Davi, 1997).
- Número de Tipos de Dados Abstratos *(NAD - Number of Abstract Data types)*: número de objetos definidos pelo usuário como atributos de uma classe que são necessários para instanciar um objeto da referida classe (Bansiya e Davi, 1997).

- Número de Atributos Públicos (*NPA - Number of Public Attributes*): contabiliza o número de atributos declarados como públicos em uma classe (Bansiya e Davi, 1997).
- Número de Referências como Atributo *(NRA - Number of Reference Attributes)*: contabiliza o número de ponteiros e referências passadas como atributos de uma classe (Bansiya e Davi, 1997).
- Encapsulamento dos Métodos (*MHF - Method Hiding Factor*): razão entre a soma de todos os métodos invisíveis em todas as classes em relação ao número total de métodos definidos em um determinado sistema (Harrison, 1998).

A.2.2 Métricas de métodos

- Média de Complexidade dos Métodos *(AMC - Average Method Complexity)*: soma da complexidade ciclomática de todos os métodos dividido pelo número total de métodos (Morris, 1988).
- MAG (*MAX V(G)*): complexidade ciclomática máxima dos métodos de uma classe (McCabe *et al.*, 1994).
- Média do Tamanho dos Métodos *(AMS - Average Method Size)*: mede o tamanho médio dos métodos do programa (Lorenz e Kidd, 1994).
- Complexidade do Método *(MCX - Method Complexity)*: relaciona a complexidade com o número de mensagens (Lorenz e Kidd, 1994).

A.2.3 Métricas de acoplamento

- Acoplamento Aferente *(AC - Afferent Coupling)*: número total de classes externas de um pacote que dependem de classes de dentro desse pacote. Quando calculada no nível da classe, essa medida também é conhecida como Fan-in da classe, medindo o número de classes das quais a classe é derivada e, assim, valores elevados indicam uso excessivo de herança múltipla (McCabe *et al.*, 1994).
- Acoplamento Eferente *(EC- Efferent Coupling ou EC)*: número total de classes dentro de um pacote que dependem de classes externas ao pacote. Quando calculada no nível da classe, essa medida também é conhecida como Fan-out da classe, ou como Acoplamento entre Objetos (*CBO - Coupling Between Objects*) na família de métricas de CK (Chidamber-Kemerer) (Chidamber e Kemerer, 1994).
- Acoplamento de Classe *(CP - Class Coupling)*: mede as relações entre as classes com base em suas trocas de mensagens (Basili *et al.*, 1996).
- Fator de Acoplamento *(COF - Coupling Factor)*: razão entre o número máximo possível de acoplamentos no sistema e o número atual de acoplamentos possíveis por herança (Harrison, 1998).

A.2.4 Métricas de herança

- Construção Efetiva *(FEF - Factoring Effectiveness)*: número de métodos únicos dividido pelo número total de métodos (Morris, 1988).
- Potencial de Métodos Sobrescritos *(PMO - Percent of Potential Method uses Overridden)*: percentual de métodos sobrescritos utilizados (Morris, 1988).
- Nível de Aninhamento Hierárquico da Classe *(HNL - Class Hierarchy Nesting Level)*: mede a profundidade de hierarquia em que cada classe está localizada (Lorenz, 1991).

- Métricas de reúso de Método *(MRE - Method Reuse metrics)*: indica o nível de reúso dos métodos (Brito e Abreu e Carapuça, 1994).
- Número de Métodos Herdados *(NMI - Number of Methods Inherited)*: mede o número de métodos que uma classe herda (Lorenz e Kidd, 1994).
- Medida de Polimorfismo (*PFA - Polymorphism Factor*): razão entre o número atual de possibilidades de polimorfismos de uma classe e o número máximo de possíveis polimorfismos distintos da referida classe (e Abreu, 1994).
- Número de Métodos Sobrescritos *(NMO - Number of Methods Overridden)*: número de métodos redeclarados pela classe herdeira (Lorenz e Kidd, 1994).
- Tipo de Especialização *(SIX - Specialisation Index)*: indica o tipo de especialização (Lorenz e Kidd, 1994).
- Medida de Herança de Atributos *(AIF - Attribute Inheritance Factor)*: razão entre a soma dos atributos herdados em todas as classes do sistema e o número total de atributos disponíveis na classe (Basili *et al.*, 1996).
- Profundidade da Árvore de Herança *(DIT - Depth of Inheritance Tree)*: mede o número de ancestrais de uma classe (Shih *et al.*, 1997).
- Número de Filhos (*NOC - Number Of Children*): número total de filhos de uma classe (Rosenberg e Hyatt, 1997).
- Proporção de reúso *(RER - Reuse Ratio)*: razão entre o número de superclasses dividida pelo número total de classes (Rosenberg e Hyatt, 1997).
- Proporção de Especialização *(SPR - Specialisation Ratio)*: razão entre o número de subclasses dividido pelo número de superclasses (Rosenberg e Hyatt, 1997).
- Medida de Herança de Método *(MIF - Method Inheritance Factor)*: razão entre a soma dos métodos herdados em todas as classes e o número total de métodos disponíveis em todas as classes (Harrison, 1998).
- Proporção entre Profundidade e Largura *(RDB - Ratio between Depth and Breadth)*: razão entre a profundidade e a largura da hierarquia de classes (Bellin *et al.*, 1999).

A.2.5 Métricas do sistema

- Granularidade da Aplicação *(APG - Application Granularity)*: número total de objetos dividido pelo número total de pontos de função (Morris, 1988).
- Eficiência da Biblioteca de Objeto *(OLE - Object Library Effectiveness)*: razão entre o número total de objetos reusados e o número total de objetos da biblioteca (Morris, 1988).
- Complexidade Associada *(ASC - Association Complexity)*: mede a complexidade da estrutura associada ao sistema (Kolewe, 1993).
- Número de Hierarquias *(NOH - Number Of Hierarchies)*: número de hierarquias distintas do sistema (Kolewe, 1993).
- reúso de Classe *(CRE - Number of time a Class is Reused)*: mede as referências a uma classe e o número de aplicações que reusam tal classe (Brito e Abreu e Carapuça, 1994).
- Número de Rejeição de Classe *(NCT - Number of Classes Thrown away)*: mede o número de vezes que uma classe é rejeitada até que seja aceita (Lorenz e Kidd, 1994; West, 1992).

- Problemas Relatados por Classe *(PRC - Problem Reports per Class)*: mede os relatos de erros da classe (Lorenz e Kidd, 1994).
- reúso do Sistema *(SRE - System Reuse)*: percentual de reúso de classes (Brito e Abreu e Carapuça, 1994).
- Categorização *(CAN - Category Naming)*: divide as classes em um conjunto semanticamente significativo (Chidamber e Kemerer, 1994).
- Profundidade Média de Herança *(ADI - Average Depth of Inheritance)*: divisão entre a soma de níveis de aninhamento de todas as classes pelo número de classes (Bansiya e Davi, 1997).
- Média de Ancestrais *(ANA - Average Number of Ancestors)*: determina o número médio de ancestrais de todas as classes (Bansiya e Davi, 1997).
- Densidade Funcional *(FDE - Functional Density)*: razão entre o número de linhas de código e pontos de função (Fenton e Pfleeger, 1998).
- Modificação de Objetos Reusados *(PRO - Percent of Reused Objects Modified)*: percentual de objetos reusados que foram modificados (Bellin *et al.*, 1999).

Apêndice B

Arquitetura do Projeto Mezuro

Toda a inteligência e processamento do Mezuro está no *web service* Kalibro Metrics. O projeto Mezuro cresceu de tal forma que deixou de ser apenas uma contribuição desta tese de doutorado. O desenvolvimento do Kalibro Metrics e todos os conceitos que foram propostos e implementados nele tornaram-se um trabalho de mestrado no IME-USP (do mestrando Carlos Morais) correlacionado e complementar do ponto de vista das contribuições tecnológicas desta tese. Além disso, as questões de escalabilidade e desempenho do Kalibro Metrics, bem como do projeto Mezuro, serão tratadas como objeto de estudo de um trabalho de fim de curso (do graduando Diego Araújo). Por fim, a forma como o Mezuro "orquestra" os acessos aos *end-points* do Kalibro Metrics e outras questões gerais de escalabilidade e desempenho de *web services* poderão proporcionar um estudo de caso real para uma tese de doutorado do IME-USP na área de orquestração e coreografias de *web services* (do doutorando Paulo Moura).

Apresentaremos em detalhes a arquitetura do Kalibro neste apêndice, uma vez que na prática, na implementação do Mezuro plugin, ela reflete os modelos definidos no Kalibro Metrics e depois faz uso dos recursos do padrão arquitetural *Model-View-Controler* (MVC) providos pelo arcabouço *Ruby on Rails*.

B.1 Interface de Programação

Esta seção descreve as classes do modelo, que representam os conceitos centrais do Kalibro, e outras poucas classes que fazem parte de sua interface de programação. Todas as classes da API encontram-se no pacote `org.kalibro`. Para uma visão geral e sumarizada das entidades, ver a Figura B.1.

Chamamos a atenção para algumas convenções utilizadas em todos os diagramas UML apresentados aqui:

- Em um diagrama convencional, os elementos precedidos pelo sinal '+' são públicos. Porém, quando um atributo estiver assinalado como público, o leitor deve entender que não se trata de violação do princípio de encapsulamento, mas que o atributo possui métodos de acesso direto - `get` e `set` se atributo simples, ou ainda `add` e `remove` para coleções.

- Assinalamos com o sinal '#' os atributos cujos valores podem ser consultados mas não devem ser alterados.

- Constantes aparecem como atributos públicos no nomes em maiúsculas separadas por '_'.

- Os diagramas não mostram todos os elementos das classes representadas. Para economizar espaço, diminuir a poluição visual e não distrair o leitor, suprimi os métodos privados ou irrelevantes no contexto, por exemplo, `toString()`.

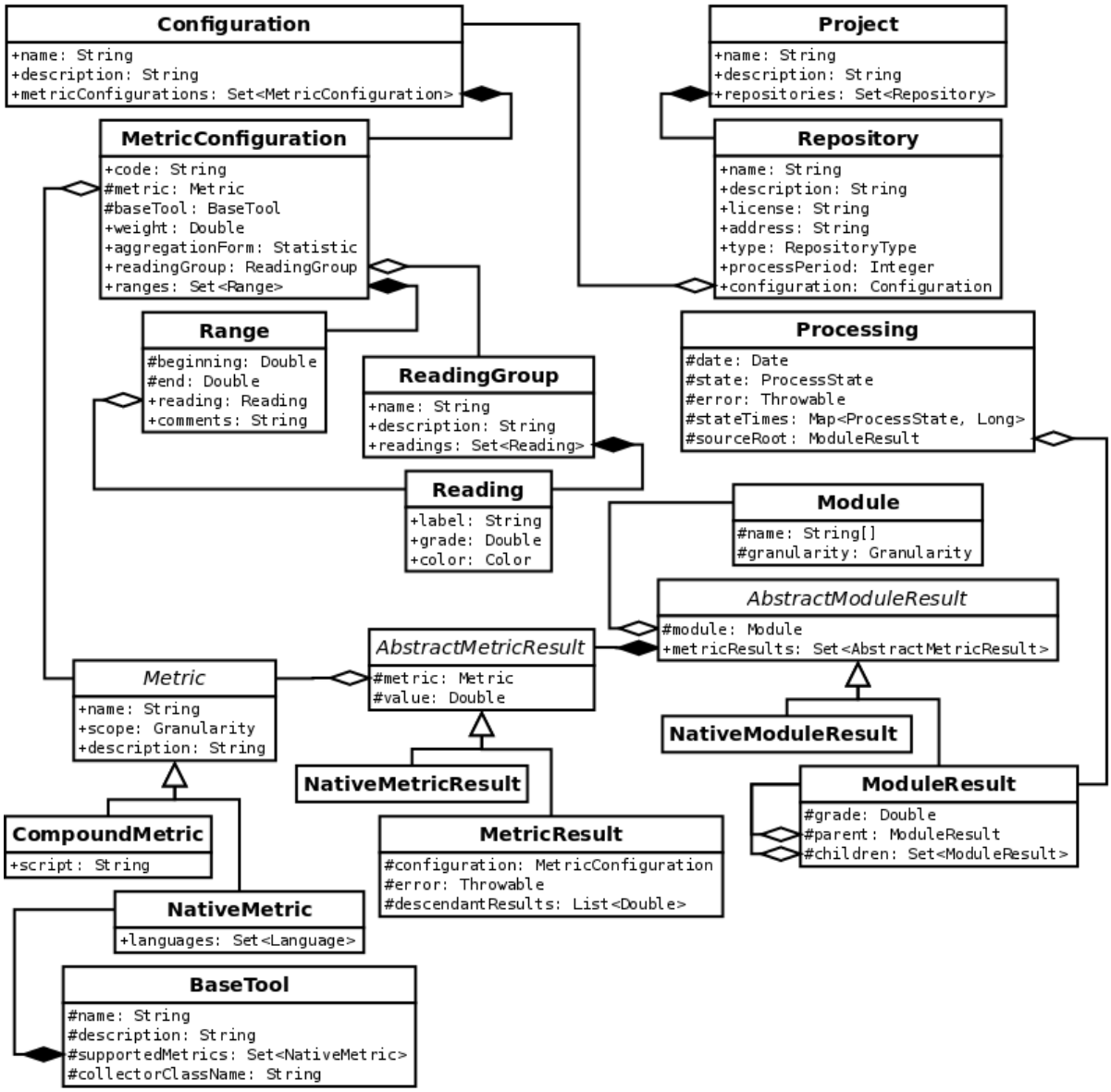


Figura B.1: *Módulo e granularidade*

B.1.1 Métricas e resultados

Projetos de software são compostos por vários módulos que se integram, cada módulo sendo divido em submódulos de granularidade cada vez menor: subcomponentes, pacotes, classes, métodos. Essa organização se reflete na estrutura de diretórios e subdiretórios, que contêm arquivos com o código-fonte, que contêm declarações.

O tipo enumerado[1] `Granularity` representa o nível de granularidade do fragmento de código a ser medido. As constantes de `Granularity` possuem nomes inspirados em Java, mas software escrito em quase qualquer linguagem de programação possui esses quatro níveis de granularidade:

- Método (`Granularity.METHOD`) é uma subrotina executada por um objeto ao receber uma mensagem. Equivalente a uma função ou procedimento em linguagens estruturadas.
- Classes (`Granularity.CLASS`) definem o comportamento de seus objetos com um conjunto

[1] Um tipo é dito enumerado quando seus valores válidos compõem em conjunto fixo de constantes. Em Java, isso era implementado usando o padrão *typesafe enum*. Segundo esse padrão, o tipo é definido por uma classe com construtor privado e por um conjunto de constantes públicas que definem as possíveis instâncias dessa classe. A partir da versão 5.0, Java passou a ter suporte linguístico para tipos enumerados, com palavra chave e sintaxe própria, que possui todas as vantagens do padrão citado e outras facilidades.

de operações e atributos. Linguagens procedimentais e funcionais não possuem o conceito de classe, mas podemos considerar que cada arquivo se encontra nesse nível de granularidade, pois geralmente possuem funções que operam sobre os mesmos conjuntos de dados.

- Pacotes (`Granularity.PACKAGE`) agrupam classes. Em linguagens não orientadas a objetos, equivalem a diretórios.
- O nível de maior granularidade é o software como um todo (`Granularity.SOFTWARE`).

A classe `Module` representa um fragmento de software (ver Figura B.2). Módulos possuem os atributos granularidade e nome. O nome é na verdade um conjunto de nomes que representam o caminho completo do módulo, desde o diretório raiz, e portanto o identifica. O nome curto (operação `getShortName`) é a última parte desse caminho. Como veremos adiante, os módulos são organizados em uma árvore, contruída a partir do caminho completo de cada módulo, o qual é fornecido pela ferramenta base.

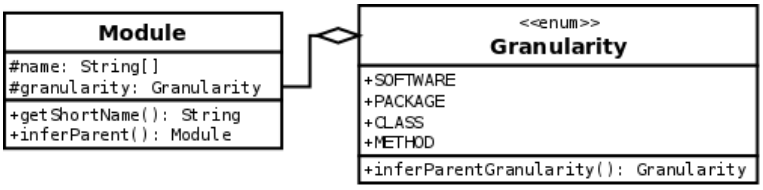


Figura B.2: *Módulo e granularidade*

Não é requisitado da ferramenta base o fornecimento de resultados para todos os módulos do código-fonte. Em geral, as ferramentas de métricas devolvem resultados por arquivo (i.e., as folhas da árvore de diretórios). Cabe aos módulos então, inferirem seus pais (método `inferParent`) para que seja possível montar a árvore. O pai de cada módulo tem o mesmo caminho do filho, exceto a última parte, por exemplo, o módulo `org.junit.Assert` possui ancestrais `org.junit` e `org`. Para inferir a granularidade do pai, as granularidades possuem o método `inferParentGranularity` que opera de acordo com a Tabela B.1.

Granularidade	Granularidade pai
METHOD	CLASS
CLASS	PACKAGE
PACKAGE	PACKAGE
SOFTWARE	SOFTWARE

Tabela B.1: *Granularidade pai inferida*

Nem sempre a granularidade inferida é a correta, por exemplo, uma classe pode estar dentro de outra classe ao invés de ter um pacote como pai. A granularidade inferida é substituída caso a ferramenta base forneça a granularidade correta.

A classe abstrata `Metric` representa métricas de código-fonte. O atributo `scope` é a granularidade dos módulos que ela mede diretamente. Por exemplo, NOP tem escopo de método, LCOM4 tem escopo de classe e COF tem escopo de software.

A Figura B.3 mostra as classes descendentes de `Metric`. `NativeMetric` representa métricas nativas, aquelas que são calculadas diretamente a partir do código-fonte por uma ferramenta base. O atributo `languages` especifica a quais linguagens de programação ela se aplica. Uma métrica composta (`CompoundMetric`) tem seu resultado calculado a partir de outras métricas. Veremos mais adiante como o atributo `script` é usado para calcular seu resultado.

Resultados de métricas nativas são representados pela classe `NativeMetricResult`, que associa a métrica ao valor do resultado. Esses resultados de métricas são obtidos medindo módulos, e portanto só fazem sentido quando agrupadas em um `NativeModuleResult` (ver Figura B.4).

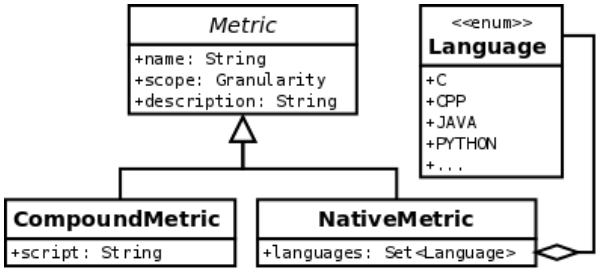


Figura B.3: *Métricas*

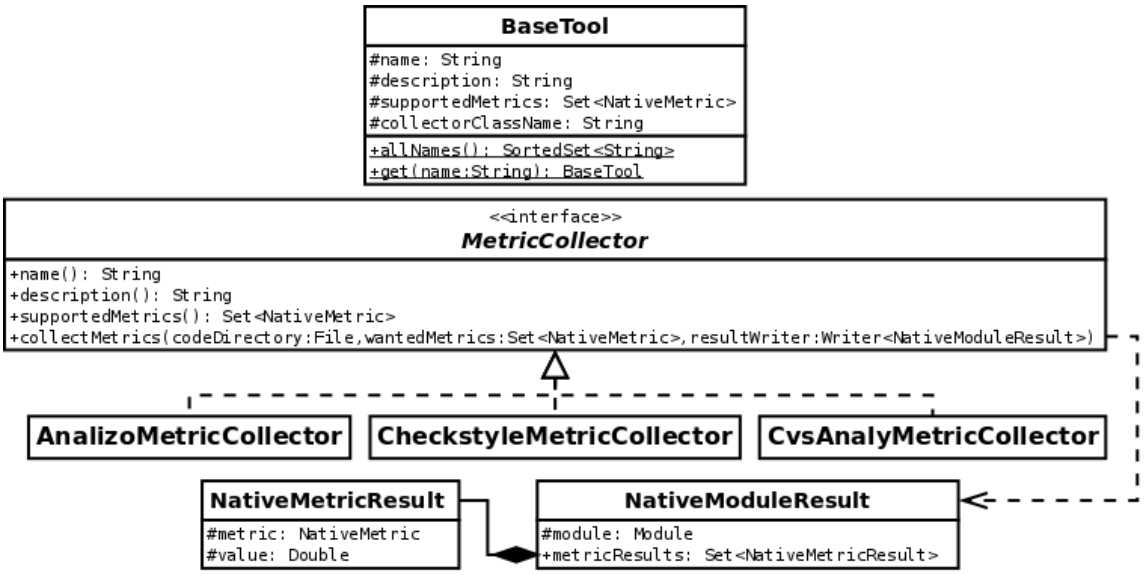


Figura B.4: *Ferramenta base - coletor de métricas*

O Kalibro dispara a coleta de métricas pelas ferramentas base através de implementações da interface `MetricCollector`. Dentre os métodos dessa interface, três fornecem apenas dados sobre a ferramenta (nome, descrição e métricas calculadas), e o último realiza a coleta em si (método `collectMetrics`). O primeiro argumento desse método é o diretório onde se encontra o código-fonte a ser analisado. O segundo argumento - um subconjunto das métricas calculadas - indica quais métricas devem ser coletadas, economizando tempo e memória caso o usuário não esteja interessado em todas as métricas que a ferramenta tem a oferecer. O último argumento é um fluxo onde os resultados devem ser escritos. A leitura dos resultados é feita concorrentemente, à medida que eles são produzidos.

As classes que implementam `MetricCollector` devem ter um construtor público sem argumentos, que lança exceção caso a ferramenta não esteja instalada corretamente ou de qualquer outra forma indisponível. Os nomes dessas classes devem ser registrados em um arquivo de configuração (`META-INF/collectors`). O Kalibro usa esse arquivo para gerenciar as ferramentas base de que dispõe. Para o usuário, informações sobre as ferramentas base são obtidas através da classe `BaseTool` e suas operações: `allNames()`, que devolve os nomes de todas as ferramentas base registradas; e `get(name)`, que devolve a ferramenta base cujo nome é igual ao argumento.

B.1.2 Interpretação e configuração

Quando um paciente faz um exame de sangue, ele precisa levar os resultados ao seu médico, que vai usar seus conhecimentos para fazer uma leitura, indicando quais índices estão saudáveis e quais são preocupantes, fazendo recomendações ou prescrevendo algum tratamento. Da mesma forma, resultados de métricas são inúteis até que sejam associados com uma interpretação que lhes dê significado. No Kalibro, essa interpretação é feita através de intervalos, e leituras desses intervalos.

A classe `Reading` é uma abstração de uma leitura, para facilitar o reúso e seus atributos:

- O rótulo (atributo `label`) é uma forma verbal concisa de chamar atenção para o conteúdo de uma interpretação. Exemplos: "Saudável", "Ruim", "Complexo", "Belo".
- A cor (atributo `color`) chama atenção de uma forma visual, ajudando a identificar rapidamente um resultado que esteja destoando dos outros, quando mostrados em conjunto.
- A nota (atributo `grade`) é um número que classifica, tornando possível comparar resultados e medir a diferença entre eles e suas expectativas.

Leituras se agrupam naturalmente. Por exemplo, "Bom" e "Ruim" devem fazer parte do mesmo grupo, assim como "Grande" e "Pequeno". A classe `ReadingGroup` dá suporte a esses agrupamentos (ver Figura B.5), dando a eles um nome para facilitar o reúso, como veremos adiante.

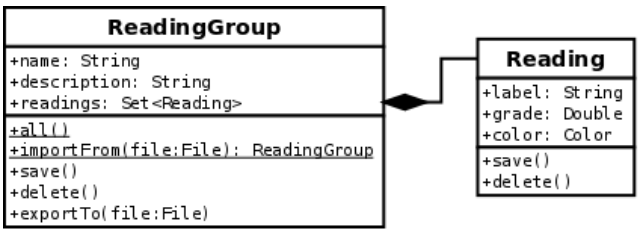


Figura B.5: *Grupo de leituras*

A classe `Range` modela intervalos numéricos que podem conter o valor do resultado produzido por uma métrica. Os atributos `beginning` e `end` são o início (limite inferior), e o fim (limite superior). Ambos os métodos baseiam-se nesses atributos: um intervalo responde se ele tem tamanho finito - `isFinite()` - e se contém um determinado valor - `contains(value)`. Os métodos assumem que o começo é fechado (exceto se for $-\infty$) e o final é aberto. Desse modo é possível criar um conjunto de intervalos contínuos e não conflitantes que cubram todos os valores reais. A escolha pelo extremo esquerdo fechado advém do fato de que a grande maioria das métricas produz resultados não negativos, com o 0 (zero) incluso.

Intervalos associam resultados de métricas com interpretações, através de uma leitura (atributo `reading`) e comentários (atributo `comments`). Comentários podem justificar o intervalo e guiar os desenvolvedores no sentido de melhorar o aspecto do código que influencia a métrica em questão. Desenvolvedores menos familiarizados com métricas podem diminuir seu custo de aprendizado usando intervalos cuidadosamente elaborados por outros mais experientes, em geral pesquisadores ou gerentes de projeto.

A classe `MetricConfiguration` modela configurações de métricas. Da mesma forma que as leituras, as configurações de métricas devem ser agrupadas em configurações gerais (classe `Configuration`, ver Figura B.6). configurações de métricas contêm os seguintes atributos:

- Um grupo de leituras (`readingGroup`), do qual cada intervalo da configuração deve obter uma leitura diferente.
- Uma forma de calcular resultados agregados (`aggregationForm`) para os ancestrais do módulo medido. Como exemplo, suponha que uma ferramenta base produza os resultados 3 e 5 para NOM nos módulos `org.junit.Assert` e `org.junit.Test`. O resultado para o módulo `org` será 4 (média de métodos por classe) se a forma de agregação for `Statistic.AVERAGE`, ou 8 (total de métodos do pacote) se a forma de agregação for `Statistic.SUM`.
- Um peso (`weight`), que serve para priorizar uma métrica em relação à outra de uma mesma configuração. O peso possibilita, junto com as notas dos intervalos associados aos resultados, a atribuição de uma nota por módulo. Essa nota serve para: ajudar a identificar módulos que requerem mais atenção; acompanhar evolução de módulos; fazer comparação entre projetos de software.

- Um identificador (`code`), para que a métrica possa ser referenciada por métricas compostas.
- Os intervalos que configuram a métrica (`ranges`).

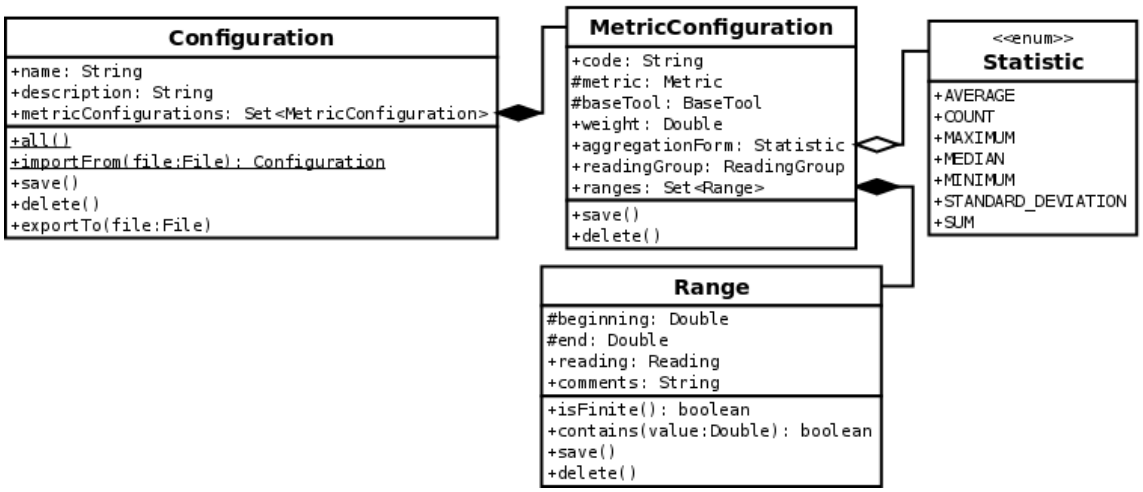


Figura B.6: *configuração de métricas*

Uma `MetricConfiguration` pode ser a configuração de uma métrica nativa ou composta. Caso a métrica configurada seja nativa, o atributo `baseTool` fornece a ferramenta base pela qual o resultado deve ser obtido. Caso a métrica seja composta, seu atributo `script` deve ser o corpo de uma função Javascript, que faz referência a outras métricas usando os identificadores (`code`): como variáveis para de métricas nativas; ou como funções para métricas compostas.

Identificador	Métrica	Script
cbo	CBO	-
lcom4	LCOM4	-
sc	composta	return cbo * lcom4;
sc2	composta	return 2 * sc();

Tabela B.2: *Exemplos de configurações de métricas*

Para tornar a explicação concreta, suponha que sejam cadastradas, em uma mesma configuração, as configurações de métricas representadas na Tabela B.2. Suponha ainda que a ferramenta base obtenha os valores CBO = 3 e LCOM4 = 2. Internamente, o Kalibro construirá o Javascript da Listagem B.1 e chamará as funções para determinar o valor das métricas compostas.

```
1 var cbo = 3;
2 var lcom4 = 2;
3 function sc(){return cbo * lcom4;}
4 function sc2(){return 2 * sc();}
```

Listagem B.1: *Exemplo de script para métricas compostas*

Javascript foi escolhido como linguagem para as métricas compostas porque: é fácil de usar em operações simples; sendo uma linguagem interpretada, suas operações podem ser restringidas pelo interpretador, portanto é segura; o Kalibro foi escrito em Java, que dispõe de bibliotecas para interpretação de Javascript.

configurações de métricas são as responsáveis por maior parte da flexibilidade do Kalibro. Ao cadastrar uma configuração, o usuário pode escolher: as métricas que lhe são relevantes; as ferramentas base que irão coletá-las; quais métricas devem ter maior peso; de que forma os resultados

serão agregados; quais intervalos serão usados na avaliação dos resultados. Além disso, ele pode criar métricas compostas a partir das disponíveis.

configurações podem ser usadas para avaliar, comparar e acompanhar projetos de software. Dependendo do objetivo, criá-las pode não ser uma tarefa trivial, mas resultado de experiência e pesquisa. Portanto, usuários devem ser capazes de reaproveitar configurações, publicá-las, compartilhá-las e derivar novas a partir de existentes. Esse é um dos principais objetivos da rede Mezuro, apresentada na seção 5.4, mas mesmo o uso direto do Kalibro dispõe de uma forma rudimentar de compartilhar configurações: exportá-las para arquivos. Com esse objetivo, as classes `ReadingGroup` e `Configuration` dispõem dos métodos `importFrom(file)` e `exportTo(file)`. O Kalibro usa o formato YAML[2], um padrão de serialização de objetos feito para ser de fácil manipulação em diferentes linguagens de programação, além de resultar em arquivos humanamente legíveis.

B.1.3 Processamento e acompanhamento

Para acompanhar as métricas de seu projeto de software, o usuário precisa cadastrar seus repositórios no Kalibro. As classes `Project` e `Repository` modelam projetos de software e seus repositórios de código (ver Figura B.7). Cada repositório possui os seguintes atributos:

- A licença do código-fonte disponibilizado (`license`).
- O endereço (atributo `address`) a partir do qual o código deve ser obtido. O significado do endereço varia de acordo com o tipo do repositório.
- O tipo de repositório, que indica como obter e atualizar o código. Pode indicar um sistema de controle de versão (`SUBVERSION`) ou um arquivo compactado remoto (`REMOTE_ZIP`), casos em que o endereço deve ser uma URL. Pode indicar um diretório (`LOCAL_DIRECTORY`) ou um arquivo compactado local (`LOCAL_ZIP`), casos em que o endereço deve ser um caminho no sistema de arquivos.
- A periodicidade (em dias) com a qual se quer monitorar o repositório (`processPeriod`). A cada período, o código é atualizado no servidor e reprocessado. Um período menor ou igual a zero indica que não se deseja processar o código periodicamente.
- A configuração com a qual se deseja processar o repositório. Ela indica quais métricas devem ser coletadas e como devem ser avaliadas.

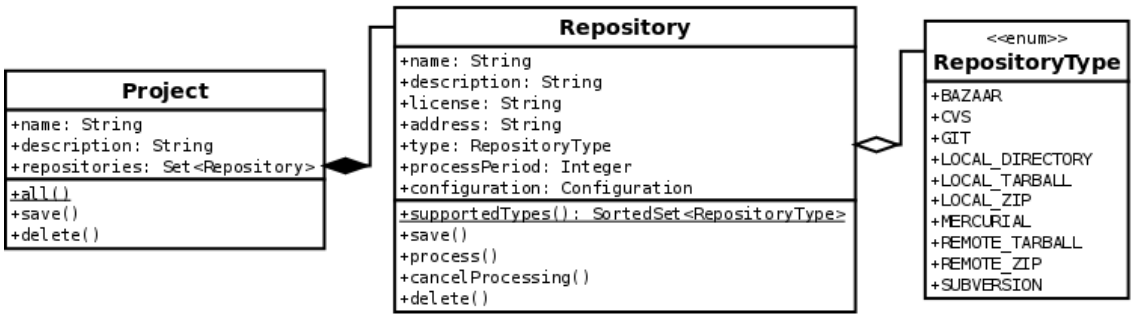


Figura B.7: *Projetos e repositórios*

O método de classe `Repository.supportedTypes()` devolve um subconjunto dos tipos mostrados na Figura B.7, indicando quais são implementados na máquina em que o Kalibro está executando. Para obter o código-fonte a partir do repositório, o Kalibro faz uso de chamadas de sistema para ferramentas instaladas. Quando a ferramenta não está instalada ou de alguma

[2]YAML Ain'T Markup Language, http://www.yaml.org/

forma inacessível, o tipo correspondente não aparece entre os oferecidos. O usuário deve levar em consideração que criar repositórios com tipos não oferecidos pode ter efeitos indesejáveis.

Além das operações usuais de salvar e apagar (`save()` e `delete()`), cada repositório possui os métodos `process()` e `cancelProcessing()`, para disparar ou cancelar seu processamento. Caso exista, o processamento anterior é cancelado toda vez que um novo é requisitado através do método `process()`. Por exemplo, se o repositório estava sendo processado diariamente, o usuário muda o período para 7 dias e executa o método `process()`, o repositório passa a ser processado apenas semanalmente.

O processamento é modelado pela classe `Processing` (ver Figura B.8). O atributo `date` marca o momento em que o repositório começou a ser processado. O atributo `state` (do tipo enumerado `ProcessState`) sinaliza em que fase o processamento se encontra:

- `ProjectState.LOADING` O processamento se encontra na fase de carregamento. O serviço está baixando o código-fonte a partir do repositório.
- `ProjectState.COLLECTING` Na fase de coleta as ferramentas base necessárias estão executando para fornecer as métricas nativas especificadas na configuração.
- `ProjectState.ANALYZING` A análise envolve o cálculo das métricas compostas, cálculo dos valores agregados e associação dos valores aos intervalos configurados.
- `ProjectState.READY` O processamento terminou com sucesso.
- `ProjectState.ERROR` O processamento terminou com erro.

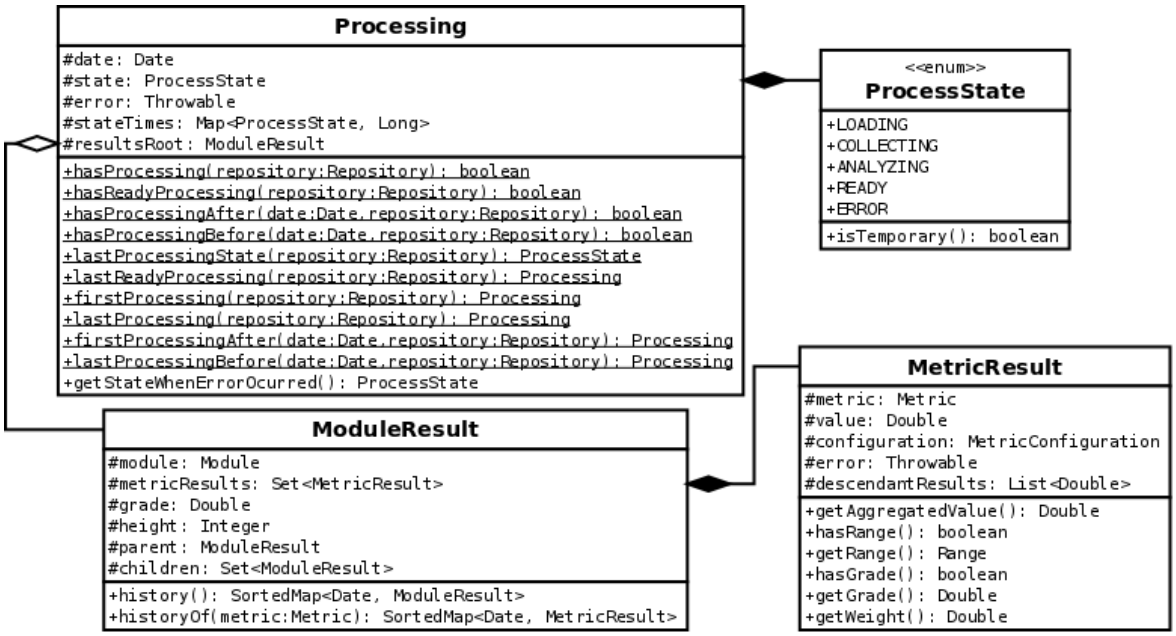


Figura B.8: *Processamento e resultados*

Caso o processamento tenha terminado com erro, o atributo `error` conterá a exceção lançada e o método `getStateWhenErrorOcurred()` devolverá em que estado o processamento estava quando ocorreu o erro. O atributo `stateTimes` mapeia quanto tempo (em milissegundos) foi gasto em cada fase do processamento. A soma dos tempos em geral não corresponde ao tempo total, pois a coleta e a análise ocorrem concorrentemente. Caso o usuário faça uma consulta e obtenha o estado `ProjectState.COLLECTING`, a análise pode já ter começado. Se o estado consultado for `ProjectState.ANALYZING`, isso significa que a coleta terminou e resta apenas terminar a análise.

A classe `Processing` possui ainda vários métodos para consultar processamentos de um repositório. Usando esses métodos é possível navegar por todos os processamentos ocorridos desde a criação do repositório.

Por fim, o atributo `resultsRoot` conterá, caso o processamento tenha terminado com sucesso, o nó raiz da árvore de resultados. A árvore de resultados reflete a estrutura do código-fonte, sendo que cada nó (um `ModuleResult`) contém o módulo, ponteiros para seus filhos e as métricas a ele associadas. As classes `ModuleResult` e `MetricResult`, apresentadas na Figura B.8, modelam os mesmos conceitos das classes `NativeModuleResult` e `NativeMetricResult`, apresentadas na Figura B.4, porém com mais dados, adicionados durante o processamento. De fato, essas classes fazem parte da mesma hierarquia (ver Figura B.9), mas todas as subclasses contêm mecanismos para assegurar o tipo correto das composições (por exemplo, um `ModuleResult` só aceita adição de `MetricResults`, e `NativeMetricResult` não pode ser criado para uma métrica composta).

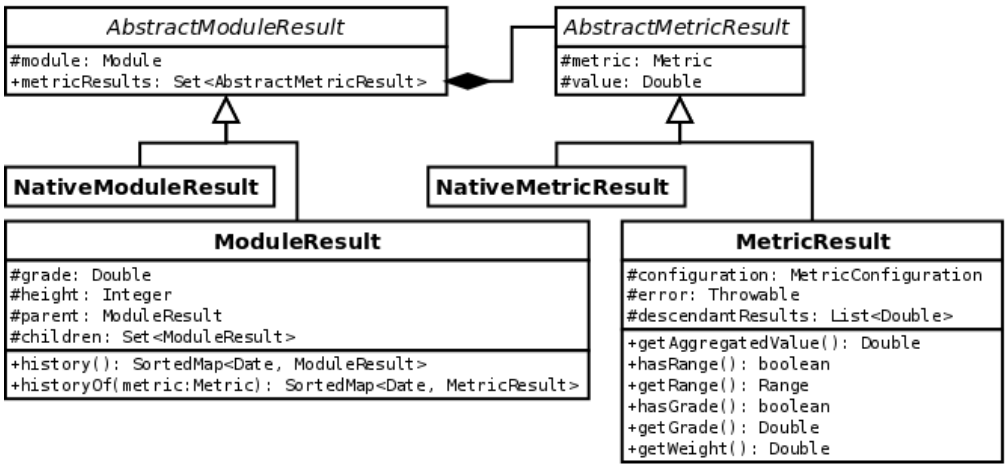


Figura B.9: *Hierarquia de resultados*

Um resultado processado de uma métrica, além de conter a métrica e o valor coletado (como no resultado nativo), contém a configuração de métrica associada (atributo `configuration`) e os valores coletados para a mesma métrica nos descendentes do módulo associado (atributo `descendantResults`). Os resultados dos descendentes são mantidos para que possa ser calculado o valor agregado (operação `getAggregateValue()`) de acordo com a forma de agregação especificada na configuração da métrica. As outras operações tratam de associar o valor (coletado ou agregado) à configuração, recuperando: o peso da métrica (`getWeight()`); o intervalo associado (`getRange()`); e a nota associada ao intervalo (`getGrade()`). Nem sempre a configuração de métrica possui um intervalo que contenha o valor obtido, e nem sempre o intervalo possui uma leitura associada, por isso as operações booleanas `hasRange()` e `hasGrade()`. Enfim, o atributo `error`, que serve apenas para métricas compostas, guarda a exceção lançada ao executar seu script, caso ocorra.

O resultado processado de um módulo, além de conter o módulo e os resultados das métricas (como no resultado nativo), contém uma nota (atributo `grade`), a altura na árvore de resultados (atributo `height`), a associação com seu pai (atributo `parent`) e com seus filhos (atributo `children`). A nota é uma média das notas para cada resultado de métrica, ponderada pelos pesos. A classe `ModuleResult` possui ainda duas operações: `history()`, que devolve todos os resultados obtidos para o módulo em questão (um para cada processamento realizado em que o módulo exista), associados às datas em foram processados; e `historyOf(metric)`, que opera da mesma forma mas apenas para uma métrica específica.

B.1.4 Preferências

As preferências são um conjunto de parâmetros que o Kalibro precisa para funcionar. A classe `KalibroSettings` modela essas preferências (ver Figura B.10). O atributo `serviceSide` indica

de que lado da relação cliente-servidor a máquina está. Caso cliente, as operações não serão executadas localmente, mas requisições serão feitas para um Kalibro Service instalado na máquina servidora, e o atributo `clientSettings` deverá conter o endereço do serviço (atributo `serviceAddress` da classe `ClientSettings`), que constitui toda informação necessária para comunicação com o serviço. Caso contrário (`ServiceSide.SERVER`), o atributo `clientSettings` é ignorado e o atributo `serverSettings` deve conter toda informação necessária para executar as operações do Kalibro localmente.

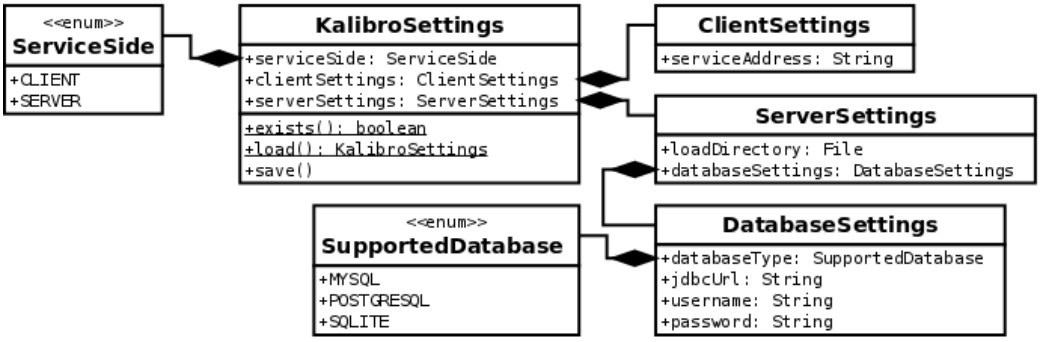


Figura B.10: *Preferências*

A classe `ServerSettings` contém os parâmetros internos do servidor: o diretório onde o código-fonte deve ser carregado para análise (atributo `loadDirectory`) e a configuração do banco de dados. A configuração do banco de dados (`DatabaseSettings`) é composta dos seguintes atributos:

- `databaseType` O Kalibro permite o uso de diferentes tipos de banco de dados: Mysql, PostgreSQL e SQLite.
- `jdbcUrl` Endereço para acessar o banco de dados via JDBC.
- `username` O nome de usuário para se conectar à base de dados.
- `password` A senha do usuário junto ao servidor de banco de dados.

Para o Kalibro executar corretamente, precisa ser criado um diretório chamado `.kalibro` dentro do diretório do usuário que o está executando (por exemplo, na minha máquina fica em `/home/<user>/.kalibro`). Esse diretório é usado com vários propósitos, entre eles manter as preferências, que são procuradas dentro dele, em um arquivo chamado `kalibro.settings`. As preferências são convertidas em formato YAML, que é de fácil leitura e edição.

O método de classe `KalibroSettings.load()` obtém a configuração a partir desse arquivo. Esse método é chamado em todos os lugares onde as preferências são usadas. O método de instância `save()` faz o contrário, persiste as preferências no arquivo. O método `exists()` verifica se o arquivo de preferências existe no local esperado. Internamente, o serviço utiliza apenas o método `load()`; os outros dois existem para serem usados por aplicativos que usem o Kalibro como API, provavelmente clientes com uma interface gráfica.

B.2 Interface do Serviço

O Kalibro Metrics possui uma interface em forma de serviço Web: Kalibro Service. Serviços Web são uma solução de interoperabilidade, com popularidade crescente na última década, que usa padrões abertos e largamente empregados (HTTP, SOAP, XML). Existe um largo número de bibliotecas, em quase todas linguagens de programação capazes de lidar com esses padrões.

Para utilizar o serviço a partir de uma aplicação Java, conhecer a API é suficiente. De acordo com a preferências (mostradas na seção B.1.4), as operações do Kalibro podem ser executadas

localmente ou remotamente. Mesmo se as operações forem executadas remotamente (lado cliente), a interface de programação Java encapsula toda comunicação com o serviço. Nenhum detalhe de como as requisições são feitas precisa ser conhecido e nenhuma outra informação além do endereço do serviço precisa ser fornecida. Porém, para utilizar o serviço fora do mundo Java, a interface Web Service precisa ser conhecida. Esta seção apresenta essa interface, ou seja, as especificações de mensagens trocadas: requisições e respostas.

Uma requisição feita a um serviço Web deve ser enviada para um de seus pontos de acesso. Cada ponto de acesso publica um documento WSDL que mostra as operações nele disponíveis e especifica o formato XML das requisições e respostas. O Kalibro utiliza o framework JAX-WS[3] para gerar seus pontos de acesso automaticamente, através de anotações (metadados presentes em declarações de classes, métodos e atributos).

Os pontos de acesso são especificados através de interfaces Java. Os métodos da interface especificam as operações do ponto de acesso, os parâmetros dos métodos especificam o formato da requisição e o tipo de retorno especifica o formato das respostas. Quando uma requisição é enviada a um ponto de acesso, o arcabouço a converte para objetos Java e executa o método correspondente com esses objetos como parâmetros. O valor devolvido pelo método é convertido de volta para XML e enviado como resposta.

As interfaces Java que especificam os pontos de acesso estão no pacote `org.kalibro.service`, e seus nomes seguem a convenção de ter o nome da entidade a que se dedica mais o sufixo `Endpoint`. As classes que especificam os formatos de XML de parâmetros e retorno se encontram no pacote `org.kalibro.service.xml`, e seguem a convenção de ter o nome da entidade a que se refere mais o sufixo `Xml`. O ponto de acesso principal não se dedica a nenhuma entidade em particular e portanto não segue essa convenção, chamando-se apenas `Kalibro`. Ele possui apenas uma operação, que devolve a versão do serviço (ver Figura B.11). Usarei diagramas de classes para as interfaces, por ser um formato de muito mais fácil apreensão do que documentos WSDL.

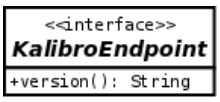


Figura B.11: *Ponto de acesso principal*

B.2.1 Ponto de acesso de ferramentas base

O ponto de acesso de ferramentas base (ver Figura B.12), possui duas operações: listagem das ferramentas base disponíveis e obtenção da ferramenta base a partir do nome. O leitor deve perceber a semelhança entre os atributos da classe `BaseToolXml` (que especificam tags XML) e os atributos da entidade `BaseTool`, assim como a semelhança entre suas operações e as da interface `BaseToolEndpoint` (compare com a Figura B.4). Isso ocorre porque os métodos do ponto de acesso foram feitos para serem usados pela entidade - quando as preferências indicam que as operações devem ser executadas remotamente. Nesse caso, `BaseTool.get("X")` chama `BaseToolEndpoint.getBaseTool("X")`. Desse modo, o objeto `BaseToolXml` serve o simples propósito de transferir dados entre cliente e servidor, seguindo o padrão de projeto *Data Transfer Object*, ou simplesmente DTO.

Esse padrão é usado em todos os pontos de acesso, portanto não listarei cada operação de cada ponto de acesso, me concentrando somente nas diferenças.

No caso das métricas, a diferença em relação às entidades é a compressão da hierarquia, pois a classe `MetricXml` serve para transferir dados tanto de métricas nativas como de métricas compostas, o atributo booleano `coumpound` sendo o diferenciador (comparar com a Figura B.3).

[3] http://jax-ws.java.net/

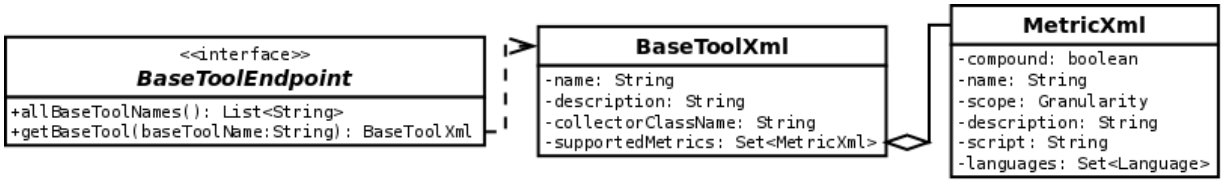


Figura B.12: *Ponto de acesso de ferramentas base*

B.2.2 Pontos de acesso de cadastros

O ponto de acesso de grupos de leituras (veja Figura B.13) possui as operações usuais para listar, salvar e remover os grupos, mas o controle é feito através do atributo `id`. O `id`, que aparece em quase todas as entidades, é sua chave primária no banco de dados do Kalibro. A operação `saveReadingGroup(readingGroup)` devolve o `id` do grupo salvo, para que se possa referenciar a mesmo grupo em operações futuras. Ela serve tanto para criar um grupo novo (se `readingGroup` não tiver `id`) como para atualizar um grupo existente (se o `id` estiver presente na requisição). O atributo `name` é obrigatório e deve ser único entre os grupos de leitura.

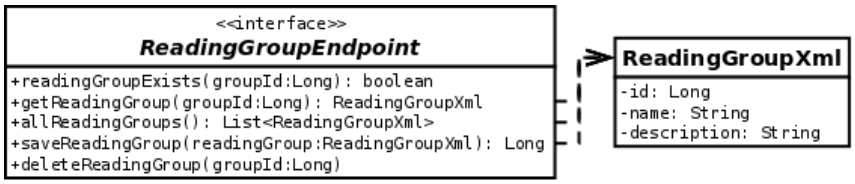


Figura B.13: *Pontos de acesso de grupos de leituras*

Além das operações usuais, o ponto de acesso de grupos de leitura também dispõe de operações para testar a existência de um grupo específico e recuperá-lo (`readingGroupExists(groupId)` e `getReadingGroup(groupId)`, respectivamente).

Ao recuperar um grupo de leitura, tanto listando como buscando, as leituras a ele associadas não são devolvidas. Da mesma forma, as leituras não são salvas junto com o salvamento do grupo. Isso ocorre para que os clientes do serviço possam fazer atualizações parciais mais facilmente e usar o padrão *lazy loading*, segundo o qual os objetos são carregados somente quando necessário. Isso torna menores o tempo de resposta e o volume de dados trocados. Para salvar ou recuperar leituras de um grupo é preciso recorrer ao ponto de acesso de leituras (ver Figura B.14).

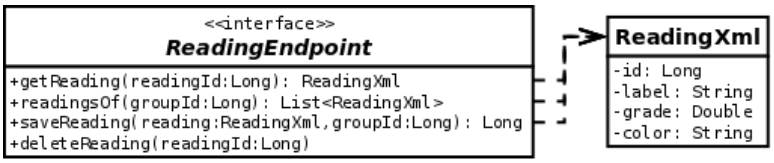


Figura B.14: *Pontos de acesso de leituras*

Leituras sempre fazem parte de um grupo, portanto listá-las e salvá-las exigem que se identifique o grupo a qual pertencem. Por isso o parâmetro `groupId` nessas operações.

Todos os campos de uma leitura são obrigatórios para salvá-la, exceto o `id` cuja ausência indica criação ao invés de atualização (analogamente aos grupos). A cor da leitura, que na entidade deve ser uma instância da classe `java.awt.Color`, no XML deve ser uma string de 6 caracteres, representado um número hexadecimal de 3 bytes, no formato RGB, formato usado para descrever cores em documentos CSS, amplamente empregados em páginas Web.

Os pontos de acesso de configurações gerais (`ConfigurationEndpoint`) e de projetos de software (`ProjectEndpoint`) são análogos ao de grupo de leituras, e os requisitos para salvar são os mesmos: nome obrigatório e único entre as entidades da mesma classe.

Os pontos de acesso de configurações de métricas (`MetricConfigurationEndpoint`) e intervalos (`RangeEndpoint`) são análogos ao de leituras, precisando do `id` das entidades das quais depende para listar e salvar. A referência que uma configuração de métricas faz ao seu grupo de leitura é feita através do seu identificador (`MetricConfigurationXml.readingGroupId`), assim como a referência do intervalo a sua leitura (`RangeXml.readingId`), mas as ferramentas base são identificadas pelo nome (`MetricConfigurationXml.baseToolName`), porque elas realmente não existem no banco de dados.

Para salvar uma configuração de métricas, a ferramenta base é obrigatória somente se a métrica for nativa, e o grupo de leituras é opcional. Os códigos devem ser únicos dentro de uma configuração geral. Para salvar um intervalo, apenas o início e o fim são obrigatórios, e o fim não pode ser menor que o início. Intervalos de uma mesma configuração de métricas não podem ter interseção.

O ponto de acesso de repositórios (`RepositoryEndpoint`) possui, além das operações usuais, outras para recuperar os tipos de repositório permitidos, iniciar e cancelar processamentos. Deve ser observado que os tipos locais (`LOCAL_DIRECTORY`, `LOCAL_ZIP` e `LOCAL_TARBALL`) nunca são oferecidos através do Kalibro Service, pois o servidor em geral não tem acesso ao sistema de arquivos do cliente. Os campos obrigatórios para salvar um repositório são: o nome, o tipo, o endereço e a referência a uma configuração. Os nomes dos repositórios de um projeto devem ser únicos.

B.2.3 Pontos de acesso de resultados

Os pontos de acesso de resultados são somente leitura, ou seja, não possuem operações para salvar entidades ou de qualquer forma alterar o estado do banco de dados, servindo apenas para consultar resultados de análise de código-fonte. Como nos outros casos, suas operações e objetos de transferência têm uma relação clara e direta com os métodos e atributos das entidades. São eles o ponto de acesso de processamentos (`ProcessingEndpoint`), o ponto de acesso de resultados por módulo (`ModuleResultEndpoint`), e o ponto de acesso de resultados por métrica (`MetricResultEndpoint`).

Um ponto que vale a pena ressaltar são os objetos de transferência de erros (ver Figura B.15). Eles podem aparecer em resultados de métricas compostas ou processamentos ou mal-sucedidos, com o objetivo de fornecer as informações necessárias para diagnosticar as causas do erro. A classe `ThrowableXml` reflete a estrutura de uma exceção Java (`java.lang.Throwable`), que contêm uma mensagem e uma pilha de execução mostrando os métodos chamados até o ponto em que o erro foi lançado. Pode ainda conter como causa uma outra exceção. O campo `targetString` contém o texto formatado da exceção, para que o cliente não precise navegar pela pilha de execução caso queira imprimi-la.

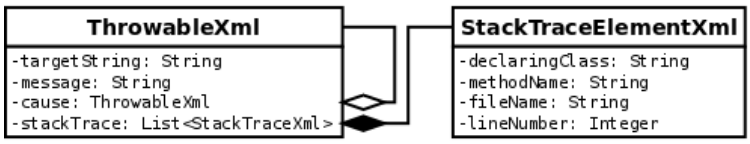


Figura B.15: *Transferência de erros por XML*

Outro ponto importante é: a configuração associada a um resultado de métrica é apenas uma fotografia da original (ver Figura B.16). Quando um repositório é processado, uma cópia da configuração associada é guardada junto com o processamento. Desse modo, a configuração usada no processamento pode ser posteriormente alterada (ou mesmo excluída), pois os resultados não estão associados a ela, mas sim à fotografia do seu estado no momento do processamento. A fotografia difere da configuração em si por não ter associação com um grupo de leituras, pois seus intervalos possuem também uma fotografia da leitura associada (campos `label`, `grade` e `color` em `RangeSnapshotXml`).

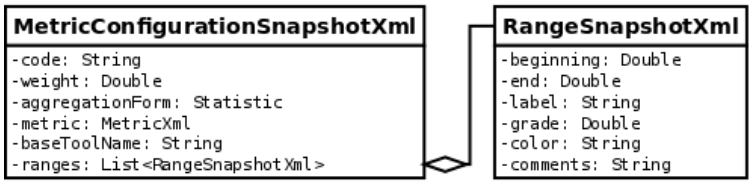


Figura B.16: *Fotografia de configuração de métricas*

B.3 Organização Interna

Esta seção e a próxima são dirigidas àqueles que querem entender o funcionamento detalhado do Kalibro. Irei percorrer a estrutura de pacotes, portanto o leitor tirará maior proveito se obtiver uma cópia do código-fonte junto ao repositório do Kalibro e explorá-lo usando o ambiente de desenvolvimento Eclipse[4]. Para tal, é necessário ter instalado o sistema de controle de versão Git[5] e executar os comandos a seguir na linha de comando:

```
git clone git://gitorious.org/kalibro/kalibro.git kalibro
cd kalibro
./configure.sh # seguir as instruções para configurar o Eclipse
git checkout -t origin/stable-1.0 # versão 1.0, tratada neste trabalho
```

O código-fonte do Kalibro está dividido em 3 projetos:

- `KalibroCore` Contém quase todo o código do Kalibro: as classes da API, incluindo suas dependências; as implementações dos coletores; interfaces dos pontos de acesso e objetos XML. Os outros projetos têm esse como dependência.
- `KalibroService` Contém apenas implementações dos pontos de acesso, bibliotecas e arquivos de configuração necessários para implantação do serviço Web.
- `KalibroTests` Contém toda estrutura de testes automatizados.

Em relação à estrutura de pacotes, podemos listar os seguintes:

- `org.analizo` Implementação do coletor de métricas para Analizo.
- `org.checkstyle` Implementação do coletor de métricas para Checkstyle.
- `org.cvsanaly` Implementação do coletor de métricas para CVSAnalY.
- `org.kalibro` Classes da API do Kalibro.
- `org.kalibro.service` Interfaces e implementações dos pontos de acesso.
- `org.kalibro.service.xml` Objetos de transferência de dados por XML.
- `org.kalibro.dto` Classes abstratas de transferência de dados.
- `org.kalibro.dao` Interfaces de acesso a dados.
- `org.kalibro.client` Implementações de acesso a dados remotos.
- `org.kalibro.core` Constitui, junto com seus subpacotes, o núcleo do Kalibro propriamente dito.

[4] http://www.eclipse.org/
[5] http://git-scm.com/

Duas classes do pacote `org.kalibro` que ainda não mencionei são `KalibroException` e `KalibroError`. Um erro do tipo `KalibroError` só é lançado quando alguma premissa básica é violada. Seu lançamento geralmente indica que o código do Kalibro contém um erro. Uma exceção do tipo `KalibroException` é lançada para fornecer contexto e encapsular exceções das bibliotecas usadas no Kalibro.

Sempre que uma exceção verificada (*checked exception*) é lançada por alguma das bibliotecas que o Kalibro usa, uma `KalibroException` (que não é verificada) é criada contendo a outra como causa e lançada no lugar dela. Evitar que classes lancem exceções verificadas deixa sua interface e o código das classes dependentes mais limpo, pois elimina a necessidade de declarar exceções na assinatura de todos os métodos entre quem lança e quem captura (Martin, 2008). Exceções verificadas frequentemente forçam a violação do OCP, um princípio de orientação a objetos que afirma: o comportamento de entidades deve poder ser alterado sem mudar sua interface (Martin, 2002).

B.3.1 Acesso e transferência de dados

As classes da API do Kalibro processam e persistem seus dados usando interfaces de acesso a dados, completamente desacopladas de como essa funcionalidade é fornecida, seguindo o padrão de projeto *Data Access Object* (DAO). Essas interfaces estão no pacote `org.kalibro.dao`, e são análogas às interfaces que especificam os pontos de acesso do serviço, mas os objetos passados como parâmetro e retorno são entidades, ao invés de simples objetos de transferência XML. De fato, os pontos de acesso foram planejados como DAOs em forma de serviço Web.

Os objetos de acesso a dados devem ser obtidos através da classe abstrata `DaoFactory`, que possui um método estático para obtenção de cada tipo de DAO, sem especificar a classe concreta que o implementa, seguindo o padrão de projeto *Abstract Factory*. As subclasses de `DaoFactory` implementam o acesso ao banco de dados ou ao serviço remoto, e a subclasse escolhida depende das preferências (seção B.1.4), como ilustrado na Figura B.17.

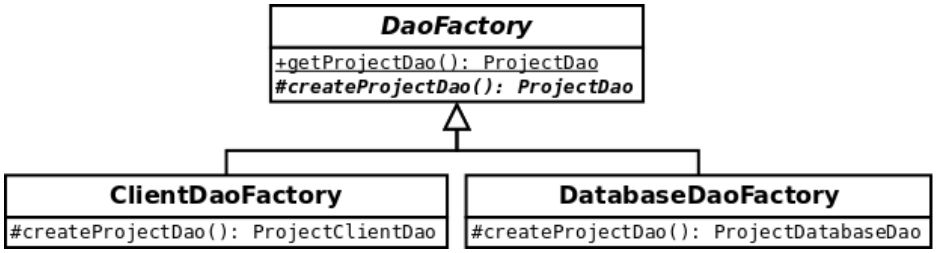


Figura B.17: *Fábricas de objetos de acesso a dados*

O pacote `org.kalibro.client` contém a fábrica `ClientDaoFactory` e os DAOs que ela produz, responsáveis por encaminhar as chamadas para os pontos de acesso de um Kalibro Service remoto. O pacote `org.kalibro.core.persistence` contém a fábrica `DatabaseDaoFactory`, classes auxiliares e os DAOs que ela produz, responsáveis por processar os dados na própria máquina, usando o banco de dados quando necessário persisti-los.

Qualquer chamada a algum método de acesso a dados na máquina cliente dispara a mesma chamada no servidor, a diferença está no arquivo de preferências de cada máquina. Observe o diagrama de sequência da Figura B.18. O método `save()` do projeto dispara o método `save(project)` usando a interface `ProjectDao`. Se estivesse na máquina servidora, a chamada concreta seria feita em um `ProjectDatabaseDao` e o projeto seria persistido no banco de dados. Mas na máquina cliente, essa chamada é feita em um `ProjectClientDao`, que prepara uma requisição ao serviço Web. No lado servidor, a implementação do ponto de acesso recebe a requisição e a transforma de novo em uma chamada usando a interface `ProjectDao`. Dessa forma, as entidades podem usar os DAOs sem precisar saber se a funcionalidade será fornecida diretamente ou através do Kalibro Service.

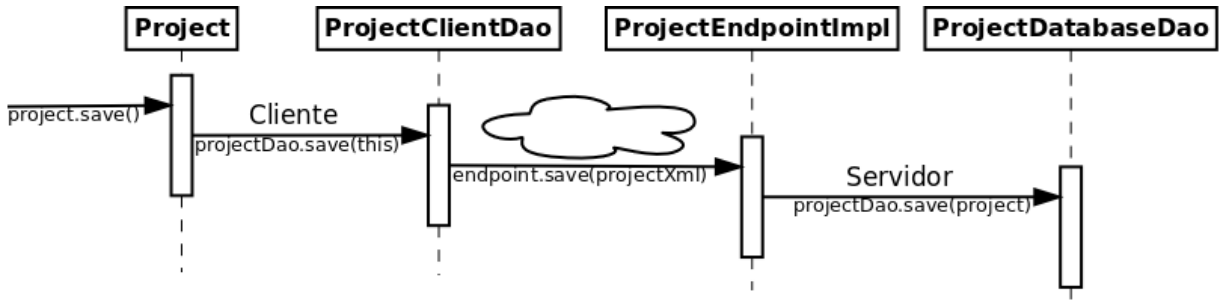


Figura B.18: *Fluxo de um acesso remoto*

Note que as entidades precisam ser representadas em XML para serem transferidas para o serviço e de volta para o cliente. Chegando no destino, a representação XML precisa ser convertida em uma entidade equivalente à original para manipulação. Da mesma forma, as entidades, para serem persistidas, precisam ser representadas como registros na base de dados. Para recuperar as entidades, os registros persistidos precisam ser convertidos de volta para entidades equivalentes às originais.

O Kalibro armazena os dados de forma persistente usando Eclipselink[6], implementação de referência da JPA[7], especificação oficial para arcabouços de mapeamento objeto-relacional em Java. Assim como no JAX-WS classes e atributos são mapeados em elementos XML, as anotações da JPA mapeiam classes a tabelas e atributos a colunas no banco de dados.

Os objetos de transferência de dados, ou DTOs, representam os dados das entidades em formato XML ou relacional, e são a ponte de comunicação entre o Kalibro e esses arcabouços (JPA e JAX-WS). Todos os DTOs descendem das classes abstratas presentes no pacote `org.kalibro.dto`. Os objetos XML encontram-se no pacote `org.kalibro.service.xml` e os registros encontram-se no pacote `org.kalibro.core.persistence.record`.

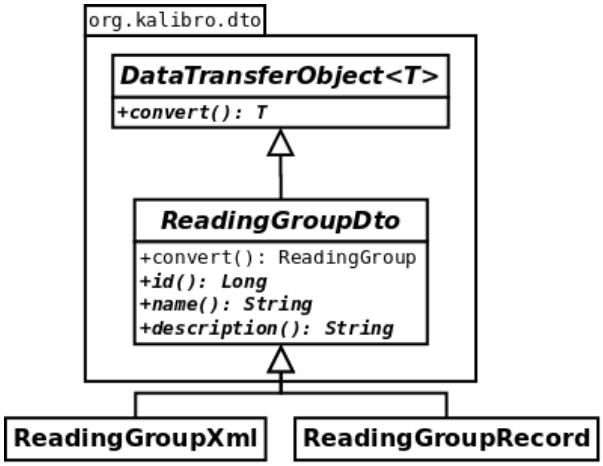


Figura B.19: *Objeto abstrato de transferência*

A classe abstrata `DataTransferObject` é superclasse de todos os objetos de transferência, possuindo alguns métodos utilitários e especificando apenas a operação `convert()` que deve recuperar a entidade, equivalente à original. Diretamente abaixo dela estão DTOs abstratos para cada tipo de entidade (veja o exemplo dos grupos de leitura na Figura B.19). Os DTOs abstratos exigem das subclasses o fornecimento de cada atributo necessário para recriar a entidade, e implementam a conversão geral com base neles - padrão de projeto *Template Method*. Essa arquitetura promove o desacoplamento entre as diferentes representações de dados e abstrai o que há de comum nas suas conversões.

[6] http://www.eclipse.org/eclipselink/
[7] http://www.oracle.com/technetwork/articles/javaee/jpa-137156.html

Cada DTO concreto deve ter:

- Atributos anotados para manter os dados na representação mais conveniente para o contexto.
- Um construtor público sem argumentos, usado pelo arcabouço.
- Um construtor que toma uma entidade como argumento, preenchendo seus atributos com os valores correspondentes aos atributos da entidade.
- Implementação dos métodos que convertem cada um de seus atributos de volta ao formato da entidade.

Muitas vezes a representação de transferência não possui todos os dados da entidade, apenas a informação necessária para obtê-los. Por exemplo, um `ReadingGroupXml` não traz as leituras do grupo, apenas os atributos diretos mostrados na Figura B.19, mas a partir do `id` do grupo é possível obter suas leituras usando a interface `ReadingDao`. Quando um grupo de leituras é criado a partir de um DTO, um *proxy* é colocado no lugar do conjunto de leituras. A classe utilitária `DaoLazyLoader` é uma fábrica desses *proxies* para carregamento preguiçoso, usada pelos DTOs abstratos. O objeto que o *proxy* substitui é carregado em seu primeiro uso, utilizando a chamada especificada na sua criação. Os parâmetros necessários para criar um *proxy* são a interface DAO, o método e os argumentos para a chamada.

Para criação de *proxies*, o Kalibro usa a biblioteca CGLIB[8] que torna possível a extensão de classes e implementação de interfaces Java em tempo de execução. CGLIB é usada for vários arcabouços populares, como Hibernate[9], Spring[10] e Powermock[11].

Os bancos de dados com os quais o Kalibro pode trabalhar são Mysql[12], PostgreSQL[13] e SQLite[14], e a escolha é feita através do arquivo de preferências. No diretório `META-INF` do projeto `KalibroCore` encontram-se scripts de criação do banco de dados, um para cada tipo. O Kalibro recomenda o PostgreSQL versão 9.1 ou superior, mas adicionar suporte a outro banco de dados envolve passos simples:

1. Adicionar uma biblioteca com *driver* de conexão JDBC para o novo banco de dados.
2. Adicionar uma constante no tipo enumerado `SupportedDatabase`, que especifica o *driver* JDBC utilizado - a única mudança no código necessária.
3. Adicionar um script análogo aos existentes para criação das tabelas no novo banco de dados.

Os valores reais são persistidos como inteiros longos. Esses inteiros são a sequência de *bits* que compõem um número de ponto flutuante de precisão dupla em Java (`Double`). Essa conversão é necessária pois os valores reais nos bancos de dados em geral não permitem os valores especiais ∞, $-\infty$ e NaN (*Not a Number*).

B.3.2 Abstração das entidades

O pacote `org.kalibro.core.abstractentity` possui a classe abstrata `AbstractEntity` e suas auxiliares, e é dedicado a minimizar os esforços para criar, manter e testar entidades. A classe `AbstractEntity` é a superclasse de todas as entidades da API do Kalibro, fornecendo a elas os seguintes métodos (ver Figura B.20):

[8] http://cglib.sourceforge.net/
[9] http://www.hibernate.org/
[10] http://www.springsource.org/
[11] http://code.google.com/p/powermock/
[12] http://www.mysql.com/
[13] http://www.postgresql.org/
[14] http://www.sqlite.org/

- `importFrom(file,entityType)` Método de classe que recebe um arquivo e recupera a entidade impressa nele. Assume que o arquivo está no formato YAML. Não é um método público, mas funciona como auxiliar para qualquer entidade que precise fornecer esta funcionalidade, como `ReadingGroup` e `Configuration` (vistas na seção B.1.2).

- `exportTo(file)` Oposto da anterior, imprime no arquivo uma representação YAML da entidade. Também só é público nas subclasses interessadas em fornecer a funcionalidade de exportação para arquivos. O método `save()` da classe `KalibroSettings` utiliza esse método internamente.

- `toString()` Sobrescreve o método de mesmo nome da classe `Object`, oferecendo uma forma padrão de representar textualmente a entidade. Sem surpreender, esse método devolve a mesma representação YAML impressa no método anterior. Esse formato é conveniente por ser compacto e legível, e a presença desse método facilita sessões de teste e depuração.

- `hashCode()` Sobrescreve o método da classe `Object`, produzindo um código de *hash* para a entidade. Esse código é usado para guardar a entidade em coleções baseadas em tabelas de *hash*.

- `equals(other)` Sobrescreve o método da classe `Object`, testando a equivalência da entidade com outro objeto.

- `deepEquals(other)` Testa se a entidade é profundamente equivalente a outra. Em geral, isso significa testar a igualdade em todos os seus atributos, e recursivamente em atributos que sejam entidades ou coleções de entidades. Esse método foi pensado para facilitar comparações completas em testes automatizados.

- `compareTo(other)` Implementa o método de mesmo nome da interface `Comparable`, comparando a entidade com o argumento para ordenação. Devolve um inteiro negativo, nulo ou positivo para sinalizar que a entidade é menor, igual ou maior que o argumento.

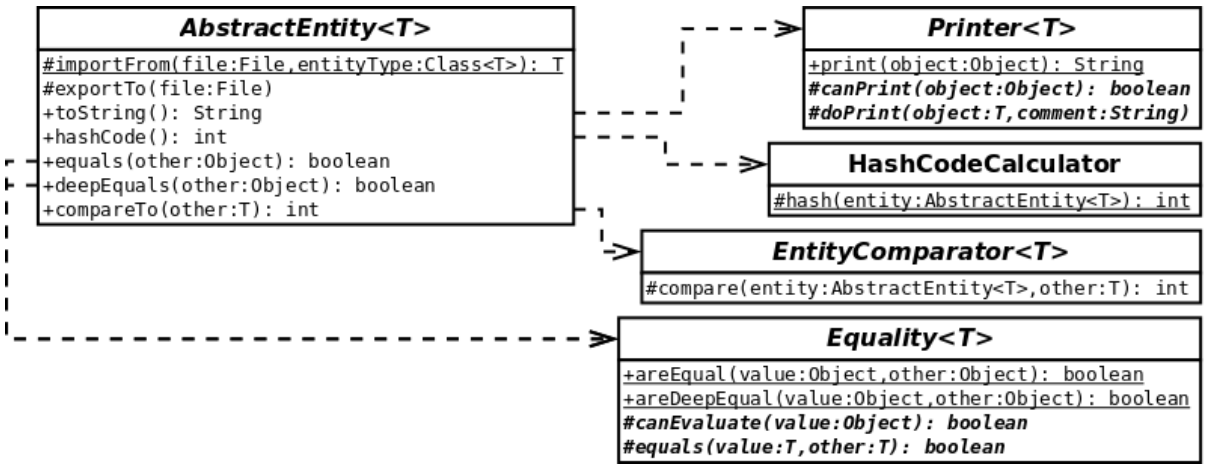


Figura B.20: *Abstração de entidade e auxiliares*

Para que a classe `AbstractEntity` possa realizar todas essas operações sem conhecer de antemão cada uma de suas subclasses, é necessário usar reflexão. Todas as operações citadas usam os atributos da entidade reflexivamente, capturando metadados - quando necessários e disponíveis - na forma de anotações definidas no pacote `org.kalibro.core.abstractentity`:

- `IdentityField` Marca um atributo como parte da identidade do objeto, e é usado pelos métodos `equals()` e `hashCode()`. Por exemplo, na classe `Repository`, o atributo `name` é o único marcado como identificador. Portanto, dois repositórios são considerados equivalentes e produzem o mesmo código de `hash` se tiverem nomes iguais. Como repositórios não existem

sem estar em um projeto, e este possui um conjunto de repositórios, uma consequência é que não pode haver dois repositórios com o mesmo nome dentro de um projeto.

- `Ignore` Marca um atributo para ser ignorado na impressão e em comparações. O método `deepEquals()` funciona como o método `equals()`, mas é recursivo e usa todos os atributos não-ignorados na comparação, ao invés de apenas os marcados como identificadores.

- `SortingFields` É usada na declaração de classe e especifica a lista dos atributos a serem usados na ordenação de coleções da entidade. As comparações ocorrem na ordem em que os atributos são listados, e cada atributo da lista só é comparado se todos os anteriores derem empate. Todos os campos listados devem ser comparáveis.

- `Print` Permite personalizar a impressão de um atributo, adicionando comentários, especificando a ordem em que os campos devem ser impressos ou marcando um atributo para não ser impresso.

Como mostrado na Figura B.20, a classe `AbstractEntity` não implementa toda essa funcionalidade sozinha, mas se utiliza das seguintes classes auxiliares:

- `HashCodeCalculator` Utiliza os campos identificadores para calcular códigos de *hash* com boa distribuição.

- `EntityComparator` Compara os atributos de ordenação para classificar as entidades.

- `Equality` Determina se dois objetos são equivalentes, tanto no caso normal como no profundo. É uma classe abstrata e tem uma subclasse para cada tipo especial de atributo. Cada subclasse deve implementar os métodos `canEvaluate(value)`, que determina se o tipo do objeto é apropriado para ela avaliar, e `equals(value,other)`, que testa a equivalência. A cada requisição de comparação, percorre as subclasses procurando a mais apropriada para fazer a avaliação.

- `Printer` Funciona de forma similar à anterior, sendo abstrata e procurando entre suas subclasses a mais apropriada. Mas tem a peculiaridade de armazenar toda a impressão em um buffer compartilhado, e as subclasses devem usar métodos auxiliares para imprimir nele.

As classes auxiliares, por sua vez, usam operações básicas de reflexão presentes no pacote `org.kalibro.core.reflection`, que também são usadas pontualmente em outras partes do Kalibro.

B.3.3 Concorrência

Concorrência aumenta consideravelmente a complexidade de um programa, porque aumenta o número de estados pelos quais o programa pode passar. Suponha uma classe feita para desempenhar determinada computação, e para isso, dispara novas *threads*[15] e espera por condições envolvendo semáforos que outras *thread* devem modificar. Essa classe é mais difícil de compreender, testar e modificar por causa do tratamento da concorrência. Fundamentalmente essa classe realiza duas funções: a computação para qual foi projetada e gerenciar a execução das *threads*, violando portanto o princípio de responsabilidade única, ou SRP (Martin, 2002).

Para minimizar esse problema, encapsulamos a complexidade do gerenciamento de concorrência no pacote `org.kalibro.core.concurrent`. As classes públicas desse pacote estão representadas na Figura B.21.

A classe abstrata `Task` modela uma tarefa, e o método `compute()` deve ser implementado nas subclasses para executar a tarefa e devolver seu resultado, quando existente. De posse de uma tarefa, é possível executá-la de formas diversas, usando seus métodos:

[15] Um processo Java pode ser dividido em *threads*, tarefas que podem ser executadas concorrentemente. A palavra inglesa significa "fio", e um termo comum é "linha de execução".

- `executeInBackground()` Executa a tarefa de forma assíncrona, ou seja, dispara a execução da tarefa sem esperar pelo seu término.
- `execute()` Executa a tarefa de forma síncrona, esperando pelo seu resultado.
- `execute(timeout)` Executa a tarefa de forma síncrona, mas espera somente o tempo especificado no argumento. Se a tarefa ainda estiver em andamento quando o tempo expirar, ela é interrompida e uma exceção é lançada.
- `executePeriodically(period)` Executa a tarefa de forma assíncrona e agendada periodicamente. Por exemplo, se o período passado como argumento for 2 horas, a execução da tarefa inicia no momento da chamada, outra vez 2 horas depois, outra vez 4 horas depois e assim por diante.
- `cancelExecution()` Cancela a execução da tarefa se agendada, ou a interrompe se em andamento.

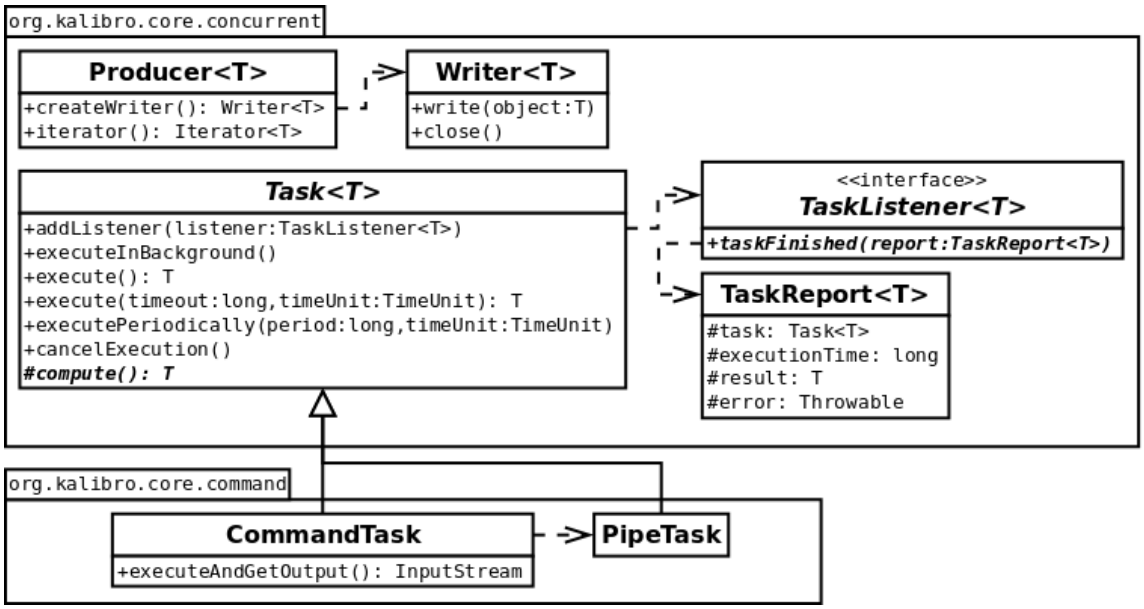


Figura B.21: *Classes que gerenciam concorrência*

A obtenção do resultado de uma tarefa executada de forma assíncrona é feita usando o padrão de projeto *Observer*. O método `Task.addListener(listener)` registra um ouvinte à tarefa. Esses ouvintes são implementações da interface `TaskListener`, e seu método `taskFinished(taskReport)` é chamado quando a tarefa termina, passando como argumento um relatório com dados da execução. Esse relatório é modelado pela classe `TaskReport`, que contém os seguintes atributos:

- `task` A tarefa executada, pois um ouvinte pode ser registrado em várias tarefas.
- `executionTime` O tempo em milissegundos entre antes e depois da execução da tarefa.
- `result` O resultado da tarefa, se ela terminou normalmente.
- `error` A exceção lançada durante a execução da tarefa, caso exista.

As classes `Producer` e `Writer` coordenam tarefas que seguem o paradigma produtor - consumidor, gerenciando o *buffer*[16] entre elas. A classe `Writer` tem apenas os métodos `write(object)` e `close()`, para que a tarefa produtora escreva objetos e sinalize o término da escrita. A classe

[16]Região da memória usada para armazenar dados temporariamente, enquanto estão sendo transferidos de um lugar para outro. Usada quando a taxa de escrita pode ser superior à taxa de leitura.

`Producer` é iterável, ou seja, os objetos produzidos podem ser obtidos em um laço simples como se estivessem em uma coleção. A Listagem B.2 mostra um exemplo de tarefa consumidora disparando a execução de tarefas produtoras e imprimindo os objetos por elas produzidos. O laço da linha 4 fica em espera enquanto não houver objetos escritos e termina quando todos os escritores associados forem fechados.

```
1 Producer<String> producer = new Producer<String>();
2 new ProducerTask(producer.createWriter()).executeInBackground();
3 new ProducerTask(producer.createWriter()).executeInBackground();
4 for (String produced : producer)
5     print(produced);
```

Listagem B.2: *Código-exemplo de consumidor criando produtores*

Uma tarefa comum é realizar chamadas de sistema. A classe `CommandTask`, única classe pública do pacote `org.kalibro.core.command` recebe um comando na sua construção e pode executá-lo das várias formas que sua superclasse (`Task`) disponibiliza. Quando uma chamada de sistema é executada, o processo gerado pode produzir dois tipos de saída: uma no *buffer* normal e outra no *buffer* de erros. As saídas são escritas em arquivos de registro para salvar as informações dos comandos executados e evitar transbordamento dos *buffers*. As saídas dos processos são redirecionadas para os arquivos assincronamente, em sua própria tarefa, `PipeTask`. Alternativamente, a saída normal pode ser capturada por quem dispara a execução do comando (método `executeAndGetOutput`); nesse caso, somente a saída de erros é registrada.

B.3.4 Processamento

O processamento de repositórios é realizado em 3 fases: carregamento, coleta e análise. Na fase de carregamento, o código-fonte é copiado para a máquina servidora do Kalibro, a partir de seu repositório. Na coleta, as ferramentas base necessárias são executadas para fornecer as métricas nativas especificadas na configuração. A análise utiliza todas as informações obtidas na fase de coleta para gerar a árvore de resultados, com todas as métricas compostas, valores agregados e associações com os intervalos configurados.

O carregamento é feito através de chamadas de sistema. O suporte aos diferentes tipos de repositórios depende da instalação da ferramenta nativa na máquina. Cada tipo de repositório possui um carregador correspondente no pacote `org.kalibro.core.loaders`, e a convenção estabelece que o nome do carregador é igual ao nome do tipo de repositório mais o sufixo `Loader`. As características comuns a diferentes carregadores foram abstraídas em carregadores abstratos. A hierarquia de carregadores está representada na Figura B.22.

A superclasse de todos os carregadores é a classe abstrata `Loader`, e seus métodos seguem o padrão de projeto *Template Method*. O método `validate()` executa os comandos devolvidos por `validationCommands()` para validar o suporte ao tipo de repositório. Caso algum dos comandos de validação não execute com sucesso, o tipo correspondente não aparece no conjunto devolvido por `Repository.supportedTypes()`. Quem realmente carrega o código-fonte é o método `load(address,loadDirectory)`, executando os comandos devolvidos por `loadCommands(address,update)`. O segundo argumento define se os comandos desejados são de carregamento inicial ou atualização. A subclasse é responsável por dizer se o diretório é atualizável através do método `isUpdatable(directory)`.

As duas subclasses de `Loader` abstraem carregadores para os tipos de repositório que são sistemas de controle de versão (`RepositoryLoader`) ou simples arquivos/diretórios (`FileLoader`).

Para sistemas de controle de versão, diretórios atualizáveis são aqueles que possuem subdiretório com metadados do sistema (por exemplo, `.svn` para Subversion). Os sistemas com os quais o

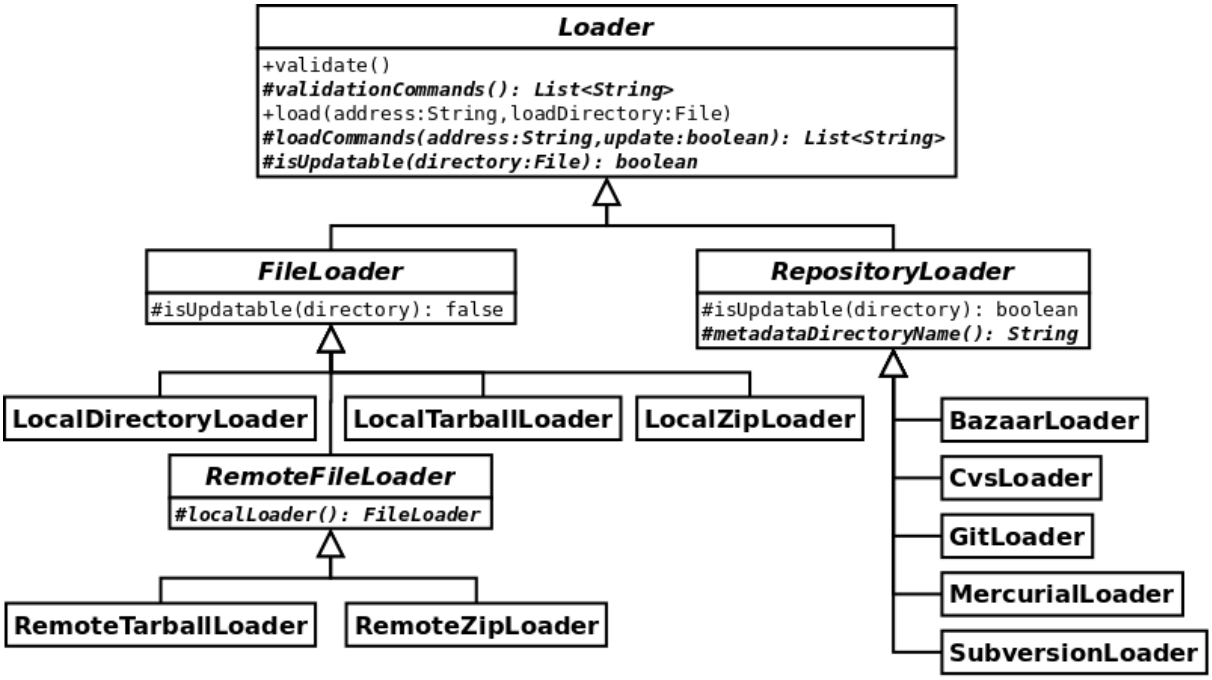


Figura B.22: *Hierarquia de carregadores*

Kalibro trabalha atualmente são Bazaar[17], CVS[18], Git[19], Mercurial[20] e Subversion[21].

Código-fonte obtido de arquivos ou diretórios precisam ser recopiados inteiramente, por isso o método `isUpdatable(directory)` da classe `FileLoader` sempre devolve `false`. A classe `LocalDirectoryLoader` copia o código-fonte de um diretório na máquina local usando o comando `cp`. As classes `LocalTarballLoader` e `LocalZipLoader` extraem o código-fonte de arquivos compactados na máquina local usando os comandos `tar` e `zip`, respectivamente. Os carregadores de arquivos remotos possuem uma superclasse comum, `RemoteFileLoader`, que utiliza o comando `wget` para baixar o arquivo e reutiliza os carregadores locais especificados pelo método `localLoader()` para extrair o código-fonte.

A principal vantagem dessa arquitetura é a facilidade em adicionar suporte a novos tipos de repositórios, pois para tal basta criar um novo carregador, que vai precisar basicamente de um comando de validação, um comando de carregamento e um de atualização. Outra vantagem é a relativa independência do Kalibro em relação a sistemas de controle de versão, pois nenhuma biblioteca é utilizada com essa finalidade, e não obriga a instalação dos sistemas para os quais o usuário não requer suporte: apenas os carregadores validados são utilizados. A desvantagem é a menor portabilidade, pois os carregadores utilizam comandos pré-estabelecidos, que podem não corresponder à utilização correta da ferramenta em diferentes sistemas operacionais. Porém, a tendência das ferramentas é ter o mesmo formato de chamada nos sistemas operacionais baseados em Unix[22], para os quais o Kalibro foi desenhado.

O pacote `org.kalibro.core.processing` possui as classes responsáveis pelo processamento de repositórios. Sua classe principal é a tarefa `ProcessTask`. Ela e suas auxiliares (invisíveis fora do pacote) estão representadas na Figura B.23.

As fases do processamento são realizadas por subtarefas, coordenadas por `ProcessTask`, a tarefa principal. A subtarefa `LoadingTask` prepara o diretório para carregamento do repositório e executa o carregador correto passando o diretório como parâmetro. A subtarefa `CollectingTask`

[17] http://bazaar.canonical.com/en/
[18] http://cvs.nongnu.org/
[19] http://git-scm.com/
[20] http://mercurial.selenic.com/
[21] http://subversion.apache.org/
[22] http://www.unix.org/

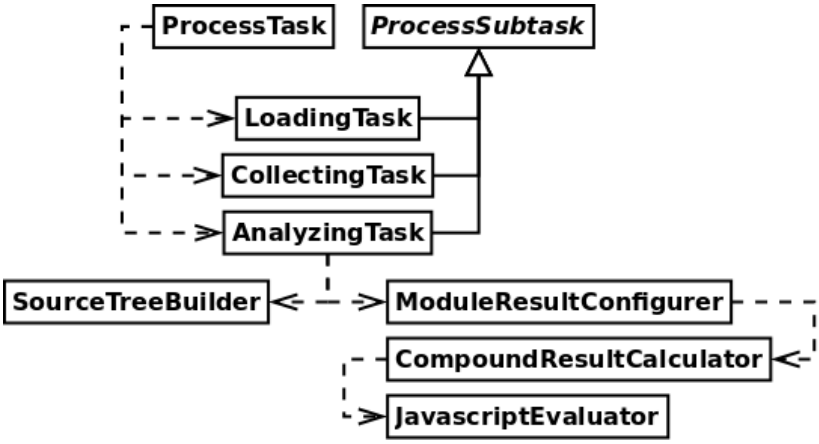


Figura B.23: *Tarefas de processamento*

separa as métricas da configuração por ferramenta base e executa cada uma delas, criando e passando como parâmetro um `Writer` para cada coletor (ver seção B.3.3). A análise é realizada pela subtarefa `AnalyzingTask`, concorrentemente à coleta, e consiste de duas fases. Primeiramente, os resultados produzidos pela subtarefa anterior são consumidos e usados para construir a árvore de resultados (usando a classe `SourceTreeBuilder`). Posteriormente, a árvore é percorrida de baixo pra cima (ordem decrescente de altura) para acumular os valores dos nós descendentes para os nós ancestrais. A cada nó percorrido, a classe `ModuleResultConfigurer` adiciona as métricas compostas e calcula a nota do módulo.

Referências Bibliográficas

Abrahamsson *et al.***(2002)** Pekka Abrahamsson, Outi Salo, Jussi Ronkainen, e Juhani Warsta. Agile software development methods. Relatório técnico, VTT Technical Research Center of Finland. Citado na pág. 7

Akaike(1974) H. Akaike. A new look at the statistical model identification. *Automatic Control, IEEE Transactions on*, 19(6):716–723. ISSN 0018-9286. doi: 10.1109/tac.1974.1100705. URL http://dx.doi.org/10.1109/tac.1974.1100705. Citado na pág. 31, 43

Albrecht e Gaffney(1983) A. J. Albrecht e J. E. Gaffney. Software Function, Source Lines of Code, and Development Effort Prediction: A Software Science Validation. *IEEE Transactions on Software Engineering*, 9(6):628–648. Citado na pág. 95, 102

Arthur(1985) Lowell Jay Arthur. *Measuring Programmer Productivity and Software Quality.* Wiley-Interscience, New York. Citado na pág. 1, 13

Azzalini e Arellano-Valle(2013) Adelchi Azzalini e Reinaldo B. Arellano-Valle. Maximum penalized likelihood estimation for skew-normal and skew-t distributions. *Journal of Statistical Planning and Inference*, 143(2):419 – 433. ISSN 0378-3758. doi: 10.1016/j.jspi.2012.06.022. URL http://www.sciencedirect.com/science/article/pii/S0378375812002352. Citado na pág. 38

Baker e Zweben(1980) A. L. Baker e S. H. Zweben. A Comparison of Measures of Control Flow Complexity. *IEEE Transactions on Software Engineering*, 6(6):506–512. Citado na pág. 103

Baldwin e Clark(2006) Carliss Y. Baldwin e Kim B. Clark. The architecture of participation: Does code architecture mitigate free riding in the open source development model? *Management Science*, 52(7):1116–1127. Citado na pág. 15

Balijepally *et al.***(2009)** Venugopal Balijepally, Radha K. Mahapatra, Sridhar P. Nerur, e Kenneth Price. Are two heads better than one for software development? the productivity paradox of pair programming. *MIS Quarterly*, 33(1):91–118. URL http://aisel.aisnet.org/misq/vol33/iss1/7/. Citado na pág. 11

Bansiya e Davi(1997) J. Bansiya e C. Davi. Automated Metrics and Object-Oriented Development: Using QMOOD++ for Object-Oriented Metrics. *Dr. Dobb's Journal*, 22(12):42, 44–48. Citado na pág. 17, 105, 106, 108

Bansiya e Davis(2002) J. Bansiya e C.G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Transactions on Software Engineering*, 28(1):4–17. ISSN 0098-5589. doi: http://doi.ieeecomputersociety.org/10.1109/32.979986. Citado na pág. 2

Barbagallo *et al.***(2008)** Donato Barbagallo, Chlara Francalenei, e Francesco Merlo. The Impact of Social Networking on Software Design Quality and Development Effort in Open Source Projects. Em *ICIS 2008 Proceedings*. URL http://aisel.aisnet.org/icis2008/201. Citado na pág. 22

Barkmann *et al.***(2009)** Henrike Barkmann, Rüdiger Lincke, e Welf Löwe. Quantitative Evaluation of Software Quality Metrics in Open-Source Projects. Em *AINA Workshops*, páginas 1067–1072. Citado na pág. 22

Basili e Rombach(1987) V. R. Basili e H. D. Rombach. TAME: Integrating Measurement Into Software Environments. Relatório Técnico TR-1764, University of Maryland, Computer Science Department, University of Maryland, Computer Science Department. Citado na pág. 101

Basili e Turner(1975) V. R. Basili e A. J. Turner. Iterative Enhancement: A Practical Technique for Software Development. *IEEE Transactions on Software Engineering*, 4(1):390–396. Citado na pág. 103

Basili *et al.***(1996)** V. R. Basili, L. C. Briand, e W. L. Melo. A Validation of Object-Oriented Design Metrics as Quality Indicators. *IEEE Transactions on Software Engineering*, 10(22):751–761. doi: http://dx.doi.org/10.1109/32.544352. Citado na pág. 15, 106, 107

Baxter *et al.***(2006)** Gareth Baxter, Marcus Frean, James Noble, Mark Rickerby, Hayden Smith, Matt Visser, Hayden Melton, e Ewan Tempero. Understanding the shape of java software. Em *Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, OOPSLA '06, páginas 397–412, New York, NY, USA. ACM. ISBN 1-59593-348-4. doi: 10.1145/1167473.1167507. URL http://doi.acm.org/10.1145/1167473.1167507. Citado na pág. 23, 24

Beck(2007) Kent Beck. *Implementation Pattens*. Addison Wesley. Citado na pág. 15

Beck(1997) Kent Beck. *Smalltalk: best practice patterns*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA. ISBN 0-13-476904-X. Citado na pág. 17

Bellin *et al.***(1999)** D. Bellin, M. Tyagi, e M. Tyler. Object Oriented Metrics: An Overview. Web Publication, 1999. Citado na pág. 107, 108

Benkler(2006) Yochai Benkler. *The Wealth of Networks: How Social Production Transforms Markets And Freedom*. Yale University Press. Citado na pág. 1

Bieman e Ott(1994) J. M. Bieman e L. M. Ott. Measuring Functional Cohesion. *IEEE Transactions on Software Engineering*, 8(20):254–259. Citado na pág. 103

Boehm(1981) B. W. Boehm. *Software Engineering Economics*. Prentice-Hall, N. J. Citado na pág. 102, 104

Boehm *et al.***(1978)** B. W. Boehm, J. R. Brown, e J. R. Kaspar. *Characteristics of Software Quality (TRW Series of Software Technology*. Elsevier, 1st edition edição. Citado na pág. 103

Briand *et al.***(1997)** Lionel Briand, Prem Devanbu, e Walcelio Melo. An investigation into coupling measures for C++. Em *ICSE '97: Proceedings of the 19th international conference on Software engineering*, páginas 412–421, New York, NY, USA. ACM. ISBN 0-89791-914-9. doi: 10.1145/253228.253367. URL http://dx.doi.org/10.1145/253228.253367. Citado na pág. 15

Briand *et al.***(1999)** Lionel C. Briand, Sandro Morasca, e Victor R. Basili. Defining and validating measures for object-based high-level design. *IEEE Trans. Softw. Eng.*, 25(5):722–743. ISSN 0098-5589. doi: 10.1109/32.815329. URL http://dx.doi.org/10.1109/32.815329. Citado na pág. 15

Briand *et al.***(2000)** Lionel C. Briand, Jürgen Wüst, John W. Daly, e D. Victor Porter. Exploring the relationship between design measures and software quality in object-oriented systems. *J. Syst. Softw.*, 51(3):245–273. ISSN 0164-1212. doi: 10.1016/S0164-1212(99)00102-8. URL http://dx.doi.org/10.1016/S0164-1212(99)00102-8. Citado na pág. 15

Brito e Abreu e Carapuça(1994) Fernando Brito e Abreu e Rogério Carapuça. Candidate Metrics for Object Oriented Software Within A Taxonomy Framework. *Journal of Systems and Software*, 26(1):87–96. doi: http://dx.doi.org/10.1016/0164-1212(94)90099-X. Citado na pág. 104, 107, 108

Brooks(1975) Frederick Phillips Brooks. *The Mythical Man-Month*. Addison-Wesley. Citado na pág. 13

Capra *et al.***(2008)** E. Capra, C. Francalanci, e F. Merlo. An Empirical Study on the Relationship Between Software Design Quality, Development Effort and Governance in Open Source Projects. *IEEE Transactions on Software Engineering*, 34(6):765–782. ISSN 0098-5589. doi: {10.1109/TSE.2008.68}. Citado na pág. 22

Card e Agresti(1988) D. N. Card e W. W. Agresti. Measuring Software Design Complexity. *Jornal of Systems and Software*, 8(3):185–197. Citado na pág. 14

Castells(2006) Manuel Castells. A Era da Intercomunicação. *Le Monde Diplomatique Brasil*, Agosto 2006. Citado na pág. 7

Chidamber e Kemerer(1991) R. S. Chidamber e C. F. Kemerer. Towards A Metrics Suite for Object-Oriented Design. Em *Proceedings of the OOPSLA 91 Conference*, páginas 197–211. Citado na pág. 15, 104

Chidamber e Kemerer(1994) S. R. Chidamber e C. F. Kemerer. A Metrics Suite for Object-Oriented Design. *IEEE Transactions on Software Engineering*, 20(6):476–493. Citado na pág. 15, 19, 104, 105, 106, 108

Clauset *et al.***(2007)** Aaron Clauset, Cosma R. Shalizi, e M. E. J. Newman. Power-law distributions in empirical data. *SIAM Reviews*. URL http://arxiv.org/abs/0706.1062. Citado na pág. 23, 36, 37

Concas *et al.***(2007)** Giulio Concas, Michele Marchesi, Sandro Pinna, e Nicola Serra. Power-laws in a large object-oriented software system. *IEEE Trans. Softw. Eng.*, 33(10):687–708. Citado na pág. 24

Conte *et al.***(1986)** S. D. Conte, H. E. Dunsmore, e V. Y. Shen. *Software Engineering Metrics and Models*. Benjamin-Cummings, California. Citado na pág. 14

Corbucci(2011) Hugo Corbucci. Métodos ágeis e software livre: um estudo da relação entre estas duas comunidades. Dissertação de Mestrado, Instituto de Matemática e Estatística da Universidade de São Paulo. URL https://github.com/hugocorbucci/dissertacao-mestrado. Citado na pág. 7, 8

Costa(2009) Joenio Costa. Extração de informações de dependência entre módulos de programas c/c++. Undergraduation course conclusion project, Universidade Católica do Salvador. URL http://wiki.dcc.ufba.br/Aside/ProjetoFinalJoenioCosta. Citado na pág. 77

Cousineau *et al.***(2004)** Denis Cousineau, Scott Brown, e Andrew Heathcote. Fitting distributions using maximum likelihood: Methods and packages. *Behavior Research Methods, Instruments, & Computers*, 36(4):742–756. ISSN 0743-3808. doi: 10.3758/BF03206555. URL http://dx.doi.org/10.3758/BF03206555. Citado na pág. 44

Crowston e Scozzi(2002) K. Crowston e B. Scozzi. Open Source Software Projects as Virtual Organizations: Competency Rallying for Software Development. Em *IEE Proceedings Software*, volume 1, páginas 3–17. Citado na pág. 10, 11, 26

Crowston *et al.***(2006)** Kevin Crowston, , e James Howison. Hierarchy and centralization in free and open source software team communications. *Knowledge Technology & Policy*, 18:65–85. Citado na pág. 11

Curtis *et al.***(1979a)** B. Curtis, S. B. Sheppard, e P. Milliman. Third Time Charm: Stronger Prediction of Programmer Performance By Software Complexity Metrics. Em *4th Int. Conf. on Software Engineering*, páginas 356–360, New York. Citado na pág. 101

Curtis *et al.*(**1979b**) B. Curtis, S. B. Sheppard, P. Milliman, M. A. Borst, e T. Love. Measuring the Psychological Complexity of Software Maintenance Tasks With the Halstead and McCabe Metrics. *IEEE Transactons Software Engineering*, 5(2):96–104. Citado na pág. 101

Darcy *et al.*(**2005**) D. P. Darcy, C. F. Kemerer, S. A. Slaughter, e J. E. Tomayko. The Structural Complexity of Software: An Experimental Test. *Software Engineering, IEEE Transactions on*, 31(11):982–995. ISSN 0098-5589. doi: {10.1109/TSE.2005.130}. Citado na pág. 19

del Bianco *et al.*(**2007**) V. del Bianco, M. Chinosi, L. Lavazza, S. Morasca, e D. Taibi. How European Software Industry Perceives OSS Trustworthiness and What Are the Specific Criteria To Establish Trust in OSS. Qualipso report, Qualipso. http://qualipso.semanticdesktop.org/xwiki/bin/view/Wiki/WP51. Citado na pág. 16

del Bianco *et al.*(**2008a**) V. del Bianco, L. Lavazza, S. Morasca, e D. Taib. Definition of Trustworthiness of Software Products and Rtefacts. Qualipso report, Qualipso. Citado na pág. 16

del Bianco *et al.*(**2008b**) V. del Bianco, L. Lavazza, S. Morasca, e D. Taib. Identification of Factors That Influence the Trustworthiness of Software Products and Artefacts. Qualipso report, Qualipso. Citado na pág. 16

del Bianco *et al.*(**2008c**) V. del Bianco, L. Lavazza, S. Morasca, e D. Taibi. Analysis of Relevant Open-Source Projects and Artifacts. Qualipso report, Qualipso. Citado na pág. 16

DeMarco(1982) T. DeMarco. *Controlling Software Projects: Management, Measurement & Estimation*. Yourdon Press, New York. Citado na pág. 102, 104

DiCiccio e Monti(2004) Thomas J DiCiccio e Anna Clara Monti. Inferential aspects of the skew exponential power distribution. *Journal of the American Statistical Association*, 99(466):439–450. doi: 10.1198/016214504000000359. URL http://www.tandfonline.com/doi/abs/10.1198/016214504000000359. Citado na pág. 38

Dunsmore e Gannon(1979) H. E. Dunsmore e J. D. Gannon. Data Referencing: An Empirical Investigation. *IEEE Computer*, 12:50–59. Citado na pág. 103

e Abreu(1994) Fernando Brito e Abreu. MOOD - Metrics for Object-Oriented Design. Em *OOPSLA 94*. Citado na pág. 107

Fenton e Melton(1990) Norman E. Fenton e A. Melton. Deriving Structurally Based Software Measures. *Journal of Systems and Software*, 12:177–187. Citado na pág. 104

Fenton e Neil(1999) Norman E. Fenton e Martin Neil. Software metrics: successes, failures and new directions. *Journal of Systems and Software*, 47(2-3):149 – 157. Citado na pág. 16

Fenton e Pfleeger(1998) Norman E. Fenton e Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. Course Technology, 2 edition edição. Citado na pág. 13, 14, 108

Ferreira *et al.*(**2009**) Kecia Aline M. Ferreira, Mariza A. S. Bigonha, Roberto S. Bigonha, L. Mendes, e H. C. Almeida. Reference Values for Object-Oriented Software Metrics. Em *XXIII Brazilian Symposium on Software Engineering*, volume 1, páginas 62–72. Citado na pág. 16, 24

Fowler(2000) Martin Fowler. The new methodology, July 2000. URL http://martinfowler.com/articles/newMethodologyOriginal.html. (Original version). Citado na pág. 7

Gaffney *et al.*(**1995**) John E. Gaffney, Henry D. Felber, e Richard W. Erling. *The Software Measurement Guidebook (Management Information Systems) (Paperback)*. Software Productivity Consortium: International Thomson Computer Press, January 1995. Citado na pág. 104

Gousios *et al.*(**2007**) Georgios Gousios, Vassilios Karakoidas, Konstantinos Stroggylos, Panagiotis Louridas, Vasileios Vlachos, e Diomidis Spinellis. Software Quality Assesment of Open Source Software. Em *Proceedings of the 11th Panhellenic Conference on Informatics*, May. Citado na pág. 1, 2, 15

Grady e Caswell(1987) R. B. Grady e D. R. Caswell. *Software Metrics: Establishing A Company-Wide Program*. Prentice-Hall, N. J. Citado na pág. 14, 101

Hair *et al.*(**2006**) J. F. Hair, W. C. Black, B. J. Babin, R. E. Anderson, e R. L. Tatham. *Multivariate data analysis*. Pearson Education In, Upper Saddle River, NJ, 6th edição. Citado na pág. 26

Halstead(1972) M. H. Halstead. Natural Laws Controlling Algoritmic Structure? *ACM SIGPLAN Notices*, 7(2):19 – 26. Citado na pág. 102

Halstead(1977) M. H. Halstead. *Elements of Software Science*. Elsevier North-Holland, New York. Citado na pág. 102

Hansen(1978) W.J. Hansen. Measurement of program complexity by the pair (cyclomatic number, operation count). *ACM SIGPLAN Notices*, 3(13):29–33. Citado na pág. 103

Harrison(1998) R. Harrison. An Evaluation of the MOOD Set of Object-Oriented Software Metrics. *IEEE Transactions on Software Engineering*, 24(6):491–496. Citado na pág. 106, 107

Harrison e Cook(1987) W. Harrison e C. Cook. A Micro/Macro Measure of Software Complexity. *Jornal of Systems and Software*, 7(3):213–219. Citado na pág. 14

Hausen(1989) H. L. Hausen. Yet Another Software Quality and Productivity Modeling — YAQUAPMO. Em *Proceedings of the Twenty-Second Annual Hawaii International Conference On System Sciences*, volume 2, páginas 978–987. Citado na pág. 104

Henderson-Sellers(1996) Brian Henderson-Sellers. *Object-Oriented Complexity*. Prentice-Hall. Citado na pág. 2

Henry e Kafura(1981) S. Henry e D. Kafura. Software Structure Metrics Based on Information Flow. *IEEE Transactions on Software Engineering*, 5(7):510–518. Citado na pág. 15, 103

Henry e Kafura(1984) S. Henry e D. Kafura. The Evaluation of Software Systems' Structure Using Quantitative Software Metrics. *Software — Practice and Experience*, 14(6):561–573. Citado na pág. 14

Herraiz *et al.*(**2012**) I. Herraiz, D. Rodriguez, e R. Harrison. On the statistical distribution of object-oriented system properties. Em *Emerging Trends in Software Metrics (WETSoM), 2012 3rd International Workshop on*, páginas 56–62. doi: 10.1109/WETSoM.2012.6226994. Citado na pág. 24

Herraiz *et al.*(**2011**) Israel Herraiz, Daniel M. Germán, e Ahmed E. Hassan. On the distribution of source code file sizes. Em María José Escalona Cuaresma, Boris Shishkov, e José Cordeiro, editors, *ICSOFT (2)*, páginas 5–14. SciTePress. ISBN 978-989-8425-77-5. Citado na pág. 24

Hill(1976) ID Hill. Algorithm as 100: Normal-johnson and johnson-normal transformations. *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, 25(2):190–192. Citado na pág. 44

Hitz e Montazeri(1995) M. Hitz e B. Montazeri. Measuring Coupling and Cohesion in Object-Oriented Systems. Em *Proceedings of International Symposium on Applied Corporate Computing*. Citado na pág. 19

Hsiao e of Toronto. Institute for Policy Analysis(1979) C. Hsiao e University of Toronto. Institute for Policy Analysis. *Causality testing in econometrics*. Working paper series. Institute for Policy Analysis, University of Toronto. URL http://books.google.com.br/books?id=bGPuAAAAMAAJ. Citado na pág. 74

Ince e Hekmatpour(1988) D. C. Ince e S. Hekmatpour. An Approach To Automated Software Design Based on Product Metrics. *Software Engineering Journal*, 3(2):53–56. Citado na pág. 104

ISO/IEC9126-1(2001) ISO/IEC9126-1. Software Engineering — Product Quality — Part 1: Quality Model, 2001. Citado na pág. 2, 13

Johnson(2002) Justin Pappas Johnson. Open source software: Private provision of a public good. *Journal of Economics and Management Strategy*, 11(4):637–662. Citado na pág. 9, 10

Jones(2005) M. C. Jones. The uniform mixture of generalized arc-sine distributions. *Journal of the Korean Statistical Society*, 34(1):35–38. URL http://oro.open.ac.uk/22561/. Citado na pág. 38

Jones e Faddy(2003) M. C. Jones e M. J. Faddy. A skew extension of the t-distribution, with applications. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 65 (1):159–174. ISSN 1467-9868. doi: 10.1111/1467-9868.00378. URL http://dx.doi.org/10.1111/1467-9868.00378. Citado na pág. 38

Jones(1986) T. C. Jones. *Programming Productivity*. McGraw-Hill, New York. Citado na pág. 101, 102

Jones(1991) T. C. Jones. *Applied Software Measurement: Assuring Productivity and Quality*. McGraw-Hill, New York. Citado na pág. 16, 102

Kafura e Canning(1985) D. Kafura e J. Canning. A Validation of Software Metrics Using Many Metrics and Two Resources. Em *8th Intl. Conf. on Software Engineering*, páginas 378–385. IEEE Computer Society Press. Citado na pág. 101

Kafura e Henry(1981) D. Kafura e S. Henry. Software Quality Metrics Based on Interconnectivity. *Journal of Systems and Software*, 2(2):121–131. Citado na pág. 102

Kemerer(1987) C. F. Kemerer. An Empirical Validation of Software Cost Estimation Models. *Communications ACM*, 30(5):416–429. Citado na pág. 101

Kolewe(1993) R. Kolewe. Metrics in Object-Oriented Design and Programming, October 1993. Citado na pág. 104, 105, 107

Lanza e Marinescu(2006) Michele Lanza e Radu Marinescu. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate and Improve the Design of Object-Oriented Systems*. Hardcover. Citado na pág. 16, 24, 41, 54, 66

Li e Cheung(1987) Hon Fung Li e William Kwok Cheung. An Empirical Study of Software Metrics. *IEEE Transactions Software Engineering*, 13(6):697–708. Citado na pág. 14, 101

Li e Henry(1993) Wei Li e Sallie Henry. Object-oriented metrics that predict maintainability. *Journal of Systems and Software*, 23(2):111–122. ISSN 01641212. doi: 10.1016/0164-1212(93)90077-b. URL http://dx.doi.org/10.1016/0164-1212(93)90077-b. Citado na pág. 15

Lorenz(1991) M. Lorenz. Real World Reuse. *Journal of object-oriented programming*, 4(7):35 – 39. Citado na pág. 106

Lorenz e Kidd(1994) M. Lorenz e J. Kidd. *Object-Oriented Software Metrics*. Prentice Hall. Citado na pág. 15, 17, 105, 106, 107, 108

Louridas *et al.*(**2008**) Panagiotis Louridas, Diomidis Spinellis, e Vasileios Vlachos. Power laws in software. *ACM Trans. Softw. Eng. Methodol.*, 18(1):2:1–2:26. ISSN 1049-331X. doi: 10.1145/1391984.1391986. URL http://doi.acm.org/10.1145/1391984.1391986. Citado na pág. 23

Macro e Buxton(1987) A. Macro e J. Buxton. *The Craft of Software Engineering*. Addison-Wesley. Citado na pág. 103

Mahnke *et al.*(**2012**) Reinhard Mahnke, Jevgenijs Kaupu, e Martins Brincs. Power Laws and Skew Distributions. *Communications in Computational Physics*. ISSN 18152406. doi: 10.4208/cicp.010411.050811a. URL http://dx.doi.org/10.4208/cicp.010411.050811a. Citado na pág. 38

Martin(2002) Robert C. Martin. *Agile Software Development: Principles, Patterns and Practices*. Prentice-Hall. Citado na pág. 123, 127

Martin(2008) Robert C. Martin. *Clean Code - A Handbook of Agile Software Craftsmanship*. Prentice Hall. Citado na pág. 15, 123

McCabe e Associates(1994) McCabe e Associates. *McCabe Object-Oriented Tool Users Instructions*, 1994. Citado na pág. 105

McCabe(1976) T. J. McCabe. A Complexity Measure. *IEEE Transactions Software Engineering*, 2(4):308–320. Citado na pág. 18, 102

McCabe(1982) T. J. McCabe. Structured Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric. Relatório técnico, National Bureau of Standards - Special Publication SP 500 - 99. Citado na pág. 18, 102

McCabe e Butler(1989) T. J. McCabe e C. Butler. Design Complexity Measurement and Testing. *Communications of the ACM*, 32:1415–1425. Citado na pág. 18, 102

McCabe *et al.*(**1994**) T. J. McCabe, L. A. Dreyer, e A. H. Watson. Testing An Object-Oriented Application. *Journal of the Quality Assurance Institute*, 8(4):21–27. Citado na pág. 105, 106

Meirelles *et al.*(**2010**) P. Meirelles, C. Santos, J. Miranda, F. Kon, A. Terceiro, e C. Chavez. A study of the relationships between source code metrics and attractiveness in free software projects. Em *Software Engineering (SBES), 2010 Brazilian Symposium on*, páginas 11–20. Citado na pág. 15

Meirelles *et al.*(**2009**) Paulo R. M. Meirelles, Raphael Côbe, Simone Hanazumi, Paulo Nunes, Geiser Challco, Straus Martins, Eduardo Morais, e Fabio Kon. Crab: Uma Ferramenta de Configuração e Interpretação de Métricas de Software para Avaliação da Qualidade de Código. Em *XXIII SBES - Simpósio Brasileiro de Engenharia de Software (XVI Sessão de Ferramentas)*. Citado na pág. 75, 82

Michlmayr *et al.*(**2005**) Martin Michlmayr, Francis Hunt, e David Probert. Quality Practices and Problems in Free Software Projects. Em Marco Scotto e Giancarlo Succi, editors, *First International Conference on Open Source Systems*, páginas 309–310, Genova, Italy. Citado na pág. 8, 22

Midha(2008) Vishal Midha. Does Complexity Matter? The Impact of Change in Structural Complexity On Software Maintenance and New Developers' Contributions in Open Source Software. Em *ICIS 2008 Proceedings*. Citado na pág. 22

Mills(1988) Everald E. Mills. Software Metrics. Relatório técnico, Software Engineering Institute, SEI - Carnegie Mellon University. Citado na pág. 13, 14, 101

Mockus *et al.*(**2000**) Audris Mockus, Roy T. Fielding, e James Herbsleb. A case study of open source software development: the apache server. Em *ICSE '00: Proceedings of the 22nd international conference on Software engineering*, páginas 263–272, New York, NY, USA. ACM. ISBN 1-58113-206-9. doi: http://doi.acm.org/10.1145/337180.337209. Citado na pág. 11

Mohanty(1981) S. N. Mohanty. Software Cost Estimation: Present and Future. *Software-Practice and Experience*, 11(2):103–121. Citado na pág. 101

Morris(1988) K.L. Morris. Metrics for Object-Oriented Software Development Environments. Dissertação de Mestrado, M.I.T. Citado na pág. 104, 106, 107

Myers(1977) G. J. Myers. An Extension To the Cyclomatic Measure of Program Complexity. *ACM SIGPLAN Notices*, 12(10):61–64. Citado na pág. 18, 102

Oviedo(1980) E. I. Oviedo. Control Flow, Data Flow and Program Complexity. Em *Proceedings of the IEEE Computer Software and Applications Conference*, páginas 146–152. Citado na pág. 103

Perlis *et al.***(1981)** A. Perlis, F. Sayward, e M. Shaw. *Software Metrics: An Analysis and Evaluation.* MIT Press, Cambridge, Mass. Citado na pág. 3

Potanin *et al.***(2005)** Alex Potanin, James Noble, Marcus Frean, e Robert Biddle. Scale-free geometry in oo programs. *Commun. ACM*, 48(5):99–103. ISSN 0001-0782. Citado na pág. 23, 24

Potier *et al.***(1982)** D. Potier, J. L. Albin, R. Ferreol, e A. Bilodeau. Experiments With Computer Software Complexity and Reliability. Em *6th Intl. Conf. on Software Engineering*, páginas 94–103, New York. IEEE. Citado na pág. 101

Raja e Tretter(2006) Uzma Raja e Marietta J. Tretter. Investigating open source project success: A data mining approach to model formulation, validation and testing. Relatório Técnico Paper-071-31, Working Paper, Texas A&M University, College Station, Texas. Citado na pág. 10, 11

Raymond(1999) Eric Steven Raymond. *The Cathedral & the Bazaar.* O'Reilly & Associates, Inc., Sebastopol, CA, USA. Citado na pág. 1, 5, 6, 8

Richter(1999) Charles Richter. *Designing Flexible Object-Oriented Systems with UML.* New Riders Publishing, Thousand Oaks, CA, USA. ISBN 1578700981. Citado na pág. 19

Rigby e Stasinopoulos(2006) Robert A Rigby e D Mikis Stasinopoulos. Using the box-cox t distribution in gamlss to model skewness and kurtosis. *Statistical Modelling*, 6(3):209–229. doi: 10.1191/1471082X06st122oa. URL http://smj.sagepub.com/content/6/3/209.abstract. Citado na pág. 41

Robles *et al.***(2006)** Gregorio Robles, Jesus M. Gonzalez-Barahona, Martin Michlmayr, e Juan Jose Amor. Mining Large Software Compilations over Time: Another Perspective of Software Evolution. Em *Proceedings of the International Workshop on Mining Software Repositories (MSR 2006)*, Shanghai, China. Citado na pág. 24, 25

Rocha *et al.***(2001)** A. R. C. Rocha, J. C. Maldonado, e K. C. Weber. *Qualidade De Software — Teoria E Prática.* Prentice Hall, São Paulo. Citado na pág. 2

Rombach(1987) H. D. Rombach. A Controlled Experiment on the Impact of Software Structure on Maintainability. *IEEE Transaction Software Engineering*, 13(3):344–354. Citado na pág. 101

Rosenberg e Hyatt(1997) L. H. Rosenberg e L. E. Hyatt. Software Quality Metrics for Object-Oriented Environments. *Crosstalk - the Journal of Defense Software Engineering*, 10. Citado na pág. 18, 107

Rubin(1987) H. A. Rubin. A Comparison of Software Cost Estimation Tools. *System Development*, 7(5):1–3. Citado na pág. 101

Santos *et al.***(2013)** Carlos Santos, George Kuk, Fabio Kon, e John Pearson. The attraction of contributors in free and open source software projects. *The Journal of Strategic Information Systems*, 22(1):26 – 45. ISSN 0963-8687. doi: 10.1016/j.jsis.2012.07.004. URL http://www.sciencedirect.com/science/article/pii/S0963868712000340. <ce:title>Service Management and Engineering in Information Systems Research</ce:title>. Citado na pág. 71

Santos Jr. *et al.***(2010)** Carlos Santos Jr., John Pearson, e Fabio Kon. Attractiveness of Free and Open Source Software Projects. Em *Proceedings of the 18th European Conference on Information Systems (ECIS)*, Pretoria, South Africa. (forthcoming). Citado na pág. xiii, 8, 9, 10, 11

Sato *et al.***(2007)** Danilo Sato, Alfredo Goldman, e Fabio Kon. Tracking the Evolution of Object Oriented Quality Metrics. Em *Proceedings of the 8th International Conference on Extreme Programming And Agile Processes in Software Engineering (XP 2007)*, páginas 84–92. Citado na pág. 2

Sauer(2005) Frank Sauer. Metrics - Eclipse Metrics Plugin. Web Site, 2005. URL http://metrics.sourceforge.net/. Citado na pág. 79

Schneidewind e Hoffmann(1979) N. F. Schneidewind e H. Hoffmann. An Experiment in Software Error Data Collection and Analysis. *IEEE Transactions on Software Engineering*, 3(5): 276–286. Citado na pág. 103

Shaikh e Cornford(2003) Maha Shaikh e Tony Cornford. Version management tools: Cvs to bk in the linux kernel. *Long Range Planning*, 34:699–725. Citado na pág. 11

Sharble e Cohen(1993) R. Sharble e S. Cohen. The Object-Oriented Brewery: A Comparison of Two Object-Oriented Development Methods. *Software Engineering Notes*, 18(2):60–73. Citado na pág. 18, 105

Shih *et al.***(1997)** T.K. Shih, Chi-Ming Chung, Chun-Chia Wang, e W.C. Pai. Decomposition of Inheritance Hierarchy DAGs for Object-Oriented Software Metrics. Em *Workshop on Engineering of Computer-Based Systems (ECBS 97)*, página 238. Citado na pág. 107

Shull e Feldmann(2008) Forrest Shull e Raimund L. Feldmann. Building theories from multiple evidence sources. Em Forrest Shull, Janice Singer, e Dag I. K. Sjøberg, editors, *Guide to Advanced Empirical Software Engineering*, páginas 337–364. Springer London. ISBN 978-1-84800-044-5. doi: 10.1007/978-1-84800-044-5_13. Citado na pág. 96

Simon e Vieira(2008) Imre Simon e Miguel Said Vieira. O rossio não rival, 2008. URL http://www.ime.usp.br/~is/papir/RNR_v9.pdf. Citado na pág. 7

Smith(1980) C. P. Smith. A Software Science Analysis of Programming Size. Em *Proceedings of the ACM National Computer Conference*, páginas 179–185. Citado na pág. 103

Stamelos *et al.***(2002)** Ioannis Stamelos, Lefteris Angelis, Apostolos Oikonomou, e Georgios L. Bleris. Code Quality Analysis in Open Source Software Development. *Information Systems Journal*, 12:43–60. Citado na pág. 22

Stetter(1984) F. Stetter. A Measure of Program Complexity. *Computer Languages*, 9(3-4):203–208. Citado na pág. 18, 102

Stewart e Gosain(2006) Katherine J. Stewart e Sanjay Gosain. The Impact of Ideology on Effectiveness in Open Source Software Development Teams. *MIS Quarterly*, 30(2):291–314. Citado na pág. 8, 10, 11

Stol *et al.***(2009)** Klaas-Jan Stol, Muhammad Ali Babar, Barbara Russo, e Brian Fitzgerald. The Use of Empirical Methods in Open Source Software Research: Facts, Trends and Future Directions. Em *FLOSS'09: Proceedings of the 2009 ICSE Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development*, páginas 19–24, Washington, DC, USA. IEEE Computer Society. ISBN 978-1-4244-3720-7. doi: http://dx.doi.org/10.1109/FLOSS.2009.5071355. Citado na pág. 22

Subramanyam e Krishnan(2003) Ramanath Subramanyam e M. S. Krishnan. Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *IEEE Trans. Softw. Eng.*, 29(4):297–310. ISSN 0098-5589. doi: 10.1109/TSE.2003.1191795. URL http://dx.doi.org/10.1109/TSE.2003.1191795. Citado na pág. 15

Tapscott e Williams(2006) Don Tapscott e Anthony D. Williams. *Wikinomics: How mass collaboration changes everything*. Portfolio – Penguin Books, New York. Citado na pág. 7

Tempero(2008) Ewan Tempero. On measuring java software. Em *Proceedings of the thirty-first Australasian conference on Computer science - Volume 74*, ACSC '08, páginas 7–7, Darlinghurst, Australia, Australia. Australian Computer Society, Inc. ISBN 978-1-920682-55-2. URL http://dl.acm.org/citation.cfm?id=1378279.1378283. Citado na pág. 3, 17

Terceiro(2010) Antonio Terceiro. Developers' contribution to structural complexity in free software projects. Em Walt Scacchi, Kris Ven, e Jan Verelst, editors, *Proceedings of the OSS 2010 Doctoral Consortium*. ISBN 978-90-337-0034-7. Citado na pág. 23

Terceiro e Chavez(2009a) Antonio Terceiro e Christina Chavez. Structural Complexity Evolution in Free Software Projects: A Case Study. Em Muhammad Ali Babar, Björn Lundell, e Frank van der Linden, editors, *QACOS-OSSPL 2009: Proceedings of the Joint Workshop on Quality and Architectural Concerns in Open Source Software (QACOS) and Open Source Software and Product Lines (OSSPL)*. Citado na pág. 48

Terceiro e Chavez(2009b) Antonio Terceiro e Christina Chavez. Structural Complexity Evolution in Free Software Projects: A Case Study. Em Muhammad Ali Babar, Björn Lundell, e Frank van der Linden, editors, *QACOS-OSSPL 2009: Proceedings of the Joint Workshop on Quality and Architectural Concerns in Open Source Software (QACOS) and Open Source Software and Product Lines (OSSPL)*. Citado na pág. 23

Terceiro *et al.***(2010a)** Antonio Terceiro, Joenio Costa, João Miranda, Paulo Meirelles, Luiz Romário Rios, Lucianna Almeida, Christina Chavez, e Fabio Kon. Analizo: an extensible multi-language source code analysis and visualization toolkit. Em *Congresso Brasileiro de Software: Teoria e Prática*. Citado na pág. 76

Terceiro *et al.***(2010b)** Antonio Terceiro, Luiz Romario Rios, e Christina Chavez. An empirical study on the structural complexity introduced by core and peripheral developers in free software projects. *Brazilian Symposium on Software Engineering*, 0:21–29. doi: http://doi.ieeecomputersociety.org/10.1109/SBES.2010.26. Citado na pág. 23

Terceiro *et al.***(2012)** Antonio Terceiro, Manoel Mendonça, Christina Chavez, e Daniela S. Cruzes. Understanding structural complexity evolution: a quantitative analysis. Em *16th European Conference on Software Maintenance and Reengineering*. Citado na pág. 23

Troy e Zweben(1981) D. A. Troy e S. H. Zweben. Measuring the Quality of Structured Designs. *J. Syst. and Software*, 2(2):113–120. Citado na pág. 14

Vandoren(1997) E. Vandoren. Maintainability Index Technique for Measuring Program Maintainability. http://www.sei.cmu.edu/str/descriptions/mitmpm.html, Maio 1997. Citado na pág. 79

VanDoren(1997) Edmond VanDoren. C4 Software Technology Reference Guide — A Prototype. Relatório técnico, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213. Citado na pág. 103

Vincenzi *et al.***(2003)** A. M. R. Vincenzi, W. E. Wong, M. E. Delamaro, e J. C. Maldonado. JaBUTi: A Coverage Analysis Tool for Java Programs. Em *XVII SBES — Brazilian Symposium on Software Engineering*, páginas 79–84. Citado na pág. 75

Warsta e Abrahamsson(2003) Juhani Warsta e Pekka Abrahamsson. Is open source software development essentially an agile method?, May 2003. URL http://agile.vtt.fi/docs/publications/2003/2003_oss.pdf. Citado na pág. 7

West(1992) M. West. An Investigation of C++ Metrics To Improve C++ Project Estimation. IBM internal paper, 1992. Citado na pág. 107

Wheeldon e Counsell(2003) Richard Wheeldon e Steve Counsell. Power law distributions in class relationships. Em *Proceedings of the Third IEEE International Workshop on Source Code Analysis and Manipulation (SCAM '03)*, páginas 45–54, Amsterdam, The Netherlands. Citado na pág. 23, 24

Williams e Carver(2010) Byron J. Williams e Jeffrey C. Carver. Characterizing software architecture changes: A systematic review. *Inf. Softw. Technol.*, 52(1):31–51. ISSN 0950-5849. doi: 10.1016/j.infsof.2009.07.002. URL http://dl.acm.org/citation.cfm?id=1645441.1645568. Citado na pág. 23

Wolverton(1974) R. W. Wolverton. The Cost of Developing Large-Scale Software. *IEEE Transactions Computers*, C-23(6):615–636. Citado na pág. 3

Woodfield *et al.***(1981)** S. N. Woodfield, V. Y. Shen, e H. E. Dunsmore. A Study of Several Metrics for Programming Effort. *J. Syst. and Software*, 2:97–103. Citado na pág. 101

Woodward *et al.***(1979)** M. R. Woodward, M. A. Hennell, e D. Hedley. A Measure of Control Flow Complexity in Program Text. *EEE Transactions. Software Engineering*, 5(1):45–50. Citado na pág. 102

Xenos *et al.***(2000)** M. Xenos, D. Stavrinoudis, K. Zikouli, e D. Christodoulakis. Object-Oriented Metrics - A Survey. Em *Proceedings of the FESMA 2000 - Federation of European Software Measurement Associations*, volume 6, Madrid, Spain. Citado na pág. 15, 104

Yao *et al.***(2009)** Yi Yao, Song Huang, Zheng-ping Ren, e Xiao-ming Liu. Scale-free property in large scale object-oriented software and its significance on software engineering. Em *Proceedings of the 2009 Second International Conference on Information and Computing Science*, ICIC 09, páginas 401–404, Washington, DC, USA. IEEE Computer Society. Citado na pág. 24

Yau e Collofello(1985) S. S. Yau e J. S. Collofello. Design Stability Measures for Software Maintenance. *IEEE Transantions Software Engineering*, 11(9):849–856. Citado na pág. 14

Yourdon e Constantine(1979) E. Yourdon e L. L. Constantine. *Structured Design*. Prentice Hall. Citado na pág. 103

Zolnowski e Simmons(1981) J. C. Zolnowski e D. B. Simmons. Taking the Measure of Program Complexity. Em *Proceedings of the National Computer Conference*, páginas 329–336. Citado na pág. 104

Zuse(1990) Horst Zuse. *Software Complexity: Measures and Methods*. Walter de Gruyter. Citado na pág. 2, 13, 14