Kalibro: interpretação de métricas de código-fonte

Carlos Morais de Oliveira Filho

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

Programa: Ciência da Computação

Orientador: Prof. Dr. Fabio Kon

Durante o desenvolvimento desta dissertação
o autor recebeu auxílio financeiro
do CNPq, da CAPES e do projeto QualiPSo da European Commission

São Paulo, 22 de setembro de 2013

Kalibro: interpretação de métricas de código-fonte

Esta dissertação contém correções e alterações sugeridas pela Comissão Julgadora durante a defesa realizada por Carlos Morais de Oliveira Filho em 07/08/2013.
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. Daniel Macedo Batista - IME-USP
- Prof. Dr. Marcos Lordello Chaim - EACH

Agradecimentos

Agradeço ao meu orientador, professor Dr. Fabio Kon, pela presença fundamental em praticamente toda a minha trajetória acadêmica. Ao meu coorientador, o doutorando Paulo Meirelles, pelo incentivo, pela cobrança e pela solicitude com que sempre se dispôs a me ajudar. Ao professor Dr. Alfredo Goldman pela colaboração durante o curso de Laboratório de Métodos Ágeis de Desenvolvimento de Software. Ao projeto QualiPSo da European Comission, à Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), ao Conselho Nacional de Desenvolvimento Científico e Tecnológico (CNPq) e ao Centro de Competência em Software Livre da Universidade de São Paulo (CCSL-USP) pelo financiamento concedido e pela estrutura disponibilizada para realização deste trabalho. Enfim, a Juliana Piroupo, minha companheira, pelo apoio carinhoso durante esse mestrado.

Resumo

Métricas de código-fonte não são novidade, mas ainda não têm sido bem exploradas no desenvolvimento de software. A maioria das ferramentas de métricas mostra valores numéricos isolados, que não são fáceis de entender porque a interpretação deles depende do contexto da implementação. Esta dissertação apresenta o software livre Kalibro Metrics, que foi desenvolvido com o objetivo de melhorar a legibilidade de métricas de código-fonte. Kalibro, ao contrário das outras ferramentas, permite que o próprio usuário crie configurações de intervalos associados a avaliações qualitativas, incluindo comentários e recomendações. Usando essas configurações, o Kalibro mostra resultados de métricas de modo amigável, ajudando: arquitetos de software a detectar falhas de projeto; gerentes de projetos a controlar a qualidade de código-fonte; usuários de software e pesquisadores a comparar características específicas do código-fonte de vários projetos. Essas configurações podem ser compartilhadas e utilizadas para intermediar discussões voltadas à evolução de critérios de avaliação baseados em métricas de código-fonte.

Palavras-chave: métricas de software, código-fonte, interpretação, Kalibro.

Abstract

Source code metrics are not new, but they have not yet been fully explored in software development. Most metric tools show isolated numeric values, which are not easy to understand because their interpretation depends on the implementation context. This dissertation presents the free software Kalibro Metrics, which was developed to improve readability of source code metrics. Kalibro, unlike the current tools, allows the user himself to create configurations of thresholds associated with qualitative evaluation, including comments and recommendations. Using these configurations, Kalibro shows metric results in a friendly way, helping: software architects to spot design flaws; project managers to control source code quality; software users and researchers to compare specific source code characteristics across software projects. These configurations can be shared and used to mediate discussions focused on the evolution of assessment criteria based on source code metrics.
Keywords: software metrics, source code, interpretation, Kalibro.

Sumário

Lista de Abreviaturas

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

Lista de Tabelas

Capítulo 1

Introdução

Medir é atribuir números ou símbolos a propriedades de entidades, de modo que elas possam ser descritas de acordo com regras bem definidas. Medidas tornam os conceitos mais fáceis de entender e controlar (Fenton e Pfleeger, 1998). Portanto, medir e monitorar a qualidade do software é fundamental, qualquer que seja a metodologia de desenvolvimento. Muitos dos fatores que compõem um bom software podem ser percebidos no código-fonte e alguns são exclusivos dele. Mesmo uma ótima bateria de testes pode produzir informação apenas sobre características externas, não refletindo qualidades como manutenibilidade, modularidade, flexibilidade e simplicidade. Nesse contexto, as métricas de código-fonte não podem ser negligenciadas em relação às outras abordagens de monitoramento da qualidade de software.

O código-fonte ganha ainda mais importância com o sucesso dos Métodos Ágeis (Marchesi *et al.*, 2003), pois vários princípios dessas metodologias valorizam o código-fonte. Dentre eles podemos citar:

- Desburocratização do processo de desenvolvimento. Código funcionando e testes são mais valorizados que documentação detalhada.
- Propriedade coletiva do código. Com cada membro da equipe cuidando do software como um todo, sem separação estática de responsáveis por partes do código, clareza e padrões de codificação são enfatizados.
- Valorização do programador e reconhecimento da codificação como um trabalho intelectual e criativo.

Outro movimento relativamente recente e de grande sucesso é o Software Livre (Raymond, 1999). Desde a década de 90, com a popularização do sistema operacional Linux e os avanços da internet, muitos projetos de Software Livre surgiram e diversas ferramentas foram produzidas. O Software Livre é baseado na distribuição e colaboração durante o processo de desenvolvimento. O código-fonte não só é modificado por toda a equipe, como fica aberto ao público. Os princípios ágeis que procuram viabilizar o compartilhamento de código são ainda mais enfatizados no contexto de Software Livre. A comunidade de Software Livre reconhece o desenvolvedor pelo código que produz. O que determina se uma funcionalidade ou melhoria será ou não incorporada ao projeto original é a qualidade de seu código-fonte, que precisa ser entendido e aceito.

Engenheiros de software e pesquisadores precisam analisar códigos-fonte para entender melhor projetos de software. Em muitos casos, o código-fonte é o único artefato disponível para se medir a qualidade do software. Projetos de software bem-sucedidos, salvo raras exceções, têm código-fonte bem estruturado. No contexto de Software Livre, a motivação para se envolver em um projeto é criticamente afetada pela modularidade do seu código (Baldwin e Clark, 2006) e sua organização e complexidade influenciam o número de downloads e membros (Meirelles *et al.*, 2010).

Funcionalidades são entregues constantemente a clientes e usuários, portanto o código-fonte é escrito gradualmente e diferentes desenvolvedores fazem atualizações e melhorias continuamente

(Martin, 2008). Enquanto estão programando, desenvolvedores tomam várias decisões, que em conjunto, influenciam na qualidade do código (Beck, 2007). Observar os atributos do código que está sendo desenvolvido auxilia na tomada dessas decisões. A partir de um conjunto 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-lo.

Métricas de código-fonte foram propostas desde a década de 70 (Perlis *et al.*, 1981; Wolverton, 1974) quando os primeiros conceitos da engenharia de software surgiram. As métricas pioneiras foram rapidamente absorvidas pela indústria de software, cujos modelos ainda são baseados em medidas simples de tamanho (como linhas de código) e contagem de defeitos (Fenton e Neil, 1999). Essas métricas são reconhecidamente limitadas (e até perigosas[1]) quando usadas isoladamente. Mas mesmo com o avanço da pesquisa em métricas desde então, a indústria continua usando apenas as métricas antigas, por serem mais fáceis de entender (Fenton e Neil, 1999). Um dos motivos dessa subutilização é que a maior parte das métricas não possuem valores com modelos populacionais conhecidos (Tempero, 2008).

Essa falta de valores de referência surpreende, uma vez que a transparência dos projetos de Software Livre cria uma disponibilidade de dados sem precedentes (von Krogh e Spaeth, 2007). Essa disponibilidade oferece uma grande oportunidade para pesquisa em métricas de código-fonte, através da análise de repositórios abertos. A análise de código-fonte pode ajudar a responder diversas perguntas:

- Qual a distribuição dos valores de uma determinada métrica?
- Quais são os valores de referência (intervalo recomendado, valores limite) de uma métrica?
- O comportamento das métricas é o mesmo em software de diferentes linguagens?
- O comportamento das métricas difere de acordo com o domínio de aplicação do software?
- Qual a relação entre determinada propriedade do código-fonte e características desejáveis em um projeto?
- Dadas duas métricas, elas estão fortemente relacionadas (redundantes) ou são independentes?
- Qual métrica ou combinação de métricas melhor reflete determinada propriedade do software?

Ferramentas de análise automática de repositórios viabilizam o processamento de grande volume de dados, potencializando o escopo e a confiabilidade dos resultados de pesquisa. Esse tipo de ferramenta também pode auxiliar o gerenciamento de projetos, fixando metas ou padrões de codificação em termos de métricas. Uma ferramenta que auxilie na interpretação de valores de métricas também tem o potencial de diminuir a curva de aprendizado para seu uso cotidiano pelo desenvolvedor. Porém, ferramentas de análise automática de código-fonte atuais (listadas na Tabela 3.1) possuem sérias limitações: em funcionalidades, abrangência ou liberdade de uso, modificação e extensão.

Nesta dissertação apresentamos o Kalibro Metrics, software livre projetado para servir como plataforma de obtenção, análise e interpretação de métricas de código-fonte de qualquer projeto de software, agregando diversas ferramentas de análise de código-fonte em um só serviço (as chamaremos de ferramentas-base). O Kalibro Metrics permite que qualquer usuário especifique um conjunto de métricas que deseja monitorar, associadas a intervalos de interpretação. Essas métricas podem ser escolhidas entre as fornecidas pelas ferramentas-base associadas ou criadas como combinação das métricas existentes. Com isso pretende-se facilitar o uso de métricas de código-fonte na gerência de projetos, no cotidiano do programador e na pesquisa em engenharia de software.

[1] Como ilustração, considere a remuneração de programadores por desempenho, avaliado pelo número de linhas de código produzidas. Esse modelo estimula a criação de código excessivamente redundante, criado com recurso de copiar e colar.

1.1 Organização do Trabalho

O Capítulo 2 descreve métricas de software, caracterizando-as e classificando-as, para então apresentar a definição das métricas de código-fonte mais recorrentes na literatura. O Capítulo 3 apresenta várias ferramentas de métricas atuais, mostrando seus usos e limitações. O Capítulo 4 descreve a arquitetura do Kalibro Metrics e como ela foi projetada para incorporar as propriedades de um serviço facilitador do uso de métricas de código-fonte. No Capítulo 5, uma configuração de valores de referência é definida com base em análises estatísticas envolvendo diversos projetos bem sucedidos. Essa configuração é utilizada para avaliar as métricas de projetos de Software Livre, ilustrando o uso do Kalibro como ferramenta de interpretação. Finalmente, o Capítulo 6 apresenta as considerações finais.

Capítulo 2

Métricas de software

> You cannot control what you cannot measure.
>
> DeMarco (1982)

Segundo o padrão ISO/IEC9126-1 (2001), métricas são compostas por procedimentos de medição e escalas de medidas. Métricas são utilizadas, entre outros fins, para: estimar cronogramas de projetos; prever custos de desenvolvimento; controlar produtividade de processos; medir a qualidade do produto.

A epígrafe deste capítulo enfatiza a importância de métricas no controle de processos e produtos. Porém, toda métrica tem um custo e a precisão com que ela reflete a qualidade analisada pode variar. O próprio DeMarco contesta sua citação (DeMarco, 2009), afirmando que nem tudo vale a pena controlar e que capacidade de adaptação é mais importante que previsibilidade em projetos de software. Portanto, não se deve procurar medir tudo, mas escolher bem as métricas utilizadas para avaliar e controlar projetos.

A abordagem mais utilizada para seleção de métricas é a *Goal-Question-Metric*, ou simplesmente GQM. Segundo essa abordagem, o processo de medição possui 3 níveis hierárquicos (Basili *et al.*, 1994):

1. **Conceitual.** Uma meta é definida para algum objeto: produto, processo ou recurso.
2. **Operacional.** São elaboradas perguntas sobre os atributos do objeto medido que devem ser controlados para alcançar a meta. Nesse nível é definido o ponto de vista sobre o qual a meta será avaliada.
3. **Quantitativo.** Cada pergunta é associada a um conjunto de dados usado para respondê-la de modo quantitativo.

Ainda segundo o ISO/IEC9126-1 (2001), as métricas de software podem ser classificadas em:

- **Métricas internas.** São métricas cuja coleta dispensa a execução do software, ou seja, são baseadas em medidas estáticas. Essas medidas geralmente são obtidas através da análise do código. São exemplos de atributos da qualidade interna: simplicidade, concisão, coesão, clareza, baixo acoplamento e generalidade.
- **Métricas externas.** Métricas aplicáveis ao software em execução. Exemplos de atributos da qualidade externa: correção, usabilidade, eficiência e robustez.
- **Métricas de qualidade em uso.** São métricas coletadas quando os usuários finais estão utilizando o software no ambiente para o qual ele foi projetado, sob as condições reais.

A qualidade interna é percebida pelo desenvolvedor ao manter e evoluir o software, enquanto a qualidade externa é percebida em seu uso. Idealmente, o software deve ao mesmo tempo preencher

os requisitos (qualidade externa) e ser fácil de entender e modificar (qualidade interna), mas pressões de agenda e orçamento podem fazer com que a qualidade interna seja negligenciada para adiantar a entrega do software. Daí a necessidade de medir e gerenciar também a dívida técnica, termo cunhado por Cunningham (1992) para indicar a discrepância entre qualidade interna e externa.

As métricas devem estar associadas a uma escala de medida que proporcione significado ao valor obtido no seu cálculo. Antes de fazer comparações entre valores de métricas, é preciso estar ciente do modelo de dados ao qual eles pertencem. Os valores podem pertencer a um desses tipos de escala estatística (Conte *et al.*, 1986):

- **Nominal.** Não existe ordem nem magnitude. Serve para separar categorias; os valores são rótulos que podem ser comparados apenas por igualdade. Exemplos: estado civil, código postal, código de barras.
- **Ordinal.** Os valores podem ser comparados segundo uma ordem, mas não existe definição clara da magnitude da diferença entre eles. Exemplo: tamanho de roupa P-M-G-GG.
- **Intervalar.** Existe ordem e tamanho dos intervalos que separam os valores, mas proporções não são válidas, pois não existe unidade natural ou zero absoluto. Exemplo: na escala Celsius de temperatura, não faz sentido dizer que 40 graus é o dobro de calor que 20 graus.
- **Racional.** Possui ordem, magnitude, zero absoluto e unidade natural. Exemplos: preço, idade e distância.

Métricas de escala racional possuem grande vantagem prática, pois seus valores podem ser usados em operações matemáticas como soma e divisão. Como exemplo desse uso temos a criação de métricas compostas, como LOC/tempo. Outras boas características que métricas deveriam ter, são (Mills, 1988):

- **Simplicidade.** O que a métrica se propõe a medir deve ser claro. Seu resultado deve ser fácil de associar com o atributo medido.
- **Validade.** A métrica deve medir efetivamente o que se propõe a medir.
- **Objetividade.** A métrica deve ser formalmente definida. A obtenção de seu valor ser repetível, sem depender do julgamento de quem coleta. Idealmente, seu valor deve ser coletado de forma automática.
- **Fácil obtenção.** Deve ser possível obter seu valor rapidamente e a baixo custo. Caso contrário, a métrica não poderá ser usada para fornecer *feedback* constante.
- **Robustez.** Pequenas mudanças no software não podem causar grandes mudanças no valor obtido.

2.1 Métricas de código-fonte

Métricas de código-fonte são aquelas obtidas através da análise do código-fonte. De acordo com a chamada de trabalhos do SCAM[1], código-fonte é "qualquer descrição completamente executável de um sistema de software". Portanto, representações gráficas de um sistema podem ser consideradas código-fonte, desde que constituam uma especificação executável do software.

A importância do código-fonte na qualidade de software é grande e tende a crescer (Harman, 2010). Apresentaremos nesta seção a definição das métricas de código-fonte mais recorrentes na literatura. Podemos dividir as métricas de código-fonte em dois grandes grupos: métricas de tamanho e complexidade, aplicáveis a qualquer programa, e métricas orientadas a objeto, que medem propriedades características de sistemas desenvolvidos sob esse paradigma.

[1] IEEE International Working Conference on Source Code Analysis and Manipulation

2.1.1 Métricas de tamanho e complexidade

A noção de tamanho ou volume de software existe desde os primórdios da computação, quando o software ocupava espaço físico, na forma de pilhas de cartões perfurados. Um método válido de medir o tamanho do software é requisito de padrões de qualidade e certificações de processos de desenvolvimento, como o *Capability Maturity Model Integration*, conhecido como CMMI (Chrissis *et al.*, 2011). Métricas de tamanho são utilizadas por modelos de estimativas de custo e esforço, a exemplo dos modelos Cocomo (Boehm *et al.*, 2000) e Cosysmo (Valerdi, 2005).

A noção de complexidade está ligada ao número de interações entre diferentes elementos do sistema. Métricas de complexidade têm uma relação forte com manutenibilidade (Lehman *et al.*, 1997) pois, quanto mais interações, maior a probabilidade de uma mudança em uma parte do sistema afetar outras partes, exigindo mais mudanças ou introduzindo falhas inesperadas. Quanto mais entidades tiver o sistema, maior o número possível de interações entre elas, portanto sistemas maiores são mais suscetíveis a maior complexidade, portanto métricas de complexidade devem ser usadas junto com métricas de tamanho.

Linhas de código, ou simplesmente LOC (do inglês *Lines Of Code*) é a métrica de código-fonte mais antiga e mais conhecida[2] (Boehm, 1984; Jones, 1985). Em linguagens mais antigas como FORTRAN e linguagem de montagem (*assembler*), o número de linhas de código era fácil de definir e o tamanho de cada linha não variava muito, pois havia um limite (80 ou 96 caracteres). As linguagens mais usadas atualmente permitem grande variedade de estilos de programação que influenciam na quantidade de linhas de código. Portanto, foram definidas variações dessa métrica:

- **Linhas de código** (LOC ou SLOC - *Source Lines Of Code*): a definição mais comum conta todas as linhas que contêm código, ou seja, comentários e linhas em branco são excluídas da contagem.
- **Linhas físicas de código**: conta o número de linhas de cada arquivo, sem analisar o código em si. Portanto, linhas em branco e comentários contam e cada linha é contada apenas uma vez, mesmo se contiver mais de uma instrução. Raramente usada, por variar muito com a formatação e estilo de codificação (pouca robustez).
- **Linhas lógicas de código** (LLOC - *Logical Lines Of Code*): conta o número de instruções. Geralmente equivale a contar o número de terminadores, como pontos-e-vírgulas. Desse modo, uma linha com um teste condicional ou declaração de função não conta e uma linha com duas instruções conta duas vezes.
- **Linhas efetivas de código** (ELOC - *Effective Lines Of Code*): conta todas as linhas que contêm código, mas exclui aquelas que possuem apenas delimitadores (p.ex. chaves, parênteses, aspas, *begin*, *end*). É melhor que SLOC para estimar esforço pois é menos vulnerável a variações de estilo, mas é menos usada, talvez porque poucas ferramentas a oferecem.

LOC é uma métrica simples de entender e fácil de computar, mas possui sérias limitações. Ela é muito sensível a diferenças na linguagem de programação. Linguagens diferentes valorizam de forma diferente clareza e brevidade, podendo diferir muito em prolixidade. Comparar linhas de código de linguagens diferentes exige certos cuidados: podem ser usados fatores de normalização ou a comparação pode ser feita apenas em escala logarítmica. Projetos de software frequentemente são desenvolvidos usando mais de uma linguagem de programação. Como não é razoável simplesmente somar, fica difícil definir medidas como erros por linhas de código.

LOC tem sido usada para medir produtividade. Porém, mais linhas de código não significa necessariamente mais funcionalidade e, muitas vezes, as melhores soluções são implementadas com menos linhas de código. LOC é particularmente ruim para medir produtividade de indivíduos. Um programador pode, eliminando repetição e introduzindo melhorias no sistema, ter saldo negativo em linhas de código. A medida de produtividade por linhas de código incentiva programadores a

[2] Um termo muito usado é KLOC, que representa 1000 linhas de código

escrever código desnecessariamente prolixo, provavelmente introduzindo repetições e prejudicando concisão.

Pontos de função, ou FP (do inglês *Function Points*), procura representar a quantidade de funcionalidade entregue aos usuários do sistema (Albrecht e Jr, 1983). Não há uma definição precisa e abrangente, mas vários padrões para análise ou obtenção automática de pontos de função, os quais têm como parâmetros o número de entradas do usuário, as consultas, as saídas e os principais arquivos (ISO/IEC19761, 2011; ISO/IEC20926, 2009; ISO/IEC20968, 2002; ISO/IEC24570, 2005; ISO/IEC29881, 2008). Pontos de função foram propostos como uma alternativa a linhas de código para medir produtividade sem incentivar prolixidade e para estimar custos, pois pontos de função podem ser calculados com base apenas nos requisitos. O custo de 1 ponto pode ser estimado com base em experiências prévias com outros projetos.

Métricas de Halstead é um conjunto de métricas baseadas na teoria da informação (Halstead, 1977). Elas se aplicam a vários elementos do software e são utilizadas para estimar esforço, tempo de desenvolvimento e até o número provável de *bugs*, com base apenas no número de operadores e de operandos.

Complexidade ciclomática, em definição atribuída a McCabe (1976), é o número de ciclos independentes no grafo de controle de fluxo do programa. Esse grafo contém cada bloco básico de código como nó, com uma aresta ligando dois nós se a execução pode passar do primeiro para o segundo. A esse grafo é adicionado: um nó de entrada ligado ao primeiro bloco; um nó de saída, ao qual se ligam os blocos terminais; uma aresta ligando o nó de saída ao nó de entrada. Pode ser provado que essa definição equivale ao número de desvios (ou estruturas condicionais) mais 1, o que é mais fácil de computar. Como a coleta consiste em contar o número de condicionais, a métrica também é chamada de complexidade condicional. Ela indica o número de testes que o fragmento de software precisa ter para cobrir todos caminhos linearmente independentes de execução.

Medidas estruturais baseadas em fluxo de informação foram propostas por Henry e Kafura (1981). A métrica de complexidade de Henry e Kafura é definida como $length \times (fanin \times fanout)^2$, onde: *length* é o comprimento do procedimento (LOC); *fan-in* é o número de chamadas ao procedimento mais o número de estruturas de dados do qual ele obtém informação; *fan-out* é o número de chamadas que o procedimento realiza mais o número de estruturas de dados que ele modifica.

Nesta subseção foram apresentadas apenas as métricas mais referenciadas e conhecidas da engenharia de software tradicional. Muitas outras métricas e uma série de processos foram propostos (Kafura e Canning, 1985; Kemerer, 1987), mas apenas algumas tem sido amplamente aceitas ou utilizadas. Mesmo as mais comuns, como as apresentadas, não possuem uma uniformidade. Seus resultados são difíceis de interpretar e comparar, especialmente quando envolvem diferentes linguagens, aplicações e metodologias de desenvolvimento.

Embora não possam ser usadas de forma descuidada, métricas simples como LOC e complexidade ciclomática se mostraram boas medidas para prever características como erros, esforço total e manutenibilidade (Li e Cheung, 1987; Rombach, 1987). A aplicação cuidadosa de algumas das métricas e modelos disponíveis podem produzir resultados úteis, se de acordo com o ambiente específico (Mills, 1988).

2.1.2 Métricas orientadas a objeto

Como a orientação a objetos usa entidades e não algoritmos como componentes fundamentais, métricas de código-fonte para software que segue esse paradigma possuem abordagens diferentes. Além de tamanho e complexidade, em um software orientado a objetos é possível medir o uso de mecanismos de herança, polimorfismo e encapsulamento, assim como o grau de interdependência entre as entidades. As métricas apresentadas nesta seção estão entre as mais comumente encontradas em ferramentas de análise de código-fonte, das quais destacam-se as dos grupos CK (Chidamber e Kemerer, 1994) e MOOD (Abreu e Carapuça, 1994). Xenos *et al.* (2000) apresenta uma revisão mais ampla desse tipo de métrica.

Grupo CK - Chidamber e Kemerer

Profundidade na árvore de herança, ou DIT (do inglês *Depth of Inheritance Tree*), é a distância máxima da classe à raiz na árvore de herança. O termo "distância máxima" aparece na definição porque há linguagens que permitem herança múltipla. Em linguagens em que cada classe pode estender diretamente apenas uma classe, uma definição equivalente é o número de ancestrais da classe. Quanto mais ancestrais a classe tiver, maior a chance dela herdar atributos e métodos, tornando mais difícil prever seu comportamento. Valores muito altos são sinais de uma hierarquia complexa, provavelmente mais difícil de entender e manter, mas valores muito baixos podem indicar que o mecanismo de herança, que promove o reúso de implementação, está sendo subutilizado.

Número de filhos, ou NOC (do inglês *Number Of Children*), é definida como o número de subclasses diretas da classe analisada. Uma mudança em uma classe pode influenciar todas as suas subclasses. Quanto mais subclasses a classe tiver, maior sua influência no sistema e maior a importância de ser bem testada. Um número excessivo de subclasses pode indicar uso inapropriado do mecanismo de herança. NOC pode ser usada junto com DIT para analisar o uso de herança em um sistema.

Acoplamento entre objetos, ou CBO (do inglês *Coupling Between Objects*), de uma classe é o número de classes acopladas a ela, ou seja, o número de classes que ela acessa somado ao número de classes do sistema que a acessam. As parcelas dessa soma são comumente chamadas, respectivamente, de conexões eferentes e conexões aferentes. Dizemos que uma classe acessa outra quando a primeira invoca métodos, lê ou modifica atributos da segunda. Quanto mais as classes forem independentes, mais fácil é reutilizá-las e menos arriscado é modificá-las. Classes mais acopladas precisam de mais rigor em testes, pois mais partes do sistema dependem delas.

Resposta para uma classe, ou RFC (do inglês *Response For a Class*), é o número de métodos da classe somado ao número de métodos que ela invoca. Em outras palavras, é o número de métodos que potencialmente podem ser executados como resultado do envio de mensagem à classe. Classes com alto RFC possuem muitos métodos e/ou acionam muitos métodos. Quanto mais métodos puderem ser acionados através da classe, mais difícil será entender seu comportamento e testá-la.

Falta de coesão em métodos mede a falta de coesão de uma classe. A definição original (Chidamber e Kemerer, 1994) recebeu várias críticas e sugestões de melhorias. Novas versões foram propostas, dentre elas a versão 4 foi a mais aceita (Hitz e Montazeri, 1995), portanto é comum ver a sigla LCOM4 (do inglês *Lack of Cohesion in Methods*). Seja $M = \{M_1, \ldots, M_n\}$ o conjunto dos métodos da classe analisada. Dois métodos M_i e M_j são coesos se ambos acessam pelo menos um mesmo atributo da classe ou se M_i chama ou é chamado por M_j. LCOM4 é o tamanho da partição formada pela separação de M subconjuntos coesos. O menor valor possível é 0, mas ele representa o caso degenerado onde a classe não possui métodos. Uma classe completamente coesa tem LCOM4 igual a 1. Valores maiores que 1 indicam que a classe possivelmente viola o princípio de responsabilidade única, ou SRP (Martin, 2002), e que pode ser quebrada em duas ou mais classes coesas.

Grupo MOOD - *Metrics for Object Oriented Design*

Os autores das métricas do grupo MOOD (Abreu e Carapuça, 1994) procuraram criar um conjunto de métricas adimensionais, independentes de tamanho e de linguagem de programação (para facilitar comparação entre projetos diferentes). Portanto, todas possuem a mesma escala: seus valores variam de 0 a 1 e representam porcentagens.

Fator de acoplamento, ou COF (do inglês *Coupling Factor*), é a métrica mais popular desse grupo e indica o quão acoplado é o sistema. Seu valor é dado por:

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

Onde n é o número de classes e $AC(C_i)$ é o número de conexões aferentes da classe C_i. O numerador é o total de ligações entre as classes e o denominador é o total possível de ligações. Um software

fortemente conectado tem baixo grau de independência entre os módulos, sendo mais difícil entender, testar e modificar.

As outras métricas desse grupo, menos populares, são:

- **Fator de ocultação de métodos** (MHF - *Method Hiding Factor*), a porcentagem de métodos não-públicos.
- **Fator de ocultação de atributos** (AHF - *Attribute Hiding Factor*), a porcentagem de atributos não-públicos, que junto com MHF mede encapsulamento.
- **Fator de herança de métodos** (MIF - *Method Inheritance Factor*), a porcentagem de métodos herdados.
- **Fator de herança de atributos** (AIF - *Attribute Inheritance Factor*), a porcentagem de atributos herdados, que junto com MIF mede o impacto da herança.
- **Fator de polimorfismo** (PF - *Polymorphism Factor*), que mede o uso de polimorfismo.

Capítulo 3

Ferramentas de Métricas

Como vimos no capítulo anterior, métricas devem ser objetivas e fáceis de coletar. Portanto, seu uso efetivo requer ferramentas que automatizem a análise do código-fonte. Neste capítulo, descrevemos brevemente a funcionalidade e a estrutura das ferramentas mais relevantes no contexto desta dissertação. A última seção apresenta uma discussão sobre características desejáveis em ferramentas de métricas e uma tabela que associa cada uma das ferramentas estudadas com essas características.

3.1 Analizo

Analizo[1] é um software livre baseado inicialmente no Egypt[2], também software livre. O Egypt teve o suporte ao cálculo de métricas implementado inicialmente por Terceiro *et al.* (2010) e recebeu diversas contribuições do nosso grupo do CCSL-USP. Por incorporar funcionalidades e subferramentas que fugiram da proposta inicial do Egypt, passou a ser chamado Analizo ("análise" em esperanto).

O Analizo é um *toolkit* livre, multilinguagem e extensível para análise e visualização de código-fonte. Devido a seu desempenho, ele atende à necessidade de analisar projetos grandes ou em grande número. Além disso, o Analizo é capaz de manipular o código-fonte que não pode ser compilado (com erros de sintaxe ou fazendo uso de bibliotecas obsoletas), importante para analisar código legado em estudos de evolução de software.

O Analizo utiliza outras ferramentas para extrair dados do código-fonte. Seus principais extratores se comunicam com as ferramentas Sloccount[3] – que obtém o número de linhas efetivas de código (ver Seção 2.1.1) – e Doxyparse (Costa, 2009), um analisador desenvolvido e mantido pelo grupo de colaboradores da Analizo. O Doxyparse é baseado no Doxygen[4], uma ferramenta livre e multilinguagem para documentação de código-fonte.

No momento, a Analizo realiza a análise de código-fonte escrito em C, C++ e Java, calculando métricas tanto no nível de projeto (p.ex., COF) quanto no nível de classe (p.ex., LCOM4). No nível de projeto, a Analizo também fornece diversas estatísticas (p.ex., média, desvio padrão, mediana, moda) para cada uma das métricas de classe.

O Analizo possui um comando para o processamento em lote de vários projetos, produzindo um arquivo CSV com os dados das métricas para cada projeto, bem como um resumo geral das métricas no nível de projeto. Esses arquivos podem ser facilmente importados em ferramentas de análise estatística ou em planilhas. O Analizo também pode processar repositórios Git[5] e Subversion[6], gerando um arquivo CSV com valores de métricas para cada revisão em que o código-fonte foi alterado.

[1] http://analizo.org
[2] gson.org/egypt
[3] dwheeler.com/sloccount/
[4] doxygen.org/
[5] http://git-scm.com/
[6] http://subversion.apache.org/

O Analizo apresenta algumas imprecisões[7], talvez devido ao fato do Doxyparse, seu principal extrator, ser baseado em uma ferramenta de documentação, que não foi inicialmente projetada para análise de métricas. Outra limitação é a falta de opções de configuração na análise de métricas. Por exemplo, não é possível reduzir o tempo de processamento reduzindo a quantidade de métricas desejadas.

3.2 Checkstyle

Checkstyle[8] é uma ferramenta para ajudar programadores Java a seguirem padrões de codificação, ideal para projetos que desejam estabelecer um padrão obrigatório. Ela toma como entrada, além do código-fonte, uma configuração que descreve o padrão de codificação, por exemplo, o padrão criado pela Sun para o Java (Vermeulen *et al.*, 2000). Sua saída consiste em um conjunto de notificações - com nível informativo, alerta ou erro - indicando os lugares em que o código viola os padrões. A ferramenta tornou-se bastante popular na forma de um *plug-in* para a IDE Eclipse[9], que fornece verificação de violações em tempo real. Atualmente, existem *plug-ins* da Checkstyle para diversas IDEs e editores de código.

Inicialmente projetado para verificar estilo, a Checkstyle teve várias melhorias em sua arquitetura e passou a oferecer diversos tipos de verificações, incluindo problemas de projeto de classes, código duplicado e padrões de erro como bloqueio com verificação dupla (*double checked locking*). Uma classe de verificações da Checkstyle é relativa a métricas de código-fonte, verificando violações de tamanho, complexidade, acoplamento, profundidade de aninhamento, entre outras.

A arquitetura da Checkstyle, seguindo o padrão de projeto *Visitor* (Gamma *et al.*, 1995), facilita a implementação de novas verificações, inclusive de métricas. Porém, as verificações de métricas atualmente implementadas permitem apenas a configuração de um valor máximo. Outra limitação é o fato do Checkstyle ser feito somente para Java, sem planos de suporte a outras linguagens.

3.3 CVSAnalY

A ferramenta CVSAnalY analisa registros de um repositório de código e monta um bancos de dados com diversas informações sobre ele. Ela também pode ser usada para gerar estatísticas e gráficos sobre o histórico de desenvolvimento do software. A CVSAnalY interage os sistemas de controle de versão mais utilizados atualmente (CVS, Subversion, Bazaar, Git, Mercurial) e também com muitas linguagens de programação. Ela obtém de cada atualização (*commit*) registrada no repositório diversas informações como autores, data, linhas adicionadas e removidas, arquivos excluídos ou renomeados etc. Consultando o banco de dados gerado, é possível responder diversas perguntas, entre elas:

- Qual o número de módulos do projeto? Quantos arquivos no repositório não são código (textos, sons, figuras)?
- Qual o comentário vinculado a um *commit* específico?
- Quem são os contribuidores?
- Quais desenvolvedores contribuíram para determinado arquivo?
- Quais os desenvolvedores mais ativos no projeto? Quais foram os mais ativos no passado?
- Quais arquivos são modificados com mais frequência?
- Quais pares de arquivos geralmente são modificados ao mesmo tempo?

[7] Foram encontrados alguns resultados incorretos de métricas em código Java, por exemplo: https://github.com/analizo/analizo/issues/5 e https://github.com/analizo/analizo/issues/6

[8] http://checkstyle.sourceforge.net

[9] http://www.eclipse.org/eclipselink/

A CVSAnalY possui algumas extensões (opcionais), entre elas a obtenção de métricas de código para cada versão de cada arquivo encontrado no repositório de código-fonte. A CVSAnalY é voltada mais para análise dos registros do que para o código-fonte em si, constituindo uma poderosa ferramenta para análise da dinâmica social do desenvolvimento do projeto. Portanto as métricas de código-fonte que ela implementa são poucas, basicamente de tamanho e complexidade: linhas de código, linhas comentadas, linhas em branco, número de funções, complexidade ciclomática e métricas de Hasltead. Também não existe uma interface clara para adicionar novas métricas.

3.4 Requisitos

A indústria de desenvolvimento de software adotou métricas simples - como linhas de código e contagem de erros - porque elas são fáceis de entender e de coletar (Fenton e Neil, 1999). O ideal seria ter uma ferramenta que colete métricas e associe seus resultados a alguma interpretação, tornando-as mais fáceis de entender. Tendo encontrado lacunas nas ferramentas existentes (listadas a seguir), nosso grupo de pesquisa elaborou um conjunto de requisitos (Meirelles, 2013) para uma ferramenta que vise a difundir o uso de métricas de código-fonte:

1. **Intervalos de aceitação:** A ferramenta deve trabalhar com múltiplos intervalos para fornecer diferentes interpretações sobre os valores das métricas. Por exemplo, usando o Eclipse Metrics é possível configurar valores mínimo e máximo (apenas um intervalo de aceitação) para cada métrica que ele fornece e associar uma dica de correção quando o resultado para um método/classe/pacote cai fora desse intervalo. Os intervalos devem ser configuráveis, pois as métricas em geral não possuem valores de referência absolutos. Os valores ideais podem variar de acordo com vários fatores, como a linguagem, domínio de aplicação e requisitos.

2. **Extensível:** A ferramenta não deve se limitar a apenas uma linguagem de programação, nem a um conjunto definido de métricas. Ela 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 determinar o conjunto de métricas que deseja usar. As métricas devem poder ser adicionadas conectando-se a diferentes coletores de métricas ou a partir das métricas existentes. O índice de manutenibilidade (VanDoren, 2002) é um exemplo de métrica criada como combinação de métricas básicas. Outro exemplo é a combinação de CBO e LCOM4, que exerce influência sobre a atratividade de projetos de software livre (Meirelles *et al.*, 2010).

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

A Tabela 3.1 apresenta uma comparação das ferramentas estudadas com esses requisitos. Percebemos que nenhuma ferramenta cumpre todos. Com essa motivação foi criado o Kalibro Metrics, software responsável pelo suporte à interpretação de métricas de código-fonte. Definimos uma interface de conexão simples para delegar a coleta de valores para ferramentas existentes. Dessa forma, o Kalibro pode ser integrada com as ferramentas mais interessantes de acordo com as necessidades de cada projeto analisado. Sua arquitetura foi projetada para incorporar outras ferramentas no futuro.

Ferramenta	Linguagens	Intervalos	Extensível	Comparação	Livre
Analizo	C, C++, Java	Não	Sim	Não	Sim
Analyst4j[10]	Java	Sim	Não	Não	Não
CCCC[11]	C++, Java	Não	Não	Não	Sim
Checkstyle[12]	Java	Sim	Sim	Não	Sim
CK Java Metrics[13]	Java	Não	Não	Não	Sim
CMetrics[14]	C	Não	Sim	Não	Sim
Cscope[15]	C	Não	Não	Não	Sim
CVSAnaly[16]	C, C++, Python	Não	Sim	Não	Sim
Dependency Finder[17]	Java	Não	Não	Não	Sim
inFusion[18]	C, C++, Java	Sim	Não	Sim	Não
JaBUTi[19]	Java	Não	Não	Não	Sim
Macxim[20]	Java	Não	Não	Não	Sim
Metrics[21]	Java	Sim	Não	Não	Sim
OOMeter[22]	Java, C#	Não	Não	Não	Não
Understand[23]	Java	Não	Não	Não	Não
VizzAnalyzer[24]	Java	Não	Não	Não	Não

Tabela 3.1: *Ferramentas existentes versus requisitos definidos*

[10] http://www.codeswat.com
[11] http://cccc.sourceforge.net
[12] http://checkstyle.sourceforge.net
[13] http://spinellis.gr/sw/ckjm
[14] http://tools.libresoft.es/cmetrics
[15] http://cscope.sourceforge.net
[16] http://metricsgrimoire.github.io/CVSAnalY/
[17] http://depfind.sourceforge.net
[18] http://intooitus.com/products/infusion
[19] http://ccsl.icmc.usp.br/pt-br/projects/jabuti
[20] http://qualipso.eu/macxim-tool
[21] http://metrics.sourceforge.net
[22] http://www.ccse.kfupm.edu.sa/~oometer/oometer
[23] http://scitools.com
[24] http://www.arisa.se/tools.php

Capítulo 4

Kalibro Metrics

Kalibro Metrics é um serviço web para configuração, interpretação e monitoramento de métricas de código-fonte. Seus conceitos e funcionalidades foram prototipados no primeiro semestre de 2009, por uma equipe da disciplina Laboratório de Programação Extrema do IME-USP. Essa equipe trabalhava na ferramenta JaBUTi - Java Bytecode Understanding and Testing (Vincenzi *et al.*, 2003), que estava sendo adaptada para o contexto do QualiPSo – Quality Platform for Open Source Software (Pezuela *et al.*, 2010). 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 para identificar as partes críticas do código – aquelas que mais precisam ser testadas. Inicialmente, a equipe foi montada com o objetivo de melhorar o módulo de cálculo de métricas de código-fonte da JaBUTi. Entretanto, o foco passou para a visualização de métricas e decidiu-se fazer um módulo separado, no qual seria possível configurar intervalos para as métricas, tornando seus resultados mais fáceis de apresentar. Não demorou para que a equipe decidisse transformar esse módulo em uma ferramenta independente, chamada Crab (Meirelles *et al.*, 2009) e disponibilizada sob a licença BSD.

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

Com esse objetivo comecei a trabalhar na Crab no contexto de meu trabalho de conclusão de curso e como colaborador de um dos grupos da disciplina de Desenvolvimento de Software Livre do IME-USP, no segundo semestre de 2009. A ferramenta-base escolhida para integrar a Crab foi o Analizo (Terceiro *et al.*, 2010), por ser bastante eficiente na análise de código C, C++ e Java, pela disponibilidade de seus desenvolvedores e por possuir um grande número de métricas embutidas. 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 em Java para o Analizo, que acionasse a Crab para visualizar os resultados. Ao invés disso, ficou decidido que fazia mais sentido que a Crab, por ser uma ferramenta de visualização, ficasse acima da ferramenta-base, chamando-a para coletar os resultados.

Essa inversão de paradigma de integração, junto com outras mudanças de impacto, fez com que a Crab fosse relicenciada como LGPL versão 3 e, por influência do grupo criador da Analizo (nome que significa "análise" em esperanto), renomeada como Kalibro ("calibrar" em esperanto). Ao final do meu trabalho de formatura, o Kalibro já possuía base de dados própria, calculava estatísticas para as métricas coletadas e exibia os resultados baseando-se nas configurações cadastradas. No entanto, ainda era necessário baixar o código em separado e selecionar o diretório local para análise, bem como a exibição e integração precisava ser melhorada.

Em 2010, início deste trabalho de mestrado, ainda no contexto do QualiPSo, implementei um extrator para a plataforma Spago4Q[1], para análise de projetos C e C++, pois até então só havia

[1] http://www.spago4q.org

suporte para Java. A exemplo da Macxim[2], construí uma interface de serviço Web para que o extrator da Spago4Q pudesse coletar as métricas usando a Analizo através do Kalibro. Desde então, ficou decidido separar a coleta, configuração e interpretação de métricas da sua visualização.

Dessa separação surgiu o Kalibro Service, um serviço Web sem interface gráfica que possui toda funcionalidade não-visual: acompanhamento de repositórios de código; cálculo de métricas através de ferramentas-base; criação e compartilhamento de configurações e de interpretações de métricas. A outra parte passou a se chamar Kalibro Desktop, um aplicativo com interface Java Swing cuja funcionalidade é apenas visual, operando como um cliente magro do Kalibro Service. Atualmente, o Kalibro Desktop é um projeto congelado, pois pretende-se usar a rede de monitoramento Mezuro como camada de visualização dos resultados processados pelo Kalibro, como será explicado na Seção 4.5.4.

Este capítulo descreve a arquitetura do Kalibro e tem como propósito servir de guia para futuros usuários e desenvolvedores. A Seção 4.1 descreve sua interface de programação[3] e se dedica àqueles que pretendem conectá-lo com uma ferramenta coletora de métricas, usá-lo como biblioteca em um aplicativo Java ou apenas conhecer sua estrutura geral. A Seção 4.2 descreve a API Web[4] e se dedica àqueles que pretendem usar o Kalibro Service de fora de um aplicativo Java. A Seção 4.3 descreve a organização interna do código e é dedicada àqueles que pretendem contribuir programando ou simplesmente entender o funcionamento do Kalibro mais a fundo. A Seção 4.4 complementa a anterior descrevendo a organização dos testes automatizados. A Seção 4.5 descreve como foi feita a integração do Kalibro com outros programas. Enfim, a Seção 4.6 discute limitações atuais do Kalibro e sua rede de testes e descreve funcionalidades planejadas para versões futuras.

4.1 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`. Todos os diagramas apresentados nesta seção também se encontram no Apêndice A, para consulta. Para uma visão geral e sumarizada das entidades, recorra à Figura A.8.

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

- 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 com 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, suprimimos os métodos privados ou irrelevantes para o contexto, p.ex. `toString()`.

[2] http://www.qualipso.org/macxim-tool

[3] Interface de programação (ou API) é o conjunto padrões e rotinas (métodos, classes, constantes e interfaces públicas) desenhados para que outros aplicativos façam uso da funcionalidade do software sem envolver-se em detalhes de seu funcionamento.

[4] API Web é o conjunto definido de mensagens de requisição e resposta HTTP de um serviço Web. No caso do Kalibro Service, expresso no formato XML.

4.1.1 Métricas e resultados

Projetos de software são compostos por vários módulos que se integram, cada módulo sendo dividido 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 `Granularity` representa o nível de granularidade do fragmento de código a ser medido. As contantes 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 de operações e atributos. Linguagens procedurais 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 4.1). 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, construída a partir do caminho completo de cada módulo, o qual é fornecido pela ferramenta-base.

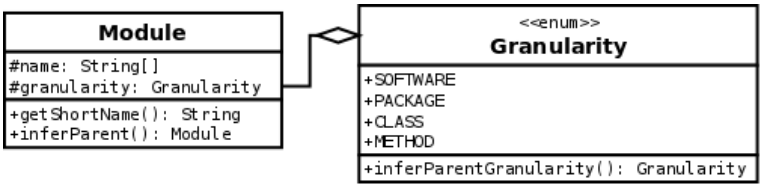


Figura 4.1: *Módulo e granularidade*

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, p.ex., 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 conforme indicado na Tabela 4.1.

Granularidade	Granularidade pai
METHOD	CLASS
CLASS	PACKAGE
PACKAGE	PACKAGE
SOFTWARE	SOFTWARE

Tabela 4.1: *Granularidade pai inferida*

Nem sempre a granularidade inferida é a correta, p.ex., 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.

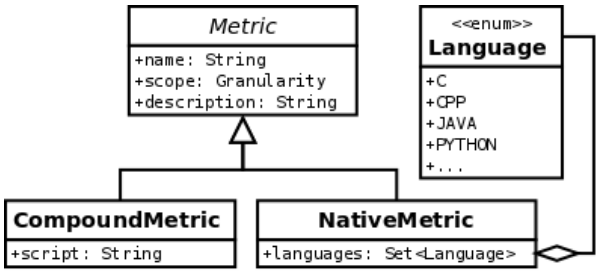


Figura 4.2: *Métricas*

A Figura 4.2 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, portanto só fazem sentido quando agrupados em um `NativeModuleResult` (ver Figura 4.3).

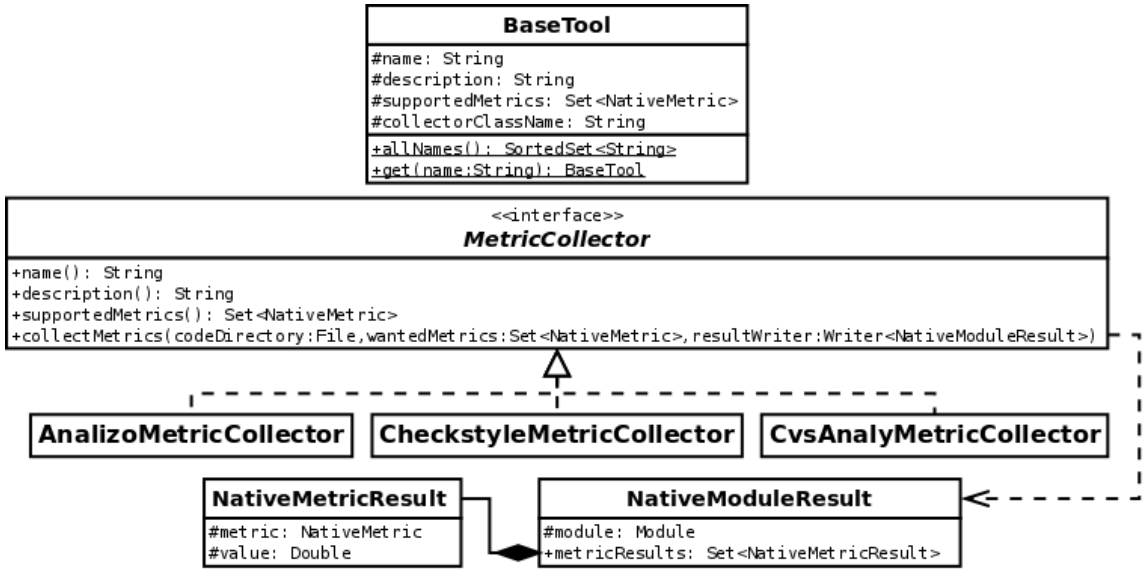


Figura 4.3: *ferramenta-base - coletor de métricas*

O Kalibro dispara a coleta de métricas pelas ferramentas-base através de implementações da interface `MetricCollector`. Dentre os métodos dessa interface, três apenas fornecem dados sobre a ferramenta (nome, descrição e métricas implementadas), 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 implementadas – 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.

4.1.2 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 suas leituras.

A classe `Reading` é uma abstração de uma leitura, para facilitar o reuso 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 4.4), dando a eles um nome para facilitar o reuso, como veremos adiante.

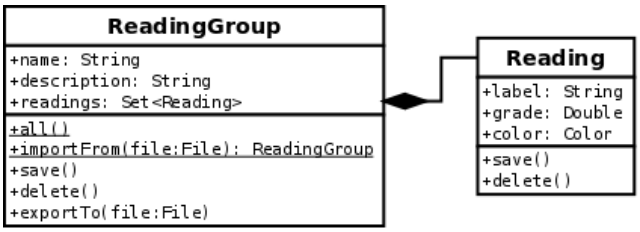


Figura 4.4: *Grupo de leituras*

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 4.5). 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`).

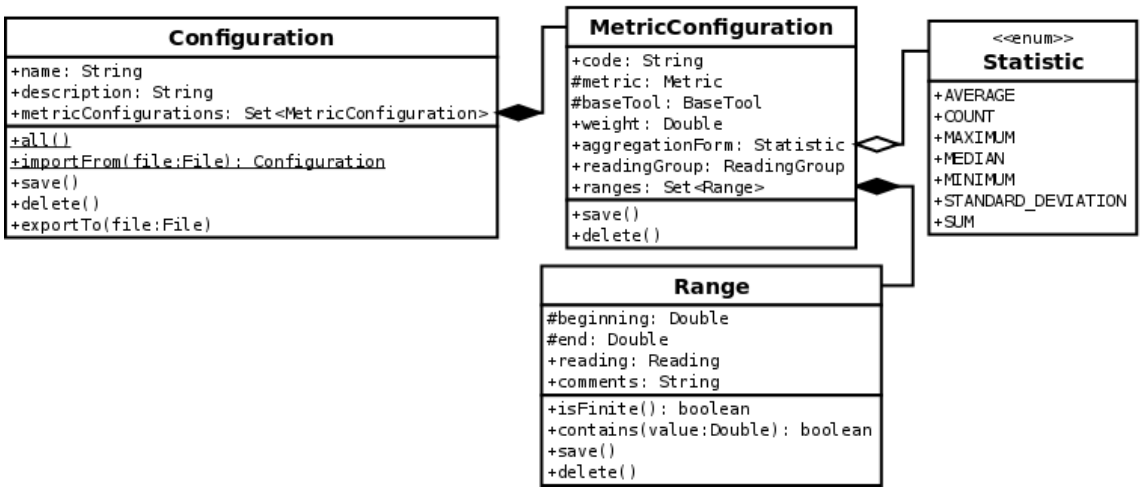


Figura 4.5: *Configuração de métricas*

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 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 4.2: *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 4.2. Suponha ainda que a ferramenta-base obtenha os valores CBO = 3 e LCOM4 = 2. Internamente, Kalibro construirá o Javascript da Listagem 4.1 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 4.1: *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; 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 e 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 4.5.4, 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[5], um padrão de seriação de objetos feito para ser de fácil manipulação em diferentes linguagens de programação, além de resultar em arquivos humanamente legíveis.

4.1.3 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 4.6). 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.

O método de classe `Repository.supportedTypes()` devolve um subconjunto dos tipos mostrados na Figura 4.6, indicando quais são suportados na máquina em que o Kalibro está executando. Para obter o código-fonte a partir do repositório, Kalibro faz uso de chamadas de sistema

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

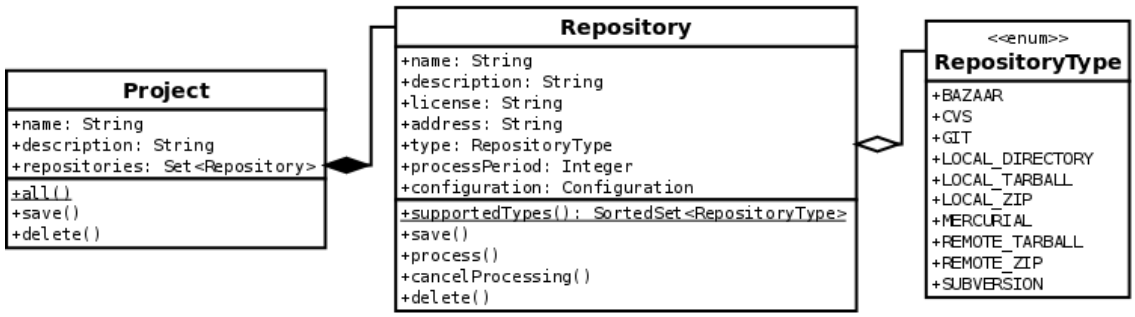


Figura 4.6: *Projetos e repositórios*

para ferramentas instaladas. Quando a ferramenta não está instalada ou de alguma forma inacessível, o tipo correspondente não aparece entre os suportados. O usuário deve levar em consideração que criar repositórios com tipos não suportados 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 4.7). 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:

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

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 `ProcessState.COLLECTING`, a análise pode já ter começado. Se o estado consultado for `ProcessState.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 4.7, modelam os mesmos conceitos das classes `NativeModuleResult` e `NativeMetricResult`, apresentadas na Figura 4.3, porém com mais dados, adicionados durante o processamento. De fato, essas classes

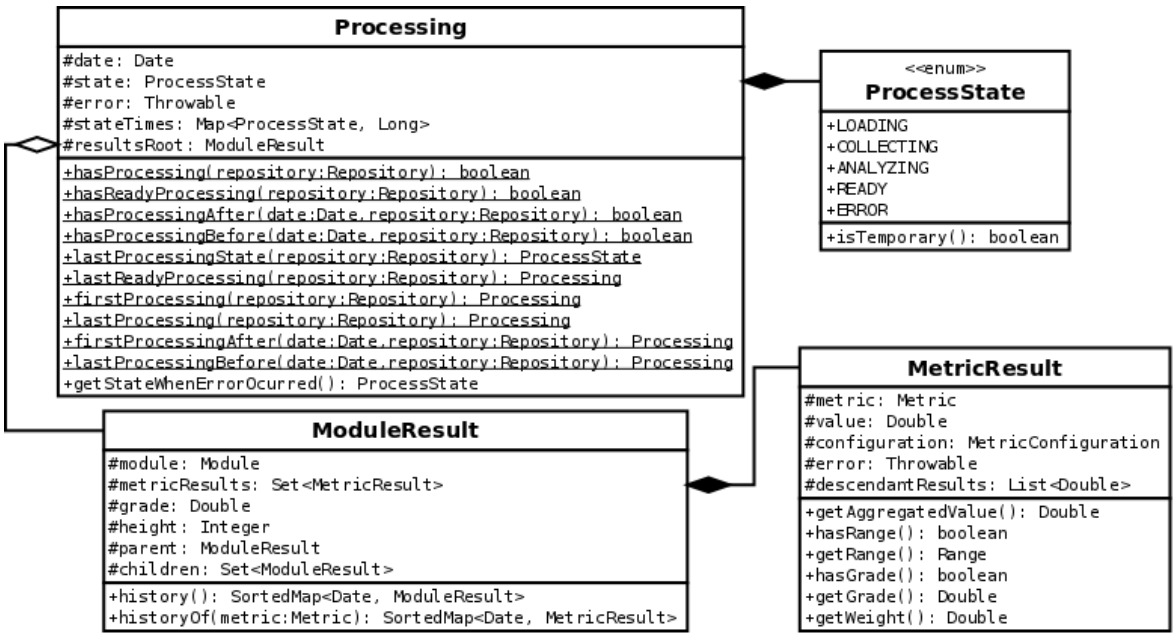


Figura 4.7: *Processamento e resultados*

fazem parte da mesma hierarquia (ver Figura 4.8), mas todas as subclasses contêm mecanismos para assegurar o tipo correto das composições (p.ex., um `ModuleResult` só aceita adição de `MetricResults`, enquanto que um `NativeMetricResult` não pode ser criado para uma métrica composta).

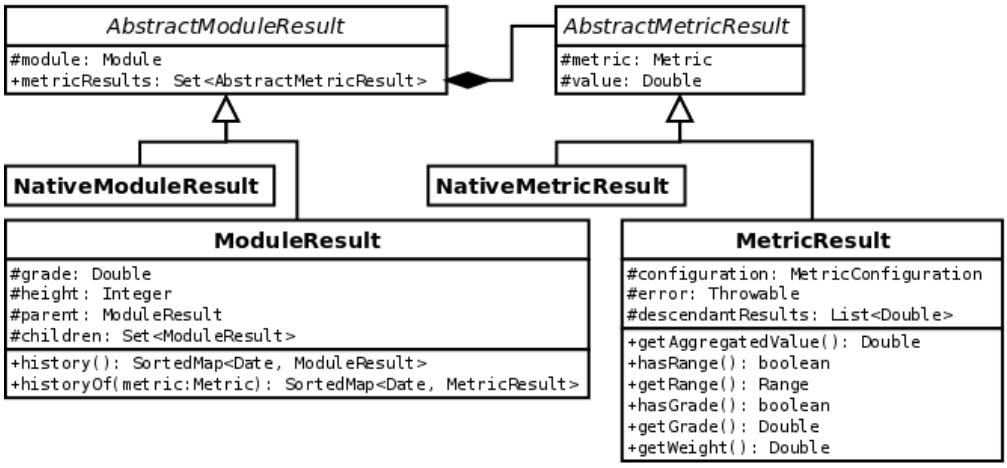


Figura 4.8: *Hierarquia de resultados*

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 que foram processados; e `historyOf(metric)`, que opera da mesma forma mas apenas para uma métrica específica.

4.1.4 Definições

As definições são um conjunto de parâmetros que o Kalibro precisa para funcionar. A classe `KalibroSettings` modela essas definições (ver Figura 4.9). 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, caso em que 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.

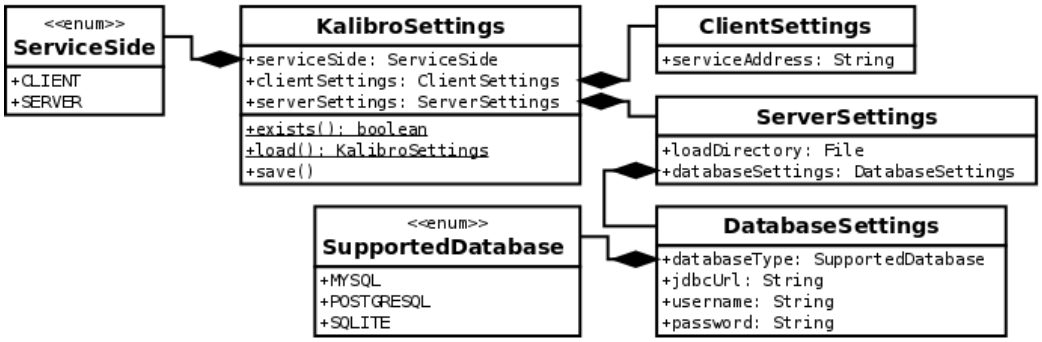


Figura 4.9: *Definições*

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` Kalibro suporta 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/carlos/.kalibro`). Esse diretório é usado com vários propósitos, entre eles manter as definições, que são procuradas dentro dele, em um arquivo chamado `kalibro.settings`. As definições são convertidas em formato YAML, que é de fácil leitura e edição (vide Listagem A.1).

O método de classe `KalibroSettings.load()` obtém a configuração a partir desse arquivo. Esse método é chamado em todos os lugares onde as definições são usadas. O método de instância `save()` faz o contrário, grava as definições no arquivo. O método `exists()` verifica se o arquivo

de definições 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.

4.2 Interface do Serviço

O Kalibro Metrics possui uma interface em forma de serviço Web chamada de 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 grande 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 as definições (descritas na Seção 4.1.4), 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 arcabouço JAX-WS[6] 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 em objetos Java e executa o método correspondente com esses objetos como parâmetros. O retorno do 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 4.10). Usaremos diagramas de classes para as interfaces, por ser um formato de muito mais fácil compreensão do que documentos WSDL.

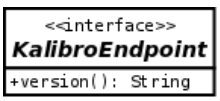


Figura 4.10: *Ponto de acesso principal*

4.2.1 Ponto de acesso de ferramentas-base

O ponto de acesso de ferramentas-base (ver Figura 4.11), 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 4.3). Isso ocorre porque os métodos do

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

ponto de acesso foram feitos para serem usados pela entidade – quando as definições 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 (Alur *et al.*, 2003).

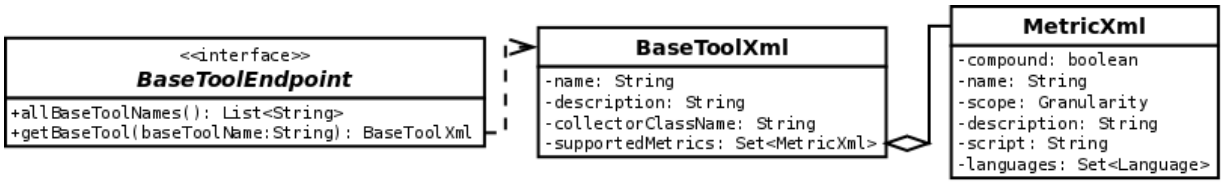


Figura 4.11: *Ponto de acesso de ferramentas-base*

Esse padrão é usado em todos os pontos de acesso, portanto não listaremos cada operação de cada ponto de acesso, nos concentrando somente nas diferenças. A lista completa dos pontos de acesso, suas operações, parâmetros e tipos de retorno pode ser encontrada no Apêndice B.

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 4.2).

4.2.2 Pontos de acesso de cadastros

O ponto de acesso de grupos de leituras (veja Figura 4.12) 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.

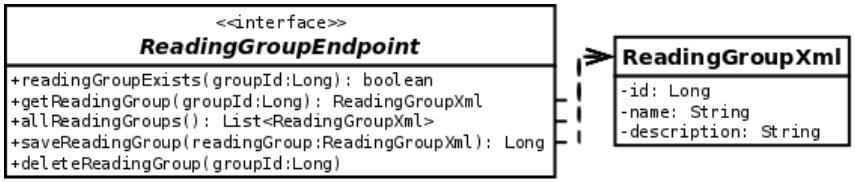


Figura 4.12: *Pontos de acesso de grupos de leituras*

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 de projeto *Lazy Load* (Fowler, 2003), 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 4.13).

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 cadeia de 6 caracteres,

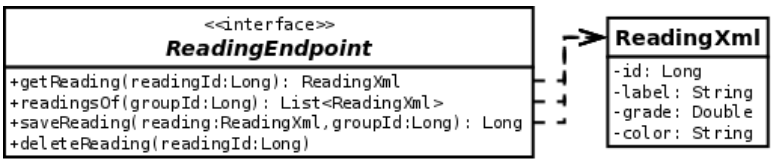


Figura 4.13: *Pontos de acesso de leituras*

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, enquanto 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, sendo que 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 suportados, iniciar e cancelar processamentos. Deve ser observado que os tipos locais (`LOCAL_DIRECTORY`, `LOCAL_ZIP` e `LOCAL_TARBALL`) nunca são suportados 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.

4.2.3 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`), que estão detalhados no Apêndice B.

Um ponto que vale a pena ressaltar são os objetos de transferência de erros (ver Figura 4.14). Eles podem aparecer em resultados de métricas compostas ou processamentos ou malsucedidos, 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.

Outro ponto importante é: a configuração associada a um resultado de métrica é apenas uma fotografia da original (ver Figura 4.15). Quando um repositório é processado, uma cópia da configuração associada é guardada junto com o processamento, seguindo o padrão de projeto *Memento*

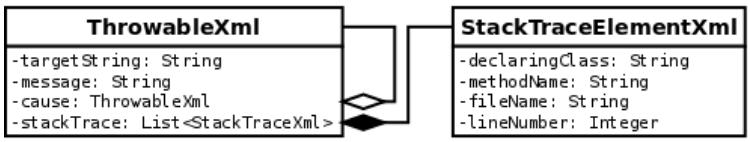


Figura 4.14: *Transferência de erros por XML*

(Gamma *et al.*, 1995). 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`).

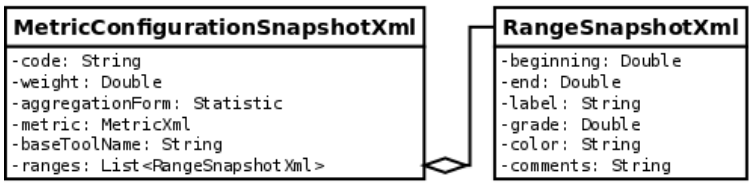


Figura 4.15: *Fotografia de configuração de métricas*

4.3 Organização Interna

Esta seção e a próxima são dirigidas àqueles que querem entender o funcionamento detalhado do Kalibro. Iremos 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[7]. Para tal, é necessário ter instalado o sistema de controle de versão Git[8] 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 vistas na Seção 4.1, incluindo suas dependências; as implementações dos coletores descritas na Seção 4.5; interfaces dos pontos de acesso e objetos XML vistos na Seção 4.2. 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 descrita na Seção 4.4.

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

- `org.analizo` Implementação do coletor de métricas para Analizo (Seção 4.5.1).
- `org.checkstyle` Implementação do coletor de métricas para Checkstyle (Seção 4.5.2).

[7] http://www.eclipse.org/
[8] http://git-scm.com/

- `org.cvsanaly` Implementação do coletor de métricas para CVSAnalY (Seção 4.5.3).
- `org.kalibro` Classes da API do Kalibro (Seção 4.1).
- `org.kalibro.service` Interfaces e implementações dos pontos de acesso (Seção 4.2).
- `org.kalibro.service.xml` Objetos de transferência de dados por XML (Seção 4.2).
- `org.kalibro.dto` Classes abstratas de transferência de dados (Seção 4.3.1).
- `org.kalibro.dao` Interfaces de acesso a dados (Seção 4.3.1).
- `org.kalibro.client` Implementações de acesso a dados remotos (Seção 4.3.1).
- `org.kalibro.core` Constitui, junto com seus subpacotes, o núcleo do Kalibro propriamente dito. Trataremos dele nas Seções 4.3.2, 4.3.3 e 4.3.4.

Duas classes do pacote `org.kalibro` que ainda não mencionamos 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).

4.3.1 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*, ou simplesmente DAO (Alur *et al.*, 2003). 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* (Gamma *et al.*, 1995). As subclasses de `DaoFactory` implementam o acesso ao banco de dados ou ao serviço remoto; a subclasse escolhida depende das definições (Seção 4.1.4), como ilustrado na Figura 4.16.

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 definições de cada máquina. Observe o diagrama de sequência da Figura 4.17. O método `save()` do projeto dispara o método `save(project)` usando a interface `ProjectDao`. Se estivesse na máquina servidora, a chamada concreta seria feita

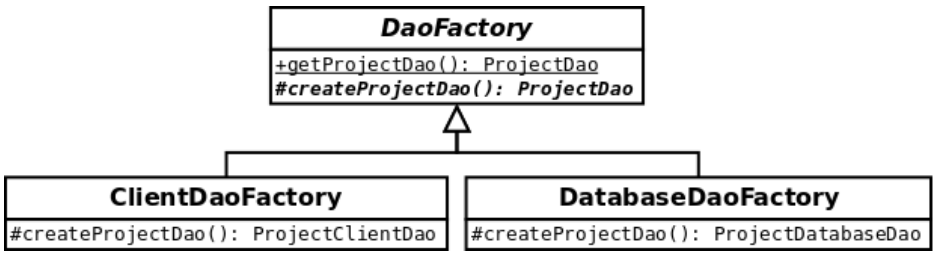


Figura 4.16: *Fábricas de objetos de acesso a dados*

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.

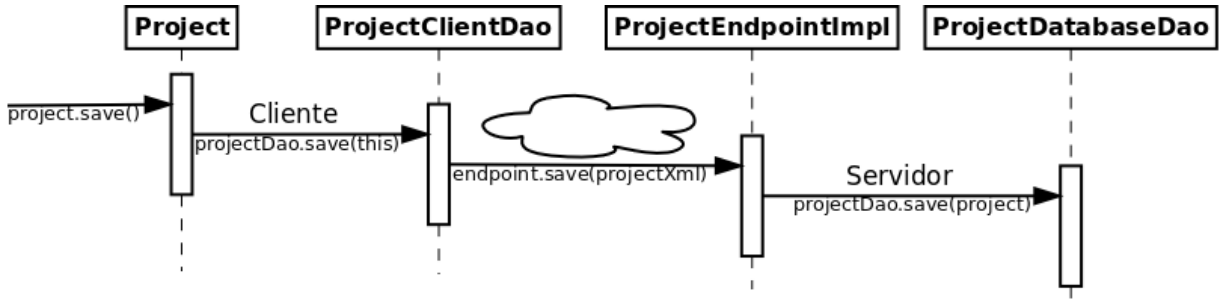


Figura 4.17: *Fluxo de um acesso remoto*

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 persiste dados usando o Eclipselink[9], implementação de referência da JPA[10], 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 (Alur *et al.*, 2003), 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`.

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 4.18). 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* (Gamma *et al.*, 1995). Essa arquitetura promove o desacoplamento entre as diferentes representações de dados e abstrai o que há de comum nas suas conversões.

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

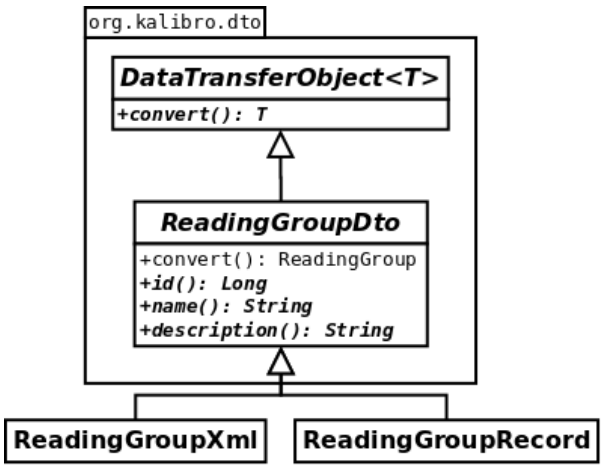


Figura 4.18: *Objeto abstrato de transferência*

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 4.18, 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*[11] 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*, Kalibro usa a biblioteca CGLIB[12] 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[13], Spring[14] e Powermock[15].

Os bancos de dados suportados pelo Kalibro são Mysql[16], PostgreSQL[17] e SQLite[18]; a escolha é feita através do arquivo de definições. No diretório `META-INF` do projeto `KalibroCore` encontram-se *scripts* de criação do banco de dados, um para cada tipo. O Kalibro tem foco no 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.

[11] Um *proxy* é um objeto que controla o acesso a outro de mesma interface, delegando qualquer operação ao objeto original (Gamma *et al.*, 1995).
[12] http://cglib.sourceforge.net/
[13] http://www.hibernate.org/
[14] http://www.springsource.org/
[15] http://code.google.com/p/powermock/
[16] http://www.mysql.com/
[17] http://www.postgresql.org/
[18] http://www.sqlite.org/

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.

O Apêndice C apresenta a estrutura das tabelas do Kalibro. Note que 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*).

4.3.2 Abstração das entidades

O pacote `org.kalibro.core.abstractentity` possui a classe abstrata `AbstractEntity` e suas auxiliares. Ele 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 4.19):

- `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 4.1.2).

- `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; 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, 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.

Para que a classe `AbstractEntity` possa realizar todas essas operações sem conhecer de antemão cada uma de suas subclasses, é necessário usar reflexão[19]. 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`:

[19]Reflexão é a capacidade de um programa consultar ou mesmo alterar a estrutura ou comportamento de objetos em tempo de execução (Sobel e Friedman, 1996)

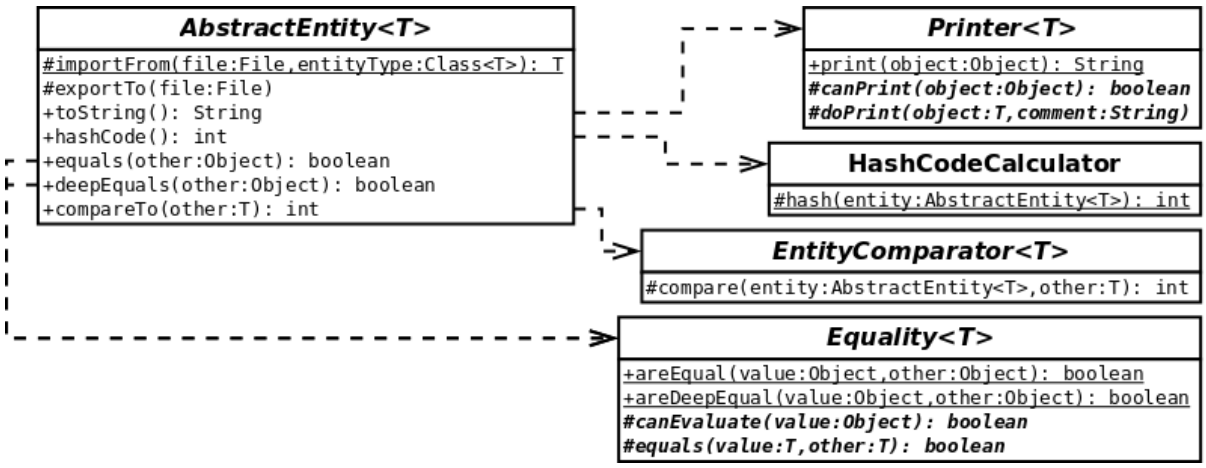


Figura 4.19: *Abstração de entidade e auxiliares*

- `IdentityField` Marca um atributo como parte da identidade do objeto; 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; 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 4.19, 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, seguindo o algoritmo recomendado por Bloch (2001).

- `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; 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.

4.3.3 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 que, para isso, dispara novas *threads* 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 o gerenciamento da 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 4.20.

A classe abstrata `Task` modela uma tarefa; 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:

- `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.

A obtenção do resultado de uma tarefa executada de forma assíncrona é feita usando o padrão de projeto *Observer* (Gamma *et al.*, 1995). O método `Task.addListener(listener)` registra um ouvinte à tarefa. Esses ouvintes são implementações da interface `TaskListener`; 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* 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 `Producer` é iterável, ou seja, os objetos produzidos podem ser obtidos em um laço simples como se

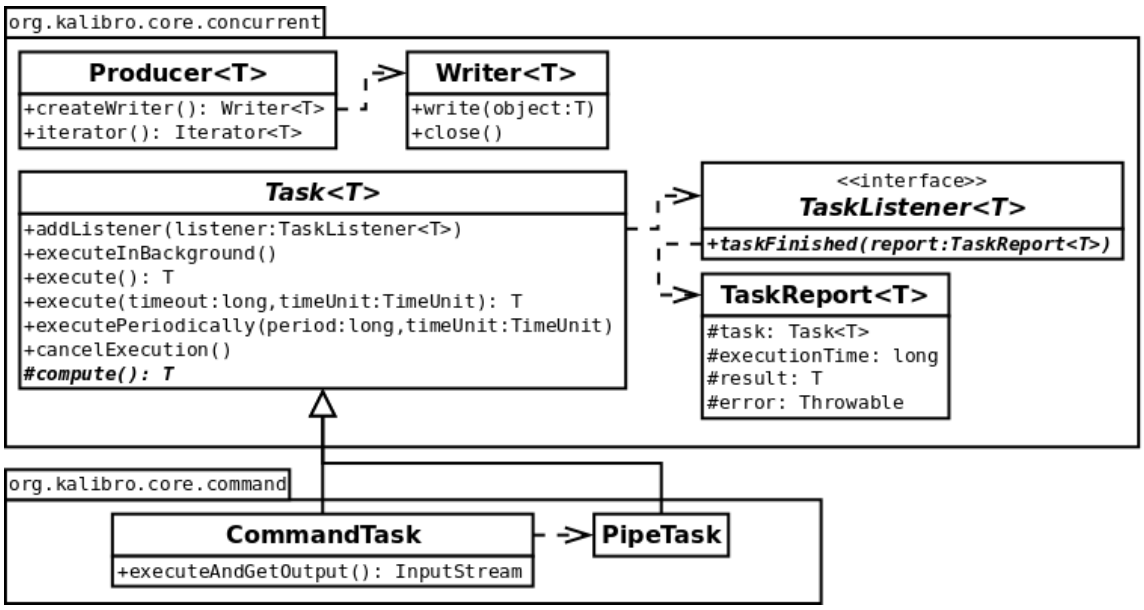


Figura 4.20: *Classes que gerenciam concorrência*

estivessem em uma coleção. A Listagem 4.2 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 4.2: *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.

4.3.4 Processamento

O processamento de repositórios é realizado em três 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`; 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 4.21.

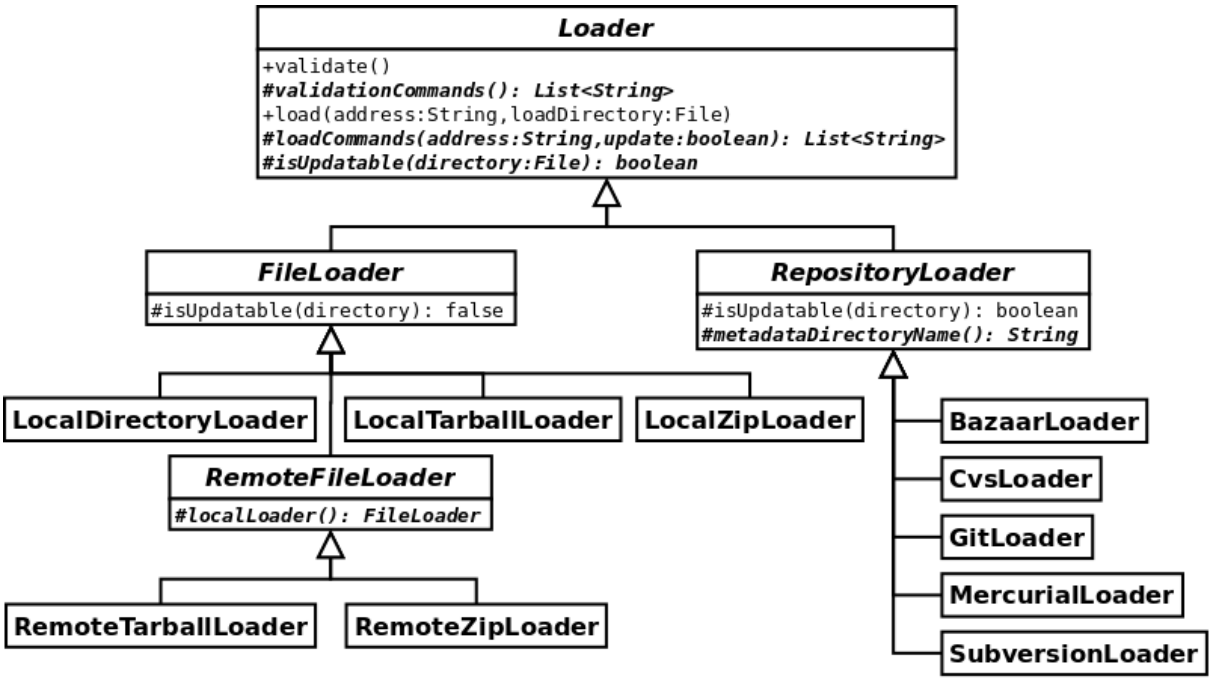


Figura 4.21: *Hierarquia de carregadores*

A superclasse de todos os carregadores é a classe abstrata `Loader`; seus métodos seguem o padrão de projeto *Template Method* (Gamma *et al.*, 1995). 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 (p.ex., `.svn` para Subversion). Os sistemas com que o Kalibro trabalha atualmente são Bazaar[20], CVS[21], Git[22], Mercurial[23] e Subversion[24].

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

[20] http://bazaar.canonical.com/en/
[21] http://cvs.nongnu.org/
[22] http://git-scm.com/
[23] http://mercurial.selenic.com/
[24] http://subversion.apache.org/

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 ele não requer a instalação dos sistemas para os quais o usuário não precisa de suporte: apenas os carregadores validados são utilizados. A desvantagem é a menor portabilidade, pois os carregadores utilizam comandos preestabelecidos, 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, 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 4.22.

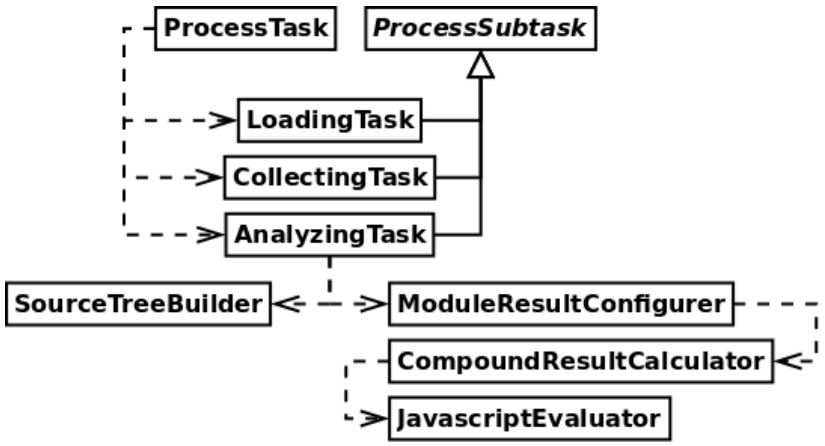


Figura 4.22: *Tarefas de processamento*

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` 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 4.3.3). 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.

4.4 Testes Automatizados

O projeto `KalibroTests` contém os testes automatizados do Kalibro e possui um diretório de código-fonte para cada um dos 3 níveis de teste:

- `unit` Contém os testes de unidade. Cada teste de unidade é direcionado a uma classe de produção, verificando se sua interface é fácil de usar e se seu comportamento é o esperado.
- `integration` Possui testes de integração. Testes de integração verificam se o sistema interage bem com código que não faz parte dele. No Kalibro, estão testadas a integração com os carregadores, as ferramentas-base integradas ao Kalibro e as principais bibliotecas utilizadas.
- `acceptance` Possui testes de aceitação. Os testes de aceitação verificam se o sistema funciona do ponto de vista externo. Cada funcionalidade é testada através de uma execução de ponta a ponta do sistema.

Os testes de unidade oferecem bastante *feedback* sobre a qualidade interna e quase nenhum sobre a qualidade externa do sistema, os testes de aceitação fazem o oposto, enquanto os testes de integração ficam em algum lugar no meio desses dois níveis de teste (Freeman e Pryce, 2009). Devido aos seus objetivos e restrições diferentes, cada nível de teste possui requisitos de qualidade diferentes. Por exemplo, se espera que testes de unidade sejam extremamente rápidos, pois cada classe deve ser simples e testada de forma isolada. Já testes de aceitação devem simular o sistema em produção, com acesso a disco e esperas por tarefas concorrentes, portanto os requisitos de velocidade são afrouxados. Enquanto que em testes de aceitação a criação de cenários complexos pode ser necessária para testar uma funcionalidade específica do sistema, a necessidade de criá-los em testes de unidade aponta que talvez a classe testada esteja demasiadamente complexa e precise ser quebrada.

De modo geral, ao desenvolver os testes automatizados do Kalibro, procuramos perseguir as seguintes qualidades (do acrônimo *FIRST* Martin (2008)):

- *Fast*: Rapidez é muito importante para qualquer teste automatizado e crucial para testes de unidade. Os programadores devem poder executar baterias de testes várias vezes sem atrapalhar o processo de desenvolvimento, para obter *feedback* rápido.

- *Isolated*: Testes devem ser independentes. Os resultados devem ser os mesmos qualquer que seja a ordem de execução dos testes. Um teste não pode preparar o ambiente de outro teste.

- *Repeatable*: Testes devem ser repetíveis, ou seja, duas execuções do mesmo teste sobre o mesmo código deve produzir sempre o mesmo resultado. Consequência disso é que os testes devem produzir os mesmos resultados em máquinas diferentes.

- *Self-validating*: O desenvolvedor deve saber instantaneamente se o sistema passou ou falhou no teste, ao olhar sua saída. Não deve ser preciso comparar textos ou analisar tabelas para determinar se houve falha. Caso contrário, seu resultado torna-se subjetivo.

- *Timely*: Os testes devem ser escritos logo antes do código que o faz passar. Desse modo, o código de produção será escrito tendo em vista sua clareza de interface e testabilidade. Esse paradigma de programação é conhecido como TDD (Beck, 1999).

O último diretório de código-fonte (`src`) contém arquivos e classes auxiliares que procuram dar suporte ao alcance dessas qualidades. Nele estão incluídos casos de teste abstratos, classes utilitárias e *fixtures*[25]. As classes mais abstratas e de uso geral estão no pacote `org.kalibro.tests`; as principais delas estão representadas na Figura 4.23 (aparecem apenas seus métodos mais ilustrativos).

Primeiramente, observemos as classes `UnitTest`, `IntegrationTest` e `AcceptanceTest`. Elas correspondem a cada nível de teste e estão organizadas seguindo uma hierarquia, mais por conveniência que por precisão conceitual, pois em geral os testes de um nível faz uso das mesmas facilidades do nível precedente. São exemplos disso os métodos de classe mostrados na Figura 4.23: `loadFixture()` devolve *fixtures*, necessárias em qualquer nível; `cleanLogs()` apaga os arquivos de registro, operação necessária a ser executada ao término dos casos de teste de integração ou aceitação, mas não de unidade.

A classe `UnitTest` define uma regra de tempo limite (*timeout*) para a execução de cada teste. Um teste é interrompido e tido como falha se ele demorar mais que seu tempo limite. Com esse artifício se evita que uma bateria inteira de testes seja travada por apenas um, que por acaso entre em laço ou espera infinita. As classes correspondentes aos outros níveis sobrescrevem a regra, a afrouxando para refletir seus requisitos de velocidade. Note que os tempos limites são bastante generosos (testes de unidade deveriam levar uma fração de segundo, não 2 segundos). Isso é necessário para alcançar a repetibilidade (um teste passando com tempo limite exigente poderia falhar em uma

[25] *Fixtures* são objetos ou simples conjuntos de dados fixos e bem conhecidos usados consistentemente para criar os contextos dos casos de teste.

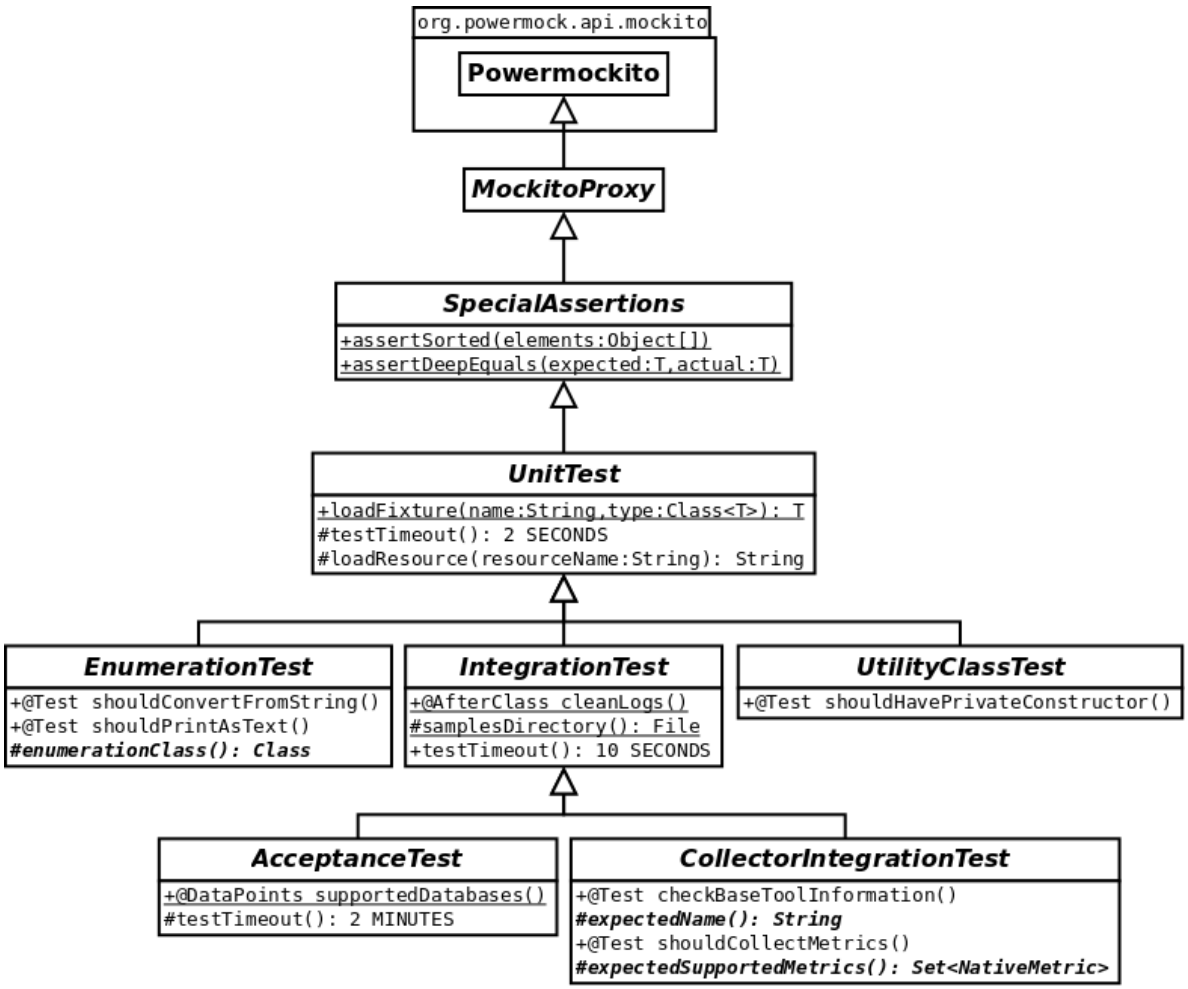


Figura 4.23: *Hierarquia de testes abstratos*

máquina mais lenta), mas o desenvolvedor deve procurar criar testes que passam em tempo muito abaixo do limite. Portanto, ultrapassar o tempo limite é clara evidência de que o teste precisa ser repensado.

A Figura 4.23 mostra ainda que acima de `UnitTest` na hierarquia de classes existem outras classes utilitárias: `SpecialAssertions`, com asserções frequentemente usadas no Kalibro, e `MockitoProxy` cujo objetivo é facilitar e resolver conflitos de importação de métodos estáticos das bibliotecas *Mockito* e *Powermock*. Esse padrão segue o mesmo da biblioteca *Powermock*, com a classe `Powermockito` descendendo de outras classes utilitárias.

Nem todos os testes descendem diretamente da classe relativa ao nível do teste (`UnitTest`, `IntegrationTest` ou `AcceptanceTest`). Alguns pertencem a uma família de testes com necessidades em comum, portanto existem outras classes de teste abstratas abaixo das 3 básicas. As necessidades em comum dos testes refletem as características em comum das classes testadas, portanto temos testes abstratos para classes irmãs e para aquelas que seguem o mesmo padrão de desenho, como tipos enumerados e classes utilitárias (`EnumerationTest` e `UtilityClassTest`).

No desenvolvimento do Kalibro, o *plug-in* do Eclipse Eclemma[26] foi utilizado para acompanhar a cobertura dos testes, principalmente os de unidade. Quando essa ferramenta mostra que o código foi 100% coberto, isso significa que todas as instruções foram executadas, mas não garante que todos os cenários foram testados, pois a mesma instrução pode ser executada com a classe em estados diferentes. O requisito mínimo de cobertura deve ser 100%, pois código não coberto ou precisa ser testado ou é inútil. Para manter a quarta qualidade de teste (auto-validável, i.e., saída booleana) é

[26] http://www.eclemma.org/

preciso cobrir até código morto[27]. Por exemplo, classes utilitárias possuem apenas métodos de classe e devem ter um construtor privado somente para proibir a criação de instâncias. Esse construtor é código morto e não precisa ser testado. Mas o teste `UtilityClassTest` verifica a existência desse construtor e o executa para completar a cobertura. Se isso não fosse feito, o Eclemma (que não distingue código morto) acusaria uma cobertura menor que 100% a cada bateria de testes e o programador perderia tempo procurando inutilmente pelo código não coberto.

Além de possuir uma interface simples para ferramentas-base, o Kalibro oferece um modelo de teste para a integração com o coletor: a classe abstrata `CollectorIntegrationTest`. Para implementá-la do modo mais simples basta seguir o padrão proposto:

1. Usar um arquivo `description`, no mesmo pacote do coletor, para obter a descrição da ferramenta.
2. Especificar o conjunto de métricas implementadas esperado.
3. Adicionar um diretório com o código-fonte de teste no arquivo `Resources/tests.tar.gz`. Esse arquivo é extraído no diretório de usuário pelo *script* `configure.sh` (ver comandos no início da Seção 4.3) e daí usado nos testes.
4. Especificar os resultados esperados para os fontes adicionados no passo anterior. Os resultados podem ser colocados em um arquivo chamado `expected-results.yml`, no mesmo pacote do coletor, para ser detectado automaticamente. Espera-se que esse arquivo esteja no formato YAML e contenha um mapa (módulo -> nome da métrica -> valor esperado). Um extrato do arquivo da Analizo é apresentado na Listagem 4.3 como exemplo.

```
1 ?
2   !!org.kalibro.Module [SOFTWARE, ["null"]]
3 :
4   "Total Abstract Classes": "0"
5   "Total Coupling Factor": "1"
6 ?
7   !!org.kalibro.Module [CLASS, ["HelloWorld"]]
8 :
9   "Average Cyclomatic Complexity per Method": "1"
10  "Average Method LOC": "4"
```

Listagem 4.3: *Exemplo de arquivo de resultados esperados*

4.5 Integração

Uma das qualidades mais procuradas ao projetar o Kalibro é a facilidade de integração. Esta seção mostra como foi feita a integração do Kalibro com outros programas. Primeiro mostraremos como a interface `MetricCollector` foi implementada para comunicação com cada uma das ferramentas-base suportadas. Depois, discutiremos como o Kalibro Service foi ou está sendo usado para dar suporte a outras plataformas.

4.5.1 Analizo

Analizo Metrics é uma ferramenta escrita em Perl que faz uso de extratores para analisar código-fonte (ver Capítulo 3). A Analizo foi escolhida como nossa primeira ferramenta base, por ser bastante eficiente na análise de código C, C++ e Java, além de possuir um grande número de métricas embutidas (Terceiro *et al.*, 2010). Seu coletor simplesmente executa a Analizo através de uma chamada de sistema e traduz a sua saída padrão em métricas e resultados. Para entender melhor como a tradução funciona, observe a saída da Analizo nas listagens a seguir[28]. A Listagem 4.4 mostra a

[27] Código morto é aquele que garantidamente nunca será executado.

[28] Ambas as listagens foram reduzidas, com algumas métricas e estatísticas suprimidas para facilitar a visualização.

saída do comando de listar as métricas e a Listagem 4.5 mostra a saída do comando que realiza a análise para um arquivo de exemplo. Note que é preciso primeiro capturar as abreviaturas para poder traduzir a saída da análise, associando-as às métricas correspondentes.

```
1 Global Metrics:
2 total_cof - Total Coupling Factor
3 total_loc - Total Lines of Code
4 total_nom - Total Number of Methods
5
6 Module Metrics:
7 amloc - Average Method LOC
8 cbo - Coupling Between Objects
9 dit - Depth of Inheritance Tree
10 lcom4 - Lack of Cohesion of Methods
11 loc - Lines of Code
12 noc - Number of Children
13 nom - Number of Methods
14 rfc - Response For a Class
```

Listagem 4.4: *Saída da listagem de métricas da Analizo*

```
1 ---
2 total_cof: 1
3 total_loc: 4
4 total_nom: 1
5 ---
6 _module: HelloWorld
7 amloc: 4
8 cbo: 0
9 dit: 0
10 lcom4: 1
11 loc: 4
12 noc: 0
13 nom: 1
14 rfc: 1
```

Listagem 4.5: *Exemplo de saída da análise de código-fonte da Analizo*

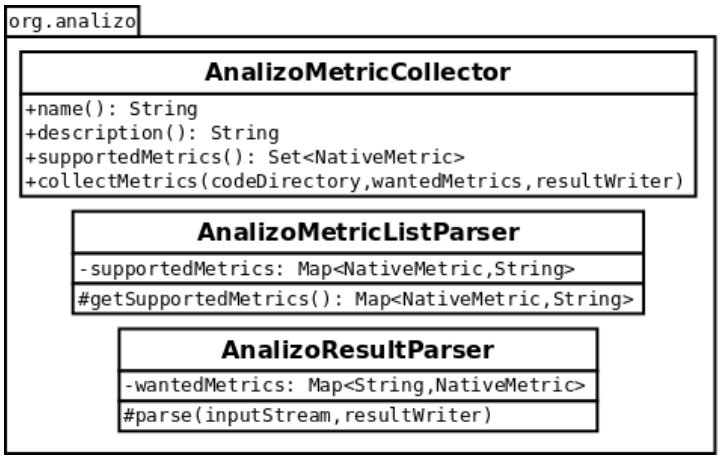


Figura 4.24: *Coletor de métricas da Analizo*

A classe coletora (`AnalizoMetricCollector`) conta com duas classes auxiliares: uma para traduzir a listagem e outra para traduzir os resultados da análise (ver Figura 4.24). A listagem é

traduzida em um mapa onde a métrica é a chave e as abreviaturas são os valores. Esse mapa é utilizado tanto pelo coletor (para listar as métricas suportadas) quanto pelo tradutor de resultados (`AnalizoResultParser`). O tradutor de resultados recebe o conjunto de métricas pedidas (parâmetro `wantedMetrics`) e filtra o mapa invertendo-o, para obter as métricas a partir das abreviaturas durante a tradução.

Analizo foi a primeira ferramenta-base integrada ao Kalibro, e a simplicidade de sua integração se reflete nas métricas do pacote `org.analizo`: 3 classes, todas com menos de 30 linhas de código; cada classe possui por volta de 5 métodos, com complexidade ciclomática média menor que 2.

4.5.2 Checkstyle

Checkstyle é uma ferramenta que analisa código Java, acusando violações no padrão de codificação configurado, e pode ser usada para definir um limite máximo ou mínimo para diversas métricas (ver Capítulo 3). Por ser uma ferramenta escrita em Java, seu código pôde ser empacotado como dependência do Kalibro, dispensando instalação separada.

As classes que realizam a integração estão representadas na Figura 4.25. Métricas da Checkstyle são modeladas pela classe `CheckstyleMetric` (descendente de `NativeMetric`) e o conjunto de métricas implementadas é lido do arquivo `supported-metrics.yml`. Cada métrica contém, além dos atributos herdados, as informações necessárias para configurar a Checkstyle de modo a poder coletá-la. A classe `CheckstyleConfiguration` modela uma configuração da Checkstyle, com o conjunto de verificações que devem ser processadas, correspondentes às métricas requisitadas. Por exemplo, se a métrica AMLOC for requisitada, a configuração incluirá uma verificação de tamanho de método com um limite máximo de -1, para que o evento de violação do limite seja gerado para todo método, qualquer que seja seu tamanho, e a média será computada por classe. A classe `CheckstyleListener` é o ouvinte da Checkstyle, que traduz os eventos de violação em resultados de métricas.

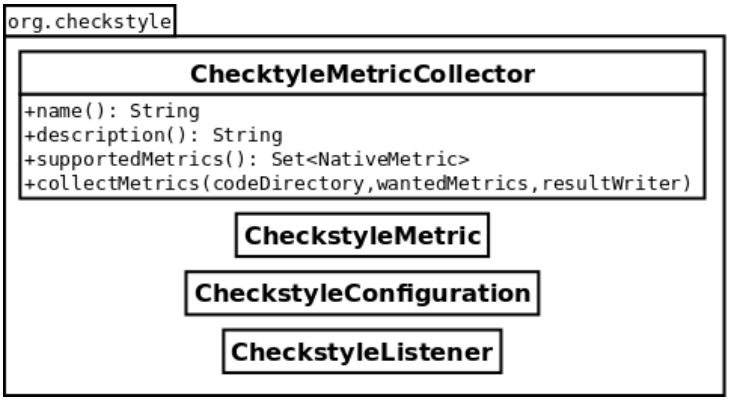


Figura 4.25: *Coletor de métricas do Checkstyle*

Como a Checkstyle não foi desenhada para coletar métricas, suas verificações tiveram que ser adaptadas. As métricas registradas refletem as adaptações de verificações existentes que foram utilizadas como métricas. Essa integração não é definitiva, pois: as verificações usadas podem, se empregadas de outra forma, gerar outras métricas; é possível que hajam outras verificações não exploradas, que resultem em métricas interessantes; e devido à arquitetura flexível da Checkstyle, é possível adicionar novas verificações.

Apesar de não ser tão simples quanto a integração da Analizo, as métricas do pacote `org.checkstyle` ainda apontam pouco esforço de implementação: 4 classes, uma média de 9 métodos por classe; os métodos possuem em média 5 linhas de código e complexidade ciclomática menor que 2. Além do esforço de implementação houve também um esforço de configuração, na forma do arquivo `supported-metrics.yml`

4.5.3 CVSAnalY

A CVSAnalY analisa registros de um repositório de código e monta bancos de dados com diversas informações sobre ele. Uma das tabelas do banco de dados gerado contém resultados de métricas, para cada atualização (*commit*). O coletor executa o CVSAnalY através de uma chamada de sistema, guarda o banco de dados em um arquivo temporário do SQLite e então consulta as métricas da atualização mais recente.

A Figura 4.26 mostra a classe coletora e suas auxiliares. De maneira similar ao Checkstyle, uma métrica do CVSAnalY é modelada pela classe `CvsAnalyMetric`, que estende `NativeMetric` adicionando um atributo para identificar o nome da coluna na tabela de métricas que fornece o valor correspondente. A classe `CvsAnalyDatabaseFetcher` monta e executa uma consulta que inclui as colunas correspondentes às métricas requisitadas.

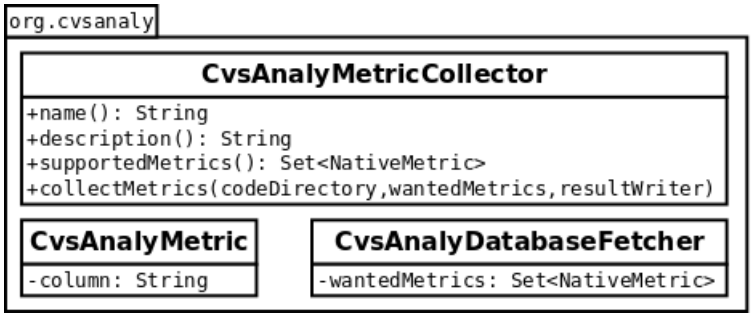


Figura 4.26: *Coletor de métricas do CVSAnalY*

Essa integração faz uso bastante superficial de todo o potencial do CVSAnalY como ferramenta coletora de métricas. O que o Kalibro requer de suas ferramentas-base é apenas a geração de resultados de métricas a partir de um código-fonte fixo. Mas o CVSAnalY vai além, lendo os registros do repositório e coletando métricas para cada atualização, o que seria bastante útil se o Kalibro suportasse a análise do histórico de repositórios (voltaremos a esse tema na Seção 4.6). Além disso, a integração atual explora apenas a tabela de métricas de código-fonte, mas as outras tabelas geradas pelo CVSAnalY poderiam ser utilizadas para geração de métricas sociais.

Nessa integração também houve esforço de implementação e configuração, mas menores ainda que na integração com o Checkstyle. Seguem algumas métricas do pacote `org.cvsanaly`: 3 classes, uma média de 5 métodos por classe; os métodos possuem em média 6 linhas de código e complexidade ciclomática menor que 2.

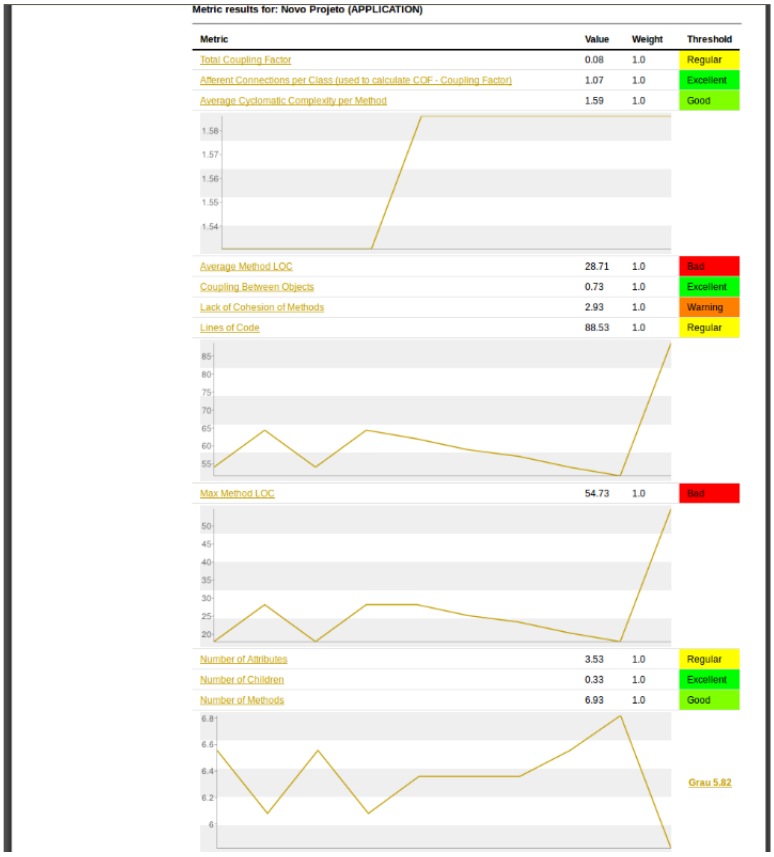
4.5.4 Mezuro

Mezuro é "uma rede sociotécnica 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" (Meirelles, 2013). O Mezuro está sendo desenvolvido com base na plataforma de redes sociais Noosfero[29], através de um *plug-in* que se comunica com o Kalibro Service para fornecer a funcionalidade de análise e avaliação de métricas de código-fonte. Na prática, ele funciona como uma camada de visualização do Kalibro Service, potencializando o compartilhamento de configurações e resultados por se basear em uma rede social.

O Noosfero foi desenvolvido com o arcabouço *Ruby on Rails*[30]. A gema[31] Savon[32] (cliente SOAP) foi utilizada para comunicação com a interface Web Service do Kalibro. As entidades do modelo do Mezuro refletem o modelo do Kalibro, mas seguindo o padrão *Model-View-Controller* do Rails. A conexão com os pontos de acesso do Kalibro Service substitui a persistência no banco de dados do Noosfero e é feita pelas entidades do modelo, ficando desacoplada do resto da aplicação.

[29] http://noosfero.org
[30] http://www.rubyonrails.com.br/
[31] Uma gema é uma biblioteca ou arcabouço Ruby.
[32] http://savonrb.com

Figura 4.27: *Configurando uma métrica com intervalos no Mezuro*



41

Figura 4.28: *Visualizando resultados de métricas no Mezuro*

As configurações criadas por líderes de projeto ou pesquisadores podem ser melhoradas ao longo do tempo. Além de fornecer as funcionalidades do Kalibro para o usuário final, o Mezuro é um ambiente facilitador do compartilhamento de configurações e discussão em torno delas. A Figura 4.27 mostra o cadastro de uma configuração de métrica, com seus intervalos, no Mezuro. A Figura 4.28 mostra resultados de métricas para um projeto Java, com gráficos para o histórico de algumas métricas.

4.6 Discussão da Arquitetura

Procuramos desenvolver as funcionalidades do Kalibro de forma modular e desacoplada. Por exemplo, existem interfaces claras para implementação de coletores de métricas, carregadores de código-fonte de diferentes tipos de repositório, prover suporte a diferentes tipos de banco de dados e disparar eventos a partir do processamento de repositórios. Cada um desses módulos pode ser modificado de forma independente, portanto a curva de aprendizado ao começar a desenvolver com o Kalibro deve ser pequena. Uma arquitetura modular é "tolerante a incertezas" e "acolhe experimentos" (Baldwin e Clark, 2000), tornando mais provável o recebimento de contribuições da comunidade de Software Livre (Baldwin e Clark, 2006).

A rede de testes engloba os três níveis (unidade, integração e aceitação) e tem ótima cobertura. Porém, ainda falta avaliação de desempenho e escalabilidade, estudo que está sendo conduzido pela equipe de desenvolvimento do Mezuro. Os testes de aceitação também apresentam limitações: alguns cenários não foram testados por limitações de tempo ou de ambiente; por testar a integração com todos os bancos de dados suportados, a bateria de testes é relativamente lenta. A bateria de testes de unidade, por envolver cada vez mais testes, também precisará ser otimizada para continuar executando rapidamente.

Como limitação em funcionalidades, temos a falta de mecanismo de autenticação e administração (contas de usuário, senhas, limites de processamentos por usuário, etc). Também falta um mecanismo mais claro e abrangente para facilitar o compartilhamento de configurações, um dos principais objetivos do serviço. Outra funcionalidade desejável e ausente é a comparação automática entre diferentes projetos segundo uma determinada configuração.

Enquanto esta dissertação é escrita, novas funcionalidades estão sendo implementadas. Nos próximos *releases* do Kalibro é provável que estejam presentes:

- Notificação do término de processamento, enviada automaticamente aos e-mails dos interessados.
- Detecção automática do alcance de intervalos críticos.
- Integração com a JaBUTi[33] como ferramenta-base.
- Suporte à coleta de métricas do "passado" de um repositório, com base nos seus registros.

[33] http://code.google.com/p/jabutimetrics/

Capítulo 5

Interpretação de métricas de código-fonte

Neste capítulo construiremos uma configuração de métricas e intervalos baseada nas análises estatísticas realizadas por Meirelles (2013). De posse dessa configuração, mostraremos como ela pode ser usada no Kalibro para ajudar a interpretar e comparar métricas de projetos. Usaremos o Mezuro como plataforma de visualização dos resultados.

Enfatizamos que a configuração aqui construída não pretende propor valores de referência absolutos, mas servir como ponto de partida para posterior aperfeiçoamento e adaptação de acordo com as necessidades específicas de seus usuários. A variedade de distribuições dos valores das métricas em projetos diferentes é um indicador de que não há um padrão absoluto, justificando novamente a necessidade do serviço que permite ao usuário configurar seus próprios valores.

5.1 Seleção de métricas

O Analizo foi utilizado para a realização dos estudos, pois sua grande eficiência mostrou-se necessária dada a quantidade de projetos analisados. Dentre as métricas suportadas pelo Analizo, Meirelles (2013) seleciona um subconjunto representativo, pois constatou-se que muitas delas eram redundantes por medir propriedades muito correlacionadas do código. As métricas selecionadas abrangem diversos aspectos de uma classe:

1. Conexões aferentes, ou ACC (*Afferent Connections per Class*), uma medida de acoplamento.
2. Média da complexidade ciclomática dos métodos, ou ACCM (*Average Cyclomatic Complexity per Method*).
3. Média do tamanho dos métodos, ou AMLOC (*Average Method Lines Of Code*).
4. Média do número de parâmetros por método, ou ANPM (*Average Number of Parameters per Method*).
5. Profundidade na árvore de herança, ou DIT.
6. Número de métodos, ou NOM (*Number Of Methods*).
7. Número de atributos públicos, ou NPA (*Number of Public Attributes*).
8. Complexidade estrutural, ou SC (*Structural Complexity*), uma medida que combina acoplamento (CBO) e coesão (LCOM4).

Dentre as métricas selecionadas, daremos destaque especial para aquelas cuja influência no sucesso, qualidade ou atratividade de projetos de software livre tenha sido evidenciada pela literatura. Midha (2008) verificou que software com altos valores para métricas de complexidade (como

ACCM) tendem a ter mais erros, levar mais tempo para corrigi-los e atrair menos interesse de novos desenvolvedores. Capra *et al.* (2008) define qualidade de software em termos de 5 métricas de orientação a objetos, entre elas CBO, que usamos indiretamente através da métrica SC. Além disso Meirelles *et al.* (2010) mostram que SC influencia a atratividade de projetos de software livre. Barbagallo *et al.* (2008) definiram qualidade de software em termos de métricas de orientação a objeto, entre elas CBO e DIT.

Métrica	**Propriedade**	**Peso**
ACC	acoplamento	2
ACCM	complexidade	2
AMLOC	tamanho	1
ANPM	*fan-in*	1
DIT	herança	1
NOM	tamanho	1
NPA	encapsulamento	1
SC	coesão e acoplamento	4

Tabela 5.1: *Pesos das métricas em nossa configuração*

Os fatores que mais parecem influenciar a manutenibilidade e qualidade percebida do software são complexidade e acoplamento, que procuraremos destacar. Vamos também balancear medidas de complexidade e tamanho pois, como vimos no Capítulo 2, devemos usá-las em conjunto. Nossa sugestão de distribuição de pesos está representada na Tabela 5.1: 2 métricas de tamanho (AMLOC e NOM) se equilibram com o peso 2 da métrica de complexidade (ACCM); as métricas que medem diretamente complexidade e acoplamento (ACC e ACCM) são destacadas com peso 2; e SC, que combina coesão e acoplamento, recebe o maior destaque.

5.2 Configuração de intervalos

Em sua tese, Meirelles (2013) teve o cuidado de separar valores de referência para cada uma das linguagens de programação analisadas: C, C++ e Java. Para nossos fins – proposta inicial de configuração, sem muito rigor, a título ilustrativo – vamos usar a mesma abordagem mas procurando generalizar os valores para as 3 linguagens.

Em geral, nossa interpretação dos valores das métricas segue o lema "quanto maior, pior", pois as propriedades medidas geralmente dificultam a manutenção, prejudicando a qualidade interna: tamanho, acoplamento, complexidade, falta de coesão. Algumas métricas podem também ter valores baixos ruins: DIT muito pequeno pode indicar pouco reaproveitamento de implementação via herança. Esses casos degenerados, que constituem exceção e não regra, podem ser detectados quando analisamos as métricas em conjunto. Por exemplo, uma classe pode ter poucos métodos (NOM baixo, visto sozinho, é bom), mas cada um deles é enorme e faz muita coisa; nesse caso, AMLOC e ACCM terão valores altos.

A seguir, escolheremos três valores de referência para cada métrica. Esses valores serão obtidos baseando-se nos resultados da análise de um ou mais "projetos modelo", analisados no Capítulo 4 da tese de Meirelles (2013). Os valores representarão percentis das métricas nos projetos modelo. Por exemplo, ACC com valor 2 referente ao percentil 75 do Linux significa que 75% das classes/módulos do Linux tem ACC menor ou igual a 2. Dependendo da métrica, escolheremos como projetos modelo aqueles cujos valores mais concordem entre si ou com os valores obtidos na literatura. Os percentis também podem variar, pois as métricas apresentam distribuições estatísticas diferentes (Meirelles, 2013).

ACC é a quantidade de classes usada pela classe medida. As mudanças em uma classe pode afetar o funcionamento de todas as classes que dependem dela. Portanto, depender de muitas classes significa estar mais vulnerável a falhas introduzidas por outras classes. Diversas razões podem estar

por trás de uma classe com muitas dependências: classes com muitas responsabilidades tendem depender de mais classes; uma classe pode ter uma interface que exige de seus clientes o uso de muitas outras, etc. Para valores de referência, escolhemos como modelos os projetos Linux (maior parte em C) e Firefox (maior parte em C++), que parecem concordar em torno dos valores 2, 7 e 15 para os percentis 75, 90 e 95, respectivamente.

ACCM mede a complexidade ciclomática dos métodos. Como vimos no Capítulo 2, maior complexidade está associada a um maior esforço de manutenção. É preferível ter muitos métodos simples do que poucos métodos complexos, pois cada método pode ser entendido, testado e modificado em separado. Métodos mais simples e focados em uma única tarefa tem mais chance de serem reutilizados. Para valores de referência, escolhemos como modelos Linux, Firefox e Java OpenJDK8, que parecem concordar em torno dos valores 3, 5 e 7 para os percentis 75, 90 e 95, respectivamente.

AMLOC mede o tamanho dos métodos em termos de linhas de código. Autores parecem concordar que métodos com boa legibilidade geralmente possuem entre 5 e 15 linhas (Beck, 2007; Martin, 2008; McConnell, 2004) e alguns dos motivos são: métodos devem fazer apenas uma tarefa; devem caber na tela, descartando necessidade de rolagem; devem atuar em apenas um nível de abstração. Linhas de código é uma métrica que varia muito com estilo e linguagem de programação, mas como estamos tratando apenas de C, C++ e Java, que possuem sintaxe muito similares, esse fator não é tão preocupante. Para valores de referência, escolhemos como modelo o Firefox que parece estar de acordo com esses limites; os valores são 8, 19 e 37 para os percentis 50, 75 e 90, respectivamente.

ANPM mede o número de parâmetros dos métodos. Métodos e construtores com um grande número de parâmetros (mais que 3) tendem a ser mais difíceis de entender e usar. Listas similares de parâmetros em diversos métodos geralmente indicam a necessidade de uma nova classe com esses atributos (padrão *ParameterObject*, em Beck (2007)). Parâmetros do mesmo tipo introduzem a dificuldade de usá-los na ordem certa. Mesmo em C, que não possui objetos, as funções tendem a possuir poucos parâmetros. O código do Linux nos traz os valores de referência 2, 3 e 5 para os percentis 75, 90 e 99, respectivamente.

Valores altos de DIT são preocupantes, pois devemos usar herança com parcimônia. Herança viola encapsulamento, visto que detalhes da implementação da classe pai são expostos nas classes filhas. Classes filhas são altamente acopladas às classes pais; uma alteração na classe pai pode ter grandes consequências em todos os seus descendentes. Quando razoável, composição deve ser preferida à herança, pois não sofre esses problemas e ainda possui a flexibilidade da troca de implementação em tempo de execução. Escolhemos o projeto Eclipse (Java) como modelo, que possui valores de referência 2, 4 e 6 para os percentis 75, 90 e 99, respectivamente.

NOM é o número de métodos de uma classe. Quanto mais métodos na mesma classe, mais difícil é entender seu funcionamento. O Chrome (predominantemente C++) e a OpenJDK8 parecem concordar com os valores de referência 10, 17 e 27, correspondentes aos percentis 75, 80 e 95.

NPA é o número de atributos públicos de uma classe/módulo. Atributos públicos são uma violação a um dos princípios mais básicos da orientação a objetos, o encapsulamento. É difícil prever sob que situações o atributo é modificado, pois ele pode ser acessado de qualquer parte do código, tornando difícil a correção de erros. Eles devem ser evitados ao máximo, portanto o valor de NPA tende a ser muito baixo. Eclipse e OpenJDK8 parecem concordar com os valores de referência 1, 2 e 3, correspondentes aos percentis 75, 90 e 95.

Uma classe com alto CBO tem muitas classes dependendo dela, portanto é mais difícil modificá-la, pois modificações acarretariam necessidade de mudança em muitas outras classes. Uma classe pouco coesa (LCOM4 alto) provavelmente viola o princípio da responsabilidade única, apresentando uma necessidade de quebrá-la em duas ou mais classes, o que representa uma grande modificação. Sendo assim, não surpreende que quando combinados, acoplamento e falta de coesão são fatores de risco. A métrica SC, que combina essas duas medidas, já mostrou ter influência na atratividade de projetos de software livre (Meirelles *et al.*, 2010). Reunir diversas responsabilidades em uma só classe a torna pouco coesa e tende a torná-la mais acoplada, pois pode ser usada com finalidades diferentes. Para essa métrica, escolhemos o Chrome como modelo, cujos valores de referência são 12, 28 e 51, correspondentes aos percentis 75, 90 e 95.

Métrica	Intervalos	Rótulo	Comentários para valores altos
ACC	[0, 2[	Excelente	A classe possui muitas dependências. Tente seguir o princípio de responsabilidade única. Talvez seja necessário reduzir a interface das classes utilizadas.
	[2, 7[	Bom	
	[7, 15[	Regular	
	[15, ∞[	Preocupante	
ACCM	[0, 3[	Excelente	Os métodos estão muito "pesados". Tente quebrá-los em métodos menores e menos complexos. Evite longos "switches", ou vários laços aninhados no mesmo método.
	[3, 5[	Bom	
	[5, 7[	Regular	
	[7, ∞[	Preocupante	
AMLOC	[0, 8[	Excelente	Os métodos estão muito longos. Tente quebrá-los em métodos menores, cada um no seu nível de abstração. Forte candidato a novo método é o código em maior aninhamento.
	[8, 19[	Bom	
	[19, 37[	Regular	
	[37, ∞[	Preocupante	
ANPM	[0, 2[	Excelente	Os métodos estão recebendo muitos parâmetros. Conjuntos de parâmetros parecidos sugerem a criação de uma classe contendo esses dados, e transferência dos métodos para essa classe.
	[2, 3[	Bom	
	[3, 5[	Regular	
	[5, ∞[	Preocupante	
DIT	[0, 2[	Excelente	Talvez o mecanismo de herança esteja sendo utilizado em demasia. Sempre que fizer sentido, prefira composição à herança.
	[2, 4[	Bom	
	[4, 6[	Regular	
	[6, ∞[	Preocupante	
NOM	[0, 10[	Excelente	Muitos métodos em uma só classe a torna mais difícil de entender. Talvez o princípio da responsabilidade única esteja sendo violado. Talvez alguns métodos não sejam necessários.
	[10, 17[	Bom	
	[17, 27[	Regular	
	[27, ∞[	Preocupante	
NPA	[0, 1[	Excelente	Em linguagens orientadas a objetos, atributos públicos são uma aberração e devem ser evitados ao máximo. Esconda os atributos e forneça métodos para controlar o acesso a eles.
	[1, 2[	Bom	
	[2, 3[	Regular	
	[3, ∞[	Preocupante	
SC	[0, 12[	Excelente	Complexidade estrutural é um fator de risco. Verifique se a classe pode ser dividida em classes menores e mais coesas. Classes frutos dessa divisão tendem a ser menos acopladas.
	[12, 28[	Bom	
	[28, 51[	Regular	
	[51, ∞[	Preocupante	

Tabela 5.2: *Intervalos sugeridos para as métricas selecionadas*

Os valores de referência e comentários aqui apresentados encontram-se sumarizados na Tabela 5.2, na forma de intervalos a serem usados em uma configuração do Kalibro. Para cada métrica, usamos os três valores de referência para montar quatro intervalos, com rótulos e notas padronizadas, como explicado na Seção 4.1.2: "Excelente" com nota 10, "Bom" com nota 7,5, "Regular" com nota 5, e "Preocupante" com nota 2,5.

5.3 Interpretação

Nesta seção demonstramos como configurações podem ser usadas para avaliar e comparar projetos de software. Submetemos ao Kalibro Service os repositórios dos projetos VLC[1] e MPlayer[2], como cadastrados no Ohloh, um diretório aberto que apresenta métricas e análises sobre milhares de projetos de software livre [3]. Escolhemos comparar esses dois projetos pelos seguintes motivos:

[1] http://www.videolan.org/vlc/
[2] http://www.mplayerhq.hu/
[3] http://www.ohloh.net/

- Ambos são populares.
- São escritos na mesma linguagem de programação (linguagem C).
- Têm o mesmo domínio de aplicação (ambos são reprodutores de mídia).
- São concorrentes de tamanho comparável.

As figuras a seguir mostram os resultados do processamento do VLC (Figura 5.1) e do MPlayer (Figura 5.2), aplicada a configuração proposta e visualizada no Mezuro. Os valores mostrados são as médias dos valores de cada métrica para todos os módulos do projeto. As figuras também mostram os rótulos dos intervalos associados a esses valores, coloridos de forma a chamar atenção para aqueles classificados como "preocupantes" (na cor laranja). Ao passar o cursor sobre o rótulo, aparece o comentário associado ao intervalo.

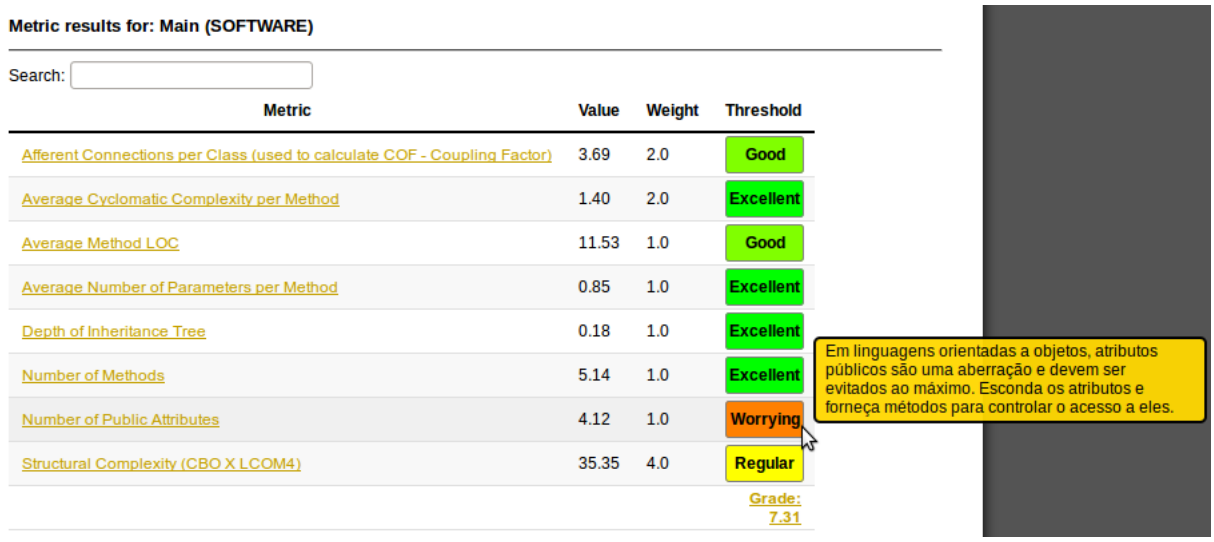


Figura 5.1: *Resultados do VLC*

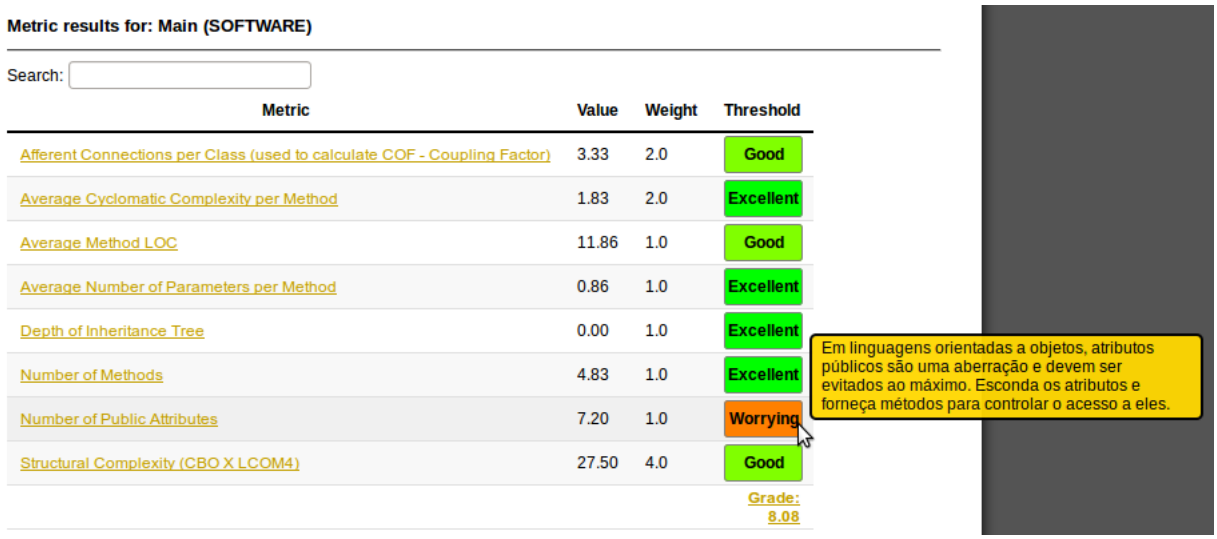


Figura 5.2: *Resultados do MPlayer*

No canto inferior direito de cada figura podemos ver a nota do projeto (7,31 para o VLC e 8,08 para o MPlayer), que é a média das notas associadas aos intervalos, ponderada pelos pesos das métricas na configuração. É interessante notar que os valores são parecidos para quase todas as métricas[4], talvez pelo fato dos projetos serem tão parecidos: ambos possuem em torno de 500 mil

[4] A única que puxa a nota do VLC para baixo em relação ao MPlayer é SC, a métrica de maior peso.

linhas de código; ambos têm aproximadamente 13 anos de existência; ambos têm algumas centenas de desenvolvedores; a atividade de ambos é intensa, com *commits* regulares (dados do Ohloh).

O único resultado classificado como "preocupante" para ambos os projetos é o da métrica NPA (número de atributos públicos), o que não surpreende, pois os projetos foram escritos em C, linguagem estruturada sem o conceito de encapsulamento. No geral, os resultados são excelentes para ambos os projetos, demonstrando a qualidade interna deles.

Metric results for: Main (SOFTWARE)

Search:

Metric	Value	Weight	Threshold
Afferent Connections per Class (used to calculate COF - Coupling Factor)	2.75	2.0	Good
Average Cyclomatic Complexity per Method	1.06	2.0	Excellent
Average Method LOC	4.36	1.0	Excellent
Average Number of Parameters per Method	0.39	1.0	Excellent
Depth of Inheritance Tree	2.27	1.0	Good
Number of Methods	5.88	1.0	Excellent
Number of Public Attributes	0.02	1.0	Excellent
Structural Complexity (CBO X LCOM4)	5.76	4.0	Excellent
			Grade: 9.42

Figura 5.3: *Resultados do Kalibro*

Processamos também, com a mesma configuração, o código do próprio Kalibro Metrics, cujos resultados podem ser visualizados na Figura 5.3. Com isso, pretendemos demonstrar como interpretar os resultados das métricas do ponto de vista do desenvolvedor, ou engenheiro de software que conhece a arquitetura do próprio projeto.

/ Main / org / kalibro / tests / **AcceptanceTest**

Metric results for: AcceptanceTest (CLASS)

Search:

Metric	Value	Weight	Threshold
Afferent Connections per Class (used to calculate COF - Coupling Factor)	20.00	2.0	Worrying
Average Cyclomatic Complexity per Method	1.48	2.0	Excellent
Average Method LOC	4.95	1.0	Excellent
Average Number of Parameters per Method	0.52	1.0	Excellent
Depth of Inheritance Tree	5.00	1.0	Regular
Number of Methods	21.00	1.0	Regular
Number of Public Attributes	0.00	1.0	Excellent
Structural Complexity (CBO X LCOM4)	105.00	4.0	Worrying
			Grade: 5.77

Figura 5.4: *Métricas da classe AcceptanceTest*

Ao contrário dos projetos anteriores, escritos em C, o valor de NPA para o Kalibro é muito baixo, como deve ser em um projeto Java, orientado a objetos. As únicas métricas cujo resultado não foi classificado como "excelente" são ACC (acoplamento) e DIT (herança). Navegando pela árvore de resultados, se aprofundando nos nós com menor nota, é possível ver que isso se deve à arquitetura das classes de teste abstratas: lembre que as classes do pacote `org.kalibro.tests`

estão organizadas segundo uma hierarquia e dão apoio à escrita de todos os testes automatizados do Kalibro via herança (ver seção 4.4). Desse modo, não surpreende que as classes de teste abstratas sejam altamente acopladas, e que a profundidade dos testes na árvore de herança seja aumentada.

Navegando na árvore de resultados encontramos facilmente a classe com pior nota segundo nossa configuração: a classe abstrata de teste `AcceptanceTest` (Figura 5.4). Como era de se esperar, seu alto acoplamento influi no resultado das métricas ACC e SC. Chegamos à conclusão que se deve ter cuidado especial ao modificar as classes abstratas de teste.

O mesmo tipo de análise mostra que as classes do núcleo do Kalibro que mais precisam de atenção são suas entidades, que fazem parte do modelo e da API, o que é condizente com nossa experiência no cotidiano do desenvolvimento.

Capítulo 6

Considerações finais

O interesse no Software Livre vem crescendo ao longo das últimas décadas. Grandes empresas como IBM, HP, Google e Nokia consideram Software Livre como parte de suas operações de pesquisa e desenvolvimento. Governos nacionais da Europa, Japão, Brasil, Índia e China consideram o Software Livre uma oportunidade para desenvolvimento de uma indústria de software independente. O projeto QualiPSo da European Comission, conduzido por diversas instituições e grupos de pesquisa pelo mundo, propôs-se a definir e implementar tecnologias, procedimentos e políticas para potencializar as práticas de desenvolvimento de Software Livre, tornando-as confiáveis, reconhecidas e estabelecidas na indústria (Pezuela *et al.*, 2010). Um dos produtos do projeto QualiPSo é um modelo de maturidade para Software Livre – *Opensource Maturity Model* (Petrinja *et al.*, 2009) – inspirado no CMMI (Chrissis *et al.*, 2011). Esse modelo identifica um conjunto de práticas que devem ser aplicadas para alcançar um desenvolvimento de Software Livre confiável.

Para viabilizar o desenvolvimento do Software Livre, os produtos devem ser submetidos a medidas padronizadas para que sejam comparáveis. No Brasil é reconhecida a falta de uso de métricas nas contratações pelo Governo Federal que, no momento, apenas recomenda o uso de Pontos de Função (Albrecht e Jr, 1983) para estimativas e acompanhamento técnico dos contratos de desenvolvimento de software. Porém, a análise de uma única métrica isolada pode levar a interpretações errôneas, como argumentado nos Capítulos 2 e 5.

Esta dissertação apresentou o Kalibro Metrics, um serviço altamente flexível para interpretação de métricas de código-fonte. Interpretar métricas é importante tanto na avaliação de produtos como na melhoria de processos de desenvolvimento de software, como vem sendo destacado por diversos estudos, entre eles:

- Basili *et al.* (1996) analisa métricas do conjunto CK (descritas na Seção 2.1.2), chegando à conclusão de que muitas delas são bons indicadores de qualidade, melhores que as métricas "tradicionais" sozinhas.
- Stamelos *et al.* (2002) relaciona métricas de código-fonte com características de qualidade de software.
- Marinescu (2004) discute estratégias para detecção automática de falhas de projeto baseada em métricas de código-fonte.
- Munro (2005) propõe interpretações automáticas de métricas de código-fonte para detectar "maus cheiros" de código e sugerir refatorações (Fowler, 1999).
- Sato *et al.* (2007) analisa a evolução de oito métricas orientadas a objeto em sete projetos, interpretando diferenças de tamanho, complexidade e acoplamento de acordo com a forma de adoção de métodos ágeis.

A seguir, explicitaremos as contribuições e limitações deste trabalho.

6.1 Contribuições

Nesse mestrado foi desenvolvido o Kalibro Metrics, que fornece um ambiente no qual engenheiros de software e pesquisadores podem definir suas próprias configurações de valores de referência para métricas de código-fonte. O uso de configurações no Kalibro pode ajudar a:

- Melhorar processos de desenvolvimento de software, monitorando a qualidade do código, com avaliações mais fáceis de interpretar.
- Detectar problemas de projeto e identificar as partes mais críticas de um sistema.
- Comparar características específicas de projetos de software.
- Definir novas métricas com base nas implementadas de forma simples.
- Realizar estudos que identifiquem relações entre métricas de código-fonte e outras características de projetos de software.

O grande diferencial do Kalibro é que ele coloca toda a flexibilidade na mão do usuário. Configurações são criadas e modificadas pelos usuários e podem ser reutilizadas e compartilhadas. Configurações de métricas têm o potencial de servir como ponto de apoio para discussões a respeito da forma de interpretar valores de métricas de código-fonte, e podem ser construídas através de contribuição coletiva, seguindo o fenômeno chamado *crowdsourcing* (Howe, 2006). Dessa forma, as melhores práticas no uso de métricas de código-fonte podem ser disseminadas.

Como ponto de partida para discussão sobre valores de referência para métricas, propusemos uma configuração que contempla projetos em C, C++ e Java. Essa configuração pode ser usada e adaptada para diferentes contextos de desenvolvimento de software. Demostramos o uso dessa configuração para interpretar os resultados de métricas, identificando pontos críticos da própria arquitetura do Kalibro e comparando dois projetos bem conhecidos de software livre.

Como contribuições a outros projetos de pesquisa e desenvolvimento, podemos citar:

- Kalibro Metrics foi integrado à QualiPSo Quality Platform através da plataforma de inteligência empresarial (BI) Spago4Q[1]. Antes dessa integração, ocorrida em 2010, a Spago4Q se limitava à análise de código Java. O Kalibro Service, em conjunto com o Analizo, passou a fornecer métricas para C e C++.
- A rede Mezuro se apoia sobre o Kalibro para fornecer um ambiente de análise, acompanhamento, visualização de métricas de código-fonte e compartilhamento de configurações. Acreditamos que o Mezuro seja um ambiente facilitador da discussão e da aplicação prática de valores de referência na interpretação de métricas de código-fonte.
- Atualmente, o Kalibro possui coletores para Analizo, Checkstyle e CVSAnalY, facilitando o uso dessas ferramentas na análise de métricas e fornecendo, para seus usuários, apoio à análise de diversas linguagens de programação. Contribuímos com o desenvolvimento dessas ferramentas, com melhorias e relatos de erros. Devido à flexibilidade do Kalibro, outras ferramentas coletoras podem ser conectadas no futuro.
- O Kalibro Service será utilizado como estudo de caso de escalabilidade de serviços Web no contexto do Mezuro e do projeto CHOReOS[2].

[1] http://www.spago4q.org
[2] http://www.choreos.eu/

6.2 Limitações e Trabalhos Futuros

Uma análise dos pontos de acesso do Kalibro foi conduzida pelo grupo de colaboradores do Mezuro, o que guiou otimizações no serviço, melhorando desempenho, granularidade das operações e organização do banco de dados. No entanto, o desempenho ainda está longe do ideal, e uma análise dinâmica certamente resultariam em novas otimizações. A bateria de testes também carece de otimização, pois seu desempenho vem ficando aquém do desejado devido a seu volume. Diversas funcionalidades desejadas não puderam ser implementadas (ver Seção 4.6).

Devido ao grande número de ferramentas e serviços de métricas, ficou difícil colocar uma descrição detalhada de todas. Gostaríamos de ter incluído no Capítulo 3 análises de outras ferramentas e serviços relacionados, em particular do Sonar[3], do CodeClimate[4] e do Hackystat[5].

Queríamos ter incluído no Capítulo 5 outras comparações, que talvez fossem interessantes:

- Firefox[6] e Chrome[7]. São navegadores extremamente populares, mas tivemos dificuldades com o Firefox: o Doxyparse, principal extrator do Analizo, falhou diversas vezes ao analisar seu código.

- PostgreSQL[8] e Mysql[9]. Não seria uma comparação apropriada pois a maior parte do código do Mysql é C++, orientado a objetos, enquanto que o PostgreSQL é quase totalmente escrito em C.

- Eclipse[10] e NetBeans[11]. É difícil montar uma versão do Eclipse comparável com o NetBeans. O Eclipse tem um núcleo relativamente pequeno e cru, na forma de uma IDE genérica que é praticamente um navegador de arquivos embelezado. A esse núcleo é possível adicionar diversas ferramentas e *plug-ins*. O NetBeans também possui arquitetura de *plug-ins*, mas seu núcleo dispõe de diversas funcionalidades voltadas para os frameworks da Sun (atualmente da Oracle).

Por fim, ainda estamos com uma comunidade de desenvolvedores pequena e concentrada no IME-USP. Mesmo com toda preocupação com a qualidade do código do Kalibro, não conseguimos atrair muitos contribuidores. A descrição detalhada da arquitetura do Kalibro exposta no Capítulo 4, em formato mais técnico, constitui um texto de referência para que os futuros colaboradores sejam integrados mais rapidamente ao projeto. As primeiras versões desse capítulo fez com que os colaboradores do IME-USP melhor entendessem o Kalibro para ajudarem no desenvolvimento dela desde o primeiro semestre de 2013.

Além disso, o Kalibro Metrics e a rede Mezuro foram colocados em ambiente de homologação em outubro de 2012, mas após detectarmos problemas de desempenho, o que nos levou a refatorar parte da estratégia de criação e comunicação com a base da dados do Kaliro, não divulgamos tal ambiente para muitos usuários, de forma que não pudemos avaliar o retorno efetivo dos mesmos. Ao final do primeiro semestre de 2013 chegamos a uma versão estável do Mezuro totalmente integrado ao Kalibro e está sendo planejado um estudo exploratório para o segundo semestre. Isso no contexto do Núcleo de Apoio à Pesquisa em Software Livre (NAPSoL[12]), sediado no CCSL-IME-USP, que apoia o projeto, incluindo financiamento a um grupo de desenvolvedores do Kalibro e do Mezuro.

Portanto, o Kalibro tem sua continuidade garantida, dentro das atividades do Mezuro e do NAPSol, o que corrobora o legado deixado por esta dissertação de mestrado ao grupo do IME-USP.

[3] http://www.sonarqube.org/
[4] https://codeclimate.com/
[5] http://code.google.com/p/hackystat/
[6] http://www.firefox.com/
[7] http://www.chromium.org/
[8] http://www.postgresql.org/
[9] http://mysql.com/
[10] http://www.eclipse.org/
[11] http://www.netbeans.org/
[12] http://www.usp.br/prp/subpagina.php?menu=6&pagina=23&subpagina=67

Apêndice A

Interface de programação do Kalibro

Este anexo apresenta diagramas de classes da interface de programação do Kalibro. Observar as convenções mencionadas na seção 4.1.

A Figura A.1 mostra a parte da API estritamente necessária para implementar um novo coletor. É um conjunto pequeno, o que reflete a facilidade de integração do Kalibro com ferramentas-base.

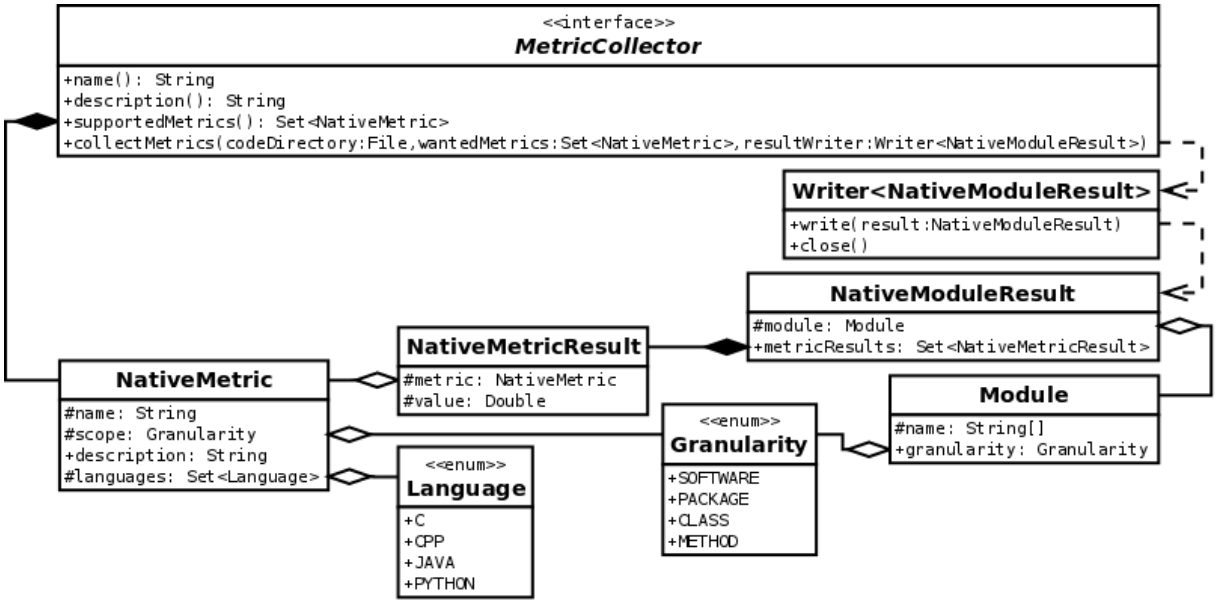


Figura A.1: *Interface de programação para coletores*

As próximas figuras são as entidades apresentadas na seção 4.1.

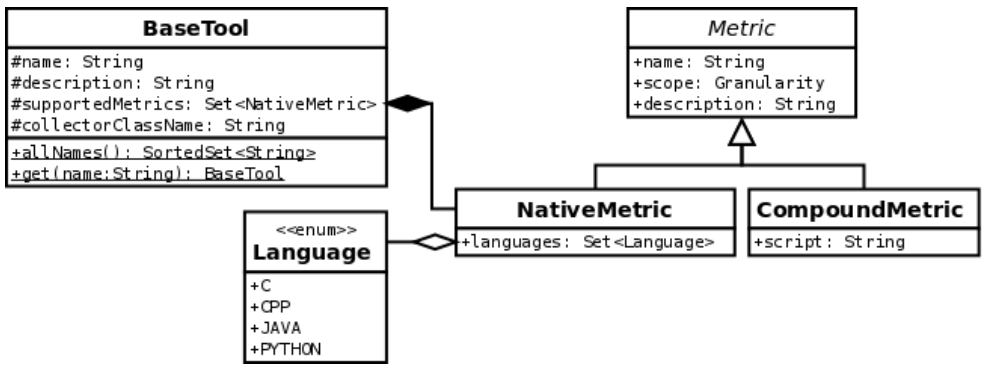


Figura A.2: *Ferramenta-base e métricas*

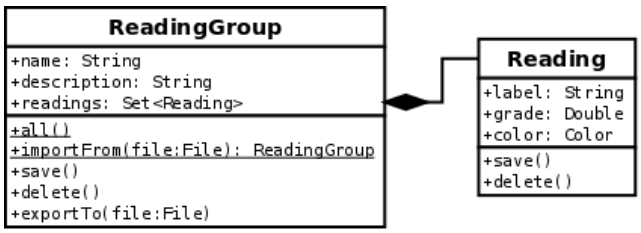


Figura A.3: *Grupo de leitura*

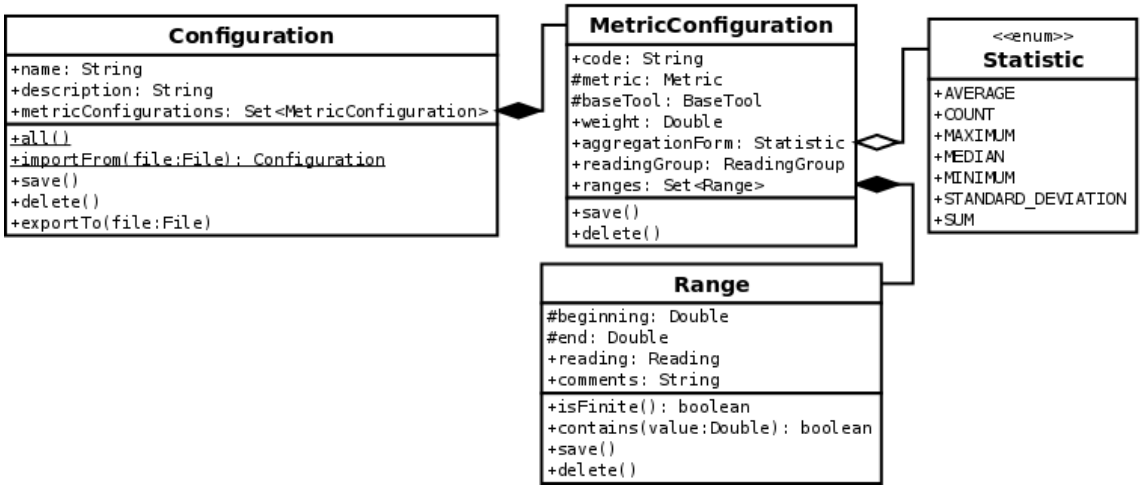


Figura A.4: *Configuração de métricas*

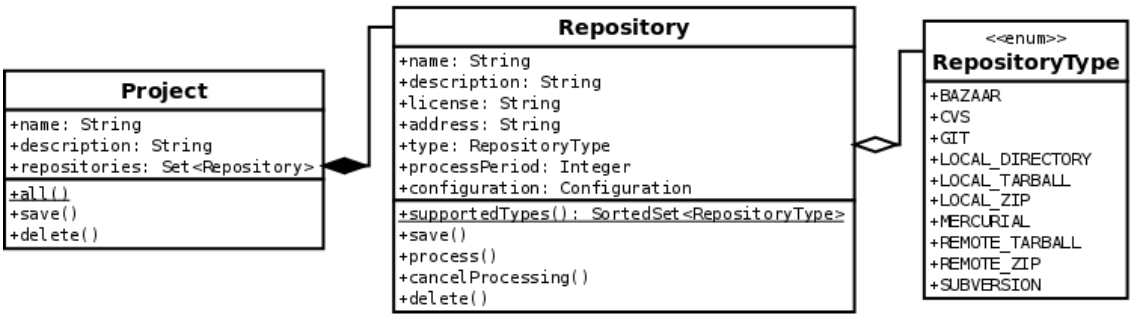


Figura A.5: *Projeto e repositório*

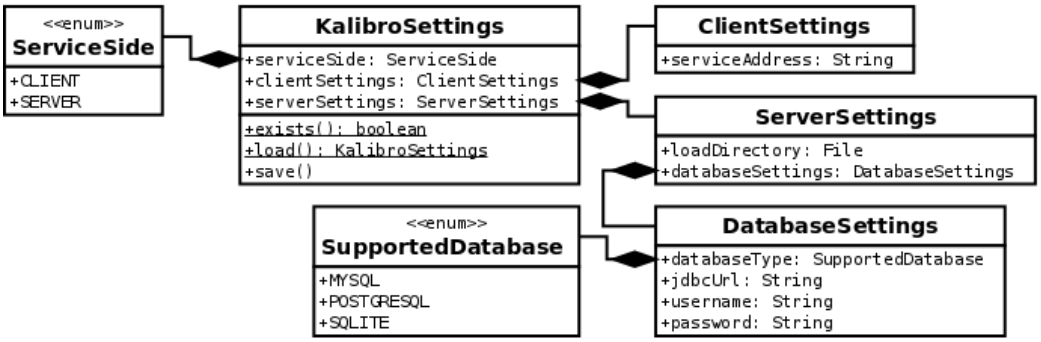


Figura A.6: *Definições*

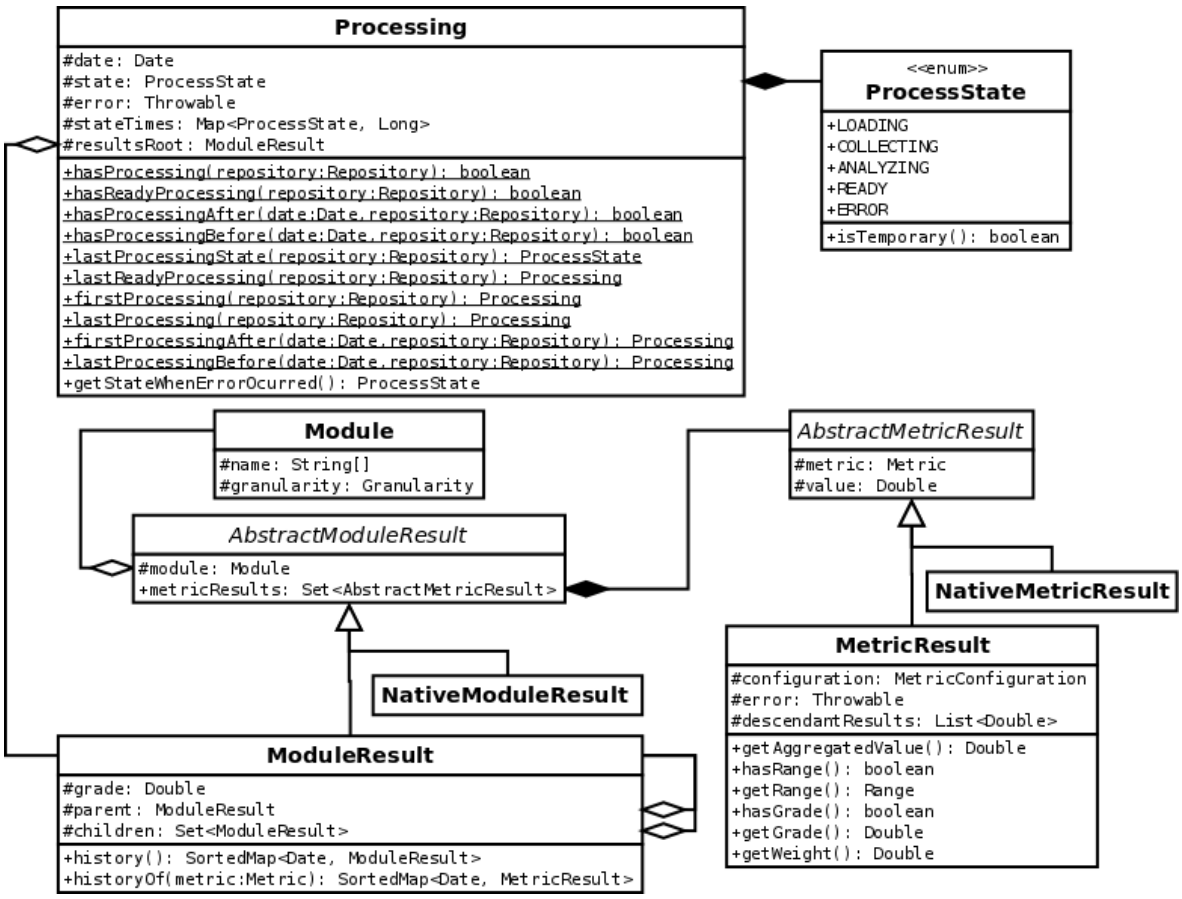


Figura A.7: *Processamento e resultados*

A Listagem A.1 mostra o arquivo `kalibro.settings` com as definições padrão (produzido por `new KalibroSettings().save()`). É um arquivo pequeno, de fácil leitura e edição.

```
1  &id1 !!org.kalibro.KalibroSettings
2 serviceSide: SERVER # CLIENT to connect to Kalibro Service; SERVER if the
      service is running on this machine
3
4 clientSettings: &id2 !!org.kalibro.ClientSettings
5   serviceAddress: "http://localhost:8080/KalibroService/" # Address of the
        remote Kalibro Service
6
7 serverSettings: &id3 !!org.kalibro.ServerSettings
8   loadDirectory: ~/.kalibro/projects # Source code will be loaded in this
        directory before analysis
9   databaseSettings: &id4 !!org.kalibro.DatabaseSettings
10    databaseType: MYSQL # Possibilities: MYSQL, POSTGRESQL, SQLITE
11    jdbcUrl: "jdbc:mysql://localhost:3306/kalibro"
12    username: "kalibro"
13    password: "kalibro"
```

Listagem A.1: *Arquivo de definições padrão*

A Figura A.8 mostra todas as entidades e suas relações. As definições, os tipos enumerados e as operações foram suprimidas para facilitar a visualização do todo.

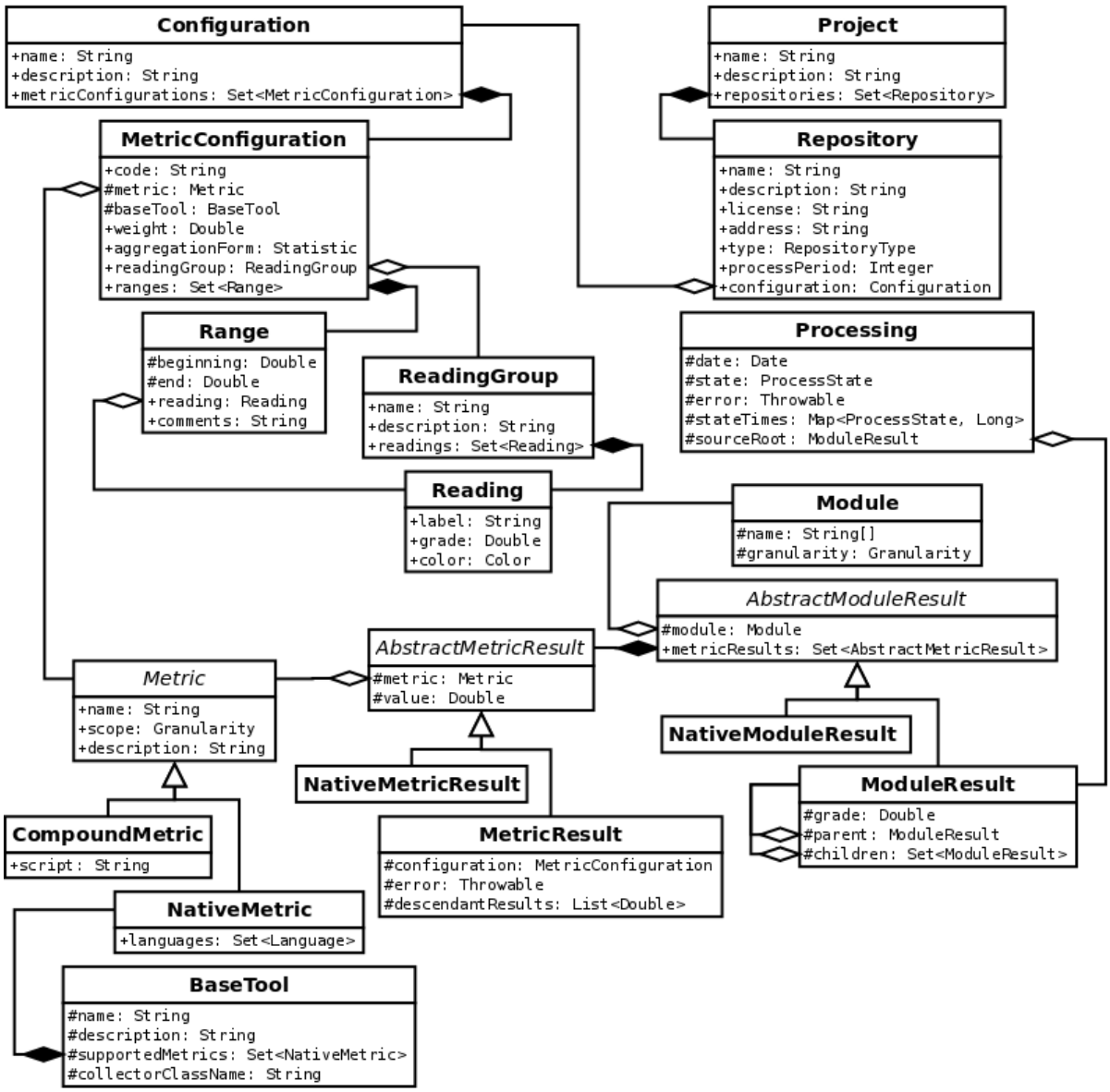


Figura A.8: *Entidades*

Apêndice B

Interface do Kalibro Service

Este anexo apresenta todos os pontos de acesso e objetos de transferência XML como apresentados na seção 4.2, inclusive os que lá foram omitidos. Comparar com as entidades presentes no Anexo A.

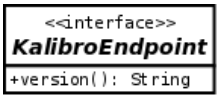


Figura B.1: *Ponto de acesso principal*

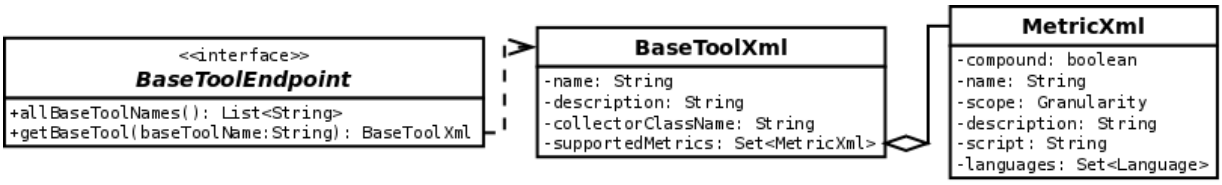


Figura B.2: *Ponto de acesso de ferramentas-base*

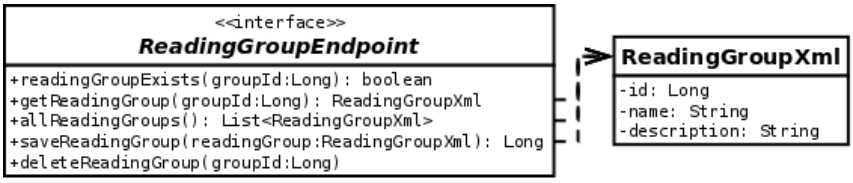


Figura B.3: *Ponto de acesso de grupos de leitura*

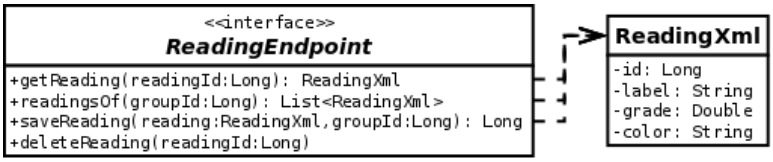


Figura B.4: *Ponto de acesso de leituras*

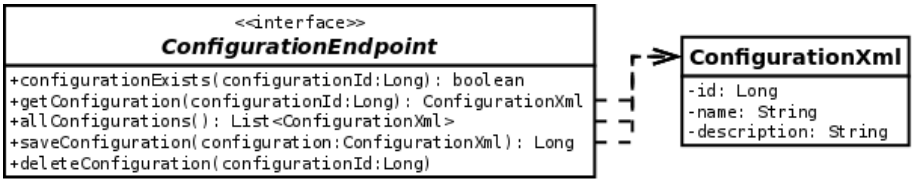


Figura B.5: *Ponto de acesso de configurações*

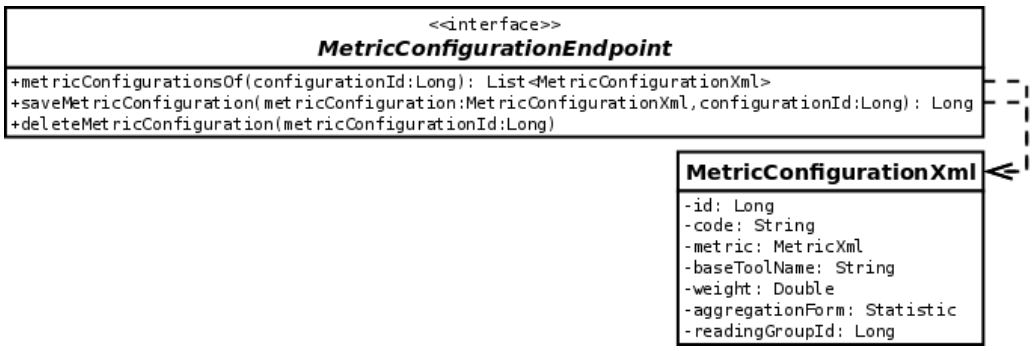


Figura B.6: *Ponto de acesso de configurações de métricas*

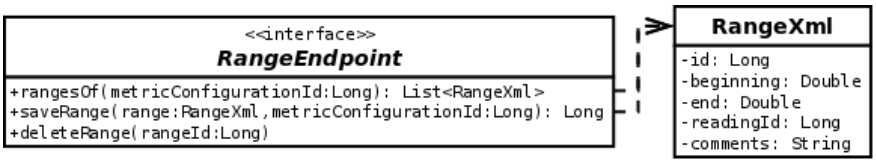


Figura B.7: *Ponto de acesso de intervalos*

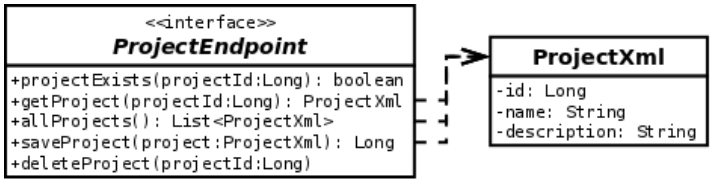


Figura B.8: *Ponto de acesso de projetos*

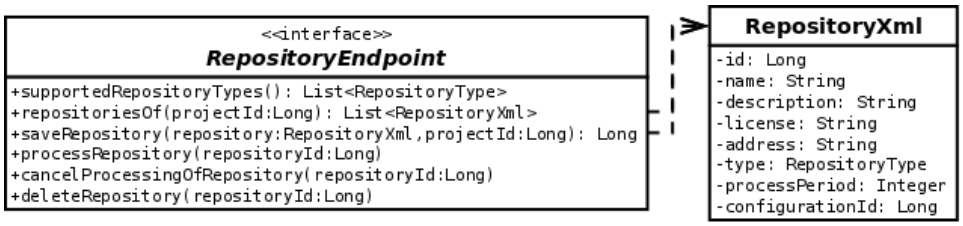


Figura B.9: *Ponto de acesso de repositórios*

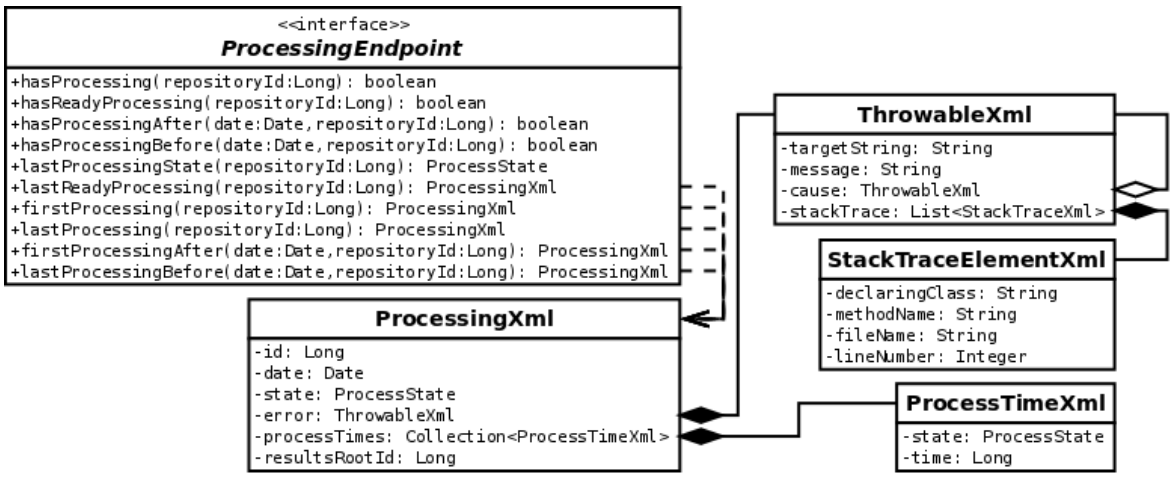


Figura B.10: *Ponto de acesso de processamentos*

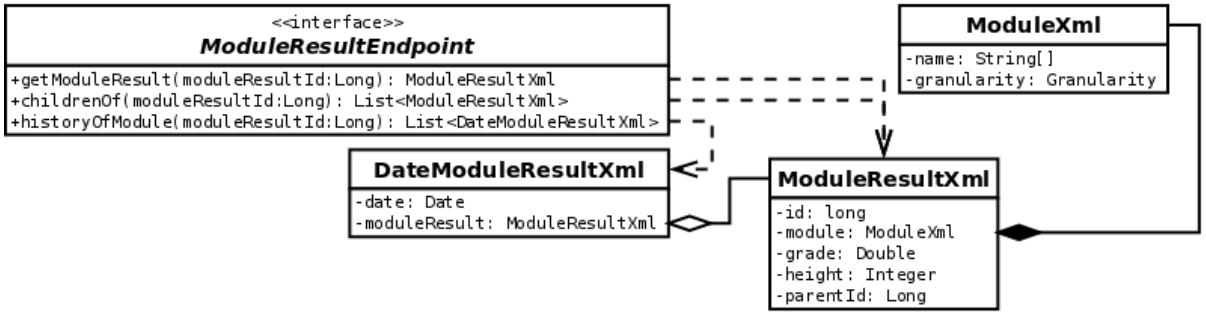


Figura B.11: *Ponto de acesso de resultados por módulo*

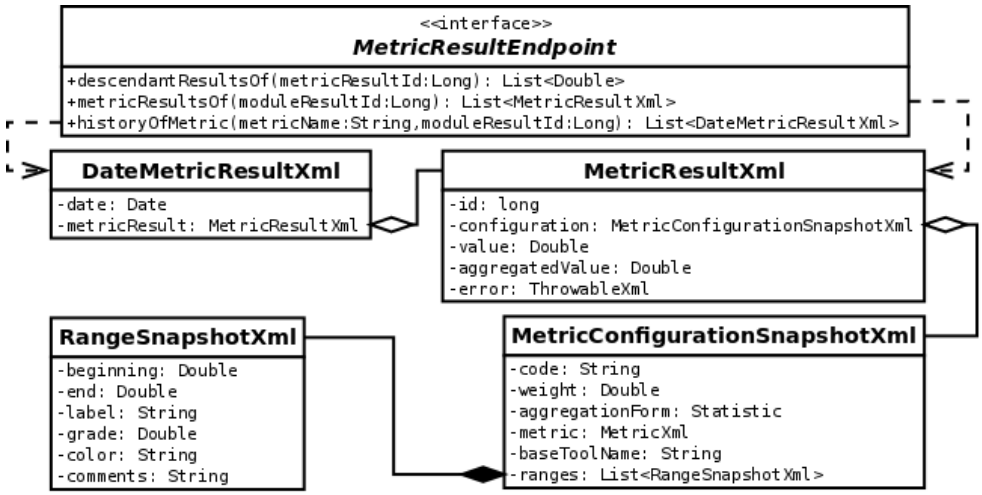


Figura B.12: *Ponto de acesso de resultados por métrica*

Apêndice C

Banco de Dados do Kalibro

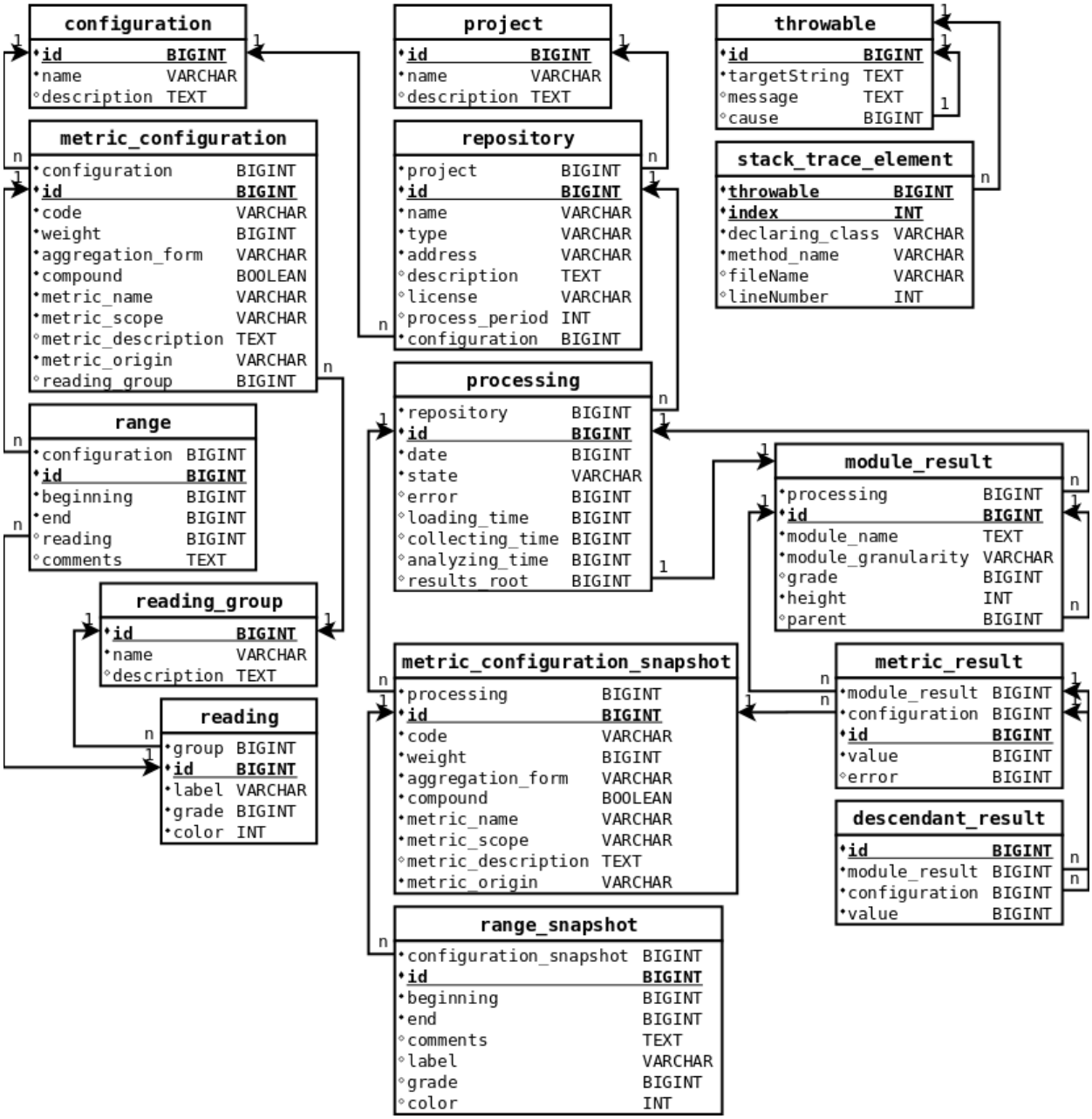


Figura C.1: *Estrutura das tabelas do Kalibro*

A estrutura das tabelas do banco de dados do Kalibro está representada na Figura C.1. Duas flechas não foram desenhadas para diminuir a poluição do diagrama: `processing.error` e `metric_result.error` referenciam `throwable.id`. Ambas são relações um-para-um.

Referências Bibliográficas

Abreu e Carapuça(1994) Fernando Brito Abreu e Rogério Carapuça. Candidate metrics for object-oriented software within a taxonomy framework. *Journal of Systems and Software*, 26(1): 87–96. Citado na pág. 8, 9

Albrecht e Jr(1983) Allan J. Albrecht e John E. Gaffney Jr. Software function, source lines of code, and development effort prediction: a software science validation. *IEEE Transactions on Software Engineering*, (6):639–648. Citado na pág. 8, 55

Alur *et al.***(2003)** Deepak Alur, John Crupi e Dan Malks. *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall. Citado na pág. 26, 29, 30

Baldwin e Clark(2006) Carliss Young 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. 1, 45

Baldwin e Clark(2000) Carliss Young Baldwin e Kim B. Clark. *Design rules: The power of modularity*, volume 1. MIT Press. Citado na pág. 45

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. *ICIS 2008 Proceedings*, página 201. Citado na pág. 48

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*, 22(10):751–761. Citado na pág. 55

Basili *et al.***(1994)** Victor R. Basili, Gianluigi Caldiera e H. Dieter Rombach. The goal question metric approach. *Encyclopedia of software engineering*, 2:528–532. Citado na pág. 5

Beck(2007) Kent Beck. *Implementation Patterns*. Addison Wesley. Citado na pág. 2, 49

Beck(1999) Kent Beck. *Extreme Programming Explained*. Addison Wesley. Citado na pág. 38

Bloch(2001) Joshua Bloch. *Effective Java: Programming Language Guide*. Addison-Wesley. Citado na pág. 33

Boehm(1984) Barry W. Boehm. Software engineering economics. *IEEE Transactions on Software Engineering*, (1):4–21. Citado na pág. 7

Boehm *et al.***(2000)** Barry W. Boehm, Clark, Horowitz, Brown, Reifer, Chulani, Ray Madachy e Bert Steece. *Software Cost Estimation with Cocomo II with Cdrom*. Prentice Hall PTR. Citado na pág. 7

Capra *et al.***(2008)** Eugenio Capra, Chiara Francalanci e Francesco 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. Citado na pág. 48

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. 8, 9

Chrissis *et al.***(2011)** Mary Beth Chrissis, Mike Konrad e Sandra Shrum. *CMMI for Development: Guidelines for Process Integration and Product Improvement*. Addison-Wesley Professional. Citado na pág. 7, 55

Conte *et al.***(1986)** Samuel Daniel Conte, Hubert E. Dunsmore e Vincent Y. Shen. *Software engineering metrics and models*. Benjamin-Cummings Publishing Co., Inc. Citado na pág. 6

Costa(2009) Joenio Costa. Extração de informações de dependência entre módulos de programas c/c++. Undergraduation course conclusion project, Universidade Católica do Salvador. Citado na pág. 11

Cunningham(1992) Ward Cunningham. The wycash portfolio management system. Em *Conference on Object Oriented Programming Systems Languages and Applications: Addendum to the proceedings on Object-oriented programming systems, languages, and applications*, volume 19, páginas 29–30. Citado na pág. 6

DeMarco(2009) Tom DeMarco. Software engineering: An idea whose time has come and gone? *IEEE Software*, 26(4):95–96. Citado na pág. 5

DeMarco(1982) Tom DeMarco. Controlling software projects. management, measurement and estimation. *ISBN*, 10(0131717111):0–13. Citado na pág. 5

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. 2, 13

Fenton e Pfleeger(1998) Norman E. Fenton e Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., Boston, MA, USA. ISBN 0534954251. Citado na pág. 1

Fowler(2003) Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley. ISBN 0-321-12742-0. Citado na pág. 26

Fowler(1999) Martin Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional. Citado na pág. 55

Freeman e Pryce(2009) Steve Freeman e Nat Pryce. *Growing Object-Oriented Software, Guided by Tests*. Addison-Wesley. Citado na pág. 38

Gamma *et al.***(1995)** Erich Gamma, Richard Helm, Ralph Johnson e John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley. ISBN 0-201-63361-2. Citado na pág. 12, 28, 29, 30, 31, 34, 36

Halstead(1977) Maurice H. Halstead. *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc. Citado na pág. 8

Harman(2010) Mark Harman. Why source code analysis and manipulation will always be important. Em *10th IEEE Working Conference on Source Code Analysis and Manipulation (SCAM)*, páginas 7–19. Citado na pág. 6

Henry e Kafura(1981) Sallie Henry e Dennis Kafura. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, (5):510–518. Citado na pág. 8

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. 9

Howe(2006) Jeff Howe. The rise of crowdsourcing. *Wired magazine*, 14(6):1–4. Citado na pág. 56

ISO/IEC19761(2011) ISO/IEC19761. Software engineering – cosmic: a functional size measurement method. *International Organization for Standardization and International Electrotechnical Commission*. Citado na pág. 8

ISO/IEC20926(2009) ISO/IEC20926. Software and systems engineering – software measurement – ifpug functional size measurement method 2009. *International Organization for Standardization and International Electrotechnical Commission*. Citado na pág. 8

ISO/IEC20968(2002) ISO/IEC20968. Software engineering – mk ii function point analysis – counting practices manual. *International Organization for Standardization and International Electrotechnical Commission*. Citado na pág. 8

ISO/IEC24570(2005) ISO/IEC24570. Software engineering – nesma functional size measurement method version 2.1 – definitions and counting guidelines for the application of function point analysis. *International Organization for Standardization and International Electrotechnical Commission*. Citado na pág. 8

ISO/IEC29881(2008) ISO/IEC29881. Information technology – systems and software engineering – fisma 1.1 functional size measurement method. *International Organization for Standardization and International Electrotechnical Commission*. Citado na pág. 8

ISO/IEC9126-1(2001) ISO/IEC9126-1. Software engineering – product quality – part 1: Quality model. *International Organization for Standardization and International Electrotechnical Commission*. Citado na pág. 5

Jones(1985) Capers Jones. *Programming productivity*. McGraw-Hill, Inc. Citado na pág. 7

Kafura e Canning(1985) Dennis Kafura e James Canning. A validation of software metrics using many metrics and two resources. Em *Proceedings of the 8th International Conference on Software Engineering*, páginas 378–385. IEEE Computer Society Press. Citado na pág. 8

Kemerer(1987) Chris F. Kemerer. An empirical validation of software cost estimation models. *Communications of the ACM*, 30(5):416–429. Citado na pág. 8

Lehman *et al.***(1997)** Manny Meir Lehman, Juan F. Ramil, Paul D. Wernick, Dewayne E. Perry e Wladyslaw M. Turski. Metrics and laws of software evolution - the nineties view. Em *Fourth International Software Metrics Symposium*, páginas 20–32. IEEE. Citado na pág. 7

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. 8

Marchesi *et al.***(2003)** Michele Marchesi, Giancarlo Succi, Don Wells e Laurie Williams. *Extreme Programming Perspectives*. Addison-Wesley. Citado na pág. 1

Marinescu(2004) Radu Marinescu. Detection strategies: Metrics-based rules for detecting design flaws. Em *20th IEEE International Conference on Software Maintenance*, páginas 350–359. Citado na pág. 55

Martin(2002) Robert C. Martin. *Agile Software Development - Principles, Patterns, and Practices*. Prentice Hall. Citado na pág. 9, 29, 34

Martin(2008) Robert C. Martin. *Clean Code - A Handbook of Agile Software Craftsmanship*. Prentice Hall. Citado na pág. 2, 29, 38, 49

McCabe(1976) Thomas J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, (4):308–320. Citado na pág. 8

McConnell(2004) Steve McConnell. *Code Complete*. O'Reilly Media, Inc. Citado na pág. 49

Meirelles *et al.***(2010)** Paulo Meirelles, Antônio Terceiro, Carlos Santos, Christina Chavez, Fabio Kon e João Miranda. A study of the relationships between source code metrics and attractiveness in free software projects. *Brazilian Symposium on Software Engineering*, 0:11–20. Citado na pág. 1, 13, 48, 49

Meirelles *et al.***(2009)** Paulo R. M. Meirelles, Raphael Cóbe, 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 de qualidade de código. Em *XXIII Simpósio Brasileiro de Engenharia De Software*. Citado na pág. 15

Meirelles(2013) Paulo Roberto Miranda Meirelles. *Monitoramento de métricas de código-fonte em projetos de software livre*. Tese de Doutorado, Universidade de São Paulo. Citado na pág. 13, 43, 47, 48

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. *ICIS 2008 Proceedings*, página 37. Citado na pág. 47

Mills(1988) Everald E. Mills. Software metrics. Relatório técnico, DTIC Document. Citado na pág. 6, 8

Munro(2005) Matthew James Munro. Product metrics for automatic identification of "bad smell" design problems in java source code. Em *Software Metrics, 2005. 11th IEEE International Symposium*, páginas 15–15. IEEE. Citado na pág. 55

Perlis *et al.***(1981)** Alan J. Perlis, Frederick Sayward e Mary Shaw. *Software Metrics: An Analysis and Evaluation*. MIT Press. Citado na pág. 2

Petrinja *et al.***(2009)** Etiel Petrinja, Ranga Nambakam e Alberto Sillitti. Introducing the open-source maturity model. Em *Proceedings of the 2009 ICSE Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development*, páginas 37–41. IEEE Computer Society. Citado na pág. 55

Pezuela *et al.***(2010)** Clara Pezuela, Matteo Melideo, Andrea Rossi, Jean-Christophe Spilmont, Julie Marguerite, Ellen Francine Barbosa, José Carlos Maldonado, Matthias Fluegge, Sandro Morasca, Jaime Garza, Alberto Silliti, Jean-Pierre Laisne e Giuseppe Lauria. Qualipso position paper: Quality platform for open source software, 2010. Citado na pág. 15, 55

Raymond(1999) Eric Steven Raymond. *The Cathedral & the Bazaar*. O'Reilly & Associates, Inc., Sebastopol, CA, USA. Citado na pág. 1

Rombach(1987) H. Dieter Rombach. A controlled expeniment on the impact of software structure on maintainability. *IEEE Transactions on Software Engineering*, (3):344–354. Citado na pág. 8

Sato *et al.***(2007)** Danilo Sato, Alfredo Goldman e Fabio Kon. Tracking the evolution of object-oriented quality metrics on agile projects. Em *Proceedings of the 8th International Conference on Agile Processes in Software Engineering and Extreme Programming*, páginas 84–92. Springer. Citado na pág. 55

Sobel e Friedman(1996) Jonathan M. Sobel e Daniel P. Friedman. An introduction to reflection-oriented programming. Em *Reflection*, volume 96, páginas 263–288. Citado na pág. 32

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(1):43–60. Citado na pág. 55

Tempero(2008) Ewan Tempero. On measuring java software. Em *ACSC '08: Proceedings of the Thirty-First Australasian Conference on Computer Science*, volume 74, páginas 7–7, Darlinghurst, Australia, Australia. Australian Computer Society, Inc. Citado na pág. 2

Terceiro *et al.***(2010)** 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. 11, 15, 40

Valerdi(2005) Ricardo Valerdi. *The constructive systems engineering cost model (COSYSMO)*. Tese de Doutorado, University of Southern California. Citado na pág. 7

VanDoren(2002) Edmond VanDoren. Maintainability index technique for measuring program maintainability. *SEI STR report*. Citado na pág. 13

Vermeulen *et al.***(2000)** Allan Vermeulen, Scott W. Ambler, Greg Bumgardner, Eldon Metz, Trevor Misfeldt, Jim Shur e Patrick Thompson. *The Elements of Java^{TM} Style*. Cambridge University Press. Citado na pág. 12

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 Simpósio Brasileiro de Engenharia De Software*. Citado na pág. 15

von Krogh e Spaeth(2007) Georg von Krogh e Sebastian Spaeth. The open source software phenomenon: Characteristics that promote research. *The Journal of Strategic Information Systems*, 16(3):236–253. Citado na pág. 2

Wolverton(1974) R. W. Wolverton. The cost of developing large-scale software. *IEEE Transactions Computers*, C-23(6):615–636. Citado na pág. 2

Xenos *et al.***(2000)** Michalis Xenos, D. Stavrinoudis, K. Zikouli e D. Christodoulakis. Object-oriented metrics - a survey. Em *Proceedings of the FESMA*, páginas 1–10. Citado na pág. 8