Carlos Morais de Oliveira Filho

Kalibro: Uma ferramenta de configuração e interpretação de métricas de código-fonte

Orientador: Prof. Dr. Fabio Kon
Coorientador: Paulo R. M. Meirelles

São Paulo, Novembro de 2009

Sumário

I Parte técnica **1**

1 Introdução **2**

2 Métricas de software **4**
2.1 Características 4
2.1.1 Classificação 5
2.1.2 Escalas de medidas 5
2.2 Métricas de código-fonte 6
2.2.1 Valores de referência 7

3 Kalibro: configuração e interpretação de métricas **11**
3.1 Arquitetura da Kalibro 12
3.2 Funcionalidades da Kalibro 14
3.3 Integração da Crab com a Jabuti 17
3.4 Integração com a Analizo 17

4 Considerações finais **20**

II Parte subjetiva **21**

5 Experiência pessoal **22**
5.1 Desafios e frustrações 22
5.2 Disciplinas relevantes 22
5.3 Estudos futuros 23

A Código de integração com a Analizo **28**

B Exemplo de saída da Analizo **29**

C Exemplo de projeto salvo pela Kalibro **30**

D Código de integração da Crab com a Jabuti **32**

Resumo

Métricas de código-fonte foram propostas desde a década de 70, mas não têm sido extensivamente usadas e disseminadas entre os desenvolvedores. Umas das causas disso pode ser a curva de aprendizado e o custo de ter um especialista em métricas na equipe. Outro problema é a falta de valores de referência para os valores das métricas obtidos na análise de um sistema. Com a difusão do software livre e dos métodos ágeis, o código passa a ser um dos principais artefatos produzidos e, portanto, deve ser alvo de avaliação. Nesse contexto, este trabalho apresenta a Kalibro, uma ferramenta que potencializa o uso de programas de cálculo de métricas de código-fonte. A Kalibro permite a configuração de um conjunto de intervalos para os valores das métricas. Com uma configuração definida, os resultados são apresentados de uma forma mais intuitiva e de fácil entendimento para a avaliação da qualidade do código.

Parte I

Parte técnica

Capítulo 1

Introdução

Para controlar o processo de desenvolvimento de qualquer produto é necessário medir sua qualidade e produtividade. Com esse objetivo, a indústria de software há décadas busca mecanismos para medir e controlar o processo de desenvolvimento de software pois, segundo alguns autores, a qualidade do processo garante a qualidade do produto (Rocha, Maldonado e Weber, 2001). Entretanto, a denominada "crise do software" (Arthur, 1985) persiste até os dias atuais. Nos últimos anos, de acordo com o Chaos Report (Extreme Chaos, 2004, 2006, 2009), 63% dos projetos atrasam, 45% custam mais do que o estimado e, ao término, apenas 67% das funcionalidades são implementadas.

Em contraponto às premissas da chamada engenharia de software tradicional, desde o início deste século, são cada vez mais disseminadas as práticas e valores dos métodos ágeis na indústria de software. Essas práticas e valores procuram: desburocratizar o processo de desenvolvimento; concentrar a maior parte dos esforços na qualidade do código desenvolvido; reconhecer a codificação como um trabalho intelectual e criativo e, estimular a construção de um ambiente de trabalho comunicativo.

Antes do sucesso dos métodos ágeis, já existia outra maneira de desenvolvimento de software que é diferente da engenharia de software tradicional (Scacchi, 2007), o software livre. Desde a década de 90, com a popularização do sistema operaciopnal GNU/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 de software (Raymond, 1999). O que determina se uma funcionalidade ou melhoria será ou não incorporada ao projeto original é a qualidade de seu código-fonte. Da mesma forma, o desenvolvedor é reconhecido dentro de uma comunidade de software livre pelo código que produz.

Alguns dos possíveis impactos dos métodos ágeis e do software livre na indústria de software podem ser observados quando analisamos os dados do Chaos Report de 1994 que apresenta os seguintes dados: 31% dos projetos não eram concluídos; apenas 16% dos projetos eram bem sucedidos; o estouro médio de custo era de 180% e, o estouro médio de prazo era 164%. Comparando com os mesmos dados de 2004, caiu para 18% os projetos não concluídos, aumentou para 29% os projetos bem sucedidos, o estouro médio de custo caiu para 56% e o estouro médio de prazo caiu para 84%. Já o relatório publicado em 2009 apresenta que os projetos cancelados aumentaram para 24%, porém passou para 32% os completados com sucesso.

Apesar da melhoria do percentual de projetos bem sucedidos nos últimos 15 anos, podemos considerar que esse dado ainda não está satisfatório. Dessa forma, outros mecanismos de observação da qualidade do software estão sendo pesquisados, como as métricas

de código-fonte (Meirelles e Kon, 2009). Principalmente no contexto dos métodos ágeis e do software livre, medir a qualidade do código-fonte passou a ser tão importante quanto medir a qualidade do processo. Em muitos casos, o que está disponível é o código-fonte, que passa a ser o principal artefato para se medir a qualidade do software. Porém, ainda não foi disseminada uma cultura de análise sistemática da qualidade do código produzido.

Mesmo com métricas de código-fonte sendo propostas em alguns trabalhos desde a década de 70 (Wolverton, 1974; Perlis, Sayward e Shaw, 1981), seu potencial de uso no desenvolvimento de software não tem sido explorado. Segundo Ewan Tempero (Tempero, 2008), um dos motivos dessa subutilização é que a maior parte das métricas não possuem valores de referência conhecidos.

Nesse cenário, identificamos dois problemas: coletar automaticamente os valores das métricas, independente da linguagem utilizada, e interpretar os resultados das métricas, associando-os à qualidadade do código-fonte. A principal motivação deste trabalho foi a constatação de que as ferramentas de avaliação de código-fonte ainda são deficientes da perspectiva de um desenvolvedor não especialista em métricas de software, pois geralmente apresentam resultados na forma de valores isolados para cada métrica. A interpretação desses valores fora de contexto pode não retratar características consideradas importantes no acompanhamento da qualidade do código (Mills, 1988), o que dificulta a tomada de decisões a respeito da qualidade do software (Meirelles et al., 2009).

Dessa forma, este trabalho apresenta a Kalibro, uma ferramenta de configuração e interpretação de métricas. Ela foi projetada para ser incorporada a ferramentas que calculam valores de métricas. Os resultados são visualizados de forma mais intuitiva, pois permite que um especialista crie uma configuração de valores de referência e avaliações qualitativas que será usada para auxiliar a interpretação da qualidade do código por parte dos desenvolvedores.

Para a elaboração desse trabalho foi feito um estudo do estado da arte de métricas de software e valores de referência para métricas de código-fonte, apresentado no capítulo 2. O capítulo 3 descreve em detalhes o objetivo da Kalibro, suas funcionalidades e estrutura. Um caso de integração é apresentado. Por fim, no capítulo 4 serão feitas as considerações finais e apresentados os trabalhos futuros.

Capítulo 2

Métricas de software

Métricas de software têm como objetivo identificar, medir e consequentemente controlar os principais parâmetros que afetam o desenvolvimento de software (Mills, 1988). A necessidade do uso de métricas de software teve maior atenção quando se constatou a chamada "crise do software" (Arthur, 1985), devido à ineficácia da gerência do desenvolvimento da maior parte das soluções de software (Mills, 1988).

Desenvolver software é um processo complexo (Brooks, 1975) e não se tinha medidas bem definidas e viáveis para sua avaliação. Porém, estimativas precisas e eficazes, planejamento e controle são aspectos difíceis de se concretizar em conjunto (Mills, 1988). Dessa forma, as métricas de software foram propostas como um poderoso recurso para melhoria do processo de gestão.

Uma métrica, segundo definição da ISO 9126 (ISO/IEC9126-1, 2001), é a composição de procedimentos para medição e escalas de medidas. O SEI (*Software Engineering Institute*) definiu o termo métricas de software como um conjunto de medidas de um processo ou produto de software, onde um produto de software pode ser visto como um objeto abstrato que evolui de uma instrução inicial para o sistema finalizado, incluindo o código-fonte e variadas formas de documentação produzidas durante o desenvolvimento (Mills, 1988). Portanto, métricas são utilizadas para estimar um cronograma e custos de desenvolvimento do software e para medir a produtividade e a qualidade do produto.

2.1 Características

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

Uma das abordagens mais conhecidas é a GQM (Goal Question Metric — objetivo-pergunta-métrica) proposta por Basili (Basili, Caldiera e Rombach, 1994). Podemos citar ainda a abordagem Lean, proposta por Mary Poppendieck (Poppendieck e Poppendieck, 2003), que adapta os conceitos Lean usados na manufatura, cadeia de suprimentos e desenvolvimento de produtos. De acordo com Everald Mills (Mills, 1988) e Norman Fenton (Fenton e Pfleeger, 1998), o ideal é que as métricas possuam as seguintes características:

- Simplicidade: o que a métrica se propõe a medir deve ser claro.

- Objetividade: a métrica deve ser formalmente definida, seu valor deve estar atrelado ao objeto medido, independente de quem o obtenha.
- Fácil obtenção: deve ser possível obter seu valor rapidamente e a baixo custo.
- Validade: a métrica deve medir efetivamente o proposto.
- Robustez: pequenas mudanças no software não podem causar grandes mudanças no valor obtido.
- Linearidade de escala: há formas de mapeamento para o entendimento do comportamento das entidades através da manipulação dos números obtidos.
- Independência da linguagem utilizada.

2.1.1 Classificação

As métricas de software podem ser classificadas quanto ao âmbito da sua aplicação, quanto ao critério utilizado na sua definição e quanto ao método de obtenção da medida.

A maneira mais ampla de classificá-las é quanto ao objeto da métrica. *Métricas de produtos* medem a complexidade e tamanho final do programa. *Métricas de processo* referem-se ao processo de concepção e desenvolvimento do software, medindo, por exemplo, o processo de desenvolvimento, tipo de metodologia usada e tempo de desenvolvimento (Mills, 1988).

As métricas também são classificadas quanto ao método de obtenção, dividindo-se em *primitivas* e *compostas* (Grady e Caswell, 1987). As métricas primitivas são aquelas que podem ser diretamente observadas em uma única medida, como o número de linhas de código, erros indicados em um teste de unidade ou ainda o total de tempo de desenvolvimento de um projeto. As métricas compostas são as combinações de uma ou mais medidas como o número de erros encontrados a cada mil linhas de código ou ainda o número de linhas de teste por linha de código.

Outra maneira de classificar as métricas de software é quanto aos critérios utilizados para determiná-las. Desse modo, as métricas são diferenciadas em *métricas objetivas* e *métricas subjetivas*. As objetivas são obtidas através de regras bem definidas, sendo a melhor forma de possibilitar comparações posteriores consistentes. Assim, os valores obtidos por elas deveriam ser sempre os mesmos, independentemente do instante, condições ou indivíduo que os determinam. A obtenção dessas métricas é passível de automatização, como por exemplo número de linhas de código (LOC) (Conte, Dunsmore e Shen, 1986). As subjetivas podem partir de valores, mas dependem de um julgamento, que também é um dado de entrada, para serem levantadas.

2.1.2 Escalas de medidas

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

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

Neste trabalho estamos interessados em métricas objetivas, que possam ser calculadas a partir do código-fonte. Essas métricas podem ser primitivas ou compostas e interpretaremos o significado dos seus dados por intervalos de valores de referência.

2.2 Métricas de código-fonte

Parte das métricas objetivas tratam características do código-fonte. Uma série de trabalhos deu início às abordagens como tamanho do programa e complexidade do software (Troy e Zweben, 1981; Henry e Kafura, 1984; Yau e Collofello, 1985). Existe uma certa quantidade de métricas desses tipos. Por exemplo, Hon Li (Li e Cheung, 1987) referencia e compara 31 diferentes métricas de complexidade. Os trabalhos de Card e Harrison (Card e Agresti, 1988; Harrison e Cook, 1987) também propuseram novas métricas de complexidade. Os tipos de métricas objetivas mais referenciados e conhecidas do estado da arte de métricas de software serão brevemente apresentados a seguir.

- **Métricas de tamanho:** Algumas métricas de software foram desenvolvidas para tentar quantificar o tamanho e auxiliar as medições na fase de concepção do software. Em sua origem, seguem o paradigma de desenvolvimento de software tradicional. Entre métricas de tamanho podemos citar: Linhas de Código (LOC) (Boehm, 1981; Jones, 1986, 1991), Pontos de Função (FP) (Albrecht e Gaffney, 1983), System Bag (DeMarco, 1982).
- **Métricas de Halstead:** É um conjunto de métricas baseadas na teoria da informação, consideradas as primeiras métricas com fundamentação teórica comum (Halstead, 1972, 1977). Essas métricas se aplicam a vários aspectos do software e também são usadas para avaliar o esforço global de desenvolvimento. O Vocabulário (n), comprimento (N) e volume (V) são métricas que aplicam-se especificamente ao software final. Também foram especificadas fórmulas para calcular o esforço total (E) e o tempo de desenvolvimento (T) de software.
- **Métricas de complexidade:** Métricas de complexidade são uma forma objetiva de medir a complexidade de um pedaço de software. Alta complexidade deve ser evitada pois prejudica a compreensão do código e o torna mais suscetível a erros. Softwares grandes são mais suscetíveis a ter alta complexidade, portanto recomenda-se usá-las juntamente com as métricas de tamanho. Dentre as principais métricas

de complexidade, temos: Complexidade Ciclomática ($v(G)$) (McCabe, 1976; Myers, 1977; McCabe, 1982; Stetter, 1984; McCabe e Butler, 1989), Número de Nós (Woodward, Hennell e Hedley, 1979) e Fluxo de Informação (Kafura e Henry, 1981).

- **Métricas de Manutenibilidade:** O SEI recomenda o uso de um índice de manutenibilidade (Maintainability Index — MI) (VanDoren, 1997). A manutenibilidade de um sistema é calculada usando uma combinação de medidas largamente usadas. O índice é um polinômio da seguinte forma:

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

  Os coeficientes são resultado de calibragem usando vários sistemas mantidos pela HP (Hewlett-Packard). Os termos são os seguintes: $aveV$ é a média do volume V de Halstead; $aveV(G)$ é a média da complexidade ciclomática por módulo, $aveLOC$ é a média de linhas de código por módulo; $perCM$ é a média da porcentagem de linhas de comentário por módulo. Esse último termo é opcional.

- **Métricas de Orientação a Objetos:** Como o paradigma da orientação a objetos usa entidades e não algoritmos como componentes fundamentais, a abordagem das métricas de código-fonte para programas orientados a objetos deve ser diferente. Além de tamanho e complexidade, em um software orientado a objetos é possível medir o uso da herança e o grau de interdependência entre as entidades. Desse conjunto de métricas, destacam-se as dos grupos CK (Chidamber e Kemerer, 1994) e MOOD (Abreu e Carapuca, 1994). Uma análise mais detalhada dessas métricas é apresentada em (Xenos et al., 2000). As métricas de orientação a objetos mais encontradas nas ferramentas de avaliação de código são: Respostas para uma Classe (RFC) (Sharble e Cohen, 1993), Falta de Coesão entre Métodos (LCOM), Acoplamento entre Objetos (CBO) (Chidamber e Kemerer, 1994), Número de Parâmetros por Método (NPM), Número de Atributos Públicos (NPA) (Bansiya e Davi, 1997), Profundidade da Árvore de Herança (DIT) (Shih et al., 1997) e Número de Filhos (NOC) (Rosenberg e Hyatt, 1997).

2.2.1 Valores de referência

Neste trabalho, selecionamos algumas, entre as inúmeras métricas propostas pela literatura, tendo em vista medir os aspectos mais relevantes à manutenibilidade do software, como sua complexidade interna, modularidade e grau de dependência entre os módulos. Outro critério utilizado na escolha das métricas que detalharemos nesta seção foi a existência de trabalhos que para elas sugerem valores de referência.

Um estudo recente (Ferreira, Bigonha e Bigonha, 2008) analisou diferentes métricas para 40 softwares desenvolvidos em Java e disponíveis em *www.sourceforge.net*. Esses softwares pertenciam a 11 diferentes domínios de aplicação, entre eles: clustering, bancos de dados, frameworks, jogos, multimídia, rede e segurança. A ferramenta EasyFit versão 5.0 foi utilizada para ajustar os dados de acordo com diferentes distribuições de probabilidade. O trabalho sugere então intervalos qualitativos para as métricas analisadas.

Lanza e Marinescu (Lanza e Marinescu, 2006) coletaram diversas métricas de tamanho, complexidade e herança de 45 sistemas feitos em Java e 37 em C++. Seus valores são apresentados no livro de sua autoria. Esse livro sugere valores considerados baixo, médio, alto e muito alto para cada métrica, obtidos através da média e desvio padrão.

Os intervalos de avaliação sugeridos nessa seção foram inspirados nos dois trabalhos citados. Esse intervalos são distribuídos como configuração padrão da Kalibro, ferramenta apresentada no próximo capítulo.

```
1  public class HelloWorld {
2          // This comment will not count
3          public static void main(String args[]){
4                  Printer printer = new HelloWorldPrinter();printer.print();
5          }
6  }
7
8  class Printer {
9          private String message;
10
11         public Printer(String msg){
12                 message = msg;
13         }
14
15         public void print(){
16                 System.out.println(message);
17         }
18 }
19
20 class HelloWorldPrinter extends Printer {
21         public HelloWorldPrinter(){
22                 super("Hello World!");
23         }
24
25         public void doNothing(){
26                 // a method
27         }
28 }
```

Listagem 2.1: Exemplo de código para ilustrar o cálculo das métricas

Para ilustrar como são calculadas as métricas, utilizaremos como exemplo o trecho de código apresentado na listagem 2.1.

- **LOC** (Lines of Code — Linhas de código) é a medida mais comum para o tamanho de um software. São contadas apenas as linhas executáveis, ou seja, são excluídas linhas em branco e comentários. No exemplo, a classe `HelloWorld` tem LOC = 3, contando as linhas 1, 3 e 4. A linha 4, apesar de possuir 2 instruções, conta como apenas 1 linha. Para efetuar comparações entre sistemas usando LOC, é necessário que ambos tenham sido feitos na mesma linguagem de programação e que o estilo esteja normalizado (Jones, 1991). Os intervalos sugeridos para o LOC de uma classe são: até 70 (bom); entre 70 e 130 (regular); de 130 em diante (ruim).

- **AMLOC** (Average Lines per Method — Número médio de linhas por método). No exemplo, a classe `Printer` tem AMLOC = 3, pois possui 2 métodos e LOC = 6. Essa medida indica se o código está bem distribuído entre os métodos. Quanto

maior, mais "pesados" são os métodos. É preferível ter muitas operações pequenas e de fácil entendimento que poucas operações grandes e complexas. Os intervalos sugeridos são: até 10 (bom); entre 10 e 13 (regular); de 13 em diante (ruim).

- **DIT** (Depth of Inheritance Tree — Profundidade da árvore de herança) é o número de superclasses ou classes ancestrais da classe sendo analisada. São contadas apenas as superclasses do sistema, ou seja, as classes de bibliotecas não são contabilizadas. No exemplo, a classe `HelloWorldPrinter` tem DIT = 1 e nas demais DIT = 0. Nos casos onde herança múltipla é permitida, considera-se o maior caminho da classe até uma das raízes da hierarquia. Quanto maior for o valor DIT, maior é o número de atributos e métodos herdados, e portanto maior é a complexidade. Os intervalos sugeridos são: até 2 (bom); entre 2 e 4 (regular); de 4 em diante (ruim).

- **LCOM** (Lack of Cohesion in Methods — Ausência de coesão em métodos). Foi originalmente proposta por Chidamber e Kemerer (Chidamber e Kemerer, 1994). A definição original recebeu várias críticas e sugestões de melhorias. Hitz e Montazeri (Hitz e Montazeri, 1995) sugeriram a métrica LCOM4. Seja $M = \{M_1, \ldots, M_n\}$ o conjunto dos métodos da classe analisada. Dois métodos M_i e M_j estão relacionados se ambos acessam pelo menos um mesmo atributo da classe ou se M_i chama ou é chamado por M_j. LCOM4 é a quantidade de partições de M formadas após separar os métodos em conjuntos de métodos relacionados. No exemplo, a classe `Printer` tem LCOM4 = 1, pois todos os métodos acessam a variável `message`. Já `HelloWorldPrinter` tem LCOM4 = 2, pois o contrutor e o método `doNothing()` não estão relacionados. Coesão entre os métodos de uma classe é uma propriedade desejável, portanto o valor ideal dessa métrica é 1. Se uma classe tem diferentes conjuntos de métodos não relacionados entre si, é um indício de que a classe deveria ser quebrada em classes menores e mais coesas. Os intervalos sugeridos são: até 2 (bom); entre 2 e 5 (regular); de 5 em diante (ruim).

- **Número de atributos públicos** mede o encapsulamento. Os atributos de uma classe devem servir apenas às funcionalidades da própria classe. Portanto, as variáveis de classe devem ser ocultadas para evitar complexidade, pois fica difícil prever os efeitos colaterais de alterar atributos públicos. Assim, o valor ideal dessa métrica é zero. Os intervalos sugeridos são: até 1 (bom); entre 1 e 9 (regular); de 9 em diante (ruim).

- **Número de métodos públicos** é o tamanho da "interface" da classe. Os métodos públicos representam os serviços que a classe disponibiliza. Valores altos para essa métrica indicam que a classe tem demasiadas funcionalidades e que poderia ser quebrada. Os intervalos sugeridos são: até 10 (bom); entre 10 e 40 (regular); de 40 em diante (ruim).

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

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

Então $ACC(C) = \sum_{i=1}^{n} cliente(C_i, C)$, onde n é o número total de classes do sistema. No nosso exemplo, $ACC($`Printer`$)$ é 2, pois ela é utilizada pelas outras duas

classes. `HelloWorldPrinter` é filha e portanto cliente de `Printer`. Se o valor dessa métrica for grande, uma mudança na classe tem potencialmente mais efeitos colaterais, tornando mais difícil a manutenção. Os intervalos sugeridos são: até 2 (bom); entre 2 e 20 (regular); de 20 em diante (ruim).

- **CBO** (Coupling Between Objects — Ligações entre objetos) é a recíproca da métrica ACC. Mede quantas classes são utilizadas pela classe analisada, ou seja $CBO(C) = \sum_{i=1}^{n} cliente(C, C_i)$.

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

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

O numerador é o total de ligações entre as classes e o denominador é o total possível de ligações. Portanto, COF é a porcentagem de ligações. No nosso exemplo, o numerador é $ACC(\texttt{Printer}) + ACC(\texttt{HelloWorldPrinter}) + ACC(\texttt{HelloWord}) = 2+1+0 = 3$. O denomindador é $3^2-3 = 6$, e portanto COF = 0,5. Um software fortemente conectado apresenta maior COF, indicando um baixo grau de independência entre os módulos, alta complexidade e difíceis entendimento e manutenção. Os intervalos sugeridos são: até 0,02 (bom); entre 0,02 e 0,14 (regular); de 0,14 em diante (ruim).

Capítulo 3

Kalibro: configuração e interpretação de métricas

Um recente estudo sobre a avaliação automática de código-fonte afirma que foi observada uma carência nas ferramentas de métricas (Meirelles et al., 2009). A interpretação livre de contexto de uma métrica pode não retratar as características consideradas importantes para um determinado software. Mesmo para um especialista é difícil avaliar a qualidade do código olhando apenas para os valores obtidos, sem pontos de referência.

Metric	Total	Mean	Std. Dev.	Maximum	Resource causing Maximum
Number of Overridden Methods (avg/max per type)	10	0,156	0,592	3	/Kalibro/src/br/usp/ime/ccsl/kal
Number of Attributes (avg/max per type)	160	2,5	2,681	11	/Kalibro/src/br/usp/ime/ccsl/kal
Number of Children (avg/max per type)	2	0,031	0,174	1	/Kalibro/src/br/usp/ime/ccsl/kal
Number of Classes (avg/max per packageFragment)	64	4,571	3,087	10	/Kalibro/src/br/usp/ime/ccsl/kal
Method Lines of Code (avg/max per method)	2952	5,766	6,424	73	/Kalibro/src/br/usp/ime/ccsl/kal
Number of Methods (avg/max per type)	482	7,531	7,902	43	/Kalibro/src/br/usp/ime/ccsl/kal
Nested Block Depth (avg/max per method)		1,309	0,565	4	/Kalibro/src/br/usp/ime/ccsl/kal
Depth of Inheritance Tree (avg/max per type)		1,953	1,595	6	/Kalibro/src/br/usp/ime/ccsl/kal
Number of Packages	14				
Afferent Coupling (avg/max per packageFragment)		8,286	9,58	29	/Kalibro/src/br/usp/ime/ccsl/kal
Number of Interfaces (avg/max per packageFragment)	1	0,071	0,258	1	/Kalibro/src/br/usp/ime/ccsl/kal
McCabe Cyclomatic Complexity (avg/max per method)		1,498	1,035	13	/Kalibro/src/br/usp/ime/ccsl/kal
Total Lines of Code	4989				
Instability (avg/max per packageFragment)		0,504	0,368	1	/Kalibro/src/org/softwarelivre/a
Number of Parameters (avg/max per method)		0,49	0,855	6	/Kalibro/src/br/usp/ime/ccsl/kal
Lack of Cohesion of Methods (avg/max per type)		0,356	0,377	1	/Kalibro/src/br/usp/ime/ccsl/kal
Efferent Coupling (avg/max per packageFragment)		3,143	2,474	8	/Kalibro/src/br/usp/ime/ccsl/kal
Number of Static Methods (avg/max per type)	30	0,469	1,51	8	/Kalibro/src/br/usp/ime/ccsl/kal
Normalized Distance (avg/max per packageFragment)		0,46	0,35	0,935	/Kalibro/src/br/usp/ime/ccsl/kal
Abstractness (avg/max per packageFragment)		0,036	0,087	0,25	/Kalibro/src/br/usp/ime/ccsl/kal
Specialization Index (avg/max per type)		0,013	0,053	0,333	/Kalibro/src/br/usp/ime/ccsl/kal
Weighted methods per Class (avg/max per type)	767	11,984	11,146	53	/Kalibro/test/br/usp/ime/ccsl/ka
Number of Static Attributes (avg/max per type)	17	0,266	1,019	7	/Kalibro/src/br/usp/ime/ccsl/kal

Figura 3.1: Metrics

Um exemplo de onde podemos constatar essa carência é a ferramenta Metrics (Sauer, 2005). Metrics é um plugin para o IDE Eclipse. Ele calcula várias métricas (23 na versão 1.3.6) cada vez que o projeto é compilado. Ela permite a configuração de valores máximo e mínimo recomendado para cada métrica e exibe alertas quando esses valores são violados. A própria Metrics sugere apenas valores máximos para 3 métricas, aplicadas a métodos: 10 para complexidade ciclomática; 5 para número de parâmetros e 5 para profundidade de aninhamento. Para cada métrica é mostrado o total, a média, o desvio padrão e o valor máximo. Observa-se que falta flexibilidade na configuração e recomendação, além de uma apresentação gráfica intuitiva. A figura 3.1 mostra uma captura de tela da Metrics.

Nesse contexto, desenvolvemos a Kalibro, que é uma evolução do que foi prototipado no software livre denominado Crab (Meirelles et al., 2009). A Crab foi pensada dentro do projeto de doutorado chamado Mangue e teve o início de sua implementação na disciplina Laboratório de Programação Extrema do IME-USP. O seu código-fonte foi disponibilizado sob a licença BSD (Berkeley Software Distribution, 2009), o que nos permitiu renomear a versão estável para Kalibro e relicenciar sob LGPL versão 3 (*http://www.opensource.org/licenses/lgpl-license.php*) para garantirmos a liberdade do software nas versões posteriores.

A Kalibro se diferencia das demais ferramentas de métricas de software, pois recebe como entrada a definição de um conjunto de métricas, que deve ser obtida a partir de uma ferramenta específica de métricas de software, chamada aqui de *ferramenta base*. Como saída, produz um resultado da avaliação da qualidade do software em um formato de mais fácil entendimento. Ela possibillita que, um usuário com conhecimento em métricas, possa cadastrar intervalos de valores de julgamento para cada métrica, juntamente com uma avaliação qualitativa. O desenvolvedor comum poderá carregar as configurações definidas por especialistas e visualizar de forma muito mais clara as classes e os métodos que estão dentro do esperado de acordo com os valores de referência para uma determinada métrica, bem como quais contêm as principais deficiências. Além disso, uma nota final para o sistema analisado é gerada, segundo essa configuração.

A Kalibro foi desenvolvida com o objetivo de potencializar o uso das métricas de código-fonte. Cada métrica é calculada para cada classe e, em seguida, sumarizada para a avaliação do software como um todo. A Kalibro também possibilita a criação de métricas compostas, definidas através de expressões que podem utilizar os valores coletados das outras métricas. Essa funcionalidade torna possível expressar melhor uma determinada característica do software.

3.1 Arquitetura da Kalibro

A Kalibro foi projetada com o objetivo de simplificar sua integração com diferentes ferramentas base. Sua arquitetura é composta de poucos pacotes, cujos componentes têm responsabilidades bem definidas (vide figura 3.2).

O modelo (`model` na figura 3.2) é constituído das classes que representam as entidades básicas manipuladas pela Kalibro. O pacote `br.usp.ime.ccsl.kalibro.model` possui as seguintes classes:

- **Range:** representa um intervalo de avaliação do valor de uma métrica. Para cada intervalo é dado um nome, uma nota, comentários e recomendações. A Kalibro associa o valor de uma métrica calculada a um dos intervalos definidos para essa métrica. No intervalo, o valor inicial é fechado e o valor final é aberto, para que seja possível determinar uma faixa contínua de intervalos. Por exemplo, podemos definir os intervalos [0,5[e [5,10[. Neste caso, uma métrica com valor 5 pertence ao intervalo [5, 10[.

- **Metric:** representa uma métrica dentro da Kalibro. Ela possui nome, descrição, categoria, peso e um conjunto de intervalos (`Range`). A categoria é como o valor da métrica em um agrupamento de classes deverá ser calculado se baseando nos valores de cada classe. As categorias suportadas são soma, média, máximo e mínimo. O peso é o quanto a métrica é importante no contexto do código avaliado, sendo usado no cálculo de uma média ponderada na nota final de avaliação de um programa.

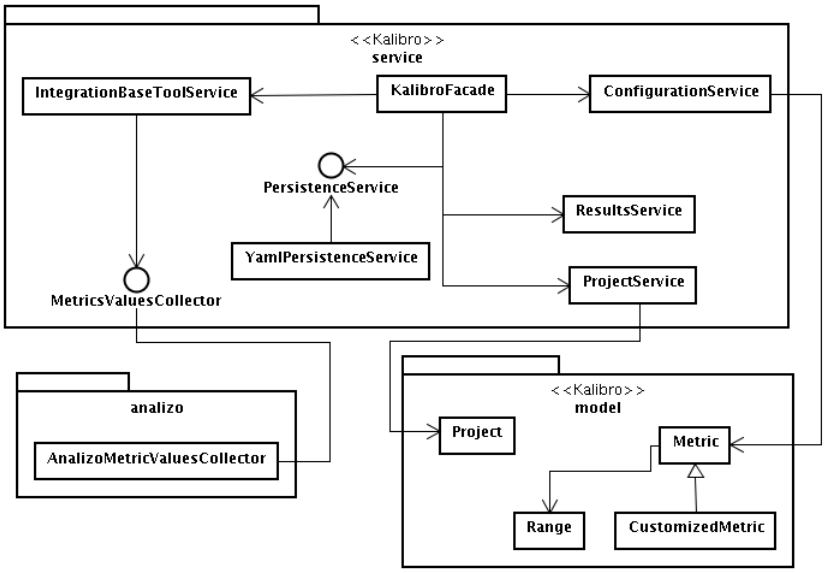

Figura 3.2: Arquitetura da Kalibro

Um conjunto de métricas, com seus intervalos de avaliação definidos formam uma configuração.

- **CustomizedMetric:** é uma especialização de `Metric` que representa uma métrica composta. Nela é definido um código JavaScript para que seu valor seja obtido a partir de expressões que podem usar as outras métricas como variáveis.

- **Project:** representa um projeto que será avaliado pela Kalibro. Possui nome, descrição e o caminho do código a ser avaliado. O projeto guarda ainda dados da última avaliação: o caminho para o arquivo onde foi salva a configuração utilizada; os valores das métricas colhidos; a data em que foram colhidos; o tempo que a ferramenta base levou para colher os valores.

O pacote serviço (service na figura 3.2) é composto pelas classes que manipulam as regras de negócio implementadas na Kalibro. As classes de serviços foram implementadas seguindo padrão de projeto Mediator (Gamma et al., 1994). As classes que seguem esse padrão desacoplam e gerenciam as colaborações entre um grupo de objetos. Isso permite a unificação da interface do sistema, diminuindo a dependência entre os objetos que estão se comunicando.

As funcionalidades do serviço somente são acessadas através de uma classe que atua como sua fachada, o que torna o núcleo da arquitetura independente da forma de interação com o usuário. O pacote `br.usp.ime.ccsl.kalibro.service` possui as seguintes classes:

- **KalibroFacade:** O componente de entrada do pacote. Representa o centro da arquitetura da Kalibro, sendo implementada seguindo o padrão de projeto Façade (Gamma et al., 1994). Essa classe é uma interface para todas as funcionalidades da Kalibro, tornando os serviços mais fáceis de entender e usar. Uma classe de interface gráfica necessita apenas desse ponto de acesso. Da mesma forma, um serviço apenas acessa outro serviço através da fachada.

- **MetricsValuesCollector:** A interface que deve ser implementada pela ferramenta base. Ela possui dois métodos: `getSupportedMetrics()`, que deve devolver a lista

de métricas providas pela ferramenta, e `calculateMetrics()`, que deve executar o cálculo e devolver os resultados. O apêndice A mosta um exemplo de implementação.

- **PersistenceService:** Essa interface foi criada para guardar em disco as configurações, os projetos avaliados pela Kalibro e a lista de projetos recentes.

- **YamlPersistenceService:** Essa classe implementa `PersistenceService`, e gerencia arquivos no formato YAML (YAML, 2009). Para que a Kalibro passe a suportar outro formato basta criar uma classe que implemente a persistência de outra forma.

- **ConfigurationService:** Uma configuração é um conjunto de objetos `Metric`, ou seja, um conjunto de métricas com seus pesos e intervalos qualitativos definidos para classificar o código. A classe `ConfigurationService` administra a configuração carregada pela Kalibro. Ela possui métodos responsáveis por carregar, validar e salvar uma configuração, consultar, incluir, editar e remover métricas e seus dados da configuração.

- **IntegrationBaseToolService:** Administra a comunicação entre a Kalibro e a ferramenta base. Possui métodos para colher resultados e criar uma configuração vazia, baseando-se nas métricas fornecidas pela ferramenta base. Realiza a validação dos dados fornecidos pela `MetricValuesCollector`.

- **ResultsService:** A classe responsável por todos os cálculos de valores de métricas que ocorrem na Kalibro. Valida e executa o código JavaScript das métricas compostas, baseando-se nos resultados das métricas nativas coletadas pela ferramenta base. Também totaliza os valores de todas as métricas para a avaliação geral do código, calculando o resultado final de cada métrica de acordo com a categoria definida.

- **ProjectService:** Responsável por guardar, carregar, validar e integrar projetos com a configuração e os resultados. Guarda também uma lista com os projetos recentes, que é atualizada cada vez que um projeto é salvo ou carregado.

3.2 Funcionalidades da Kalibro

A Kalibro é usada em conjunto com qualquer ferramenta de coleta de métricas. Dessa forma, ao ser integrada a uma ferramenta base, a Kalibro passa a utilizar suas métricas. As funcionalidades da Kalibro são as seguintes:

- **Definir um projeto de análise de código-fonte:** Um projeto é composto pelo código-fonte do software analisado, os valores das métricas colhidos pela ferramenta base e a configuração de intervalos para avaliação. Para criar um novo projeto é preciso o nome e o caminho do código fonte (vide figura 3.3).

- **Carregar métricas nativas providas por uma ferramenta base:** A Kalibro permite adicionar, editar e excluir de uma configuração as métricas definidas pela ferramenta-base (vide figura 3.4).

- **Criar uma configuração de métricas:**. Definir descrição, categoria de contabilização (soma, média, mínimo e máximo) e peso para cada métrica (vide figura 3.5).

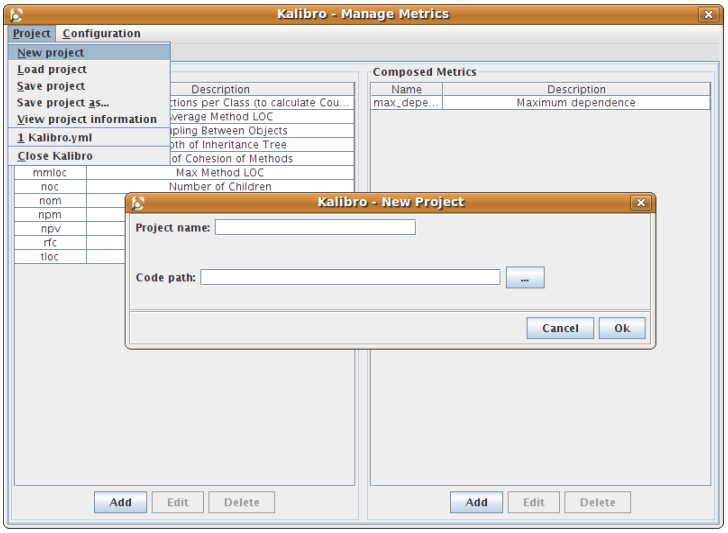

Figura 3.3: Kalibro: Criando novo projeto

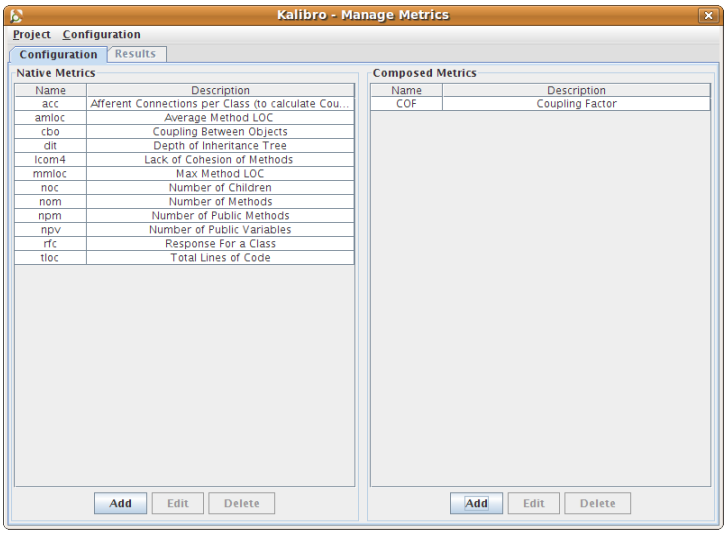

Figura 3.4: Kalibro: tela de configuração

- **Adicionar métricas compostas a partir das métricas nativas:** É possível definir novas métricas através de um código Javascript, que pode utilizar como variáveis valores de métricas nativas (providas pela ferramenta base) ou de outras métricas compostas previamente definidas (a figura 3.5 mostra o script que gera o COF).

- **Definir intervalos qualitativos para as métricas:** Para cada métrica da configuração é possível definir intervalos de avaliação. Os intervalos possuem uma avaliação qualitativa (ex.: péssimo, ruim, regular, bom, ótimo), uma nota (avaliação numérica), comentários e recomendações. É possível definir uma faixa contínua de intervalos, com valor inicial aberto e final fechado (vide figura 3.6).

- **Contabilizar os valores das métricas:** A Kalibro chama a ferramenta base, que calcula os valores das métricas nativas para cada classe. De posse desses valores, é realizado o cálculo das métricas compostas (definidas por Javascript) e a totalização dos valores de cada métrica, de acordo com sua categoria. Para cada métrica da

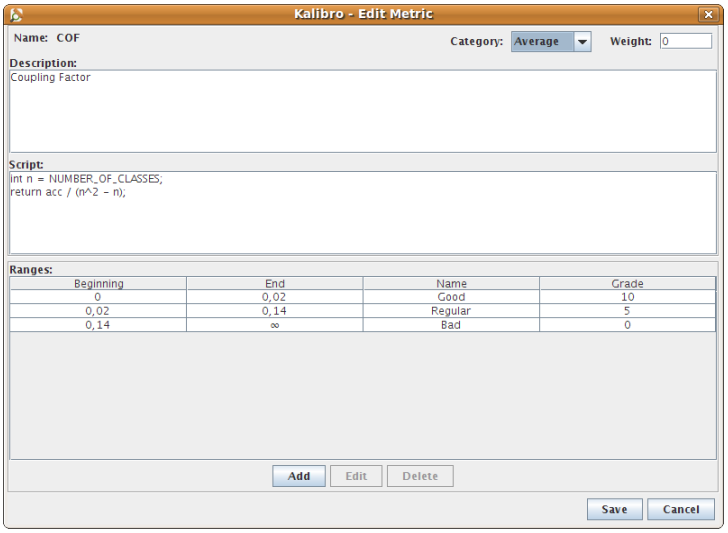

Figura 3.5: Kalibro: criando uma métrica composta

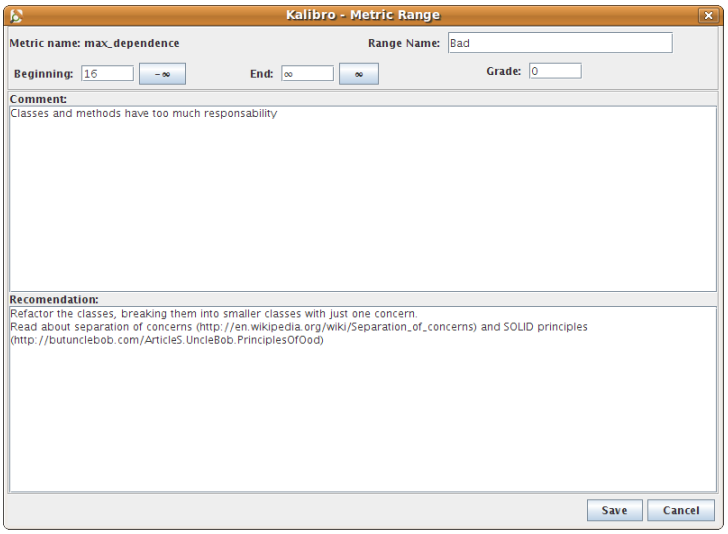

Figura 3.6: Kalibro: configurando um intervalo

configuração, verifica em que intervalo o resultado é classificado.

- **Visualizar resultados:** O resultados podem ser vistos de forma detalhada (para cada classe) ou sumarizada (todo o programa). Ao usuário são apresentadas a avaliação qualitativa, a nota, os comentários e as recomendações do intervalo. É mostrado ainda, através de barras, onde está situado o resultado em comparação com o valor mínimo e máximo do intervalo. Com base nas notas (definidas nos intervalos) e nos pesos (definidos nas métricas), é calculada uma nota (via média ponderada) para cada classe e para o sistema como um todo (vide figura 3.7).

- **Persistir resultados e configurações:** A configuração pode ser salva e associada ao projeto. Isso possibilita definir valores de referência para as métricas em diversos contextos, estabelecidos por diferentes especialistas em métricas, de tal forma que o desenvolvedor comum poderá escolher a configuração a ser usada. Os resultados também são gravados em arquivo para que não seja necessário recalculá-los, quando se pretende apenas visualizar a última análise.

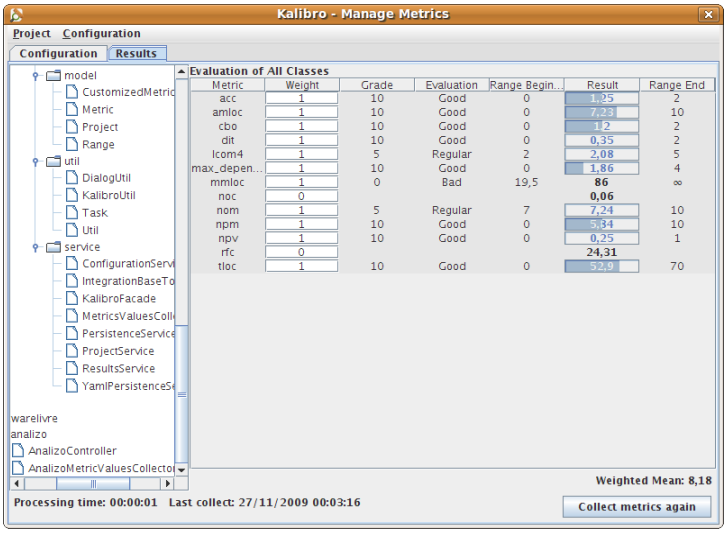

Figura 3.7: Kalibro: vizualização de resultados

3.3 Integração da Crab com a Jabuti

A ferramenta prototipada como Crab foi integrada à Jabuti (Vincenzi et al., 2003), que calcula 27 métricas através do *bytecode*, código Java compilado. A integração foi feita estendendo a classe abstrata `MetricValuesCollectorServiceAbstract`, que na Kalibro foi substituída pela interface `MetricValuesCollector`. A classe coletora de métricas da Jabuti (anexo D), realiza a integração com uma simples mudança no formato dos dados manipulados pela Jabuti para preencher as estruturas de dados requeridas pela Crab.

O cálculo de métricas a partir do código compilado possui duas grandes desvantagens em relação ao cálculo a partir do código-fonte. Uma delas é a perda de informação, uma vez que constantes, comentários e macros não aparecem no código compilado. A outra é a impossibilidade de colher qualquer informação de um software que não possa ser compilado no momento.

3.4 Integração com a Analizo

O programa Analizo é constituído por um conjunto de ferramentas que realizam a análise de um software. Uma dessas ferramentas é a Analizo Metrics, que produz um relatório de métricas a partir do caminho do código-fonte. Neste trabalho estamos chamando de Analizo apenas a ferramenta Analizo Metrics.

Doxygen (http://www.doxygen.org) é um sistema de documentação para várias linguagens, enrte elas C, C++, C#, Objective-C, Java, Python, Fortran, IDL. O Doxygen extrai informações de hierarquia e colaborações entre módulos de um projeto. A Analizo baseia-se nas informações extraídas pelo Doxygen para cálculo das métricas, portanto foi feita para ser extensível e independente de linguagem.

O nome "analizo" significa "análise" em esperanto. Ela começou como uma versão modificada do sotware livre Egypt, criado por Andreas Gustafsson (Egypt, 2006). Porém, tantas funcionalidade foram adicionadas e removidas por Antonio Terceiro (Terceiro e Chavez, 2008) que em Outubro de 2009 se tornou um outro software. A Analizo suporta a análise de código C, C++ e Java. Ela calcula as seguintes métricas por classe:

- ACC: Conexões aferentes

- AMLOC: Média de linhas por método
- CBO: Acoplamento entre objetos
- DIT: Profundidade da árvore de herança
- LCOM4: Falta de coesão em métodos
- MMLOC: Máximo de linhas por método
- NOC: Número de filhos imediatos de uma classe
- NOM: Número de métodos
- NPM: Número de métodos públicos
- NPV: Número de variáveis públicas
- RFC: Resposta de uma classe
- LOC: Total de linhas de código

```
---
_module: HelloWorld
acc: 0
amloc: 3
cbo: 1
dit: 0
lcom4: 1
mmloc: 3
noc: 0
nom: 1
npm: 1
npv: 0
rfc: 2
tloc: 3
```

Listagem 3.1: Saída da Analizo para o código 2.1

Também, calcula resultados resumidos para as seguintes métricas pelo total de classes:

- Média de Acoplamento (CBO)
- Média da falta de coesão (LCOM4)
- Fator de acoplamento (COF)
- Número total de classes
- Número total de métodos
- Número total de métodos públicos
- Número total de variáveis públicas

- Número total de linhas de código

A saída da Analizo é um relatório escrito no formato YAML (YAML, 2009), um padrão de serialização de objetos feito para ser de fácil leitura e manipulação em diferentes linguagens. A listagem 3.1 mostra o trecho da saída da Analizo relativo à classe `HelloWorld` do código de exemplo usado na seção 2.1.

A Kalibro foi integrada com a Analizo através da classe `AnalizoMetricValuesCollector` (mostrada no apêndice A), que implementa a interface `MetricValuesCollector` (vide seção 3.1). O método `getSupportedMetrics()` executa uma chamada de sistema através do comando *analizo-metrics --list*, processa a saída e devolve a lista de métricas suportadas, com nome e descrição. O método `calculateMetrics()` colhe os resultados da análise de um código-fonte, ou seja, faz uma outra chamada de sistema, passando como argumento o caminho do código-fonte.

Desse modo, verificamos como a Kalibro é uma ferramenta facilmente acoplável, pois nesse caso de integração, apenas foi necessário que os métodos da classe coletora de métricas funcionassem como conversores da saída da Analizo, preenchendo as estruturas de dados que os serviços da Kalibro manipulam.

Assim, mantivemos uma das principais características da ferramenta quando foi prototipada como Crab, uma vez que Meirelles (Meirelles et al., 2009) apresentou sua integração com a Jabuti de forma similar ao que fizemos com a Analizo.

Capítulo 4

Considerações finais

Como resultado do nosso trabalho, temos uma ferramenta facilmente acoplável que potencializa o uso de métricas de código-fonte. A Kalibro será distribuída com um arquivo de configuração padrão, que possui os intervalos sugeridos pelos valores de referência estudados. A ferramenta à qual integramos a Kalibro, a Analizo, apresentou ótima performance, levando poucos minutos na avaliação de projetos grandes (com centenas de milhares de linhas de código).

A Kalibro e a Analizo motivaram a criação de um projeto maior denominado Mezuro, que significa "medida de qualidade" em esperanto (Mezuro, 2009). O Mezuro será uma plataforma web que terá como base as funcionalidades da Kalibro e da Analizo, porém com a possibilidade de se conectar ao repositório do código-fonte que será avaliado, o que permitirá o cálculo das métricas para diferentes versões de um mesmo software.

Desse modo poderemos coletar valores de métricas para um número bastante grande de projetos, para que, em um trabalho futuro, esses dados possam ser utilizados em uma análise estatística para encontrar mais valores de referência.

Parte II
Parte subjetiva

Capítulo 5

Experiência pessoal

5.1 Desafios e frustrações

Neste trabalho de formatura, senti minha falta de prática com pesquisa e elaboração de texto acadêmico. Acredito que se tivesse feito iniciação científica teria menos problemas nesse aspecto.

Outra coisa que aprendi neste ano e que será de grande valia no futuro, foi ser mais proativo. Ao contrário das outras disciplinas, o trabalho de formatura nos dá grande liberdade, portanto é necessário planejar com antecedência e cautela as atividades e saber cobrar de si mesmo os resultados.

Este projeto foi uma boa experiência de trabalho em equipe. Os trabalhos em equipe feitos em outras disciplinas tiveram sempre um caráter de urgência desde o início, e houve mais divisão de tarefas que colaboração nas atividades, excetuando-se apenas Laboratório de Programação Extrema. Foi compensador colaborar com um software livre, interagindo com pesquisadores de vários estados brasileiros.

Entre as dificuldades de ordem técnica, tivemos que modificar substancialmente a estrutura da Crab para incorporar todas as novas funcionalidades que pretendíamos ter na Kalibro. E tendo em vista que a Kalibro não é um aplicativo que executa sozinho, mas sim uma ferramenta feita para ser acoplada a outro software, as possibilidades de erros que devem ser tratados é maior que em um projeto comum, e foi necessário melhorar consideravelmente a cobertura dos testes.

5.2 Disciplinas relevantes

MAC0323 — Estruturas de Dados Ensina a manipular estruturas de dados básicas. Qualquer sistema usa algum dos conceitos aqui apresentados. Na Kalibro são usados hashes, árvores e listas ligadas.

MAE0121 e MAE0212 — Introdução a Probabilidade e Estatística No início parece não ter muita relação com o resto do curso, mas depois fica evidente sua importância. Sem essas disciplinas o entendimento de vários conceitos importantes seria impossível ou muito superficial. Principalmente no que diz respeito à próxima disciplina desta lista.

MAC0338 — Análise de Algoritmos Tem papel central no curso. Nessa disciplina se aprende a analisar a complexidade dos softwares. Nela vimos como os programas feitos de forma "ingênua" são ruins e como melhorar substancialmente a eficiência. Também aqui são introduzidas as classes NP, NP-completo, conceitos muito importantes no decorrer do curso.

MAC0328 — Algoritmos em Grafos Os conceitos e os teoremas apresentados nessa disciplina são extensivamente usados para provar muitos fatos em Computação. Ela tem relação direta com este trabalho, pois a maior parte das métricas de código-fonte são calculadas baseando-se em grafos construídos a partir de instruções, atributos, métodos ou classes.

MAC0438 — Programação Concorrente Frequentemente surgem problemas onde devemos coordenar mais de um processo. Portanto, entender e saber utilizar semáforos e monitores é fundamental. E como hoje vivemos a ascenção dos processadores multicore, conhecer esses conceitos é uma grande vantagem. A Kalibro, mesmo sendo um programa relativamente simples, já possui algoritmos que usam mais de uma thread.

MAC0342 — Laboratório de Programação Extrema Essa disciplina foi das mais importantes na minha formação. Ela provocou grandes mudanças na minha maneira de trabalhar. Nela são vistos os princípios e valores dos métodos ágeis, desenvolvimento dirigido por testes, técnicas de refatoração, acompanhamento do processo de desenvolvimento (papel do *tracker*). Trabalhar em um dos projetos dessa disciplina traz grandes vantagens: aprender na prática, contribuindo com a solução de um problema real e envolver-se com pessoas e ideias às quais estará ligado mesmo após o fim do semestre. Foi a partir dela que surgiu meu interesse em trabalhar com métricas, e a Kalibro nasceu dentro dessa disciplina (vide capítulo 3).

5.3 Estudos futuros

Tenho interesse em aprofundar meus conhecimentos e ampliar minha experiência com métodos ágeis. Pretendo continuar contribuindo com a Kalibro e com o projeto Mezuro. As áreas de interesse que pretendo estudar na minha pós-graduação são: desenvolvimento de sistemas, serviços web e computação paralela/distribuída.

Bibliografia

Abreu, Fernando Brito e; Carapuca, R. Candidate metrics for object oriented software within a taxonomy framework. *Journal of Systems and Software*, v. 26, n. 1, 1994.

Albrecht, A. J.; Gaffney, J. E. Software function, source lines of code, and development effort prediction: A software science validation. *IEEE Transactions on Software Engineering*, v. 9, n. 6, p. 628–648, November 1983.

Arthur, Lowell Jay. *Measuring Programmer Productivity and Software Quality*. New York: Wiley-Interscience, 1985. 292 p.

Bansiya, J.; Davi, C. Using qmood++ for object-oriented metrics. *Dr. Dobb's Journal*, December 1997.

Basili, V. R.; Caldiera, G.; Rombach, H. D. *Goal question metric paradigm*. http://www.cs.umd.edu/projects/SoftEng/ESEG/papers/gqm.pdf, 1994.

BERKELEY Software Distribution. 2009. Web Site. Disponível em: <http://www.opensource.org/licenses/bsd-license.php>.

Boehm, B. W. *Software Engineering Economics*. N. J.: Prentice-Hall, 1981.

Brooks, Frederick Phillips. *The Mythical Man-Month*. [S.l.]: Addison-Wesley, 1975.

Card, D. N.; Agresti, W. W. Measuring software design complexity. *Jornal of Systems and Software*, v. 8, n. 3, p. 185–197, June 1988.

Chidamber, S. R.; Kemerer, C. F. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, v. 20, n. 6, p. 476–493, 1994.

Conte, S. D.; Dunsmore, H. E.; Shen, V. Y. *Software Engineering Metrics and Models*. California: Benjamin-Cummings, 1986.

DeMarco, T. *Controlling Software Projects: Management, Measurement & Estimation*. New York: Yourdon Press, 1982.

EGYPT. 2006. Web Site. Disponível em: <http://www.gson.org/egypt/>.

EXTREME Chaos. [S.l.], 2004, 2006, 2009.

Fenton, Norman E.; Pfleeger, Shari Lawrence. *Software Metrics: A Rigorous and Practical Approach*. 2 edition ed. [S.l.]: Course Technology, 1998. 656 p.

Ferreira, Kécia Aline M.; Bigonha, Mariza A. S.; Bigonha, Roberto S. Reestruturação de software dirigida por conectividade para redução de custo de manutenção. In: *Revista de Informática Teórica e Aplicada*. [S.l.: s.n.], 2008. v. 15, p. 155–180.

GAMMA, Erich et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. [S.l.]: Addison-Wesley, 1994.

GRADY, R. B.; CASWELL, D. R. *Software Metrics: Establishing a Company-Wide Program*. N. J: Prentice-Hall, 1987.

HALSTEAD, M. H. Natural laws controlling algoritmic structure? *ACM SIGPLAN Notices*, v. 7, n. 2, p. 19 – 26, Febuary 1972.

HALSTEAD, M. H. *Elements of Software Science*. New York: Elsevier North-Holland, 1977.

HARRISON, W.; COOK, C. A micro/macro measure of software complexity. *Jornal of Systems and Software*, v. 7, n. 3, p. 213–219, September 1987.

HENRY, S.; KAFURA, D. The evaluation of software systems' structure using quantitative software metrics. *Software — Practice and Experience*, v. 14, n. 6, p. 561–573, June 1984.

HITZ, M.; MONTAZERI, B. Measuring coupling and cohesion in object-oriented systems. *Int. Symposium on Applied Corporate Computing*, 1995.

ISO/IEC9126-1. *Software engineering — Product quality — Part 1: Quality Model*. 2001.

JONES, T. C. *Programming Productivity*. New York: McGraw-Hill, 1986.

JONES, T. C. *Applied Software Measurement: Assuring Productivity and Quality*. New York: McGraw-Hill, 1991.

KAFURA, D.; HENRY, S. Software quality metrics based on interconnectivity. *Journal of Systems and Software*, v. 2, n. 2, p. 121–131, June 1981.

LANZA, Michele; MARINESCU, Radu. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate and Improve the Design of Object-Oriented Systems*. [S.l.]: Hardcover, 2006. 206 p.

LI, Hon Fung; CHEUNG, William Kwok. An empirical study of software metrics. *IEEE Transactions Software Engineering*, v. 13, n. 6, p. 697–708, June 1987.

MCCABE, T. J. A complexity measure. *IEEE Transactions Software Engineering*, v. 2, n. 4, p. 308–320, December 1976.

MCCABE, T. J. *Structured Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric*. [S.l.], December 1982.

MCCABE, T. J.; BUTLER, C. Design complexity measurement and testing. *Communications of the ACM*, v. 32, p. 1415–1425, December 1989.

MEIRELLES, Paulo R. M. et al. Crab: Uma ferramenta de configuração e interpretação de métricas de software para avaliação de qualidade de código. In: *XXIII SBES - Simpósio Brasileiro de Engenharia de Software (XVI Sessão de Ferramentas)*. [S.l.: s.n.], 2009.

MEIRELLES, Paulo R. M.; KON, Fabio. Mangue: Metrics and tools for automatic quality evaluation of source code. In: *VIII SBQS - Simpósio Brasileiro de Qualidade de Software (VII Workshop de Teses e Dissertações em Qualidade de Software (WTDQS))*. [S.l.: s.n.], 2009.

MEZURO. 2009. Web Site. Disponível em: <http://softwarelivre.org/mezuro/>.

MILLS, Everald E. *Software Metrics*. SEI - Carnegie Mellon University, 1988.

MYERS, G. J. An extension to the cyclomatic measure of program complexity. *ACM SIGPLAN Notices*, v. 12, n. 10, p. 61–64, October 1977.

PERLIS, A.; SAYWARD, F.; SHAW, M. *Software Metrics: An Analysis and Evaluation*. Cambridge, Mass: MIT Press, 1981.

POPPENDIECK, Mary; POPPENDIECK, Tom. *Lean Software Development: An Agile Toolkit for Software Development Managers*. [S.l.]: Addison-Wesley Professional, 2003.

RAYMOND, Eric Steven. The cathedral & the bazaar. O'Reilly Media, 1999.

ROCHA, A. R. C.; MALDONADO, J. C.; WEBER, K. C. *Qualidade de software — Teoria e prática*. São Paulo: Prentice Hall, 2001.

ROSENBERG, L. H.; HYATT, L. E. Software quality metrics for object-oriented environments. *Crosstalk Journal*, 1997.

SAUER, Frank. *Metrics - Eclipse Metrics Plugin*. 2005. Web Site. Disponível em: <http://metrics.sourceforge.net/>.

SCACCHI, W. Free/open source software development: Recent research results and methods. *Zelkowitz, M. V., editor, Advances in Computers*, v. 69, p. 243–269, 2007.

SHARBLE, R.; COHEN, S. The object-oriented brewery: A comparison of two object-oriented development methods. *Software Engineering Notes*, v. 18, n. 2, p. 60–73, 1993.

SHIH, T.K. et al. Decomposition of inheritance hierarchy dags for object-oriented software metrics. In: *Workshop on Engineering of Computer-Based Systems (ECBS 97)*. [S.l.: s.n.], 1997. p. 238.

STETTER, F. A measure of program complexity. *Computer Languages*, v. 9, n. 3-4, p. 203–208, 1984.

TEMPERO, Ewan. On measuring java software. *ACSC2008. Wollongong, Australia. Conferences in Research and Practice in Information Technology (CRPIT)*, v. 74, 2008.

TERCEIRO, Antonio; CHAVEZ, Christina. Structural complexity evolution in free software projects: A case study. 2008.

TROY, D. A.; ZWEBEN, S. H. Measuring the quality of structured designs. *J. Syst. and Software*, v. 2, n. 2, p. 113–120, June 1981.

VANDOREN, Edmond. *C4 Software Technology Reference Guide — A Prototype*. Carnegie Mellon University, Pittsburgh, Pennsylvania 15213, 1997. 231-236 p.

VINCENZI, A. M. R. et al. Jabuti: A coverage analysis tool for java programs. In: *XVII SBES — Brazilian Symposium on Software Engineering*. [S.l.: s.n.], 2003. p. 79–84.

WOLVERTON, R. W. The cost of developing large-scale software. *IEEE Transactions Computers*, C-23, n. 6, p. 615–636, June 1974.

WOODWARD, M. R.; HENNELL, M. A.; HEDLEY, D. A measure of control flow complexity in program text. *EEE Transactions. Software Engineering*, v. 5, n. 1, p. 45–50, January 1979.

XENOS, M. et al. Object-oriented metrics - a survey. In: *Proceedings of the FESMA 2000 - Federation of European Software Measurement Associations*. Madrid, Spain: [s.n.], 2000. v. 6.

YAML. *YAML Ain't Markup Language*. 2009. Web Site. Disponível em: <http://www.yaml.org/>.

YAU, S. S.; COLLOFELLO, J. S. Design stability measures for software maintenance. *IEEE Transantions Software Engineering*, v. 11, n. 9, p. 849–856, September 1985.

ZUSE, Horst. *Software Complexity: Measures and Methods*. [S.l.]: Walter de Gruyter, 1990.

Apêndice A

Código de integração com a Analizo

```
1   package org.softwarelivre.analizo;
2
3   import br.usp.ime.ccsl.kalibro.exception.NoCodeAvailableException;
4   import br.usp.ime.ccsl.kalibro.model.Metric;
5   import br.usp.ime.ccsl.kalibro.service.MetricsValuesCollector;
6
7   import java.io.BufferedReader;
8   import java.io.File;
9   import java.io.IOException;
10  import java.io.InputStreamReader;
11  import java.util.*;
12
13  public class AnalizoMetricValuesCollector implements MetricsValuesCollector {
14
15          @Override
16          public Map<String, Map<String, Double>> calculateMetrics(String codePath) throws NoCodeAvailableException {
17                  try {
18                          Process process = Runtime.getRuntime().exec("analizo-metrics " + codePath);
19                          BufferedReader reader = new BufferedReader(new InputStreamReader(process.getInputStream()));
20
21                          Map<String, Map<String, Double>> modulesMap = new HashMap<String, Map<String, Double>>();
22                          Map<String, Double> metricsMap = null;
23
24                          String line;
25                          for (line = reader.readLine(); ! line.matches("_module.*"); line = reader.readLine()) {}
26
27                          for (; line != null; line = reader.readLine()) {
28                                  if (line.matches("_module: .*")) {
29                                          metricsMap = new HashMap<String, Double>();
30                                          String moduleName = line.substring(9).replaceAll(":+", File.separator);
31                                          modulesMap.put(moduleName, metricsMap);
32                                  } else if (! line.equals("---") && metricsMap != null) {
33                                          int separatorIndex = line.indexOf(": ");
34                                          String metricName = line.substring(0, separatorIndex);
35                                          Double result = Double.parseDouble(line.substring(separatorIndex + 2));
36                                          metricsMap.put(metricName, result);
37                                  }
38                          }
39
40                          return modulesMap;
41                  } catch (Exception e) {
42                          throw new NoCodeAvailableException();
43                  }
44          }
45
46          @Override
47          public List<Metric> getSupportedMetrics() {
48                  try {
49                          List<Metric> metricsList = new ArrayList<Metric>();
50                          Process process = Runtime.getRuntime().exec("analizo-metrics --list");
51                          BufferedReader reader = new BufferedReader(new InputStreamReader(process.getInputStream()));
52
53                          for (String line = reader.readLine(); line != null; line = reader.readLine()) {
54                                  int separatorIndex = line.indexOf(" - ");
55                                  Metric metric = new Metric(line.substring(0, separatorIndex));
56                                  metric.setDescription(line.substring(separatorIndex + 3));
57                                  metricsList.add(metric);
58                          }
59                          Collections.sort(metricsList);
60                          return metricsList;
61                  } catch (IOException e) {
62                          e.printStackTrace();
63                          return null;
64                  }
65          }
66  }
```

Apêndice B

Exemplo de saída da Analizo

```
---
average_cbo: 0.333333333333333
average_lcom4: 1.33333333333333
cof: 0.333333333333333
sum_classes: 3
sum_nom: 5
sum_npm: 5
sum_npv: 0
sum_tloc: 15
---
_module: HelloWorld
acc: 0
amloc: 3
cbo: 1
dit: 0
lcom4: 1
mmloc: 3
noc: 0
nom: 1
npm: 1
npv: 0
rfc: 2
tloc: 3
---
_module: Printer
acc: 2
amloc: 3
cbo: 0
dit: 0
lcom4: 1
mmloc: 3
noc: 1
nom: 2
npm: 2
npv: 0
rfc: 4
tloc: 6
---
_module: HelloWorldPrinter
acc: 0
amloc: 3
cbo: 0
dit: 1
lcom4: 2
mmloc: 3
noc: 0
nom: 2
npm: 2
npv: 0
rfc: 2
tloc: 6
```

Apêndice C

Exemplo de projeto salvo pela Kalibro

```
--- !br.usp.ime.ccsl.kalibro.model.Project
codePath: /media/Dados/workspace/Eclipse/Kalibro/docs/TCC_Carlos
computationTime: "00:00:00"
configurationPath: /home/carlos/.kalibro/settings/default-settings.yml
name: HelloWorld
resultTimeStamp: "27/11/2009 08:34:56"
results:
  All Classes: !java.util.HashMap
    noc: !java.lang.Double 0.3333333333333333
    tloc: !java.lang.Double 15.0
    amloc: !java.lang.Double 3.0
    cbo: !java.lang.Double 0.3333333333333333
    max_dependence: !java.lang.Double 0.3333333333333333
    npm: !java.lang.Double 1.6666666666666667
    acc: !java.lang.Double 0.6666666666666666
    nom: !java.lang.Double 1.6666666666666667
    rfc: !java.lang.Double 2.6666666666666665
    npv: !java.lang.Double 0.0
    dit: !java.lang.Double 0.3333333333333333
    lcom4: !java.lang.Double 1.3333333333333333
    mmloc: !java.lang.Double 3.0
  HelloWorldPrinter: !java.util.HashMap
    noc: !java.lang.Double 0.0
    tloc: !java.lang.Double 6.0
    amloc: !java.lang.Double 3.0
    cbo: !java.lang.Double 0.0
    max_dependence: !java.lang.Double 0.0
    npm: !java.lang.Double 2.0
    acc: !java.lang.Double 0.0
    nom: !java.lang.Double 2.0
    rfc: !java.lang.Double 2.0
    npv: !java.lang.Double 0.0
    dit: !java.lang.Double 1.0
    lcom4: !java.lang.Double 2.0
    mmloc: !java.lang.Double 3.0
  Printer: !java.util.HashMap
    noc: !java.lang.Double 1.0
    tloc: !java.lang.Double 6.0
    amloc: !java.lang.Double 3.0
    cbo: !java.lang.Double 0.0
    max_dependence: !java.lang.Double 0.0
    npm: !java.lang.Double 2.0
    acc: !java.lang.Double 2.0
    nom: !java.lang.Double 2.0
    rfc: !java.lang.Double 4.0
    npv: !java.lang.Double 0.0
    dit: !java.lang.Double 0.0
    lcom4: !java.lang.Double 1.0
    mmloc: !java.lang.Double 3.0
```

```
HelloWorld: !java.util.HashMap
  noc: !java.lang.Double 0.0
  tloc: !java.lang.Double 3.0
  amloc: !java.lang.Double 3.0
  cbo: !java.lang.Double 1.0
  max_dependence: !java.lang.Double 1.0
  npm: !java.lang.Double 1.0
  acc: !java.lang.Double 0.0
  nom: !java.lang.Double 1.0
  rfc: !java.lang.Double 2.0
  npv: !java.lang.Double 0.0
  dit: !java.lang.Double 0.0
  lcom4: !java.lang.Double 1.0
  mmloc: !java.lang.Double 3.0
```

Apêndice D

Código de integração da Crab com a Jabuti

```
1   package br.crab.service;
2
3   import br.crab.exception.NoClassesAvailableException;
4   import br.crab.model.MetricEvaluation;
5   import br.jabuti.metrics.Metrics;
6   import br.jabuti.project.JabutiProject;
7
8   import java.util.*;
9
10  public class MetricsValuesCollectorServiceJabuti extends
11  MetricsValuesCollectorServiceAbstract {
12
13
14
15          private JabutiProject mainProject;
16
17          public MetricsValuesCollectorServiceJabuti(JabutiProject mainProject){
18                  this.mainProject = mainProject;
19          }
20
21          @Override
22          public void collectMetrics() throws NoClassesAvailableException {
23                  try{
24                          Metrics metrics = new Metrics(mainProject.getProgram());
25
26                          String[]classesName = mainProject.getProgram().getCodeClasses();
27
28                          for (String className : classesName) {
29                                  Map<String,Double> specificClassMetricsValues= new HashMap<String,Double>();
30                                  for (int i = 0; i < Metrics.metrics.length; i++) {
31                                          String metricName = (Metrics.metrics[i][0]).toUpperCase();
32
33                                          specificClassMetricsValues.put(metricName, (Double)metrics.getClassMetrics(className)[i]);
34                                  }
35                                  this.metricsValues.put(className, specificClassMetricsValues);
36                          }
37                  } catch (Exception e){
38                          throw new NoClassesAvailableException();
39                  }
40          }
41
42          @Override
43          public List<MetricEvaluation> getAvailableMetrics() {
44                  List<MetricEvaluation> toReturn = new ArrayList<MetricEvaluation>();
45                  for(int i = 0; i < Metrics.metrics.length; i++){
46                          MetricEvaluation metricEvaluation = new MetricEvaluation(Metrics.metrics[i][0]);// metric name
47                          metricEvaluation.setMetricDescription(Metrics.metrics[i][1]);// metric description
48                          toReturn.add(metricEvaluation);
49                  }
50                  Collections.sort(toReturn);
51                  return toReturn;
52          }
53
54          @Override
55          public br.crab.jabuti.Program getProgram() {
56                  br.crab.jabuti.Program program = null;
57                  try {
58                          program = new br.crab.jabuti.Program(mainProject.getProgram().getClassesHash());
59                  } catch (Exception e) { }
60                  return program;
61          }
62  }
```