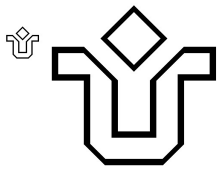

UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO
CENTRO DE CIÊNCIAS EXATAS E TECNOLOGIA
PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA

COdeMI - SOURCE CODE AS XMI. UMA REPRESENTAÇÃO DE CÓDIGO-FONTE PARA COLETA DE MÉTRICAS ESTRUTURAIS

João Paulo Oliveira dos Santos

Orientador
Márcio de Oliveira Barros

RIO DE JANEIRO, RJ - BRASIL
SETEMBRO DE 2009

COdeMI – SOURCE CODE AS XMI. UMA REPRESENTAÇÃO DE CÓDIGO-FONTE PARA COLETA DE MÉTRICAS ESTRUTURAIS

João Paulo Oliveira dos Santos

DISSERTAÇÃO SUBMETIDA AO CORPO DOCENTE DO PROGRAMA DE PÓS-GRADUAÇÃO EM INFORMÁTICA DA UNIVERSIDADE FEDERAL DO ESTADO DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM INFORMÁTICA.

Aprovada por:

_____

Prof. Márcio de Oliveira Barros, D.Sc. - UNIRIO

_____

Profa. Renata Mendes de Araujo, D.Sc. - UNIRIO

_____

Prof. Guilherme Horta Travassos, Ph.D. - UFRJ

RIO DE JANEIRO, RJ - BRASIL
SETEMBRO DE 2009

Santos, João Paulo Oliveira dos.
S237 COdeMI – Source code as XMI. Uma representação de código-fonte para coleta de métricas estruturais / João Paulo Oliveira dos Santos, 2009.
113f. + DVD

Orientador: Márcio de Oliveira Barros.
Dissertação (Mestrado em Informática) – Universidade Federal do Estado do Rio de Janeiro, Rio de Janeiro, 2009.

1. Sistemas de apoio a negócios. 2. Evolução de software. 3. Representações de código-fonte. 4. Coleta de métricas. I. Barros, Márcio de Oliveira. II. Universidade Federal do Estado do Rio de Janeiro (2003-). Centro de Ciências Exatas e Tecnologia. Curso de Mestrado em Informática. III. Título.

CDD – 005.5

Dedico este trabalho aos meus pais
Edson e Irene.

Agradecimentos

A Deus, pelas bençãos a mim concedidas durante toda minha vida.

Aos meus pais Edson e Irene pelo incentivo, apoio, amizade e amor incondicional dedicados durante todas as etapas desta jornada.

Aos amigos Dilermando Baptista Mello e Francisco Paulo Jardim Araújo, pelo incentivo acadêmico e pelas oportunidades oferecidas desde o curso de graduação.

Aos amigo(a)s Juliana, Claudia, Janaina, Clauciene, Marcos e Rúbio pelos momentos de descontração e lazer oferecidos.

Ao meu orientador Márcio Barros pelo excelente trabalho de orientação e auxílio por diversas vezes decisivo e norteador, que possibilitaram a conclusão desta dissertação.

Aos colegas que partilharam comigo problemas e alegrias do dia a dia e tantos outros que colaboraram de forma especial neste árduo e prazeroso caminho de aprendizagem, convivência e construção.

A Alessandra, secretária presente, atenciosa e competente, por sua paciência e dedicação aos nossos problemas administrativos e acadêmicos.

A todos os professores do Programa de Pós Graduação em Informática da UNIRIO, pelo convívio, pelo apoio e pela compreensão.

Resumo da Dissertação apresentada ao PPGI/UNIRIO como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

COdeMI - SOURCE CODE AS XMI. UMA REPRESENTAÇÃO DE CÓDIGO-FONTE PARA COLETA DE MÉTRICAS ESTRUTURAIS.

João Paulo Oliveira dos Santos

Setembro/2009

Orientador: Márcio de Oliveira Barros

Representações de software baseadas em XML possuem nível de abstração adequado para o processamento, análise e manipulação de código-fonte por ferramentas. Estas representações fornecem detalhes expressivos sobre a estrutura do software. A alta verbosidade de algumas representações e a exposição de detalhes do código-fonte em excesso dificultam a realização de estudos de evolução de softwares industriais. Neste trabalho, apresentamos a CodeMI, uma representação de código-fonte baseada em XML que utiliza a extensão do formato XMI para definir elementos, com ênfase na estrutura do código-fonte, que viabilizem a coleta de métricas sem revelar a dinâmica de execução do software.

Abstract of Dissertation presented to PPGI/UNIRIO as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

COdeMI - SOURCE CODE AS XMI. A REPRESENTATION OF SOURCE CODE FOR COLLECTION OF STRUCTURAL METRICS.

João Paulo Oliveira dos Santos

September/2009

Advisor: Márcio de Oliveira Barros

XML based software representations allow easier processing, analysis, and manipulation of source code by automated tools. These representations provide significant details about the structure of the software, but the verbosity of some representations may bring difficulties to code sharing and to the execution of software evolution studies in the industry. In this work, we present CodeMI, a XML-based source code representation that uses XMI extension to define elements that emphasize the structure of the source code, enabling the collection of structural metrics without publishing the dynamics of the software execution.

Sumário

Lista de Figuras

Lista de Tabelas

CAPÍTULO 1

INTRODUÇÃO

1.1. Motivação

A representação de código-fonte em formato texto é largamente utilizada por programadores para a codificação de algoritmos desde as primeiras linguagens de programação de computador até os dias de hoje. É também considerada como o formato universal para a construção de código-fonte, dadas as facilidades de manipulação e intercâmbio entre ferramentas, como editores de texto, sistemas de controle de versão, entre outras [Badros 2000]. No entanto, o formato texto não oferece a mesma versatilidade quando se deseja obter informações do software a partir do código-fonte.

A principal limitação da representação textual é a dificuldade de se obter, a partir de um processo automático, a estrutura do software, pois isto exige a construção de um analisador sintático (*parser*) específico para cada linguagem de programação. Por estrutura, entendemos os pacotes e classes que formam o programa, os atributos e métodos destas classes e os comandos de controle de fluxo que fazem parte da implementação destes métodos. Em geral, a análise sintática do código-fonte é desempenhada pelos compiladores e por uma pequena parte das ferramentas de Engenharia de Software. Ferramentas como LEX/YACC [Levine 1992] suportam o desenvolvimento dos analisadores sintáticos, porém a dependência deles implica em que cada ferramenta que envolva obtenção de informações de código-fonte implemente um *parser* próprio. Esta implementação, além de dispendiosa, envolve o conhecimento de teoria de compiladores e não é tarefa simples.

O recente crescimento do interesse por pesquisas em evolução de software, que tangem ao modo como o desenvolvimento e a manutenção de sistemas ocorrem ao longo do tempo, demanda a utilização de grandes repositórios de código-fonte para a coleta de informações sobre a forma como o software foi desenvolvido. A partir destes repositórios são extraídas métricas do software ao longo do tempo e são formadas séries temporais para serem analisadas.

1.2. Definição do Problema

A utilização de sistemas de software livre é bastante comum na realização de estudos de evolução de software, dada a facilidade de se obter o seu código-fonte a partir de repositórios públicos [Godfrey & Tu 2000] [Fisher et al. 2003] [Gall et al 1998] [Zimmermann & WeiBgerber 2004]. No entanto, muitas características referentes ao desenvolvimento de software livre diferem da forma adotada no desenvolvimento e manutenção de sistemas industriais [Raymond 1999].

O website SourceForge [SourceForge 2009] é considerado o maior repositório de projetos de software gratuito, livre e de código aberto. Este tipo de repositório é também conhecido como FLOSS (Free, Libre and Open Source Software). Ele fornece uma diversidade de dados, disponíveis na forma de repositórios de projetos de software livre, tornando-se uma atraente fonte de dados para pesquisas na área da Engenharia de Software. Contudo, diversas armadilhas podem ser encontradas neste tipo de repositório como: (a) projetos extintos; (b) revisões iniciais de sistemas podem ter sido descartadas; e (c) dados cruciais de um software podem ser hospedados fora do SourceForge, poluindo os dados recuperados. Estas questões comprometem diretamente os resultados da análise destes repositórios, tornando o SourceForge teoricamente perigoso para as pesquisas [Howison & Crowston 2004].

A facilidade de acesso aos itens de dados de qualquer projeto do SourceForge influencia pesquisadores a formularem suas teorias baseando-se somente na análise deste tipo de repositório [Howison & Crowston 2004]. Conseqüentemente, os repositórios de softwares industriais são comumente excluídos destas análises.

Por outro lado, os sistemas de informação utilizados nas empresas são, em sua maioria, protegidos por direitos autorais e/ou de propriedade intelectual. A proteção de software é um instrumento peculiar ao segmento empresarial/industrial para assegurar que segredos, normalmente alinhados à estratégia de condução de negócio da organização, não se tornem de domínio público.

Estas proteções, assim como a dificuldade de acesso ao código-fonte decorrente delas, inviabilizam a generalização das conclusões dos estudos realizados com base em software livre e dificultam a realização de qualquer tipo de estudo que tenha a necessidade de acesso direto ao código-fonte desenvolvido pelas empresas. Portanto, para a realização de pesquisas em evolução de softwares comerciais, é necessária a criação de um repositório de dados, onde a representação de código-fonte tenha um papel central, preservando as características necessárias para a realização de estudos de evolução de software, ao mesmo tempo em que esconda os detalhes do código-fonte, de modo a não infringir a propriedade intelectual das empresas que o desenvolveram. Do mesmo modo, estas representações de código-

fonte devem ser armazenadas para que sejam utilizadas futuramente sem a necessidade de um novo acesso ao código-fonte diretamente.

Com o objetivo de facilitar a análise da estrutura do código-fonte de sistemas por ferramentas automatizadas, surgiram as primeiras representações de código no formato XML (*eXtensible Markup Language*) [XML 2006]. Este formato, embora possa ser exposto em uma representação textual, é capaz de apresentar elementos e relações entre estes elementos de forma estruturada. Assim, uma representação de código-fonte em formato XML evidencia os elementos estruturais de um código fonte e viabiliza a extração de medidas do software. Dada a gama de ferramentas existentes para a realização de consultas, manipulação e transformação de informações em formato XML, a realização de análises sobre estas medidas se torna uma tarefa mais simples do que o processamento do código-fonte original.

Sendo assim, a representação XML do código fonte de um sistema traz uma série de benefícios: (a) estrutura o código de forma explícita, o que facilita sua manipulação; (b) possibilita criar consultas mais poderosas sobre os componentes do código-fonte, pois ao invés de uma pesquisa textual através de expressões regulares será realizada uma busca em marcadores onde se pode utilizar linguagens e ferramentas apropriadas; (c) possui representação extensível, o que facilita criar extensões no código, característica que o formato textual não permitia de forma simples; (d) possibilita fazer referência cruzada entre elementos de código, o que não é possível em representações textuais, pois os elementos são referenciados através de linhas e colunas; e (e) suporte amplo, uma vez que XML é suportado por várias plataformas [Mendonça et al 2004].

Embora existam diversas representações de código-fonte em formatos mais estruturados do que o formato textual das linguagens de programação (entre elas, algumas que utilizam o próprio formato XML), nenhuma das representações analisadas em um estudo comparativo se mostrou capaz de atender aos seguintes critérios: (a) ser uma representação genérica, ou seja, independente de linguagem de programação; (b) possuir granularidade adequada e em nível de pacote; (c) baixa verbosidade; e (d) baixa exposição do código-fonte. Estes critérios foram estabelecidos como essenciais à extração de métricas do código-fonte de sistemas industriais.

1.3. Objetivo

Visando criar uma representação de código-fonte independente de linguagem de programação, que não evidencie os segredos industriais refletidos na lógica de programação do software, que permita a extração de métricas estruturais (ou seja, métricas obtidas por meio dos elementos estruturais do código-fonte) e auxilie na realização de estudos de Evolução de Software propomos a CodeMI (*Source Code as XML Metadata Interchange*). *CodeMI é* uma representação de código-fonte em formato XML que utiliza o formato XMI v2.1[XMI 2007] para descrever elementos da estrutura de classes de um projeto de software orientado a objetos, como os pacotes, as classes, os atributos e os métodos do sistema. XMI é um formato padronizado pela OMG *(Object Management Group)* para a troca de informações sobre modelos UML [UML 2009]. Este formato é amplamente utilizado por ferramentas de modelagem baseadas em repositórios MOF *(Meta-Object Facility)* [MOF 2002].

Para descrever a estrutura da implementação dos métodos das classes que formam um programa foi utilizado o mecanismo de extensão do formato XMI. Por extensão, entende-se a introdução de novos elementos ao formato XMI padronizado, sem gerar incompatibilidade para as ferramentas que já utilizam o formato original. Assim, as ferramentas que conhecem as extensões poderão fazer uso destas informações, enquanto as demais ferramentas as ignorarão. Este mecanismo de extensão foi utilizado para representar estruturas de repetição, desvios condicionais e incondicionais, comandos e outros aspectos da estrutura do código-fonte que não são relevantes para os fins mais usuais do formato XMI (como a criação e compartilhamento de diagramas UML, por exemplo), mas devem ser considerados para a extração de métricas estruturais. Sendo o formato XMI um tipo específico de XML, será possível utilizar as mesmas ferramentas de análise e manipulação existentes sobre os dados obtidos na representação CodeMI para a análise e coleta de métricas, com a vantagem de poder analisar estruturas do código-fonte.

Adicionalmente foi idealizado um ambiente de suporte à realização de pesquisas em evolução de softwares comerciais, que utiliza representações de código-fonte na CodeMI para facilitar a aquisição e distribuição de informações históricas sobre sistemas de software.

1.4. Estrutura da Dissertação

Este documento está dividido em 6 capítulos. O capítulo 1 apresenta a introdução ao tema. Neste é apresentada a motivação para a realização deste

trabalho, a contextualização do problema de modo amplo, os principais objetivos e a delimitação do escopo de solução.

No capítulo 2 são apresentados trabalhos relacionados à representação de código-fonte em XML e um breve estudo comparativo entre estas representações.

No capítulo 3 é discutido o conceito de métrica estrutural de software através de sua definição e de exemplos de métricas que se enquadram neste conceito. É dada ênfase ao conjunto de métricas clássicas, orientadas a objetos, em função da possibilidade destas serem obtidas a partir da representação que será proposta no capítulo seguinte.

No capítulo 4 é apresentada a representação CodeMI (*Source Code as XML Metadata Interchange*) obtida por meio da extensão do formato XMI, visando estabelecer um formato para a extração de métricas estruturais a partir do código-fonte de sistemas desenvolvidos segundo o paradigma da orientação a objetos. Posteriormente, é realizada uma prova de conceito utilizando o sistema *Compiere*, onde abordamos o processo de extração de métricas estruturais deste software por meio da conversão de seu código-fonte para a representação CodeMI. Em seguida, serão discutidos e aplicados alguns métodos e ferramentas de coleta de métricas estruturais das suítes previamente apresentadas no capítulo 3.

No capítulo 5 é apresentada a proposta de um ambiente de suporte à realização de pesquisas sobre a evolução de softwares, focando, sobretudo softwares comerciais / industriais. São também apresentados alguns trabalhos realizados na área de evolução de software nas últimas décadas.

No capítulo 6 encerramos este trabalho, apresentando suas conclusões e sugestões de trabalhos futuros.

CAPÍTULO 2

REPRESENTAÇÕES DE CÓDIGO-FONTE EM XML

2.1. Considerações Iniciais

Com o objetivo de facilitar a análise da estrutura do código-fonte de sistemas por ferramentas automatizadas, surgiram as primeiras representações de código no formato XML (*eXtensible Markup Language*) [XML 2006]. Este formato, embora possa ser exposto em uma representação textual, é capaz de representar elementos e relações entre estes elementos de forma estruturada. Assim, uma representação de código-fonte em formato XML evidencia os elementos estruturais do código-fonte e viabiliza a extração de medidas baseadas nesta estrutura. Dada a gama de ferramentas existentes para a realização de consultas, manipulação e transformação de informações em formato XML, a realização de análises sobre estas medidas se torna uma tarefa mais simples do que o processamento do código-fonte original.

Sendo assim, a representação XML do código-fonte de um sistema traz uma série de benefícios: (a) apresenta a estrutura do código de forma explícita, o que facilita sua manipulação; (b) possibilita criar consultas mais poderosas sobre os componentes do código-fonte, pois não será mais feita uma pesquisa textual através de expressões regulares, mas uma busca em marcadores onde se podem utilizar linguagens e ferramentas apropriadas; (c) possui representação extensível, o que facilita criar extensões no código, característica que o formato textual não permitia; (d) possibilita fazer referência cruzada entre elementos de código, o que não é possível em representações textuais, pois os elementos são referenciados através de linhas e colunas; e (e) oferece suporte amplo por diversos tipos de ferramenta, uma vez que XML é suportado por várias plataformas [Mendonça et al 2004].

Os formatos de representação de código-fonte em XML existentes são classificados em representação específica de uma linguagem de programação e representação genérica. As seções 2.2 e 2.3 apresentam, respectivamente, exemplos de representações de código para estes formatos. A seção 2.4 fornece um estudo

comparativo entre estas representações. A seção 2.5 apresenta as conclusões deste capítulo.

2.2. Representações Específicas de Linguagem

As representações específicas de linguagem são assim denominadas por utilizarem marcadores para especificar estruturas e comandos restritos à linguagem de programação para a qual foram planejadas.

Utilizando as ferramentas de busca tradicionais da web, como Google, Google Acadêmico, Citeseer e Scopus, foi observado que apesar de existirem diversas linguagens de programação, foram encontradas representações em XML somente para as linguagens C++ e Java, sendo as representações para a linguagem Java encontradas em maior número.

Para o propósito deste trabalho restringimos nossa análise às representações específicas da linguagem Java, por esta apresentar uma gramática relativamente simples quando comparada a das demais linguagens e por ser a linguagem orientada a objetos de maior popularidade e utilização entre programadores, de acordo com resultados da pesquisa realizada mundialmente pelo LangPop (Programming Language Popularity) [LangPop 2009], como pode ser visto na figura 2.1.

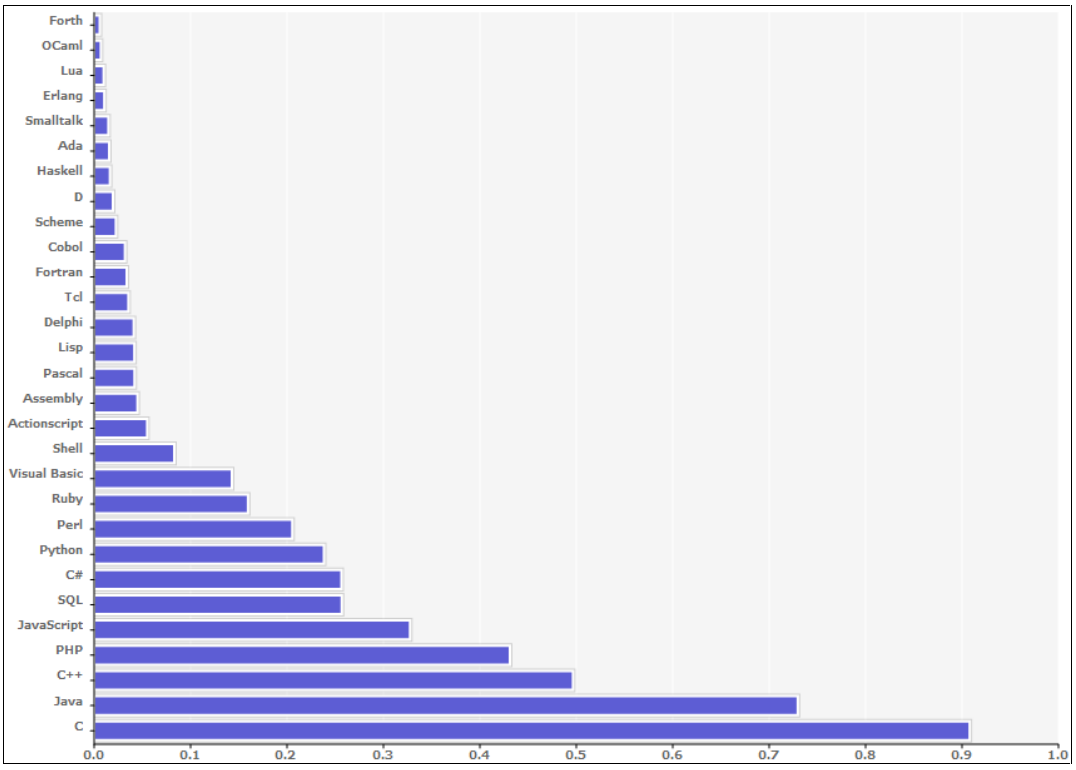

Figura 2.1. Popularidade das Linguagens de Programação

Entre as representações específicas da linguagem Java destacam-se a JavaML de Mamas e Kontogiannis [Mamas e Kontogiannis 2000], a JavaML de Greg Badros [Badros 2000], a XJava [Xjava 2008] e JavaML 2.0 de Ademar Aguiar [Aguiar 2004].

Nas próximas subseções serão abordadas as principais características de cada representação utilizando como exemplo o código fonte da classe *Funcionario* que pode ser observado na figura 2.2.

```
package funcionario;

public class Funcionario {

      private int salario;

      public void calculaSalario (int valorHora) {
            int salarioBase;
            // 22 dias de trabalho a 8 horas por dia
            salarioBase = 22*8*valorHora ;
            salario = salarioBase;
      }
}
```

Figura 2.2. Exemplo de Classe Java

2.2.1. JavaML de Mamas e Kontogiannis

Na JavaML de Mamas e Kontogiannis os elementos do código-fonte são mapeados para o formato XML através de um parser desenvolvido por meio da ferramenta Java Compiler Compiler ou JavaCC [JavaCC 2003] em conjunto com a gramática da linguagem Java versão 1.1. Este parser realiza a leitura da árvore sintática abstrata (*AST - Abstract Syntax Tree*) do código-fonte e adiciona elementos ao XML de acordo com as operações de redução detectadas [Mamas e Kontogiannis 2000].

Ao executar este parser sobre um sistema em Java serão gerados diversos arquivos XML, sendo um para cada classe do sistema. Tal fato, apesar de preservar a quantidade de arquivos, dificulta a extração de métricas que envolvam mais de uma classe, como métricas de pacote, pois será necessário analisar um conjunto de arquivos ao invés de um único local que centralize as informações.

Nesta representação, os elementos são especificados nos marcadores (*tags*) da representação XML e os valores destes elementos são representados nos atributos destes marcadores (*tag attributes*). O corpo do marcador (*tag body*) não é utilizado para armazenar informações do código-fonte. Esta representação é capaz de descrever os elementos essenciais da linguagem de programação, como classes, seus métodos e atributos, porém não armazena informações sobre a implementação

dos métodos, como número de linha e coluna dos comandos, início e fim de bloco, formatação e comentários. Outra desvantagem desta representação é a alta verbosidade, o que torna sua representação em XML bastante longa e pouco legível.

Como observado na figura 2.3, o elemento raiz desta representação é o *CompilationUnit*. Este representa o arquivo Java após sua transformação pelo *parser*. Este elemento possui três elementos filhos: (a) *PackageDeclaration,* que representa o pacote na qual a classe está inserida; (b) *ImportDeclaration,* que representa uma declaração de importação; e (c) *ClassDeclaration*, que representa toda declaração de uma classe.

O elemento *PackageDeclaration* é único para cada arquivo XML, uma vez que cada classe Java, por restrição da linguagem, pode pertencer somente a um único pacote.

O elemento *ImportDeclaration* repete-se tantas vezes quantas forem as declarações *import* no código-fonte Java.

O elemento *ClassDeclaration* pode conter vários sub-elementos, entre estes: (a) *FieldDeclaration,* que representa um atributo da classe; (b) *ConstructorDeclaration*, que representa um construtor da classe; e (c) *MethodDeclararion*, que representa a declaração de um método da classe.

Os demais elementos estão dispostos como sub-elementos da declaração do método ou construtor. São eles: (a) *FormalParameter,* que representa um parâmetro do método; (b) *LocalVariableDeclaration,* para representar uma declaração de variável local e (c) *Block*, para representar um bloco de código-fonte.

Para representar atributos, parâmetros e variáveis é utilizada uma estrutura de elementos comum, onde abaixo do marcador que representa um destes elementos existem mais dois marcadores: *Type* e *VariableDeclarator*. O primeiro possui um marcador interno que contém o tipo de dados do elemento, que pode ser *PrimitiveType*, se for um tipo primitivo, ou *Name*, se for uma *string* ou outra classe; o segundo possui um marcador interno chamado *VariableDeclaratorId*, que armazena o nome do elemento no código. O elemento *Type* também pode ser encontrado internamente ao elemento *ReturnType*, que indica o tipo de dado retornado por um método. Contudo, caso o método retorne *void*, ou seja, não retorne nada, o marcador *ReturnType* ficará vazio sem atributos e sub-elementos.

A parte suprimida da figura 2.3 compreende o marcador *ReturnType*, que neste exemplo não possui atributos e sub-elementos, pois o método retorna *void*. Também foi excluído um conjunto de elementos relacionados à multiplicação e atribuição de variáveis. Optamos por omiti-los com o intuito de reduzir o tamanho da figura 2.3 e

para destacar a alta verbosidade desta representação que para apenas estas duas ultimas operações utilizou um total de dezenove marcadores.

```
<CompilationUnit>
   <PackageDeclaration>
      <Name Identifier="funcionario" />
   </PackageDeclaration>
   <TypeDeclaration>
      <ClassDeclaration isAbstract="False" isFinal="False"
         isPublic="True">
         <UnmodifiedClassDeclaration Extends="False"
            Identifier="Funcionario">
            <ClassBody>
               <FieldDeclaration isFinal="False" isPrivate="True"
                  isProtected="False" isPublic="False"
                  isStatic="False" isTransient="False"
                  isVolatile="False">
                  <Type ArraySize="0">
                     <PrimitiveType Type="int" />
                  </Type>
                  <VariableDeclarator>
                     <VariableDeclaratorId ArraySize="0"
                        Identifier="salario" />
                   </VariableDeclarator>
               </FieldDeclaration>
               <MethodDeclaration isAbstract="False" isFinal="False"
                  isNative="False" isPrivate="False"
                  isProtected="False" isPublic="True"
                  isStatic="False" sSynchronized="False">
                  <ResultType />
                  <MethodDeclarator ArraySize="0"
                     Identifier="calculaSalario">
                     <FormalParameter isFinal="False">
                        <Type ArraySize="0">
                           <PrimitiveType Type="int" />
                        </Type>
                        <VariableDeclaratorId ArraySize="0"
                           Identifier="valorHora" />
                     </FormalParameter>
                  </MethodDeclarator>
                  <Block>
                     <LocalVariableDeclaration isFinal="False">
                        <Type ArraySize="0">
                           <PrimitiveType Type="int" />
                        </Type>
                        <VariableDeclarator>
                           <VariableDeclaratorId ArraySize="0"
                              Identifier="salarioBase" />
                        </VariableDeclarator>
                     </LocalVariableDeclaration>
                  </Block>
                  ...
               </MethodDeclaration>
            </ClassBody>
         </UnmodifiedClassDeclaration>
      </ClassDeclaration>
   </TypeDeclaration>
</CompilationUnit>
```

Figura 2.3. Representação em JavaML de Mamas e Kontogiannis da classe Funcionario

2.2.2. JavaML de Badros

A representação JavaML de Badros *(*ou *JavaML 1.0)* [Badros 2000] assemelha-se bastante à JavaML de Mamas e Kontogiannis, pois ambas preocupam-se em manter informações da AST (Abstract Syntax Tree) do código-fonte no formato XML e são obtidas por meio da utilização de um *parser*. Ambas as representações também

descrevem cada elemento do código através de marcadores, apresentando seus valores em atributos, sem utilizar o corpo dos marcadores. A representação JavaML de Badros, no entanto, fornece informações que podem ser obtidas na representação textual, mas que foram suprimidas na representação de Mamas e Kontogiannis, como o número da linha de código inicial e final de cada elemento, além de ser mais sucinta (menos verborrágica que a representação de Mamas e Kontogiannis). Além disso, utiliza uma quantidade menor de marcadores para representar os elementos do código-fonte, com nomes menores e intuitivos. Outro ponto positivo é que esta representação atribui um identificador (*id*) para cada método e variável, e uma referência ao identificador (idref) para todo relacionamento que utilize estas entidades, como invocação de métodos e atribuição de variáveis, subsidiando o cruzamento de informações para análises. Por outro lado, as desvantagens desta representação envolvem a não preservação de informações de formatação, linhas em branco e comentários.

Como observado na figura 2.4, o elemento raiz desta representação é o *java-source-program,* seguido pelo sub-elemento *java-class-file*. Este último representa o arquivo Java após sua transformação pelo *parser*. Este elemento possui três elementos filhos: (a) *package-decl,* que representa a declaração do pacote na qual a classe está inserida; (b) *import,* que representa uma declaração de importação; e (c) *class*, que representa toda declaração de uma classe.

```
<java-source-program>
   <java-class-file name="C:/Funcionario.java">
      <package-decl name="funcionario" />
      <class name="Funcionario" visibility="public" line="3" col="0"
         end-line="14" end-col="0">
         <superclass name="Object" />
         <field name="salario" visibility="private" line="5" col="8"
            end-line="5" end-col="27">
            <type name="int" primitive="true" />
         </field>
         <method name="calculaSalario" visibility="public" line="7"
            id="Funcionario_mth-18" col="5" end-line="13" end-col="5">
            <type name="void" primitive="true" />
            <formal-arguments>
               <formal-argument name="valorHora" id="Func_frm-16">
                  <type name="int" primitive="true" />
               </formal-argument>
            </formal-arguments>
            <block line="7" col="48" end-line="13" end-col="5">
               <local-variable name="salarioBase" id="Func_var-32">
                  <type name="int" primitive="true" />
               </local-variable>
               ...
            </block>
         </method>
      </class>
   </java-class-file>
</java-source-program>
```

Figura 2.4. Representação em JavaML de Badros da classe Funcionario

O elemento *package-decl* é único para cada arquivo XML, uma vez que cada classe Java, por restrição da linguagem, pode pertencer somente a um único pacote.

O elemento *import* repete-se tantas vezes quantas forem as declarações *import* no código-fonte Java.

O elemento *class* pode conter vários sub-elementos, entre estes: (a) *field,* que representa um atributo da classe; (b) *constructor,* que representa um construtor da classe; (c) *method*, que representa a declaração de um método da classe; (d) *superclass*, que refere-se à classe base através da qual a classe representada foi estendida.

O elemento *method* pode conter o sub-elemento *formal-arguments*, onde são especificados os parâmetros do método. Cada parâmetro é representado: (a) por um elemento *formal-argument*, que possui os atributos *name* e *id*, correspondentes ao nome e ao identificador do parâmetro, respectivamente; e (b) um sub-elemento *type* para especificar o tipo do parâmetro.

O elemento *localvariable,* que representa a declaração de variável local, pode ser observado como um sub-elemento direto de *method* ou em qualquer nível hierárquico abaixo deste.

O elemento *block* é utilizado para representar: (a) o bloco do código-fonte relativo à implementação do método; (b) sub-blocos de código-fonte, através do aninhamento de elementos *block*; ou (c) sub-blocos de estruturas de repetição ou desvios. Deste modo, os sub-elementos de um bloco de código-fonte variam sua disposição de acordo com a lógica do programa original.

Uma parte da representação da classe *Funcionario* foi suprimida para diminuir o tamanho da figura 2.4. A parte retirada reflete um conjunto de elementos relacionados à multiplicação e atribuição de variáveis, utilizando os seguintes elementos: (a) *binary-expr,* que é utilizado para a realização de operações binárias representadas no atributo *op*; e (b) *assignment-expr*, que representa uma operação definida pelo atributo *op*, neste caso, atribuição.

As estruturas de repetição são representadas pelo elemento *loop* e o seu tipo definido no atributo *kind,* seguidos de sub-elementos: (a) de controle que variam conforme o tipo da estrutura; e (b) sub-elementos que denotam o bloco que será executado repetidamente.

Para estruturas do tipo *for* tem-se os seguintes sub-elementos de controle : (a) *init*, que representa a inicialização do laço; (b) *test*, que representa o teste de continuidade de execução; e (c) *update*, que especifica o incremento da inicialização.

Para estruturas do tipo *while* é utilizado somente o sub-elemento de controle *test* para verificar se a condição de execução do laço é atendida.

As estruturas de desvio são representadas pelo elemento *if*. Este possui os sub-elementos: (a) *test*, que representa o teste condicional; (b) *true-case*, que representa o bloco de execução caso o teste condicional seja verdadeiro; e (c) *false-case*, que representa o bloco de execução caso o teste condicional seja falso.

O nome de cada elemento do código pode ser obtido no atributo *name* de seu próprio marcador. Tipos de dados de atributos, variáveis e parâmetros, bem como o valor de retorno de métodos, são sempre representados pelo atributo *name* do marcador *type*, internamente ao que representa o elemento correspondente.

Esta representação possui a vantagem de ser mais legível e menos verbosa que a JavaML de Mamas e Kontogiannis. No entanto, algumas desvantagens também são compartilhadas por ambos formatos. Com intuito de suprir essas deficiências, Aguiar *et al.* propuseram a versão 2 do padrão JavaML [Aguiar 2004], que inclui diversos tipos de informações que estavam ausentes na proposta original, como preservação de formatação, informações estruturais e definições de símbolos, referências e tipos. Em função disto o padrão se tornou mais verboso, gerando representações mais extensas.

2.2.3. JavaML 2.0 de Ademar Aguiar

A representação JavaML 2.0 de Ademar Aguiar [Aguiar 2004] apresenta diversas melhorias em relação à primeira versão apresentada por Badros. Entre estas, se destaca: (a) a preservação da formatação e o armazenamento dos comentários realizados pelos programadores; (b) maior riqueza de detalhes sobre informações estruturais, totalizando noventa marcadores distintos; e (c) informações semânticas relacionadas com a definição de símbolos, referências e tipos.

O principal objetivo desta representação é a preservação completa do código-fonte original e a manutenção das referências entre os símbolos do programa e suas respectivas definições. Estas referências, assim como na JavaML de Badros, são obtidas através do mecanismo padrão do XML utilizando o atributo *id* nas definições e o atributo *idref* nas referências [Aguiar 2004].

O processo de conversão do código-fonte para a JavaML 2.0 é realizado por meio de uma atualização realizada no compilador IBM Jikes v1.18 [IBM 1998].

Como pode ser observado na figura 2.5, o elemento raiz desta representação é o *java-source-program*. Este possui os seguintes elementos filho: (a) java-class-file, que representa o arquivo Java; (b) *type-dependences,* que representa as relações de dependência com outros arquivos ou bibliotecas; e (c) *java-source-code,* que representa o código-fonte da classe.

```
<java-source-program xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xsi:noNamespaceSchemaLocation="F:\javaML\xml\javaml-2.xsd">
   <java-class-file name="C:/Jikes/Funcionario.java">
      <package-decl name="funcionario" />
      <class name="Funcionario" id="Lfuncionario/Funcionario;"
         idkind="type" startToken="4" endToken="36">
         <modifiers>
            <modifier name="public" />
         </modifiers>
         <superclass name="Object" idref="Ljava/lang/Object;"
            idkind="type" startToken="4" endToken="36" />
         <field name="salario" id="Lfuncionario/Funcionario;salario"
            idkind="field" startToken="8" endToken="11">
            <modifiers>
               <modifier name="private" />
            </modifiers>
            <type name="int" primitive="true" />
         </field>
         <method name="calculaSalario"
            id="Lfuncionario/Funcionario;calculaSalario(I)V"
            idkind="method" startToken="12" endToken="35">
            <modifiers>
               <modifier name="public" />
            </modifiers>
            <type name="void" primitive="true" />
            <formal-arguments>
               <formal-argument name="valorHora"
                  id="Lfuncionario/Funcionario;arg16" idkind="formal"
                  startToken="16" endToken="17">
                  <type name="int" primitive="true" />
               </formal-argument>
            </formal-arguments>
            <block startToken="19" endToken="35">
               <local-variable-decl>
                  <type name="int" primitive="true" />
                  <local-variable name="salarioBase"
                     id="Lfuncionario/Funcionario;var35" idkind="local-variable"
                     startToken="20" endToken="22"></local-variable>
               </local-variable-decl>
               <assignment-expr op="=">
                  <lvalue>
                     <var-set name="salarioBase"
                        idref="Lfuncionario/Funcionario;var35"
                        idkind="local-variable" />
                  </lvalue>
                  <binary-expr op="*">
                     <binary-expr op="*">
                        <literal-number kind="integer" value="22" />
                        <literal-number kind="integer" value="8" />
                     </binary-expr>
                     <formal-ref name="valorHora"
                        idref="Lfuncionario/Funcionario;arg16" idkind="formal" />
                  </binary-expr>
               </assignment-expr>
               <assignment-expr op="=">
                  <lvalue>
                     <field-set name="salario"
                        idref="Lfuncionario/Funcionario;salario" idkind="field" />
                  </lvalue>
                  <var-ref name="salarioBase"
                     idref="Lfuncionario/Funcionario;var35" idkind="local-variable" />
               </assignment-expr>
            </block>
         </method>
      </class>
   </java-class-file>
   <type-dependences>
      <type-dependence filename="C:/Jikes/Funcionario.java"
         signature="Lfuncionario/Funcionario;">
         <type-ref signature="Ljava/lang/Object;" />
      </type-dependence>
      <type-dependence
         filename="C:\Arquivos de
         programas\Java\jre6\lib\rt.jar/java/lang(Object.class)"
         signature="Ljava/lang/Object;" />
   </type-dependences>
   <java-source-code>
```

```
      <codeline no="1">
         <token idx="1" afterEol="true" line="1" column="1"
            type="preprocessor" deprecated="false" lexeme="package" />
         <sp />
         <token idx="2" afterEol="false" line="1" column="9"
            type="normal" deprecated="false" lexeme="funcionario" />
         <token idx="3" afterEol="false" line="1" column="20"
            type="normal" deprecated="false" lexeme=";" />
      </codeline>
      ...
      <codeline no="8">
         <token idx="31" idref="Lfuncionario/Funcionario;salario"
            idkind="field" afterEol="true" line="8" column="1" type="normal"
            deprecated="false" lexeme="salario" />
         <sp />
         <token idx="32" afterEol="false" line="8" column="9"
            type="normal" deprecated="false" lexeme="=" />
         <sp />
         <token idx="33" idref="Lfuncionario/Funcionario;var35"
            idkind="local-variable" afterEol="false" line="8" column="11" type="normal"
            deprecated="false" lexeme="salarioBase" />
         <token idx="34" afterEol="false" line="8" column="22"
            type="normal" deprecated="false" lexeme=";" />
      </codeline>
      <codeline no="9">
         <token idx="35" id="Lfuncionario/Funcionario;var35"
            idkind="local-variable" afterEol="true" line="9" column="1" type="normal"
            deprecated="false" lexeme="}" />
      </codeline>
      ...
   </java-source-code>
</java-source-program>
```

Figura 2.5. Representação em JavaML 2.0 da classe Funcionario

O elemento *java-class-file* e seus sub-elementos possuem similaridade funcional com os elementos usados na representação da JavaML de Badros, conforme apresentado na seção 2.2.2.

O elemento *type-dependences* é utilizado para especificar as dependências com outras classes através do sub-elemento *type-dependence*. Este possui o atributo *filename*, que indica o nome do arquivo a qual é realizada a dependência. Esta dependência refere-se à cláusula *import* de um programa Java.

O elemento *java-source-code* é utilizado para especificar o código-fonte, suas linhas de código e tokens. As linhas de código são especificadas pelo elemento *codeline* e por sub-elementos *token*. Cada elemento *codeline* possui um atributo *no* que representa o número da linha de código originalmente implementada na classe Java e um conjunto de tokens. O elemento *token* possui um atributo *idx* que representa um índice numérico e um *id* referente ao tipo do token.

A JavaML 2.0 é, entre as representações apresentadas, a que possui a maior quantidade de marcadores e conseqüentemente a maior verbosidade. Embora uma grande parte destes marcadores tenha suas origens na também verbosa JavaML, diversos novos elementos foram adicionados nesta representação, sobretudo os sub-elementos de *type-dependences* e *java-source-code*. Como visto anteriormente, nestes elementos concentram-se as melhorias propostas por esta representação como

a especificação de dependências com demais classes e a descrição do código-fonte. A adição destes elementos e seus respectivos sub-elementos contribuem ainda mais para o aumento de sua verbosidade.

2.2.4. XJava

A representação XJava [XJava 2008] é obtida através da transformação do código-fonte por um *parser* Java desenvolvido através da ferramenta BeaultyJ [Xjava 2008]. Esta representação, diferentemente das anteriores, não representa a AST do código-fonte, não armazena informações estruturais, de formatação e espaços em branco. No entanto, ela também se baseia no formato XML e utiliza marcadores e atributos para representar pacotes, classes, métodos e atributos de forma simples e semelhante à representação JavaML de Badros. A principal vantagem da XJava é o fato desta apresentar os comentários feitos pelos programadores no código-fonte e também por apresentar baixa verbosidade, dada sua semelhança com JavaML de Badros.

Nesta representação, diferentemente das anteriores, todos os arquivos Java são armazenados em um único XML, organizado internamente por pacotes. Isto sugere que para sistemas com poucos arquivos esta estrutura não fornecerá obstáculos à sua análise e manipulação. Porém, para sistemas com centenas de arquivos, esta representação pode não ser a mais adequada, pois será gerado um documento XML extremamente grande, trazendo problemas de processamento para as ferramentas de consulta e manipulação. Em contrapartida, o fato de todo o sistema estar representado num único arquivo XML facilita a idealização das consultas sobre o código-fonte.

Como observado na figura 2.6, o elemento raiz desta representação é o *xjava*, que representa todo o sistema após sua transformação pelo *parser*. Este elemento possui três elementos filhos: (a) *package,* que representa o pacote na qual a classe está inserida; (b) *import,* que representa uma declaração de importação; e (c) *class*, que representa toda declaração de uma classe. Os demais elementos, que representam métodos, construtores e atributos da Xjava, possuem nomenclatura similar aos elementos da JavaML de Badros. Do mesmo modo, os elementos que representam os tipos de retorno de método, de dados, atributos e parâmetros também são especificados pelo marcador *type* e têm seu valor armazenado no atributo *name*.

```
<xjava>
   <package name="java">
      <class name="Funcionario" public="yes"
         unqualifiedName="Funcionario">
         <extends class="java.lang.Object" />
         <field name="Funcionario.salario" private="yes"
            unqualifiedName="salario">
            <type dimension="0" fullName="int" name="int"
               unqualifiedName="int" />
         </field>
         <method name="Funcionario.calculaSalario" public="yes"
            unqualifiedName="Funcionario">
            <type dimension="0" fullName="void" name="void"
               unqualifiedName="void" />
            <parameter name="valorHora">
               <type dimension="0" fullName="int" name="int"
                  unqualifiedName="int" />
            </parameter>
            <code>int sal; // 22 dias de trabalho a 8 horas por dia
               sal =
               22*8*valorHora ; setSalario(sal);
            </code>
         </method>
      </class>
   </package>
</xjava>
```

Figura 2.6. Representação em XJava da classe Funcionario

Devido ao esquema simplificado desta representação, algumas consultas ao XML podem ser facilitadas; outras, porém, tornam-se mais difíceis dada a principal desvantagem da XJava em não representar estruturadamente a implementação dos construtores e métodos, apenas transcrevendo o conteúdo do formato texto para o corpo do marcador *code*. Isto ocasiona a perda de informações importantes, como definições de variáveis, invocação de métodos e instanciação de outras classes. Para se obter tais informações, é necessário utilizar pesquisa em texto, o que vai de encontro aos benefícios oferecidos pela linguagem XML.

2.3. Representações Genéricas

As representações genéricas não se destinam a representar apenas uma linguagem de programação, mas características comuns a um conjunto de linguagens, como as linguagens de programação orientadas a objetos. Entre as representações genéricas de código-fonte destacam se a OOML de Mamas e Kontogiannis [Mamas e Kontogiannis 2000], GXL de Holt [Holt 2000] e srcML de Maletic [Maletic 2002].

2.3.1. OOML

A representação OOML foi proposta por Mamas e Kontogiannis [Mamas e Kontogiannis 2000] e, sendo uma representação genérica, apresenta características comuns utilizadas pelas linguagens Java e C++. Esta representação não especifica os elementos do código-fonte, ficando restrita somente a estrutura da classe, seus

métodos e atributos. É obtida através da transformação XSLT [XSLT 2001] que realiza o mapeamento de seus elementos a partir de um arquivo na representação JavaML, para código-fonte em Java, ou CppML, para código-fonte em C++.

Apesar de ser uma representação que serviria para muitas linguagens, sua principal desvantagem é a perda das características próprias de cada linguagem, já que nem todas possuem as mesmas estruturas. Outra desvantagem é o fato de não representar o código-fonte da implementação dos métodos.

2.3.2. GXL

A representação Graph Exchange Language - GXL, proposta por Holt [Holt 2000], é uma representação de estruturas de código em forma de grafos direcionados. As entidades são representadas como nós e as arestas simbolizam os relacionamentos entre elas. Foi originalmente projetada para ser um padrão de troca de artefatos de software em diferentes níveis de abstração. No entanto, estes artefatos possuem relação com os elementos representados pelo diagrama de classes da notação UML [UML 2009]. Devido a isto, não armazena informações relativas às estruturas do código-fonte da implementação dos métodos.

2.3.3. srcML

A representação srcML, proposta por Maletic [Maletic 2002], foi projetada para representar o código-fonte por completo, sem perda de informação. Para isto, adiciona marcadores ao código-fonte, preservando informações como espaços em branco, comentários e formatação do texto. Isto torna o código-fonte muito extenso e com abundância de informações não essenciais para sua manipulação.

2.4. Estudo Comparativo entre as Representações

A realização deste estudo comparativo foi motivada pela necessidade de escolher, entre as representações apresentadas na seção 2.2, a que atenda aos critérios estabelecidos como essenciais para a extração de métricas estruturais do código-fonte citados no capítulo 1. Estes critérios serão detalhadamente apresentados no capítulo 3.

As representações genéricas não foram incluídas neste estudo comparativo por apresentarem propósitos de utilização bastante específicos, conforme descrito na seção 2.3. Cada representação genérica foi projetada para uma determinada finalidade, não apresentando características comuns entre si que possam ser

utilizadas como critério comparativo. Portanto, neste estudo, foram consideradas somente as representações específicas da linguagem Java.

Para a realização deste estudo foram adotados alguns critérios comparativos obtidos a partir dos principais fatores de convergência ou divergência entre as representações específicas da linguagem Java. Neste sentido, temos:

- *Elementos essenciais*: indica se a representação é capaz de descrever os elementos essenciais da linguagem, como classe, métodos, atributos e suas iterações;
- *Armazenamento de informações textuais*: indica se a representação guarda informações textuais, como linha, coluna, início e fim de blocos e formatação;
- *Verbosidade*: este critério está diretamente relacionado à legibilidade da representação, pois em geral, quanto maior a quantidade de elementos utilizados, maior a verbosidade e menor a sua facilidade de leitura;
- *Preservação das linhas em branco*: indica se a representação mantém as linhas em branco do código-fonte;
- *Preservação dos comentários*: indica se a representação mantém as linhas de comentário do código-fonte. Apesar deste ser um subconjunto do critério: *armazenamento de informações estruturais*, optamos por analisá-lo separadamente pelo fato de algumas representações preservarem linhas de comentários e não armazenarem as demais informações estruturais;
- *Preservação da AST*: indica se a árvore sintática abstrata (*Abstract Syntax Tree - AST*) do código-fonte é mantida na representação, obedecendo a sequência original de comandos;
- *Exposição do código*: indica se a representação expõe o código-fonte, permitindo assim sua extração a partir de alguma ferramenta de manipulação XML;
- *Granularidade do Arquivo*: indica a granularidade máxima que a representação é capaz de fornecer para cada arquivo gerado.

A comparação entre as representações apresentadas na seção 2.2 (JavaML de Mamas e Kontogiannis, JavaML de Badros, XJava e JavaML 2.0 de Aguiar) segundo os critérios propostos (elementos essenciais, informações estruturais, verbosidade, linhas em branco, comentários, representa AST e exposição do código) pode ser observada na tabela 2.1. Nesta, verificou-se que todas as representações especificam os elementos essenciais da linguagem, fornecendo informações sobre classes, métodos, atributos e suas iterações.

Tabela 2.1. Comparativo entre representações específicas para Java

	JavaML M&K	JavaML 1.0	XJava	JavaML 2.0
Elementos essenciais	Sim	Sim	Sim	Sim
Informações textuais	Não	Não	Não	Sim
Verbosidade	Alta	Média	Baixa	Alta
Linhas em branco	Não	Não	Não	Sim
Comentários	Não	Não	Sim	Sim
Representa AST	Sim	Sim	Não	Sim
Exposição do código	Sim	Sim	Sim	Sim
Granularidade	Classe	Classe	Sistema	Classe

Em relação às informações textuais, foi observado que as representações JavaML de Mamas e Kontogiannis, JavaML de Badros e XJava não armazenam dados relativos a linha, coluna, início e fim de blocos, formatação e comentário. Estas informações foram observadas somente na representação JavaML 2.0 de Aguiar. Os dados de linha e coluna são vistos nos atributos *line* e *column* do atributo *token*, enquanto comentários são encontrados no atributo *comment* de diversos elementos, variando conforme o local onde o comentário foi digitado pelo programador.

Em relação à verbosidade das representações, destacamos a JavaML de Mamas e Kontogiannis e JavaML de Aguiar como as que possuem o maior número de marcadores para especificar seus elementos e, conseqüentemente, as mais verbosas deste estudo. A JavaML de Badros possui um conjunto reduzido de marcadores, quando comparada às anteriores, visto que nesta utiliza-se em média 5 marcadores para representar uma operação de atribuição, contra 19 marcadores para a mesma operação na JavaML de Mamas e Kontogiannis. Neste sentido, a JavaML foi classificada como sendo de média verbosidade. A XJava foi classificada como sendo de baixa verbosidade, por possuir um conjunto bastante reduzido de marcadores para representação de código-fonte, tendo em vista que todo o código-fonte é inserido sem representação estrutural no corpo do marcador *code*.

A baixa verbosidade de uma representação pode ser vista como um critério de qualidade da mesma quando se deseja analisar sistemas de larga escala. Quanto menor a quantidade de marcadores utilizados para representar uma determinada linguagem de programação, mais simples será a eleboração de consultas para extrair informações de sua estrutura.

Ainda relacionado a verbosidade, é fáctivel assumir que um arquivo em XML com proposta de representar o código-fonte parcial ou total de um sistema, deve ocupar um espaço em disco inferior ao do código-fonte original para que exista vantagens de se utilizar tal representação em estudos de Evolução de Software..

A preservação de linhas em branco no código-fonte foi detectada somente na representação JavaML 2.0. Neste caso, tem-se um marcador *codeline* para representar a linha de código-fonte, porém sem nenhum sub-elemento.

A preservação dos comentários realizados pelo programador foi observada somente nas representações XJava e JavaML 2.0. Como visto anteriormente, na XJava, todo o código-fonte fica inserido no marcador *code*. Sendo o comentário parte deste código, sua preservação ocorre de modo natural dentro deste mesmo marcador. Já na representação JavaML 2.0, os comentários são armazenados no atributo *comment* que pode pertencer a diversos marcadores, de acordo com o local onde o programador digitou o comentário.

A preservação da árvore sintática abstrata – AST foi observada na maioria das representações comparadas, exceto na XJava. A árvore sintática é a representação da estrutura sintática do código-fonte escrito em alguma linguagem de programação, neste caso Java. Cada nó desta árvore denota uma construção realizada no código-fonte. Portanto, a preservação da AST nas representações JavaML de Mamas e Kontogiannis, JavaML de Badros e JavaML 2.0 significa que cada marcador destas representações é oriundo dos nós de suas respectivas árvores sintáticas.

A exposição do código-fonte foi observada em todas as representações utilizadas no estudo comparativo. Isto significa que a partir de representação XML podemos reconstruir o seu código-fonte original. Para este processo de reconstrução são geralmente utilizadas ferramentas de transformações XSLT, facilitando uma possível reengenharia do sistema.

Em relação à granularidade do arquivo, foi observado que a maioria das representações analisadas gera um arquivo para cada classe. Isto dificulta a coleta de métricas baseadas em mais de uma classe, por exemplo, métricas de pacote. Por outro lado, o mesmo não ocorre na XJava, que fornece um único arquivo para todo o sistema. Caso o sistema seja extremamente grande, podem surgir problemas de processamento para as ferramentas de consulta e manipulação. Em contrapartida, as consultas a este arquivo podem ser facilmente elaboradas.

Após a análise comparativa das representações específicas da linguagem Java, destacamos a JavaML 2.0 por atender a totalidade dos critérios adotados, conforme mostrado na Tabela 2.1. No entanto, apesar da JavaML 2.0 possuir riqueza de detalhes sobre a estrutura da linguagem Java, sua expressiva verbosidade dificulta

a extração de métricas, sobretudo às relacionadas ao código-fonte implementado internamente ao corpo do método.

A dificuldade de se extrair medidas do código-fonte a partir das representações XML está diretamente relacionada à dificuldade de construir consultas ou transformações sobre arquivos com muitos marcadores. A realização de consultas exige que a ferramenta percorra toda a hierarquia de marcadores. Por isso, quanto maior a quantidade de marcadores, maior será a complexidade da consulta.

2.5. Conclusão

A principal desvantagem das representações específicas da linguagem Java é que a maioria permite que o código-fonte possa ser extraído a partir do XML, facilitando a engenharia reversa do software, que em diversas situações deve ser proibida, pois caracteriza violação de restrições de propriedade de software ou intelectual. Por outro lado, as representações com baixa verbosidade, como XJava, possuem a desvantagem da falta de informações estruturais da implementação dos métodos.

Outra desvantagem é a diversidade de marcadores e atributos utilizados pela JavaML 2.0 para representar a AST do código-fonte, que torna a tarefa de extrair métricas estruturais bastante árdua, dada a dificuldade de identificar quais marcadores serão utilizados na extração de determinada medida. Por exemplo, quais marcadores devem ser considerados quando desejamos extrair uma simples métrica, como o número de comandos em um método? Ou quando calcular a complexidade ciclomática [McCabe 1976] de um determinado método?

Deste modo, segundo os critérios apresentados e as representações analisadas, consideramos inviável a utilização de uma representação específica da linguagem Java para os propósitos deste estudo.

CAPÍTULO 3

MÉTRICAS ESTRUTURAIS DE SOFTWARE

3.1. Medição de Software

À medida que a engenharia de software amadurece, a medição de software passa a desempenhar um papel cada vez mais importante no entendimento e controle das práticas e produtos do desenvolvimento de software [Kitchenham et al 1995]. Os desenvolvedores medem características do software para tentar identificar se os requisitos estão consistentes e completos, se o projeto tem boa qualidade ou se o código está pronto para ser testado. Os gerentes de projetos medem atributos do processo e do produto para serem capazes de dizer quando o software estará pronto para entrega ou se o orçamento será ultrapassado. As equipes de manutenção de software podem usar a medição para avaliar as abordagens existentes, identificando aspectos positivos e negativos e mesmo alternativas para melhor estimar este tipo de serviço [Toffano et al 2005].

As principais motivações para se medir software estão concentradas em como: (a) entender e aperfeiçoar o processo de desenvolvimento; (b) melhorar a gerência de projetos e o relacionamento com clientes; (c) reduzir frustrações e pressões de cronograma; (d) gerenciar contratos de software; (e) indicar a qualidade de um produto de software (f) avaliar a produtividade do processo; (g) avaliar os benefícios (em termos de produtividade e qualidade) de novos métodos e ferramentas de engenharia de software; (h) avaliar o retorno de investimento de um projeto; (i) identificar as melhores práticas de desenvolvimento de software; (j) embasar solicitações de novas ferramentas e treinamento; (k) avaliar o impacto da variação de um ou mais atributos do produto ou do processo na qualidade e/ou produtividade; (l) formar uma *baseline* para estimativas; (m) melhorar a precisão das estimativas e; (n) oferecer dados qualitativos e quantitativos ao gerenciamento de desenvolvimento de software, de forma a realizar melhorias em todo o processo de desenvolvimento de software [Fenton & Pfleeger 1997] [Pressman 2006].

Em termos gerais, medição é o processo pelo qual números ou símbolos são designados a atributos de entidades do mundo real de forma a descrevê-los de acordo com regras claramente definidas [Fenton & Pfleeger 1997]. Portanto, a medição captura informações sobre atributos de entidades. Uma entidade é um objeto (como uma pessoa ou um quarto) ou um evento (como uma viagem ou o projeto de desenvolvimento de um software). Um atributo é uma característica ou propriedade de uma entidade. Exemplos de atributos são: a área de um quarto, o tempo de uma viagem ou o custo de um projeto de desenvolvimento de um software.

A primeira tarefa de qualquer atividade de medição é identificar as entidades e atributos que se deseja medir. Na Engenharia de Software, existem três classes de entidades: processos, produtos e recursos. Processos são coleções de atividades relacionadas ao desenvolvimento de software. Produtos são quaisquer artefatos que resultam de uma atividade do processo. Recursos são entidades requeridas para realizar uma atividade do processo [Kitchenham et al 1995].

Considerando as métricas de produto, que são o escopo desta dissertação, observamos uma ampla variedade de medidas encontradas na literatura. As áreas mais importantes de concentração destas métricas são: (a) métricas para o modelo de análise; (b) métricas para o modelo de projeto; (c) métricas para código-fonte; e (d) métricas de teste.

As métricas para código-fonte podem ser utilizadas para avaliar a complexidade, manutenibilidade e testabilidade, entre outras características do código-fonte [Pressman 2006]. Diversas métricas para código-fonte, como o número de linhas de código (*lines of code - LOC*) e a complexidade ciclomática de McCabe, não dependem do paradigma de programação.

No entanto, com o surgimento do paradigma da orientação a objetos, houve a necessidade de medir softwares desenvolvidos sob esta concepção. A programação orientada a objetos vem crescendo bastante desde a última década, sendo cada vez mais adotada pelas empresas de software e pelas comunidades de software livre. Diversas métricas para a análise de projeto e código orientados a objeto [Chidamber & Kemerer 1994] [Lorenz & Kidd 1994] foram propostas na literatura. Uma grande parte destas métricas pode ser obtidas a partir do código-fonte.

Classes, objetos, métodos, mensagens, variáveis de instância, variáveis de classe e herança são conceitos básicos da orientação a objetos. Assim, métricas orientadas a objetos são medidas de como estas construções são usadas e influenciam no processo de desenvolvimento.

Para a concepção deste trabalho, convencionamos chamar de *Métricas Estruturais* as métricas de produto, independente de paradigma de programação, que

podem ser obtidas através dos elementos estruturais do código-fonte. Por elemento estrutural do código-fonte são considerados: (a) os desvios condicionais e incondicionais; (b) as estruturas de repetição; (c) os comandos de manipulação de variáveis e de chamada de métodos; (d) as declarações de variáveis; e (e) os blocos de tratamento de exceções. Estes elementos são de fundamental importância no desenvolvimento de software, pois capturam a complexidade do fluxo de execução dos métodos. Em uma análise mais ampla, o mesmo entendimento pode ser aplicado às classes e aos pacotes.

Nas próximas seções, serão abordadas suítes de métricas orientadas a objeto, métricas clássicas de complexidade e demais métricas derivadas de nossas observações. No próximo capítulo, tais métricas serão analisadas quanto a viabilidade de serem extraídas a partir da representação de código-fonte CodeMI.

3.2. Suíte de Métricas de Lorenz & Kidd

Mark Lorenz e Jeff Kidd [Lorenz & Kidd 1994] propuseram um conjunto de métricas com intuito de mensurar o progresso e a qualidade de projetos de software orientado a objetos. Este conjunto foi dividido em duas categorias: métricas de sistema e métricas de projeto.

Métricas de sistema são usadas para predizer as necessidades gerenciais do sistema, tais como pessoal alocado para uma tarefa e esforço total para a construção do sistema. Também medem as mudanças no estado do projeto, o quanto tem sido feito e o quanto ainda é necessário fazer para concluí-lo.

Já as métricas de projeto medem o estado estático do projeto em um ponto particular no ciclo de desenvolvimento. Estas métricas focalizam algum aspecto do projeto mais intimamente e, portanto, são mais descritivas. Elas se preocupam com a qualidade do sistema que está sendo construído do ponto de vista do desenvolvedor.

Dentro de cada uma destas categorias, as métricas são subdivididas em grupos logicamente relacionados. As métricas de sistema foram agrupadas em: (a) tamanho da aplicação; (b) tamanho da equipe; e (c) cronograma. As métricas de projetos foram agrupadas em: (a) tamanho do método; (b) aspectos internos do método; (c) tamanho da classe; (d) herança; (e) aspectos internos da classe; (f) aspectos externos da classe; e (g) acoplamento de subsistema.

A tabela 3.1 contém as principais métricas apresentadas por Lorenz e Kidd de acordo com cada categoria. Uma parcela destas métricas foi selecionada para utilização nesta pesquisa, mais precisamente as métricas de projeto, dada sua aderência em expressar características de qualidade e aspectos do projeto utilizados em estudos de evolução de software. Os grupamentos de métricas de projeto serão descritos em maiores detalhes nas subseções a seguir.

Tabela 3.1. Suíte de métricas de Lorenz e Kidd

Métricas de Sistema	Métricas de Projeto
Tamanho da Aplicação Número de *Scripts* de Cenário Número de Classes Chave Número de Classes de Suporte Número de Subsistemas **Tamanho da Equipe** Pessoas-dia por Classe Classes por Desenvolvedor **Cronograma** Número de Iterações Número de Contratos Completados	**Tamanho do Método** Número de Mensagens enviadas (NOM) Linhas de Código (LOC) **Aspectos Internos do Método** Complexidade do Método (MCX) Mensagens Enviadas (SMS) **Tamanho da Classe** Número de Métodos Públicos de Instâncias (PIM) Número de Métodos de Instâncias (NIM) Número de Atributos de Instâncias (NIA) Número de Métodos de Classe (NCM) Número de Atributos de Classe (NCA) **Herança** Profundidade da Herança (HNL) Herança Múltipla (MUI) Número de Métodos Sobrescritos (NMO) Número de Métodos Herdados (NMI) Número de Métodos Adicionados (NMA) Índice de Especialização (SIX) **Aspectos Internos da Classe** Coesão (CCO) Uso Global (GUS) Uso de Instâncias de Variáveis (IVU) Parâmetros por Método (PPM) Funções Amigas (FFU) Código Orientado a Função (FOC) Linhas Comentadas por Método (CLM) Percentual de Métodos Comentados (PCM) Problemas por Classe (PRC) **Aspectos Externos da Classe** Acoplamento (CCP) Reuso (CRE) Número de Colaborações (NCO)

3.2.1. Tamanho do Método

O tamanho de um método pode ser numericamente medido de diversos modos, como o número de mensagens enviadas, número de comandos (*statements*) ou número de linhas de código.

O número de linhas de código, também conhecido por LOC (*lines of code*), é uma das métricas mais simples que pode ser obtida de um software. Entretanto, vários fatores, como linguagem e estilo de programação, podem gerar diferenças significativas no resultado da medição quando comparamos uma mesma funcionalidade programada em diferentes contextos.

Diferentes linguagens fornecem diferentes capacidades de representação funcional para o mesmo tamanho de código-fonte. Por exemplo, a funcionalidade em um LOC em Assembly não é a mesma de um LOC em Java.

O estilo do programador é outro fator que influencia a contagem dos LOC's. O modo como é realizada a indentação das linhas de código, a criação de novas linhas e a utilização de nomes longos para a declaração de variáveis, irão afetar esta medição.

Outros fatores que podem influenciar a contagem de LOC's estão relacionados à definição sobre o que deve ser considerado como LOC. Estas divergências giram em torno dos comentários, linhas em branco e/ou declarações que não são executadas e também por particularidades da própria linguagem (diretivas e comentários para ferramentas específicas). Os autores, em sua maioria, descartam estes tipos de linha, desconsiderando-as na contagem do LOC.

O tamanho médio do método é um indicador de qualidade de projetos orientados a objetos. Grandes números indicam uma alta probabilidade de o código-fonte ter sido implementado orientado a funções, enquanto números pequenos indicam uma alta probabilidade de o código-fonte ter sido implementado orientado a objetos [Lorenz & Kidd 1994]. Esta métrica é obtida dividindo-se o somatório das linhas de código-fonte dos métodos pelo total de métodos de uma classe. Os autores também destacam que para a linguagem C++, o valor ideal para esta medida é 18 (dezoito).

3.2.2. Aspectos Internos do Método

Este grupo de métricas visa medir características internas dos métodos das classes. Lorenz & Kidd destacam duas medidas: (a) complexidade do método (MCX) e (b) mensagens enviadas (SMS).

Diversos trabalhos foram realizados na área de complexidade de código-fonte [McCabe 1976] [Myers 1977] [Dreger 1989]. Em grande parte, estes trabalhos focam

no total de pontos de decisão do código-fonte de uma função, que são normalmente representados por if/then/else e outras construções similares.

Existem algumas características na programação orientada a objetos que limitam a utilização destas medidas de complexidade. Tais características são baseadas no conceito de que bons projetos de software orientado a objetos devem utilizar poucos comandos do tipo *if* e *case*, além de sustentar a prerrogativa de que métodos devem possuir menos de seis linhas de código-fonte [Lorenz & Kidd 1994].

Para o cálculo da complexidade dos métodos, os autores utilizam os valores apresentados na tabela 3.2 para totalizar as ocorrências dos eventos identificados em cada método.

Tabela 3.2. Complexidade do Método [Lorenz & Kidd 1994]

Eventos	Medida
Chamada a API	5.0
Associações	0.5
Operações aritméticas	2.0
Mensagem com parâmetros	3.0
Expressões Aninhadas	0.5
Parâmetros	0.3
Chamadas primitivas	7.0
Variáveis temporárias	0.5
Mensagem sem parâmetro	1.0

Outra medida deste grupamento é o total de *mensagens enviadas*, que está relacionada com o recurso de encadeamento de mensagens disponível em algumas linguagens como Smalltalk. Esta métrica relaciona as mensagens enviadas para manipuladores de exceção como indicador de qualidade do projeto, à medida que os erros são detectados e tratados.

3.2.3. Tamanho da Classe

O tamanho da classe pode ser numericamente medido de diversos modos. As métricas agrupadas nesta seção abordam tais modos de quantificação individual de uma classe.

Os métodos públicos são aqueles disponibilizados como serviços às demais classes. Estes serviços são o melhor modo de julgar a quantidade de trabalho realizado por uma classe [Lorenz & Kidd 1994]. Em contrapartida, os métodos *private* e *protected* são aqueles utilizados pelas classes para cumprir o trabalho dos serviços que foram disponibilizados como públicos.

O *número de métodos públicos de instância numa classe (PIM)* é uma boa medida da responsabilidade da classe [Lorenz & Kidd 1994]. Os autores apresentam o

valor 20 (vinte) como um número a ser seguido para esta medida, porém afirmam que na média este número é geralmente um pouco maior.

O *número de métodos de instância numa classe (NIM)* considera em sua contagem todos os métodos *public*, *protected* e *private* definidos para instâncias da classe.

O número de métodos em uma classe está relacionado com a parcela de serviços que está sendo executada pela mesma. Classes muito grandes podem estar fazendo um trabalho excessivamente maior do que deveriam, acumulando responsabilidades em vez de distribuí-las a quem realmente pertencem, tornando sua manutenção cada vez mais complexa. Classes pequenas tendem a ser mais reutilizáveis à medida que fornecem um conjunto coeso de serviços em vez de um conjunto misto de capacidades.

A existência de muitos métodos em uma classe é um alerta de que muita responsabilidade pode estar sendo alocada em um único tipo de objeto. Uma alternativa para detectar esta situação é utilizar a média de métodos de instância por classe. Esta métrica deve focar os métodos públicos, pois estes representam as responsabilidades que revelam o trabalho que a classe é capaz de realizar. Para a obtenção desta métrica, Lorenz e Kidd propuseram que a *média de métodos de instância por classe (ANIM)* será o resultado da divisão do total de métodos de instância pelo total de classes.

O *número de atributos de instância numa classe (NIA)* é considerado uma medida de seu tamanho. São considerados como atributos de instância os atributos *private* ou *protected* de escopo global disponíveis para as instâncias da classe. O fato da classe possuir muitos atributos de instância indica que ela possui muitos relacionamentos com outros objetos do sistema. Estes relacionamentos podem ser objetos simples, como *strings* ou *integers*, ou objetos complexos do domínio, como estoques ou contas.

Lorenz e Kidd destacam alguns fatores importantes a serem considerados durante a análise desta métrica. São eles: (a) a preocupação com o fato do objeto modelado pela classe possuir um número de atributos compatível com os atributos de objetos do mundo real, como forma de assegurar a qualidade do projeto; (b) objetos de dados são assim denominados por assumirem uma função passiva dentro do sistema, tendo sua existência justificada apenas pelo fato de armazenar e prover informações a outros objetos; (c) a necessidade de um objeto realmente armazenar um valor deve ser considerada em função da possibilidade deste valor ser calculado somente quando requisitado ou delegado a outro objeto; (d) classes de interface com usuário geralmente possuem grande quantidade de atributos de instância, pois lidam

com a manipulação de objetos do modelo através de dados impostados pelos usuários.

Ao longo do projeto, uma indicação da evolução do tamanho da classe é a média do número de atributos de instância. A existência de muitos atributos de instância, na média, indica a possibilidade de que a classe está realizando mais trabalho do que deveria. Conseqüentemente, a classe pode conter diversos relacionamentos com outros objetos do sistema. Para a obtenção desta métrica, Lorenz e Kidd propuseram que a *média de atributos de instância por classe (ANIA)* será o resultado da divisão do total de atributos de instância pelo total de classes.

As classes em si, são objetos que podem prover serviços e estados globais às suas instâncias. Isto faz sentido para manipular valores comuns e inicialização de instância, porém não deveria ser o modo principal de realização desta tarefa. O total de métodos disponíveis para a classe (e não para sua instância) afeta o tamanho da classe. Este total de métodos da classe é geralmente menor comparado ao número de métodos de instância [Lorenz & Kidd 1994].

O *número de métodos de classe numa classe (NCM)* pode indicar o nível de serviços comumente manipulado por todas as instâncias. Isto também pode indicar deficiências no projeto, que ocorrem quando as instâncias utilizam-se em maior escala dos serviços da classe ao invés dos serviços da própria instância. Uma indicação deste tipo de ocorrência é a abundância de lógicas condicionais na implementação destes métodos de classe.

Para a obtenção da *média dos métodos de classe por classe (ANCM)*, Lorenz e Kidd propuseram que esta será o resultado da divisão do total de métodos de classe pelo total de classes.

Atributos de classe são dados globais, fornecendo objetos comuns para todas as instâncias da classe. Em geral, tem-se um baixo *número de atributos de classe (NCA)* quando comparado ao total de seus atributos de instância. Os atributos de classe são freqüentemente utilizados para fornecer valores que são usados para definir o comportamento de todas as instâncias. Podem ser utilizados, por exemplo, na determinação de um valor único para uma determinada transação.

Para a obtenção da *média do número de atributos de classe por classe (ANCA)*, Lorenz e Kidd propuseram que esta será o resultado da divisão do total de atributos de classe pelo total de classes. Este resultado geralmente é um valor baixo, pois apesar de algumas classes possuírem maior quantidade de atributos de classe, geralmente os atributos de instância aparecem em maior proporção.

3.2.4. Herança

Este grupo de métricas aborda o relacionamento de herança entre superclasses e subclasses. A herança é um conceito importante no desenvolvimento de software orientado a objeto. As subclasses herdam o comportamento através dos métodos e estados de sua superclasse.

A *profundidade da herança (HNL)* é igual do nível de aninhamento da classe em relação à superclasse mais distante em sua árvore hierárquica. Segundo Lorenz e Kidd, esta métrica deve ter seu valor máximo aproximadamente igual a 6 (seis). Árvores hierárquicas muito profundas indicam problemas no projeto, à medida que podem ser criadas subclasses que não representam especializações de suas superclasses. Uma subclasse deve idealmente estender as funcionalidades de sua superclasse.

Algumas linguagens, como C++, permitem que uma classe herde o comportamento de múltiplas superclasses. Outras linguagens, como Java, suportam apenas a herança simples, a partir de única superclasse. Neste sentido, o total de superclasses imediatas define a medida de *herança múltipla (MUI)*. A indústria de software aceitou que a herança múltipla não é necessária em função das diversas maneiras de modelagem de seus negócios [Lorenz & Kidd 1994]. Além disso, a herança múltipla também não melhora a precisão na modelagem, ou seja, não apresenta vantagens na identificação de classes de objetos com comportamentos semelhantes aos do mundo real.

Na implementação das subclasses é possível definir métodos com os mesmos nomes dos métodos das suas respectivas superclasses. Esta prática é normalmente utilizada para sobrescrever métodos, desde que sua assinatura seja preservada na subclasse. Um grande *número de métodos sobrescritos (NMO)* pode indicar problemas no projeto do software, pois uma subclasse deve ser criada com intuito de especializar uma superclasse, estendendo seus serviços disponíveis através da criação de novos métodos e não somente reescrevendo os já existentes.

3.2.5. Aspectos Internos da Classe

Este conjunto de métricas descreve o projeto interno das classes visando identificar como são utilizados seus atributos de instância, quais referências externas são realizadas, entre outros. As principais métricas deste grupo são: coesão (CCO), uso global (GUS), parâmetros por método (PPM), funções amigas (FFU), código orientado a função (FOC), linhas de comentário por método (CLM) e percentual de métodos comentados (PCM).

A *coesão (CCO)* é uma medida que reflete a alocação lógica do comportamento da classe internamente no sistema. O inter-relacionamento entre os métodos da classe e o modo de utilização dos atributos pelos métodos denotam a coesão da classe. Isto auxilia a identificação de possíveis limites para dividir classes com baixa coesão em múltiplas classes.

O *uso global (GUS)* é uma medida que reflete os artefatos de uso global disponíveis a todos os objetos do sistema, promovendo um acoplamento desnecessário. Em geral, o uso global deve ser minimizado, pois tal prática indica um projeto pobremente orientado a objetos.

A utilização da passagem de parâmetro ao invés do uso de atributos de instância, em alguns casos, tem a ver com a forma como os relacionamentos entre objetos são mantidos. Em outros, tem a ver com o que é necessário ao código-fonte do método para execução de suas atividades. Para a obtenção desta métrica, Lorenz e Kidd propuseram que a média de *parâmetros por método (PPM)* será o resultado da divisão do total de parâmetros de métodos pelo total de métodos.

A medida de *funções amigas (FFU)* visa contabilizar no sistema o total de funções deste tipo. Uma função amiga é assim denominada por permitir a quebra do encapsulamento entre os objetos. Esta pratica é permitida em algumas linguagens de programação como C++, Delphi[1], entre outras.

Algumas linguagens, como C++, permitem ao desenvolvedor escrever código externamente aos objetos. Um exemplo óbvio é a rotina *main* em C++. Visando medir esta prática, Lorenz & Kidd propuseram a métrica *percentual de código orientado a função (FOC)*.

A *média de linhas de comentário por método (CLM)* e o *percentual de métodos comentados (PCM)* são métricas relacionadas às linhas de comentários que primam pelas convenções e definições do projeto, sem impacto direto na qualidade do produto.

3.2.6. Aspectos Externos da Classe

Este conjunto de métricas examina como a classe relaciona-se com outras classes, subsistemas, entre outros. As principais métricas deste conjunto são: *acoplamento entre classes (CCP)* e *reuso da classe (CRE)*.

[1] A linguagem Delphi possibilita a quebra do encapsulamento, permitindo que uma classe acesse os atributos de outra classe, desde que estas sejam declaradas na mesma Unit (arquivo .pas).

O *acoplamento (CCP)* visa identificar as conexões entre as classes. Dentro do possível tais conexões devem ser evitadas, pois estabelecem dependências entre as classes. Este grau de dependência pode ser medido através do número de classes colaboradoras e a parcela de colaboração que é fornecida.

A métrica de *reuso (CRE)* está relacionada à quantidade de linhas de código que são reutilizadas em outras classes. Entre os modos de reutilização, os autores destacam: (a) caixa branca, através de copiar/colar partes do código-fonte dos componentes, aproveitando-os em novas implementações; e (b) caixa preta, que reutiliza o código-fonte a partir da definição de uma interface para sua execução sem examinar como foi realizada sua implementação.

3.3. Suíte de Métricas de Chidamber & Kemerer

Chidamber & Kemerer [Chidamber & Kemerer 1994] propuseram um conjunto de métricas, também conhecido por métricas CK, com intuito de subsidiar melhorias nos processos da área de desenvolvimento de software. Este conjunto é composto por seis métricas, das quais serão utilizadas quatro em nosso estudo, dado o alinhamento destas ao conceito de métricas estruturais apresentado no início deste capítulo.

3.3.1. Métodos Ponderados por Classe (WMC)

Esta métrica expressa a complexidade de uma classe individualmente. É conhecida originalmente por WMC (*Weighted Methods per Class*) e pode ser obtida através do somatório da complexidade de todos os métodos por classe, onde o número e a complexidade de métodos envolvidos são um prognóstico de quanto tempo e esforço são requeridos para desenvolver e manter uma classe. Dado que classes com grande quantidade de métodos tornam a aplicação mais específica e limitam sua possibilidade de reutilização, esta métrica pode ser utilizada como medida da reusabilidade da classe.

Sua fórmula pode ser definida como: *WMC = C(M1) + C(M2) + ... + C(Mn)*, onde *C(Mx)* é o valor da complexidade do método x. Alguns exemplos de medidas de complexidade de métodos que podem ser utilizadas no cálculo da WMC incluem a complexidade ciclomática [McCabe 1976] e as métricas *software science* propostas por Halstead [Halstead 1977].

3.3.2. Profundidade da Árvore de Herança (DIT)

Esta métrica define a complexidade do grafo de herança de uma classe como sendo o número total de seus ancestrais, até que seja obtida a raiz da classe. É normalmente relacionada com a reusabilidade, pressupondo que classes que são mais profundas na árvore de herança especializam funcionalidades de suas classes ancestrais, tornando-se mais complexas, específicas e menos reutilizáveis.

Na figura 3.1, a classe *Hashtable* tem nível dois e a classe *Dictionary* tem nível um em relação à raiz da árvore de herança em Java (*Object*). O número de filhos de uma classe é uma métrica que conta as subclasses imediatas de uma classe. Pela figura 3.1, o número de filhos conhecidos da classe *Dictionary* é um.

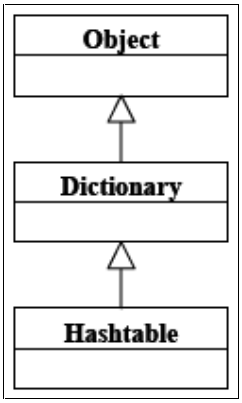

Figura 3.1. Superclasses de HashTable

3.3.3. Acoplamento entre Objetos de Classe (CBO)

Esta métrica, também conhecida como CBO (*Coupling Between Objects*), representa o número de classes com as quais uma dada classe está acoplada. Uma classe está acoplada a outra se a primeira usa métodos, atributos de instância e/ou atributos de classe definidas na segunda. Classes que possuem um alto acoplamento estão mais sujeitas a falhas do que classes de baixo acoplamento. Do mesmo modo, classes com acoplamento excessivo são prejudiciais à modularidade do projeto e comprometem sua reutilização.

3.3.4. Falta de Coesão nos Métodos (LCOM)

No paradigma da orientação a objetos, a coesão é a medida de quão interligados estão os métodos de uma classe em relação ao compartilhamento de atributos de instância. A métrica falta de coesão da classe é obtida pelo número de pares de métodos que compartilham atributos de instâncias, menos o número de pares de métodos que não compartilham nenhum atributo de instância.

3.3.5. Número de Filhos (NOC)

Esta métrica, também conhecida como NOC (*Number of Children*), determina o número de subclasses imediatas subordinadas a uma classe na hierarquia de classes. É uma medida relacionada à noção de extensão de propriedades, que determina como as subclasses herdam as funcionalidades (métodos) da classe pai.

3.3.6. Resposta para uma Classe (RFC)

Esta métrica, também conhecida como RFC (*Response For a Class*), determina o número de métodos que podem potencialmente ser executados, em resposta a uma mensagem recebida por um objeto de uma classe.

A execução de muitos métodos em resposta a uma mensagem recebida de um objeto torna a realização de testes e depuração da classe mais complexas, passando a exigir um maior nível de entendimento por parte da equipe de testes [Chidamber & Kemerer 1994].

3.4. Complexidade Ciclomática e Variações

A complexidade ciclomática, inicialmente proposta por McCabe, e algumas variações de sua abordagem em função de mudanças de paradigma ou anomalias específicas de um determinado tipo de código-fonte serão apresentas como métricas passíveis de serem obtidas a partir da representação estrutural do código-fonte.

3.4.1. Complexidade Ciclomática de McCabe

O número de complexidade ciclomática de McCabe [McCabe 1976] é uma métrica desenvolvida para avaliar a facilidade de entendimento, manutenção e teste de um sistema. É baseada na teoria clássica de número ciclomático de um grafo, ou seja, o número de regiões do grafo. Em um software, esta métrica é definida pelas decisões existentes no fluxo de execução. Em outra análise, esta medida expressa o número mínimo de casos de teste a serem construídos para assegurar que o software seja inteiramente testado.

Para a realização do cálculo da complexidade ciclomática, a partir do grafo de fluxo, é utilizada a fórmula: V(G) = E – N + 2, onde E é o numero de arestas e N o total de nós. Aplicando-se esta fórmula ao grafo de fluxo do algoritmo de busca binária apresentado na figura 3.2, teremos que V(G) = 11 – 9 + 2 = 4, dado que na figura 3.2 é

possível observar a existência de 9 (nove) nós e 11 (onze) arestas. Efetuando esta soma obtem-se um resultado igual a 4 (quatro) para a complexidade ciclomática do grafo apresentado.

De outro modo, a medida de complexidade ciclomática não necessita de grafos de fluxos para ser obtida. McCabe também mostrou que V(G) é igual a um (1) somado ao número de decisões do programa. Deste modo, o V(G) pode ser facilmente calculado por uma simples inspeção no programa.

Observando o código-fonte da busca binária apresentado na figura 3.2, notamos a existência de três instruções de decisão, sendo um *while* e dois *if's*. Logo, ao aplicar a fórmula apresentada acima, teremos: V(G) = 1 + 3 = 4.

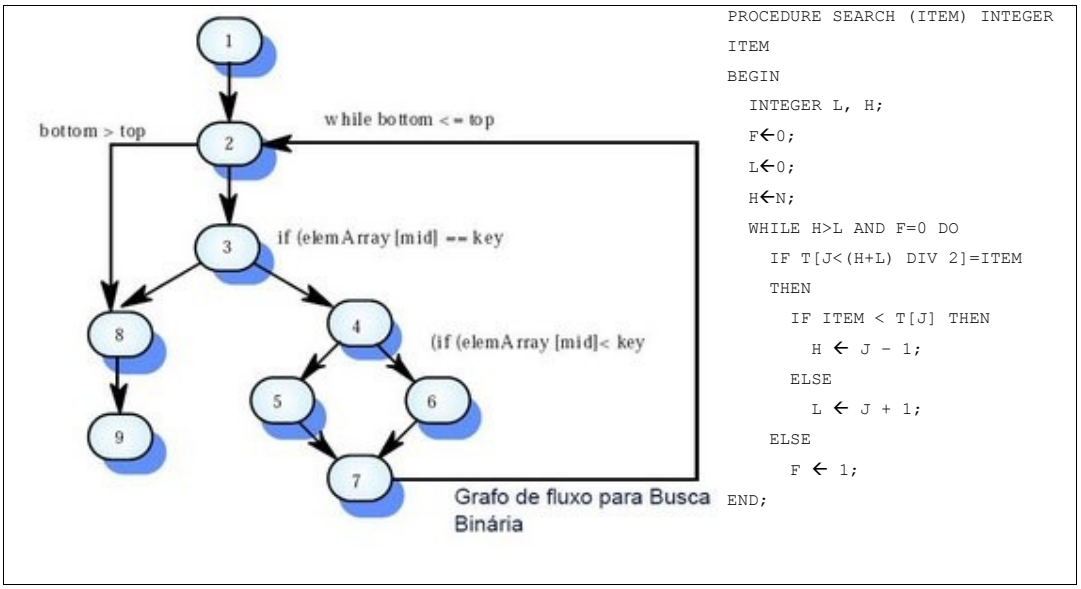

Figura 3.2. Grafo de fluxo e Código-fonte da Busca Binária [Adaptado de McCabe 1976]

3.4.2. Complexidade Ciclomática de Myers

Outra medida de complexidade, proposta por Myers [Myers 1977], apresenta uma solução para anomalias detectadas no modo de calcular a complexidade ciclomática proposta por McCabe [McCabe 1976] tomando por base a estrutura de decisão do programa ou o grafo de fluxo.

A anomalia detectada por Myers surge quando existem dois modos de se desenhar o grafo de fluxo ou de se contar decisões. Por exemplo, para o seguinte fragmento de um programa: IF (A=0) AND (B=1) THEN... poderemos desenhar dois grafos de fluxo, conforme a figura 3.3:

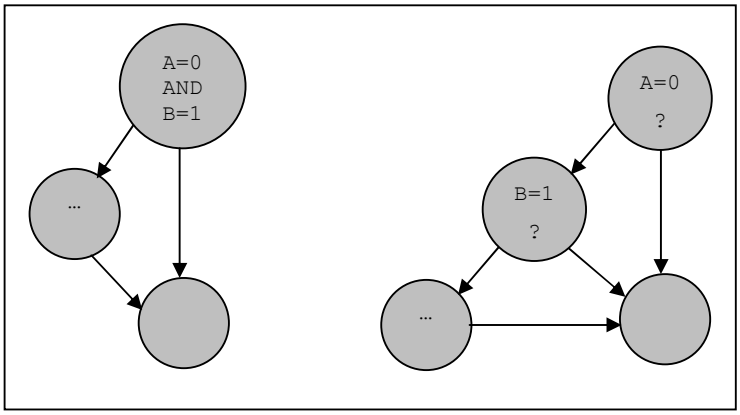

Figura 3.3. Dupla possibilidade de grafo de fluxo [Adaptado de Myers 1977]

A figura a esquerda possui V(G) = 2, enquanto a direita possui V(G) = 3. Embora os dois grafos representem o mesmo código-fonte, o modo como foi desenhado na figura 3.3 sugere a dupla interpretação e divergências nos valores de complexidade ciclomática. Isto porque na contagem de decisões, a primeira abordagem totaliza o IF inteiramente como um (1) e a segunda totaliza o número de condições, que neste mesmo IF, são duas (2).

A questão da divergência no modo de contagem das decisões já era reconhecida por McCabe, que recomenda a abordagem à direita da figura 3.3, ou seja, contar individualmente cada condição no cálculo do V(G). No entanto, segundo Myers, nenhuma das duas formas apresentadas é capaz de evitar freqüentes anomalias.

A anomalia detectada por Myers pode ser percebida através da comparação dos trechos de código-fonte apresentados acima, na figura 3.4. Desprezando-se momentaneamente o cálculo da V(G) e analisando somente os trechos de código-fonte A, B e C, Myers postula a premissa de que o trecho B fornece maior complexidade que o trecho A, e o trecho C fornece maior complexidade que o trecho B. Sendo assim, por este ponto de vista, podemos dizer que a complexidade ordenada destes trechos de código-fonte é A<B<C.

```
A:  IF (X=0) THEN ...
             ELSE ...

B:  IF (X=0) & (Y>1) THEN ...
                     ELSE ...

C:  IF (X=0) THEN
                  IF (Y>1) THEN ...
                           ELSE ...
             ELSE ...
```

Figura 3.4. Anomalia detectada por Myers [Myers 1977]

Ao calcularmos o V(G) através da contagem do total de decisões (mais um) teríamos V(A) = 2 (dois), V(B) = 2 (dois) e V(C) = 3 (três). Comparando estes valores com a premissa de Myers temos que 2<2<3 é falso. Do mesmo modo, ao calcularmos V(G) através da contagem individual das condições (mais um), teríamos V(A) = 2 (dois), V(B) = 3 (três) e V(C) = 3 (três). Novamente 2<3<3 é falso, indicando divergência com a premissa V(A)<V(B)<V(C).

O problema é que tanto o número de comandos de decisão quanto o número de condições contribuem para a complexidade. Então, ambos devem ser considerados neste cálculo. Isto pode ser realizado e a anomalia removida através do cálculo do V(G) como um intervalo de complexidade, onde o limite inferior é definido como o número de comandos de decisão mais um, enquanto o limite superior é definido como o número de condições individuais mais um.

Ao calcular o V(G) para os trechos de código-fonte apresentados na figura 3.4, teremos V(A) = 2:2, V(B) = 2:3 e V(C) = 3:3, o que satisfaz a relação intuitiva assumida como premissa por Myers para a realização de seu estudo.

3.5. Variantes de Métricas Estruturais

As métricas aqui propostas foram idealizadas tomando por base as suítes de métricas apresentadas nas seções anteriores. As idéias concebidas nas suítes foram expandidas em componentes de maior granularidade na arquitetura do software, como as classes e os pacotes.

3.5.1. Tamanho da Classe

Esta métrica segue a mesma fundamentação teórica da métrica *Tamanho do Método*, apresentada por Lorenz e Kidd [Lorenz & Kidd 1994]. No entanto, devemos considerar

que a classe é um conjunto de métodos. Esta medida poderá ser calculada a partir do somatório do tamanho de todos os métodos de uma classe do software.

3.5.2. Tamanho Médio da Classe

Esta métrica segue a mesma fundamentação teórica da métrica *Tamanho Médio do Método*, já apresentada por Lorenz e Kidd [Lorenz & Kidd 1994]. No entanto, devemos considerar que as classes estão normalmente agrupadas em subsistemas, comumente chamados pacotes. Esta métrica é obtida dividindo-se o somatório do tamanho das classes de um pacote pelo total de classes do pacote.

3.5.3. Tamanho do Subsistema

Esta métrica segue a mesma fundamentação teórica da métrica *Tamanho do Método*, apresentada por Lorenz e Kidd [Lorenz & Kidd 1994]. No entanto, devemos considerar que a classe é um conjunto de métodos e que, numa visão mais ampla, um subsistema (ou pacote) é um conjunto de classes. Esta medida poderá ser calculada a partir do somatório do tamanho de todas as classes do subsistema.

3.5.4. Complexidade Ciclomática da Classe

Esta métrica segue a mesma fundamentação teórica da métrica *Complexidade Ciclomática*, apresentada por McCabe [McCabe 1977], estendendo-a ao nível das classes. Pode ser calculada através do somatório da *Complexidade Ciclomática* de todos os métodos da classe.

3.5.5. Complexidade Ciclomática do Subsistema

Esta métrica pode ser obtida através do somatório da *Complexidade Ciclomática da Classe* de todas as classes que compõem o subsistema.

3.5.6. Complexidade Ciclomática Média das Classes do Subsistema

Esta métrica pode ser obtida a partir da *Complexidade Ciclomática do Subsistema* dividido pelo seu total de classes.

3.5.7. Classes Ponderadas por Subsistemas

Esta métrica expressa a complexidade de um subsistema ou pacote individualmente. É obtida através do somatório da complexidade de todas as classes por pacote, onde

o número e a complexidade das classes envolvidas são um prognóstico de quanto tempo e esforço são requeridos para desenvolver e manter um pacote. Esta medida é uma expansão da visão em nível de método para uma visão mais ampla, em nível de classe, da métrica: *métodos ponderados por classe (WMC)* de Chidamber & Kemerer. Sua fórmula pode ser definida como: C*PS = C(C1) + C(C2) + ... + C(Cn)*, onde *C(Cx)* é o valor da complexidade da classe x.

3.6. Considerações Finais

Neste capítulo foram apresentadas diversas métricas que se enquadram no conceito estabelecido de métricas estruturais de software. Entre estas, destacamos as suítes de métricas de Lorenz & Kidd, Chidamber & Kemerer e algumas medidas de Complexidade Ciclomática, que foram discutidas no âmbito de seus impactos no projeto de software. Tais métricas foram selecionadas por serem bastante referenciadas na literatura, em função de sua relevância e utilização em medições de software para a realização de estudos de evolução de software. Foram também propostas outras métricas estruturais a partir de pequenas variações no contexto inicialmente proposto pelos autores, porém, mantendo a natureza da medida original.

No próximo capítulo será apresentada a representação CodeMI, que será o formato utilizado para a coleta de métricas estruturais apresentadas neste capítulo.

CAPÍTULO 4

A REPRESENTAÇÃO COdeMI

4.1. Considerações Iniciais

Neste capítulo será apresentada a representação CodeMI (*Source Code as XML Metadata Interchange*), obtida por meio da extensão do formato XMI. A CodeMI visa estabelecer um formato para representar os elementos que formam a estrutura do código-fonte dos métodos para viabilizar a extração de métricas estruturais a partir do código-fonte de sistemas desenvolvidos segundo o paradigma da orientação a objetos.

Na seção 4.2 apresentaremos o formato XMI, suas principais aplicações, suas relações com o formato XML e com a notação UML. Na seção 4.3 apresentamos a representação CodeMI através da descrição de seus marcadores e atributos, suas respectivas funcionalidades e um exemplo de código-fonte convertido para esta representação. Na seção 4.4 foram abordadas questões relativas a engenharia reversa e reengenharia a partir da representação CodeMI. Na seção 4.5 foram descritos os mecanismos de consulta para a coleta de métricas estruturais, como as apresentadas no capítulo 3. Na seção 4.6 foi realizada uma prova de conceito com uma versão do sistema Compiere convertida para CodeMI, de onde foi possível coletar algumas métricas estruturais. Terminando este capítulo, a seção 4.7 apresenta as conclusões a respeito da viabilidade de utilização da representação proposta.

4.2. O Formato XMI

O formato XMI *(XML Metadata Interchange)* é, segundo a OMG *(Object Management Group)*, o formato para representação, troca e compartilhamento de modelos de sistemas orientados a objeto. Este formato representa uma tentativa de solucionar o problema de interoperabilidade entre ferramentas de modelagem, repositórios de meta-dados e outras ferramentas de desenvolvimento.

O principal objetivo do formato XMI é facilitar a troca de meta-dados entre ferramentas de modelagem baseadas na UML e repositórios de meta-dados baseados

no MOF. O termo *meta-dados* é usado para definir qualquer dado que, de alguma forma, descreve uma informação. Outro objetivo do formato XMI é definir uma notação padronizada para os esquemas conceituais do MOF *(Meta Object Facility)* [MOF 2002], que é um padrão utilizado para definir, manipular e integrar meta-modelos para o desenvolvimento de software orientado a objetos.

O MOF não é utilizado para descrever gramáticas para linguagens, mas sim para descrever a estrutura de objetos que podem ser representados em uma linguagem [Matula 2005].

O OMG normalmente se refere ao MOF como uma arquitetura em quatro camadas, conforme mostra a figura 4.1. No quarto nível (M3) encontra-se o MOF. No terceiro nível (M2) estão as linguagens descritas a partir do MOF. A UML é uma dessas linguagens. Outras linguagens incluem Common Warehouse Metamodel (CWM) [CWM 2005], Java [NetBeans 2008], entre outros. O CWM, embora tenha sido originalmente definido para atuar na área de banco de dados, inclui também um metamodelo de XML baseado no MOF.

Nível	Descrição
M3	MOF
M2	Metamodel UML (por exemplo, o elemento “Classe”)
M1	Elemento UML (por exemplo, a classe “Hóspede”)
M0	Instâncias de elementos UML (por exemplo, o hospede “João”)

Figura 4.1. Arquitetura de meta-dados do MOF.

No nível M1 encontram-se os modelos, como por exemplo, a classe “Hóspede” de um modelo de sistema de reserva de quartos em hotéis. Já no nível M0 têm-se as instâncias desses modelos, como por exemplo, o hóspede “João”. Ambos os exemplos podem ser observados na figura 4.1. Vale ressaltar que nesta arquitetura cada nível descreve objetos que podem ser instanciados por seus níveis inferiores.

Em linhas gerais, o formato XMI especifica como devem ser gerados documentos XML para a representação de modelos compostos pelos elementos definidos no MOF. A especificação define regras de produção de definição de tipos de documentos (DTD) para a geração de DTD XML e regras de produção de documentos XML para a geração de meta-dados num formato compatível com XML.

Nas ferramentas de desenvolvimento de sistemas estão concentradas as maiores utilizações do formato XMI, dentre as possibilidades apresentadas na figura 4.2. Um exemplo desta utilização são as ferramentas de geração de código-fonte a partir de modelos XMI. Para citar exemplos mais amplos, o formato XMI também pode ser utilizado por ferramentas de apoio a modelagem de projetos de bancos de dados geográficos baseados em extensões do modelo UML, como o ArgoCASEGEO [Lisboa Filho et al. 1999]. No contexto Web, este formato também pode ser utilizado como base para a troca de informações. Para a visualização destas informações são aplicadas transformações XSLT, gerando como saída páginas no formato HTML [Khun 2001].

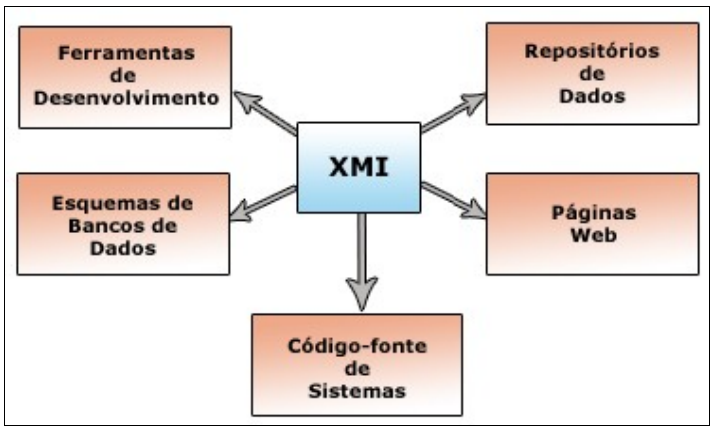

Figura 4.2. Uso do formato XMI.

Existem diversas versões de XMI disponíveis: 1.0, 1.1, 1.2, 2.0 e 2.1. Entretanto, não há compatibilidade entre as versões 1.x e 2.x, devido a modificações significativas de sintaxe, como o desaparecimento dos elementos *header* e *content* existentes na versão 1.x [XMI 2007]. Para o propósito desta pesquisa, optamos por utilizar a versão 2.1, por ser a última versão disponível na presente data.

Através do XMI são integrados três padrões da indústria de software: (a) o XML, da W3C [XML 2006]; (b) a UML [UML 2009]; e (c) o MOF [MOF 2002], ambos da OMG. A integração destes três padrões dentro do XMI unifica as tecnologias de meta-dados e modelagem da W3C e da OMG, permitindo que desenvolvedores de sistemas compartilhem modelos de objetos e outros meta-dados, pois a extensão das informações que podem ser trocadas entre duas ferramentas é limitada ao quanto desta informação pode ser compreendida por ambas as ferramentas. Caso compartilhem do mesmo modelo de meta-dados, todo o conteúdo transferido entre elas poderá ser compreendido e utilizado.

4.2.1. XMI e o Modelo UML

A Unified Modeling Language (UML) é conhecida por ser uma linguagem de diagramação ou notação para especificar, visualizar, construir e documentar modelos de sistemas de software orientados a objeto [UML 2009]. Tem suas origens na compilação das melhores práticas de engenharia, que provaram ter sucesso na modelagem de sistemas grandes e complexos. Sucedeu aos conceitos de Booch'93 [Booch 1993], OMT-2 (Object Modeling Technique) de Rumbaugh [Rumbaugh 1997] e OOSE (Object Oriented Software Engineering) de Jacobson [Jacobson 1992], fundindo-os numa única linguagem de modelagem comum e largamente utilizada.

A UML não é um método de desenvolvimento, pois não fornece uma seqüência de atividades a serem seguidas e também não descreve como desenhar o sistema. Porém, auxilia na visualização do projeto e da comunicação entre objetos. A UML é controlada pelo OMG (Object Management Group) e é um padrão da indústria para descrever graficamente um software.

Na UML 2.0 existem treze tipos de diagramas, divididos em três categorias: seis tipos de diagramas para representar a estrutura da aplicação, três tipos de diagramas para representar o comportamento da aplicação e quatro diagramas para representar diferentes aspectos das interações entre objetos que implementam a aplicação. Estes diagramas podem ser organizados hierarquicamente de acordo com a figura 4.3.

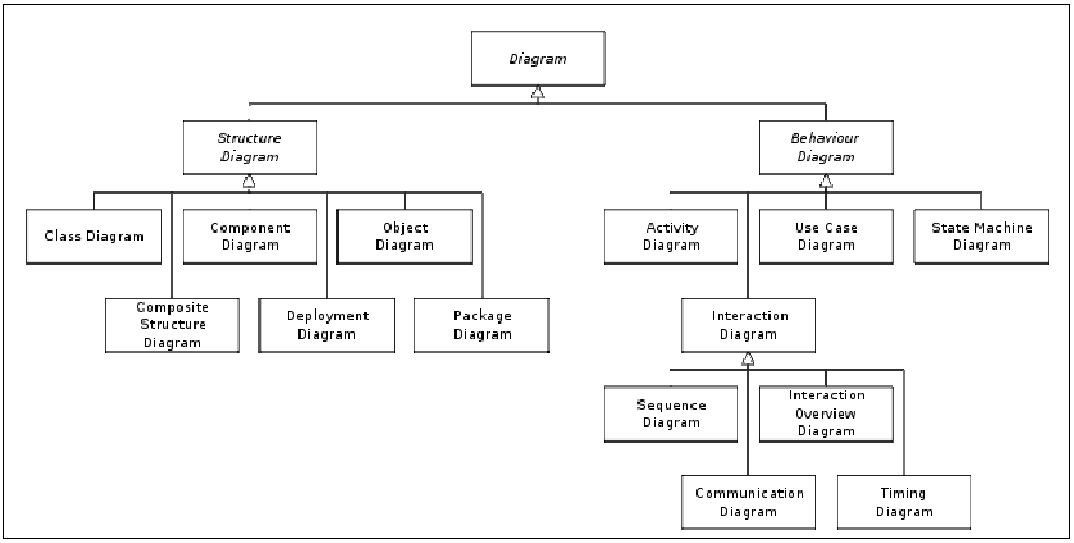

Figura 4.3. Hierarquia de diagramas UML [UML 2009]

Os diagramas estruturais são utilizados para visualizar, especificar, construir e documentar aspectos estáticos de um sistema. São eles: diagrama de classes, de pacotes, de objetos, de componentes, de implantação e de estrutura composta.

Os diagramas comportamentais são utilizados para descrever o sistema computacional modelado quando de sua execução. São eles: diagrama de casos de uso, de estados, de atividades, de visão geral da interação, de seqüência, de comunicação e de temporização. Os quatro últimos são conhecidos como diagramas de interação.

Todos os elementos utilizados na confecção destes diagramas estão definidos no meta-modelo UML (M2). De acordo com a hierarquia de modelos do MOF, qualquer modelo UML (M1) é considerado uma instância do seu meta-modelo. Tendo em vista que o formato XMI é utilizado para representar modelos baseados no MOF em formato XML, e que o modelo UML é uma instância deste meta-meta-modelo, temos que o formato XMI também é capaz de representar elementos do modelo UML. É importante destacar que a estreita relação entre um modelo UML e o formato XMI ocorre em nível de meta-modelos UML e não de seus diagramas.

Para ilustrar um exemplo da utilização de XMI, foi criado um diagrama de classes na ferramenta Case BoUml [BOUML 2002] contendo somente a classe *CompiereService*, apresentada na Figura 4.4. Esta classe foi obtida a partir do sistema *Compiere* [Compiere 2009], que é um sistema de informação que oferece diversos recursos para a gestão de empresas de forma integrada.

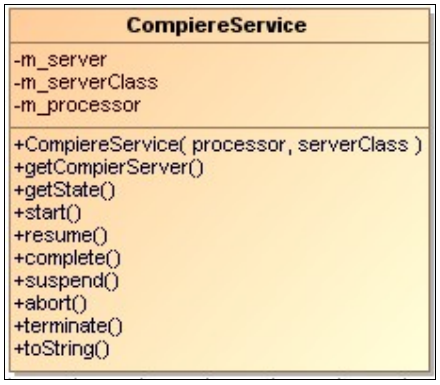

Figura 4.4. Classe *CompiereService* em UML.

Na figura 4.5 temos a representação da classe *CompiereService* no formato XMI v2.1, que foi obtida através da funcionalidade de exportação da ferramenta BoUML [BOUML 2002]. Nesta representação podem ser observados os elementos *packagedElement*, com atributo *xmi:type="uml:Package"* para representar o pacote

org.compiere.process. O elemento *packagedElement* também pode ser visto com o atributo *xmi:type="uml:Class"*, para representar a classe *CompiereService.* Os elementos *ownedAttribute* e *ownedOperation* são utilizados para representar, respectivamente, atributos e métodos da classe. Estes elementos são provenientes da representação XMI originalmente proposta pela OMG.

```
<?xml version="1.0" encoding="UTF-8"?>
<xmi:XMI xmi:version="2.1"    xmlns:uml="http://schema.omg.org/spec/UML/2.1"
       xmlns:xmi="http://schema.omg.org/spec/XMI/2.1">
<uml:Model xmi:type="uml:Model" xmi:id="modelo" name="XmiJavaParser">
<packagedElement xmi:type="uml:Package" xmi:id="PARSER_120" name="org.compiere.process">
       <packagedElement xmi:type="uml:Class" name="CompiereService" xmi:id="PARSER_130"
       visibility="public" isAbstract="???" >
              <ownedAttribute xmi:type="uml:Property" name="m_server"
              xmi:id="PARSER_140" visibility="private" />
              <ownedAttribute xmi:type="uml:Property" name="m_serverClass"
              xmi:id="PARSER_150" visibility="private" />
              <ownedAttribute xmi:type="uml:Property" name="m_processor"
              xmi:id="PARSER_160" visibility="private" />
              <ownedOperation xmi:type="uml:Operation" name="CompiereService"
              xmi:id="PARSER_170" visibility="public" isAbstract="???">
              <ownedParameter xmi:type="uml:Parameter" name="processor"
              xmi:id="PARSER_180" direction="in" />
              <ownedParameter xmi:type="uml:Parameter" name="serverClass"
              xmi:id="PARSER_190" direction="in" />
              </ownedOperation>
              <ownedOperation xmi:type="uml:Operation" name="getCompierServer"
              xmi:id="PARSER_200" visibility="public" isAbstract="???">
              </ownedOperation>
               ...
              <ownedOperation xmi:type="uml:Operation" name="toString"
              xmi:id="PARSER_280" visibility="public" isAbstract="???">
              </ownedOperation>
       </packagedElement>
</packagedElement>
</uml:Model>
</xmi:XMI>
```

Figura 4.5. Classe *CompiereService* em XMI.

O trecho suprimido da figura 4.5 descreve os demais métodos da classe *CompiereService*, através do marcador *ownedOperation*, de modo análogo aos métodos apresentados (*getCompierServer* e *toString*).

Ainda na figura 4.5, é possível observar que na representação XMI da classe *CompiereService* não há indício algum que referencie o fato deste elemento ter sido originado a partir da exportação de um diagrama de classes. De fato, há somente marcadores que especificam elementos do modelo e não diagramas.

O XMI gerado através de uma ferramenta CASE pode ser utilizado em outras ferramentas já existentes, que utilizem a linguagem de modelagem UML e sejam capazes de importar documentos XMI. Durante este trabalho, o XMI criado pela ferramenta BoUML foi testado em outras ferramentas como, por exemplo, Magic Draw [MagicDraw 2009].

A utilização do formato XMI para representar estruturas de software orientado a objetos possui a limitação de não conter elementos em sua concepção original para representar o código-fonte implementado no interior dos métodos (intramétodo) das classes. Para esta finalidade propomos a CodeMI, apresentada na próxima seção.

4.3. A Representação CodeMI

Com o objetivo de estabelecer um formato para viabilizar a extração de métricas estruturais do código-fonte e auxiliar na realização de alguns estudos de evolução de software, apresentamos a CodeMI (*Source Code as XML Metadata Interchange*). Suas principais características são:

a) é uma representação genérica, pois não se destina a representar apenas uma linguagem de programação, mas características comuns ao conjunto de linguagens de programação orientadas a objetos;
b) permite a geração de um arquivo para cada pacote do sistema, facilitando a obtenção de métricas de classes e de pacotes, tendo em vista que todas as suas classes estarão representadas no mesmo arquivo, o que facilita a realização de consultas;
c) baixa verbosidade em função da utilização de um conjunto enxuto de marcadores para representar o código-fonte relativo à implementação dos métodos (*intramétodo*), facilitando sua leitura, consulta e manipulação por ferramentas;
d) baixa exposição, pois não revela detalhes da implementação do código-fonte do sistema, inviabilizando a obtenção do código-fonte a partir desta representação. Isto assegura a não violação de eventuais restrições de propriedade intelectual, normalmente encontradas em softwares industriais. Isto, por conseguinte, viabiliza a colaboração acadêmico-industrial, permitindo que as empresas desenvolvedoras de software colaborem com iniciativas de pesquisa focadas em evolução de software industrial.

A representação CodeMI foi assim designada por introduzir ao formato XMI elementos para representar a implementação do código-fonte dos métodos, utilizando para isto o mecanismo de extensão deste formato. Os novos elementos adicionados visam representar a estrutura do código-fonte sem fornecer detalhes sobre sua dinâmica de execução.

Os marcadores pré-existentes do formato XMI, que especificam os elementos do modelo UML passíveis de utilização em diagramas de classes, como declarações de pacote, classes, métodos, atributos e elementos de relacionamento, como

associações, dependências e generalizações, continuam sendo utilizados em conjunto com os elementos especificados como extensão na representação CodeMI.

Na tabela 4.1 podemos visualizar os principais elementos adotados pela representação CodeMI. O marcador *if* é utilizado para representar um desvio condicional do tipo *if/then*. Neste, pode ser encontrado o atributo *conditions*, utilizado para armazenar o número de expressões condicionais que fazem parte da decisão do desvio. Este marcador também pode apresentar o sub-elemento *else*, que representa o bloco de código-fonte que será executado caso a expressão do desvio condicional seja *falsa.* Para descrever os blocos de código-fonte selecionados pelo resultado da expressão condicional pode ser utilizado qualquer tipo de elemento da representação CodeMI.

Tabela 4.1. Elementos da representação CodeMI

Marcadores	Atributos	Descrição
<if> </if>	conditions	Representa o desvio condicional *if*. O atributo *conditions* armazena o total de expressões condicionais que fazem parte da decisão do desvio condicional.
<else> </else>		Representa o *else* do *if*, aplicado no corpo do marcador *if*.
<switch> </switch>		Representa a estrutura de controle *switch*, ou seja, um aninhamento de expressões condicionais.
<case> </case>		Representa estrutura de controle *case*, uma das opções que podem ser resultado de um *switch*
<break>		Representa o comando *break*.
<for> </for>	conditions	Representa a estrutura de repetição *for*. O atributo *conditions* armazena o total de expressões condicionais que determinam o encerramento do loop.
<while> </while>	conditions	Representa a estrutura de repetição *while*. O atributo conditions armazena o total de expressões condicionais que determinam o encerramento do loop.
<try> </try>		Representa a estrutura de tratamento de exceção *try*.
<catch> </catch>		Representa a estrutura de tratamento de exceção *catch*
<statement>		Representa um comando.
<localvar>	type	Representa uma declaração de variável. O atributo *type* armazena o tipo da variável.
<return>		Representa o retorno do método.

O elemento *switch* é utilizado para representar a estrutura de controle semelhante a vários desvios condicionais *if/then* seguidos. Esta estrutura também é

considerada um desvio condicional, sendo normalmente utilizada para testar valores de variáveis enumeradas. Cada opção de execução desta estrutura é definida pelo sub-elemento *case*. Este representa a estrutura case, que normalmente tem seu término de execução definido pelo marcador *break*. No entanto, este marcador também pode ser utilizado para representar o término de execução de estruturas de repetição.

Os marcadores destinados a representar as estruturas de repetição são: (a) *for*, que representa um loop tradicional das linguagens de programação, indicando que seus sub-elementos pertencem a repetição que será executada uma quantidade de vezes predefinida; e (b) *while*, que representa uma estrutura que será executada enquanto uma condição de repetição for satisfeita. Em ambos, podem ser encontrado o atributo *conditions*, utilizado para armazenar o total de expressões condicionais, que determinam o encerramento do loop.

O elemento *try* é utilizado para delimitar um bloco de código-fonte potencialmente gerador de exceções de software, enquanto o elemento *catch* realiza a captura e tratamento de exceções.

O elemento *statement* é utilizado para representar comandos de manipulação de variáveis, como atribuições e operações aritméticas. Este elemento também é utilizado para representar uma chamada de método. Já o resultado do processamento dos métodos e seus pontos de retorno são representados pelo elemento *return*.

O elemento *localvar* é utilizado para representar uma declaração de variável local ao método. Neste marcador, o atributo *type* especifica o tipo da variável através do nome de um tipo primitivo ou de uma classe.

A utilização deste conjunto reduzido de marcadores pela CodeMI, para representar a estrutura do código-fonte de sistemas desenvolvidos segundo o paradigma da orientação a objetos, favorece a criação de uma representação com alta legibilidade e, conseqüentemente, baixa verbosidade.

Com intuito de verificar a aderência da CodeMI à classificação de representação genérica, bem como verificar a compatibilidade de seus marcadores com as principais linguagens de programação orientada a objetos, selecionamos algumas das linguagens OO de maior popularidade (Java, PHP e C++), segundo critérios adotados pelo LangPop [LangPop 2009], e realizamos a correspondência entre os comandos que representam a estrutura destas linguagens e os marcadores da CodeMI, como pode ser observado na tabela 4.2.

Tabela 4.2. Equivalência entre comandos e marcadores

Marcadores	Comandos JAVA	Comandos PHP	Comandos C++
<if> </if>	if	if	if
<else> </else>	else	else	else
<switch> </switch>	switch	switch	switch
<case> </case>	case	case	case
<break>	break	break	break
<for> </for>	for	for foreach	for
<while> </while>	while	while do while	while
<try> </try>	try	try	try
<catch> </catch>	catch	catch	catch
<statement>	atribuições chamada de método, operações aritméticas, entre outros.	atribuições chamada de método, operações aritméticas, entre outros.	atribuições chamada de método, operações aritméticas, entre outros.
<localvar>	Declaração de variável	Declaração de variável	Declaração de variável
<return>	return	return	return

Dentre as linguagens selecionadas, podemos observar que todos os marcadores da representação CodeMI possuem equivalência com os comandos das linguagens Java, Php e C++.

Para demonstrar a conversão de um código-fonte Java para CodeMI utilizamos o exemplo da implementação do método *terminate* da classe *CompiereService,* exibido na figura 4.6. Este método foi escolhido para o nosso exemplo por ser um trecho de código-fonte curto, mas que apresenta algumas estruturas normalmente utilizadas por programadores durante a codificação, como desvios condicionais e tratamento de exceções.

O processo de conversão do código-fonte para a representação CodeMI deve ser realizado por um *parser*, capaz de: (a) ler a árvore sintática abstrata (AST) do código-fonte dos métodos; (b) identificar os elementos estruturais na AST e convertê-los para um dos marcadores da CodeMI, descartando detalhes implementados pelo programador; e (c) gerar, ao final deste processo, um arquivo XMI para cada pacote. Deste modo, cada arquivo XMI armazenará somente informações sobre as classes do pacote analisado.

```
public boolean terminate()
{
       if (super.terminate())
       {
              if (m_server != null && m_server.isAlive())
              {
                     try
                     {
                            m_server.interrupt();
                     }
                     catch (Exception e)
                     {
                     }
              }
              log.info("terminate - done");
              return true;
       }
       return false;
}
```

Figura 4.6. Exemplo do método *terminate* em Java.

Para a realização deste estudo, foi desenvolvido um *parser* capaz de converter código-fonte em Java para a representação CodeMI. Para isto, foi utilizada a ferramenta JRefactory [Seguin 2000] para percorrer a AST *(Abstract Syntax Tree)* do código dos métodos. A JRefactory é uma ferramenta desenvolvida com intuito de detectar e realizar operações de refatoração sobre o código-fonte de sistemas desenvolvidos em Java. No entanto, a utilização desta ferramenta na confecção do *parser* é justificada somente pela necessidade de navegação pela AST. A ferramenta JRefactory encontra-se em sua versão 3.0, disponível no web site SourceForge [SourceForge 2009]. A principal funcionalidade desta versão é o suporte completo a versão 1.5 do JDK *(Java Development Kit)*.

Após a execução do *parser* sobre a classe *CompiereService* obtemos sua representação em CodeMI, como pode ser parcialmente visualizado na figura 4.7, onde focamos apenas o método *terminate* desta classe.

```
<ownedOperation xmi:type="uml:Operation" name="terminate"
<xmi:extension>
       <if conditions = "1" >
              <if conditions = "2" >
                     <try>
                            <statements/>
                            <catch>
                            </catch>
                     </try>
              </if>
              <statement/>
              <return/>
       </if>
       <return/>
</xmi:extension>
</ownedOperation>
```

Figura 4.7. Exemplo do método *terminate* em CodeMI.

É possível observar, na figura 4.7, que os elementos que representam a estrutura do código-fonte possuem fácil leitura, dada similaridade com a nomenclatura de comandos da linguagem de programação, e baixa verbosidade, devido a inexistência de excessivos detalhes e atributos. Estas características decorrem do conjunto reduzido de marcadores utilizados na CodeMI, enquanto que as demais representações abordadas no estudo comparativo do capítulo 2 demonstram um expressivo conjunto de marcadores e detalhes não desejáveis para esta pesquisa, devido ao requisito (d) de baixa exposição do código-fonte.

Outro destaque que pode ser observado na *CodeMI* é a impossibilidade de se recuperar o código-fonte original do software a partir desta representação, assegurando que eventuais restrições de propriedade intelectual existentes nos softwares industriais não sejam violadas. Também é permitido realizar diferentes análises sobre um arquivo na *CodeMI* sem a necessidade de acessar novamente o repositório de código-fonte original do software.

O marcador mais externo *ownedOperation* é oriundo do XMI e, neste exemplo, é utilizado para representar o método *terminate.* Os marcadores da representação *CodeMI* aparecem como sub-elementos de *xmi:extension*, como pode ser observado na figura 4.7. Este marcador é utilizado para delimitar os marcadores de extensões do formato XMI dos marcadores originalmente proposto pela OMG. Deste modo, asseguramos a compatibilidade dos arquivos CodeMI com as demais ferramentas de manipulação do formato XMI, com destaque para as ferramentas CASE *(Computer-Aided Software Engineering)* que utilizam este formato para a troca de modelos UML.

A troca de modelos UML entre ferramentas CASE utilizando arquivos na representação CodeMI ocorre de modo semelhante à troca de modelos do formato XMI. Deste modo, os marcadores do XMI que representem algum símbolo da UML serão interpretados pela ferramenta CASE, enquanto os marcadores de extensão do formato serão descartados por estas ferramentas.

De posse de elementos da UML é possível gerar diversos tipos de diagrama, entre estes, o diagrama de classes. A possibilidade de obter o diagrama de classes do sistema a partir de sua representação em CodeMI nos obriga a refletir sobre a possibilidade de realização de engenharia reversa e seus possíveis impactos sobre as restrições de propriedade intelectual.

4.4. CodeMI, Engenharia Reversa e Reengenharia

Uma das características da CodeMI, já mencionada na seção 4.3, é que esta representação não expõe o código-fonte do sistema. No entanto, sendo esta uma

representação estendida do formato XMI, a importação de um arquivo CodeMI pode ser facilmente realizada por qualquer ferramenta CASE, obtendo assim todos os elementos de seu modelo UML. Posteriormente, estes elementos podem ser utilizados na confecção de qualquer diagrama desta notação, concretizando a prática da engenharia reversa.

Segundo Pressman [Pressman 2006], a engenharia reversa pode extrair informação de projeto a partir do código-fonte, mas o *nível de abstração*, a *completeza* da documentação, o grau em que ferramentas e analistas trabalham juntos e a *direcionalidade* do processo são altamente variáveis.

O nível de abstração de um processo de engenharia reversa e as ferramentas usadas para executá-lo limitam a sofisticação da informação de projeto que pode ser extraída do código-fonte.

A completeza de um processo de engenharia reversa refere-se ao nível de detalhe que é fornecido em um nível de abstração. Na maioria dos casos, a completeza diminui à medida que o nível de abstração aumenta [Pressman 2006].

Se a direcionalidade do processo de engenharia reversa for única, toda a informação extraída do código-fonte será fornecida ao engenheiro de software, que pode então usá-la durante as atividades de manutenção. Se a direcionalidade for dupla, a informação será alimentada em uma ferramenta de reengenharia que tentará reestruturar ou regenerar o programa antigo.

A engenharia reversa aplicada sobre arquivos da CodeMI ocorre parcialmente devido a baixa completeza dos dados obtidos, visto que na CodeMI somente podem ser encontrados elementos estruturais de linguagens de programação orientadas a objetos, impedindo a elaboração de diagramas UML mais complexos.

De posse dos elementos da CodeMI que representam os pacotes, classes, métodos, atributos, associações, generalizações e dependências da UML, será possível rascunhar somente alguns diagramas estruturais. Dificilmente, com essas informações será possível criar outros diagramas, como os comportamentais ou de interações.

Ao final de um eventual processo de engenharia reversa realizado a partir de arquivos na CodeMI, alguns diagramas estruturais da UML podem ser obtidos e deles pode-se iniciar um processo de reengenharia em busca do software original. A reengenharia de software, também chamada de recuperação ou renovação, recupera informações de projeto de um software existente e usa estas informações para alterar ou reconstituir o sistema, preservando as funções existentes, ao mesmo tempo em que adiciona novas funções ao software, num esforço para melhorar sua qualidade global [Pressman 2006].

A tentativa de reengenharia, tomando por base a documentação obtida no processo de engenharia reversa descrito acima, seria bastante dificultada pela falta de informação a respeito do sistema. Em função da impossibilidade de obtenção do código-fonte e das dificuldades de realização de um processo de reengenharia a partir da representação CodeMI, consideramos, nestas circunstâncias, como satisfatório o nível de preservação dos detalhes do código-fonte da CodeMI.

4.5. Coletando Métricas Estruturais

Para a extração de métricas estruturais sobre a representação CodeMI podem ser utilizadas transformações XSLT - *Extensible Stylesheet Language Transformations* [XSLT 2001]. Uma transformação XSLT é expressa como um documento XML que inclui elementos definidos pela linguagem de transformação e elementos externos, definidos pelo usuário. Uma transformação expressa em XSLT descreve regras de transformação de uma árvore XML em outra árvore XML, através da associação de padrões a modelos. Um padrão é casado com elementos do documento, como nós ou atributos. Um processador XSLT percorre todos os nós do documento, compara-os a cada padrão do modelo e, caso eles sejam compatíveis, a regra associada ao padrão é aplicada. Uma de suas principais aplicações é a transformação de dados XML em HTML [XSLT 2001].

As duas principais linguagens de consulta a dados XML, que também podem ser utilizadas em conjunto com a representação CodeMI, são XPath [XPath 1999] e XQuery [XQuery 2007], ambas padronizadas pelo W3C [W3C 2009].

XPath acessa partes de documentos XML através de expressões de caminho. Sua grande vantagem é a simplicidade com que as consultas podem ser expressas. Por outro lado, esta é também sua principal desvantagem, uma vez que o mecanismo de expressões de caminho não suporta consultas avançadas, como consultas aninhadas, recursivas ou condicionais.

XQuery, por sua vez, é uma extensão de XPath. Ela também utiliza o conceito de expressão de caminho para navegar em documentos XML, mas se diferencia de XPath por definir consultas baseadas nas cláusulas FOR, LET, WHERE e RETURN (FLWR – lê-se “flower”), que dão maior legibilidade e poder às consultas e permitem a criação de consultas complexas, que não eram possíveis apenas com XPath.

A tecnologia XSLT será utilizada neste trabalho para fazer a transformação do XMI da representação CodeMI em um HTML contendo as métricas estruturais coletadas. Para exemplificar a coleta de métricas estruturais sobre a representação CodeMI, descreveremos nas próximas subseções alguns exemplos de transformações

XSLT que podem ser utilizadas para coletar métricas estruturais componentes das suítes apresentadas no capítulo 3.

4.5.1. Suíte de Lorenz & Kidd

Nesta subseção apresentaremos transformações XSLT elaboradas para a coleta de algumas métricas da suíte de Lorenz & Kidd. Entre estas destacamos:

- Tamanho do Método (LOC);
- Tamanho Médio dos Métodos;
- Número de Métodos Públicos de Instância (PIM);
- Número de Métodos de Instância numa Classe (NIM);
- Média dos Métodos de Instância por Classe (ANIM);
- Número de Atributos de Instância numa Classe (NIA);
- Média de Atributos de Instância por Classe (ANIA);
- Número de Métodos de Classe numa Classe (NCM);
- Média dos Métodos de Classe por Classe (ANCM);
- Número de Atributos de Classe numa Classe (NCA);
- Média de Atributos de Classe por Classe (ANCA);
- Número de Parâmetros por Método (PPM).

Entre os diversos modos de coletar o tamanho de um método, foi adotado nesta abordagem, o *total de linhas de código-fonte do método (LOC)*. Tendo em vista que cada linha de código-fonte originalmente implementada terá correspondência a um único marcador em sua representação CodeMI, podemos obter esta medida a partir da contagem dos marcadores existentes no método. Em virtude da CodeMI não possuir marcadores para representar as linhas em branco e comentários do código-fonte, seu total de LOC também reflete a ausência destes marcadores.

Como pode ser observado na figura 4.8, o código XSLT realiza um *loop* para percorrer todos os métodos de todas as classes do pacote que está sendo analisado. Para cada método de uma classe, identificado pelo caminho *"packagedElement/ packagedElement/ownedOperation"*, é realizada a contagem de todos marcadores internos ao *xmi:extension*, através da expressão *"count(xmi:extension//*)".*

```
<xsl:for-each select="packagedElement/packagedElement/ownedOperation">
      <xsl:value-of select="count(xmi:extension//*)"/>
</xsl:for-each>
```

Figura 4.8. Coleta do tamanho do método.

O *tamanho médio dos métodos* de uma classe pode ser obtido a partir da contagem dos marcadores existentes em todos os métodos, ou seja, o tamanho de todos os métodos, dividido pelo total de métodos.

Como pode ser observado na figura 4.9, o código XSLT realiza um *loop* para percorrer todas as classes do pacote que está sendo analisado. Para cada classe, identificada pelo caminho *"packagedElement/packagedElement"*, é realizada a contagem de todos marcadores internos ao *xmi:extension* de todos os seus métodos, através da expressão *"count(ownedOperation/xmi:extension//*)"*, obtendo o tamanho de todos os métodos da classe. Este resultado é dividido pelo total de métodos, obtido pela expressão *"count(ownedOperation)"*, resultando no tamanho médio dos métodos da classe.

```
<xsl:for-each select="packagedElement/packagedElement">
      <xsl:value-of select="count(ownedOperation/xmi:extension//*) div
            count(ownedOperation) "/>
</xsl:for-each>
```

Figura 4.9. Coleta do tamanho médio dos métodos.

O *número de métodos públicos de instância numa classe (PIM)* pode ser obtido através da contagem dos métodos com modificador de visibilidade igual a público em uma classe. Como pode ser observado na figura 4.10, o código XSLT realiza um *loop* para percorrer todas as classes do pacote que está sendo analisado. Para cada classe deste pacote é realizada a contagem de todos os métodos públicos, identificados pela expressão *"count(ownedOperation[@visibility='public'])".*

```
<xsl:for-each select="packagedElement/packagedElement">
      <xsl:value-of select="count(ownedOperation[@visibility=
            'public'])"/>
</xsl:for-each>
```

Figura 4.10. Coleta de métodos públicos de instância numa classe.

O *número de métodos de instância numa classe (NIM)* pode ser obtido a partir da contagem dos métodos que não possuem modificador *static* em sua declaração. Como pode ser observado na figura 4.11, o código XSLT realiza um *loop* para percorrer todas as classes do pacote que está sendo analisado. Para cada classe deste pacote é realizada a contagem de todos os métodos de instância, ou seja, que possuam o atributo de visibilidade diferente de *static*. Esta contagem é realizada pela expressão *"count(ownedOperation[@visibility!='static'])".*

```
<xsl:for-each select="packagedElement/packagedElement">
      <xsl:value-of select="count(ownedOperation[@visibility!=
            'static'])"/>
</xsl:for-each>
```

Figura 4.11. Coleta de métodos de instância numa classe.

A média de métodos de instância por classe (ANIM) é calculada dividindo-se o total de métodos de instância numa classe pelo total de classes. Como pode ser observado na figura 4.12, o código XSLT realiza a contagem de todos os métodos que possuam o atributo de visibilidade diferente de *static* (como na métrica anterior). Este valor é dividido pelo total de classes do pacote, que é obtido pela expressão *"count(packagedElement)".* Esta métrica pode ser obtida em nível de pacote ou de sistema.

```
<xsl:for-each select="packagedElement">
      <xsl:value-of select="round((count(packagedElement/
            ownedOperation[@visibility!='static']) div
            count(packagedElement)) * 100) div 100"/>
</xsl:for-each>
```

Figura 4.12. Coleta da média de métodos de instância por classe.

O *número de atributos de instância numa classe (NIA)* representa o total de atributos de instância de uma classe. São considerados atributos de instância os que possuem modificadores iguais à *private* ou *protected*. Como pode ser observado na figura 4.13, para cada classe do pacote sendo analisado é realizada a contagem de todos os atributos de instância, ou seja, todos os elementos *ownedAttribute* que possuam o atributo de visibilidade igual a *private* ou *protected*. Esta contagem é realizada pela expressão *"count(ownedAttribute[@visibility='private' or @visibility='protected'])".*

```
<xsl:for-each select="packagedElement/packagedElement">
      <xsl:value-of select="count(ownedAttribute[@visibility='private'
            or @visibility='protected'])"/>
</xsl:for-each>
```

Figura 4.13. Coleta de atributos de instância numa classe.

A *média de atributos de instância por classe (ANIA)* representa o total de atributos de instância numa classe dividido pelo total de classes. Como pode ser observado na figura 4.14, o código XSLT realiza a contagem de todos os atributos de instância, ou seja, que possuam o atributo de visibilidade igual à *private* ou *protected*.

Este valor é dividido pelo total de classes do pacote. Esta métrica pode ser obtida em nível de pacote ou de sistema.

```
<xsl:for-each select="packagedElement">
      <xsl:value-of select="round((count(packagedElement/
            ownedAttribute[@visibility='private' or
            @visibility='protected']) div
            count(packagedElement)) * 100) div 100 "/>
</xsl:for-each>
```

Figura 4.14. Coleta da média de atributos de instância por classe.

O *número de métodos de classe numa classe (NCM)* representa o total de métodos de classe, ou seja, os métodos compartilhados por todas as instâncias de uma classe e que possuam o modificador *static* em sua declaração. Como pode ser observado na figura 4.15, para cada classe do pacote é realizada a contagem de todos os métodos de classe, ou seja, que possuam o atributo de visibilidade igual a *static*. A contagem é realizada pela expressão *"count(ownedOperation[@visibility='static'])".*

```
<xsl:for-each select="packagedElement/packagedElement">
      <xsl:value-of select="count(ownedOperation[@visibility=
            'static'])"/>
</xsl:for-each>
```

Figura 4.15. Coleta de métodos de classe numa classe.

A *média de métodos de classe por classe (ANCM)* representa o total de métodos de classe dividido pelo total de classes. Como pode ser observado na figura 4.16, o código XSLT realiza a contagem de todos os métodos de classe, ou seja, os que possuam o atributo de visibilidade igual a *static*. Esta contagem é realizada pela expressão *"count(packagedElement/ownedOperation[@visibility='static'])".* Este valor é dividido pelo total de classes do pacote. Esta métrica pode ser obtida em nível de pacote ou de sistema.

```
<xsl:for-each select="packagedElement">
      <xsl:value-of select="round((count(packagedElement/
            ownedOperation[@visibility='static']) div
            count(packagedElement)) * 100) div 100"/>
</xsl:for-each>
```

Figura 4.16. Coleta média de métodos de classe por classe.

O *número de atributos de classe numa classe (NCA)* representa o total de atributos de classe, ou seja, os atributos compartilhados por todas as instâncias de uma classe e que possuam o modificador *static* em sua declaração. Como pode ser

observado na figura 4.17, para cada classe deste pacote é realizada a contagem de todos os atributos de classe, ou seja, todos os elementos *ownedAttribute* que possuam o atributo de visibilidade igual a *static.* Esta contagem é realizada pela expressão *"count(ownedAttribute[@visibility='static'])".*

```
<xsl:for-each select="packagedElement/packagedElement">
      <xsl:value-of select="count(ownedAttribute[@visibility=
            'static']"/>
</xsl:for-each>
```

Figura 4.17. Coleta de atributos de classe numa classe.

Outra métrica é a *média de atributos de classe por classe (ANCA)*, como pode ser observado no XSLT da figura 4.18. Esta medida e calculada de modo similar ao cálculo do *total de variáveis de classe numa classe*. Porém, no cálculo da média, este resultado é dividido pelo total de classes do pacote. Este último é obtido pela expressão: *"count(packagedElement)"*. Esta métrica pode ser obtida em nível de pacote ou de sistema.

```
<xsl:for-each select="packagedElement">
      <xsl:value-of select="round((count(packagedElement/
            ownedAttribute[@visibility='static']) div
            count(packagedElement)) * 100) div 100"/>
</xsl:for-each>
```

Figura 4.18. Coleta da média de atributos de classe por classe.

O *número de parâmetros por método (PPM)* representa o total de parâmetros de entrada de um método da classe. Como pode ser observado na figura 4.19, para cada classe do pacote é realizado outro *loop* para todos os métodos da classe que está sendo analisada. Para cada método identificado pelo caminho *"ownedOperation"* é realizada a contagem de todos os parâmetros do método, ou seja, todos os elementos "*ownedParameter".* Esta contagem é realizada pela expressão *"count(ownedParameter)".*

```
<xsl:for-each select="packagedElement/packagedElement">
      <xsl:for-each select="ownedOperation">
            <xsl:value-of select="count(ownedParameter)"/>
      </xsl:for-each>
</xsl:for-each>
```

Figura 4.19. Coleta do número de parâmetros por método.

4.5.2. Suíte de Chidamber & Kemerer

Nesta subseção apresentaremos transformações XSLT elaboradas para a coleta de métricas da suíte de Chidamber & Kemerer. Entre estas destacamos:

- Métodos Ponderados por Classe (WMC);
- Número de Filhos (NOC).

A métrica de *métodos ponderados por classe (WMC)* pode ser obtida através do somatório da complexidade de todos os métodos por classe. Como pode ser observado na figura 4.20, para cada classe é realizada a contagem de todas as decisões existentes no fluxo de execução do método, ou seja, todos os elementos *if, for, while e case*, adicionando-se 1 ao resultado deste somatório para representar o fluxo principal do método. Esta contagem é necessária para o cálculo da complexidade dos métodos (neste caso, foi utilizada a complexidade ciclomática).

```
<xsl:for-each select="packagedElement/packagedElement">
      <xsl:value-of select="(
            count(ownedOperation/xmi:extension//if)    +
            count(ownedOperation/xmi:extension//for)   +
            count(ownedOperation/xmi:extension//while) +
            count(ownedOperation/xmi:extension//case)  +
            count(ownedOperation))"/>
</xsl:for-each>
```

Figura 4.20. Coleta de métodos ponderados por classe.

O *número de filhos (NOC)* pode ser obtido a partir do total de subclasses imediatas, subordinadas a uma classe em uma hierarquia. Como pode ser observado na figura 4.21, para cada classe é criada uma variável para armazenar o identificador da classe. Em seguida, é realizada a contagem de todas as classes do pacote que são filhas desta, ou seja, as classes que possuem o elemento *generalization* com atributo *general* igual ao identificador da classe selecionada.

```
<xsl:for-each select="packagedElement/packagedElement">
      <xsl:variable name="id" select="./@xml:id" />
      <xsl:value-of select="(
            count(//packagedElement/packagedElement/
                  generalization[@general = "$id"])"/>
</xsl:for-each>
```

Figura 4.21. Coleta do número de filhos da classe.

Não foi elaborada a transformação XLST para coleta da *Profundidade da Árvore de Herança (DIT)* devido a ausência mecanismos neste formato para percorrer

recursivamente a hierarquia de uma determinada classe dentro do sistema ou de um pacote. O mesmo ocorre para a *Falta de Coesão nos Métodos (LCOM)* devido a ausência de marcadores da CodeMI para expressar a dinâmica de utilização e compartilhamento dos atributos de instância nos métodos da classe. No entanto, a utilização de bancos de dados XML com suporte a execução de consultas XQuery podem auxiliar a coleta da DIT, tendo em vista que este formato fornece recursos complementares aos da XSLT, como a operação *junção*, capazes manipular múltiplos arquivos simultaneamente, correlacionar seus marcadores e extrair informações, supomos ser possível obter a arvóre hierárquica da classe de um sistema.

4.5.3. Complexidade Ciclomática

Nesta subseção apresentaremos transformações XSLT elaboradas para a coleta de algumas métricas de complexidade ciclomática. Entre estas destacamos:

- Complexidade Ciclomática de McCabe;
- Complexidade Ciclomática de Myers.

A medida de complexidade ciclomática de McCabe pode ser obtida a partir da contagem de todas as decisões existentes no código-fonte implementado no método.

Como pode ser observado na figura 4.22, o código XSLT realiza um *loop* para todas as classes do pacote que está sendo analisado. Para cada classe deste pacote é realizado um outro loop para todos os métodos do pacote da classe. Para cada método desta classe, identificado pelo caminho *"ownedOperation"*, é realizada a contagem de todas as decisões existentes no fluxo de execução do método, ou seja, todos os elementos *if, for, while e case, somando-se 1 para representar o fluxo principal.*

```
<xsl:for-each select="packagedElement/packagedElement">
      <xsl:for-each select="ownedOperation">
      <xsl:value-of select="(
            count(xmi:extension//if)    +
            count(xmi:extension//for)   +
            count(xmi:extension//while) +
            count(xmi:extension//case)  +
            count(xmi:extension))"/>
      </xsl:for-each>
</xsl:for-each>
```

Figura 4.22. Coleta de complexidade ciclomática de McCabe.

A medida de complexidade ciclomática de Myers, que determina um intervalo de complexidade, tem seu limite inferior determinado pela contagem das decisões do

método e seu limite superior determinado pela contagem das condições existentes no código-fonte implementado no método.

Como pode ser observado na figura 4.23, o código XSLT realiza um *loop* para todas as classes do pacote que está sendo analisado. Para cada classe deste pacote é realizado outro loop para todos os métodos da classe. Para cada método desta classe é realizada a contagem dos elementos que determinam os limites inferior e superior desta métrica. Para obtenção do limite inferior são utilizadas as decisões existentes no fluxo de execução do método, de modo semelhante ao cálculo da complexidade ciclomática de McCabe apresentada acima. Para o cálculo do limite superior são utilizados os totais de condições, acrescidos de uma unidade para representar o fluxo principal.

```
<xsl:for-each select="packagedElement/packagedElement">
      <xsl:for-each select="ownedOperation">
      <xsl:variable name="cc" select="
            count(xmi:extension//if) +
            count(xmi:extension//for) +
            count(xmi:extension//while) +
            count(xmi:extension//case) + 1" />
      <xsl:variable name="ccm" select="
            sum(xmi:extension//if/@conditions) +
            sum(xmi:extension//for/@conditions) +
            sum(xmi:extension//while/@conditions) +
            sum(xmi:extension//case/@conditions) + 1" />
      <xsl:value-of select="concat($cc,':',$ccm)"/>
      </xsl:for-each>
</xsl:for-each>
```

Figura 4.23. Coleta de complexidade ciclomática de Myers.

4.5.4. Variações

Nesta subseção apresentaremos transformações XSLT elaboradas para a coleta de algumas variações de métricas das suítes apresentadas anteriormente. Entre estas destacamos:

- Tamanho da Classe;
- Tamanho Médio da Classe;
- Tamanho do Subsistema;
- Complexidade Ciclomática da Classe;
- Complexidade Ciclomática do Subsistema;
- Complexidade Ciclomática Média das Classes do Subsistema.

O *tamanho da classe* pode ser obtido através do somatório da métrica *tamanho do método* de todos os métodos da classe. Como pode ser observado na figura 4.24, o código XSLT, realiza um *loop* para percorrer todas as classes do pacote que está sendo analisado. Para cada classe é realizada a contagem de todos marcadores internos ao *ownedOperation/xmi:extension* através da expressão *"count(ownedOperation/xmi:extension//*)".*

```
<xsl:for-each select="packagedElement/packagedElement/">
      <xsl:value-of select="count(
            ownedOperation/xmi:extension//*)"/>
</xsl:for-each>
```

Figura 4.24. Coleta do tamanho da classe.

O *tamanho médio das classes* de um pacote pode ser obtido a partir do somatório da métrica *tamanho da classe* de todas as classes do pacote, dividido pelo total de classes. Como pode ser observado na figura 4.25, o código XSLT realiza a contagem de todos marcadores internos ao *xmi:extension* de todos os métodos de todas as classes do pacote, obtendo o tamanho do pacote. Este resultado é dividido pelo total de classes, *"count(packagedElement)"*, obtendo-se o tamanho médio das classes do pacote.

```
<xsl:for-each select="packagedElement ">
      <xsl:value-of select="count(
            packagedElement/ownedOperation/xmi:extension//*) div
            count(packagedElement) "/>
</xsl:for-each>
```

Figura 4.25. Coleta do tamanho médio das classes.

O *tamanho do subsistema*, ou pacote, pode ser obtido a partir do somatório da métrica *tamanho da classe* de todas as classes do pacote. Como pode ser observado na figura 4.26, o código XSLT realiza a contagem de todos marcadores internos ao *xmi:extension* de todos os métodos de todas as classes do pacote, obtendo o tamanho do pacote.

```
<xsl:for-each select="packagedElement ">
      <xsl:value-of select="count(
            packagedElement/ownedOperation/xmi:extension//*)"/>
</xsl:for-each>
```

Figura 4.26. Coleta do tamanho do subsistema.

A *complexidade ciclomática da classe* pode ser obtida a partir do somatório da métrica *complexidade ciclomática* de cada um dos seus métodos. Como pode ser observado na figura 4.27, o código XSLT realiza um *loop* para todas as classes do pacote que está sendo analisado. Para cada classe é realizada a contagem de todas as decisões existentes no fluxo de execução do método, ou seja, todos os elementos *if, for, while e case, adicionando uma unidade para cada método da classe.*

```
<xsl:for-each select="packagedElement/packagedElement">
      <xsl:value-of select="(
            count(ownedOperation/xmi:extension//if)    +
            count(ownedOperation/xmi:extension//for)   +
            count(ownedOperation/xmi:extension//while) +
            count(ownedOperation/xmi:extension//case)  +
            count(ownedOperation))"/>
</xsl:for-each>
```

Figura 4.27. Coleta de complexidade ciclomática da classe.

A *complexidade ciclomática do subsistema* pode ser obtida a partir do somatório da métrica *complexidade ciclomática* de todas as classes do subsistema. Como pode ser observado na figura 4.28, é realizada a contagem de todas as decisões existentes no fluxo de execução dos métodos de todas as classes do subsistema.

```
<xsl:for-each select="packagedElement">
      <xsl:value-of select="(
      count(packagedElement/ownedOperation/xmi:extension//if)    +
      count(packagedElement/ownedOperation/xmi:extension//for)   +
      count(packagedElement/ownedOperation/xmi:extension//while) +
      count(packagedElement ownedOperation/xmi:extension//case)  +
      count(packagedElement/ownedOperation))"/>
</xsl:for-each>
```

Figura 4.28. Coleta de complexidade ciclomática do subsistema.

A *complexidade ciclomática média das classes do subsistema* pode ser obtida a partir do somatório da métrica *complexidade ciclomática da classe* para todas as classes do subsistema dividido pelo *total de classes do subsistema.* Como pode ser observado na figura 4.29, o código XSLT realiza a contagem de todas as decisões existentes no fluxo de execução dos métodos de todas as classes do subsistema. Este valor é dividido pelo total de classes do subsistema.

```
<xsl:for-each select="packagedElement">
      <xsl:value-of select="(
      count(packagedElement/ownedOperation/xmi:extension//if)    +
      count(packagedElement/ownedOperation/xmi:extension//for)   +
      count(packagedElement/ownedOperation/xmi:extension//while) +
      count(packagedElement/ownedOperation/xmi:extension//case)  +
      count(packagedElement/ownedOperation) ) div
      count(packagedElement/packagedElement)"/>
</xsl:for-each>
```

Figura 4.29. Coleta de complexidade ciclomática das classes do subsistema.

4.6. Prova de Conceito: Compiere

Nesta seção descrevemos uma prova de conceito realizada com intuito de verificar a viabilidade de coletar métricas de um Sistema de Informação de larga escala utilizando a representação CodeMI. Na realização desta prova de conceito foi utilizado o código-fonte do software *Compiere*, que segue os moldes do desenvolvimento *open source* [Compiere 2009]. Tal escolha foi motivada pelo fato deste software ser de aplicabilidade essencialmente comercial e industrial, além de possuir volume de código-fonte entre médio e alto, com aproximadamente 247 Megabytes.

O software *Compiere* foi originalmente proposto com foco em empresas que atuam no setor do comércio [Compiere 2009]. Porém, também pode ser utilizado por outros setores, como indústria e serviços, desde que sejam realizadas algumas customizações. É considerado um sistema de gestão completo para as empresas de micro e pequeno porte, por utilizar ferramentas de gestão empresarial ERP (Enterprise Resource Planning) e gestão de relacionamento com cliente CRM (Customer Relationship Manager), integrando as informações dos departamentos e funções das diversas áreas de negócio da empresa.

Como pode ser observado na figura 4.30, esta prova de conceito foi dividida em quatro etapas: (1) obtenção do repositório do sistema de controle de versões onde reside o código-fonte do *Compiere*; (2) extração de uma versão do software a partir do repositório do sistema de controle de versões; (3) conversão do código-fonte da versão selecionada para CodeMI; e (4) extração de métricas estruturais a partir da representação CodeMI da versão selecionada do Compiere. As duas primeiras etapas (obtenção do repositório de controle de versões e extração de uma versão do software) não serão profundamente exploradas nesta prova de conceito, pois compreendem atividades que variam conforme o processo de desenvolvimento de software utilizado no projeto sob análise e as ferramentas adotadas em seu desenvolvimento.

Figura 4.30. Etapas da prova de conceito

Esta prova de conceito iniciou pela obtenção do repositório de controle de versões do sistema *Compiere* (disponível em http://svn.compiere.org). Posteriormente, foi realizada a extração de seu código-fonte a partir de operação de *checkout* em seu repositório. Para a manipulação do repositório do *Compiere* foi utilizado o *TortoiseSVN* [TortoiseSVN 2009] que é uma ferramenta cliente do SVN (SubVersioN) [SubVersion 2009] capaz de realizar diversas tarefas, como *checkout*, *update* e *commit* nos módulos do sistema. A versão do *Compiere* utilizada na realização das próximas atividades é datada de 30/10/2009.

A próxima etapa desta prova de conceito foi a conversão do código-fonte para a representação CodeMI. Esta conversão foi realizada utilizando-se um parser construído com a finalidade de identificar os elementos estruturais do código-fonte de sistemas em Java e gerar arquivos XMI para representar tais estruturas no formato CodeMI.

Após a execução deste parser, obtivemos um arquivo CodeMI para cada pacote do sistema Compiere, totalizando sessenta e seis pacotes, como pode ser observado na tabela 4.3. Sendo assim, ao final desta etapa foram gerados sessenta e seis arquivos com o nome de cada pacote e extensão .xmi, representando o código-fonte do pacote na representação CodeMI.

Tabela 4.3. Pacotes do sistema Compiere

compiere.model.xmi	org.compiere.common.constants.xmi	org.compiere.print.xmi
org.apache.ecs.filter.xmi	org.compiere.common.xmi	org.compiere.process.xmi
org.apache.ecs.storage.xmi	org.compiere.controller.xmi	org.compiere.report.core.xmi
org.apache.ecs.xhtml.xmi	org.compiere.db.xmi	org.compiere.report.xmi
org.apache.ecs.xmi	org.compiere.esb.xmi	org.compiere.server.xmi
org.apache.ecs.xml.xmi	org.compiere.excel.xmi	org.compiere.session.xmi
org.compiere.acct.xmi	org.compiere.framework.xmi	org.compiere.sla.xmi
org.compiere.api.xmi	org.compiere.grid.ed.xmi	org.compiere.sqlj.xmi
org.compiere.apps.form.xmi	org.compiere.grid.tree.xmi	org.compiere.startup.xmi
org.compiere.apps.graph.xmi	org.compiere.grid.xmi	org.compiere.swing.xmi
org.compiere.apps.info.xmi	org.compiere.images.xmi	org.compiere.test.xmi
org.compiere.apps.search.xmi	org.compiere.impexp.xmi	org.compiere.tools.xmi
org.compiere.apps.wf.xmi	org.compiere.install.xmi	org.compiere.translate.xmi
org.compiere.apps.xmi	org.compiere.interfaces.xmi	org.compiere.udf.xmi
org.compiere.cm.cache.xmi	org.compiere.intf.xmi	org.compiere.util.xmi
org.compiere.cm.extend.xmi	org.compiere.layout.xmi	org.compiere.vos.xmi
org.compiere.cm.invoice.xmi	org.compiere.ldap.xmi	org.compiere.web.sample.xmi
org.compiere.cm.request.xmi	org.compiere.minigrid.xmi	org.compiere.web.xmi
org.compiere.cm.utils.xmi	org.compiere.model.xmi	org.compiere.wf.xmi
org.compiere.cm.wiki.xmi	org.compiere.plaf.xmi	org.compiere.wstore.xmi
org.compiere.cm.xmi	org.compiere.pos.xmi	org.compiere.xmi
org.compiere.cm.xml.xmi	org.compiere.print.layout.xmi	org.compiere.xuom.xmi

De posse dos arquivos, foram selecionadas algumas métricas estruturais a fim de serem coletadas a partir da representação CodeMI do *Compiere*. Tais métricas foram selecionadas entre as suítes de Lorenz & Kidd, Chidamber & Kemerer, complexidade ciclomática e variações destas medidas, conforme apresentado no capítulo 3.

Diversas técnicas e ferramentas para manipular arquivos baseados no formato XML/XMI podem ser utilizadas na coleta de métricas estruturais sobre a representação CodeMI. Para esta prova de conceito foram criados três arquivos de transformações XSLT, concatenando em cada um as rotinas para realizar a coleta das métricas das suítes apresentadas no capítulo 3. Estas transformações podem ser visualizadas em detalhes nos apêndices A (Lorenz_Kidd.xslt), B (Chidamber_Kemerer.xslt) e C (Complexidade_Ciclomatica.xslt). As métricas variantes das suítes também foram obtidas a partir destas transformações.

A medição do sistema *Compiere* foi realizada utilizando-se o aplicativo XML Notepad 2007 [Notepad 2007] para manipular os arquivos na CodeMI e aplicar as transformações XSLT, obtendo o resultado das métricas. Estes resultados são fornecidos através de um arquivo HTML.

Cada transformação XSLT realizada resulta em um arquivo HTML. Obtivemos, ao final desta prova de conceito, um total de cento e noventa e oito arquivos HTML (66 pacotes x 3 transformações = 198 arquivos) com resultados das métricas das suítes apresentadas. Em função do grande volume de arquivos HTML obtidos, optamos por apresentar, na próxima página, somente os valores relativos a um pacote do sistema, o pacote *org.compiere.grid*. Os demais arquivos podem ser visualizados no DVD em anexo a este trabalho.

Aplicando-se as transformações *Lorenz_Kidd.xslt*, *Chidamber_Kemerer.xslt* e *Complexidade_Ciclomatica.xslt* sobre o arquivo *org.compiere.grid.xmi*, obtivemos ao final do processamento, um arquivo HTML para cada transformação realizada, totalizando três arquivos. Os arquivos gerados contém medidas obtidas para o pacote *org.compiere.grid.xmi* das suítes de métricas conforme proposto por cada transformação descrita.

Um resumo dos valores coletados através deste processamento pode ser observado na tabela 4.4. Entre as métricas da suíte de Lorenz & Kidd coletamos: (a) LOC - Linhas de Código; (b) NCM – Número de Métodos de Classe em uma Classe; (c) NCA – Número de Atributos de Classe em uma Classe; (d) NIM – Número de Métodos de Instância numa Classe; (e) NIA – Número de Atributos de Instância numa Classe; (f) NA – Número de Atributos da Classe; (g) NM – Número de Métodos da Classe; (h) PIM - Métodos Públicos de Instância numa Classe; e (i) PPM - Parâmetros por Método. Para a suíte de métricas de Chidamber & Kemerer coletamos: (a) WMC - Métodos Ponderados por Classe; e (b) NOC - Número de Filhos. Finalizando pelas medidas de complexidade: (a) CC - Complexidade Ciclomática de McCabe; e (b) CCM - Complexidade Ciclomática de Myers.

De modo análogo, coletamos algumas destas medidas diretamente do código-fonte, ou seja, sem intemédio de representação intermediária. Para isto utilizamos uma ferramenta que funciona como plugin do ambiente de desenvolvimento de software Eclipse, conhecida como Metrics [Metrics 2009]. Assim, foi possível realizar uma comparação entre as métricas colhidas a partir da CodeMI e as métricas colhidas pela ferramenta. Após a compilação do sistema Compiere, o plug-in Metrics calcula automaticamente grande parte das métricas, como pode ser visualizado na tabela 4.5.

Tabela 4.4. Resumo de métricas do pacote *org.compiere.grid (*segundo XSLT gerado pela CodeMI)

Total de Classes	Total de Metodos	Total de Comandos	ANIM	ANIA	ANCA	ANCM	C.C. Média	C.C. Total
13	160	2658	12.08	17.46	0.23	0.23	4.17	667

Nome da Classe	LOC/NM	PIM	NIM	NIA	NCM	NCA	NA	NM	LOC	WMC	NOC	CC Média	CC Total	CCM Média	CCM Total
ApanelTab	0	4	4	0	0	0	0	4	0	4	0	1	4	1:1	4:4
GridController	11.72	34	36	22	0	0	22	36	422	134	0	3.72	134	3.72:4.86	134:175
RecordAccessDialog	19.63	2	8	21	0	0	21	8	157	28	0	3.5	28	3.5:3.63	28:29
VCreateFrom	12.24	7	17	32	0	0	32	17	208	42	3	2.47	42	2.47:2.76	42:47
VCreateFromInvoice	18.73	5	11	4	0	0	4	11	206	50	0	4.55	50	4.55:5.91	50:65
VCreateFromShipment	18.08	5	11	3	1	0	3	12	217	45	0	3.75	45	3.75:5.25	45:63
VCreateFromStatement	11.3	4	10	2	0	0	2	10	113	22	0	2.2	22	2.2:2.9	22:29
VPanel	14	8	14	18	2	2	21	16	224	65	0	4.06	65	4.06:5.69	65:91
VPayment	63.82	4	11	94	0	0	94	11	702	151	0	13.73	151	13.73:22.36	151:246
VSortTab	19.31	9	13	29	0	1	30	13	251	59	0	4.54	59	4.54:5.92	59:77
VTabbedPane	8.67	9	9	5	0	0	5	9	78	31	0	3.44	31	3.44:3.89	31:35
VTable	7.33	5	6	2	0	0	2	6	44	17	0	2.83	17	2.83:3.67	17:22
XLookup	5.14	7	7	2	0	0	2	7	36	19	0	2.71	19	2.71:3.71	19:26

Tabela 4.5. Resumo de métricas do pacote *org.compiere.grid* (segundo plugin Metrics)

Total de Classes	Total de Metodos	Total de Comandos	ANIM	ANIA	ANCA	ANCM	C.C. Média	C.C. Total
13	154	3603	-	16	2,30	1.15	4.79	738

Nome da Classe	PIM	NIM	NIA	NCM	NCA	NA	NM	LOC	WMC	NOC	CC
APanelTab	4	4	0	0	0	0	0	0	0	0	0
GridController	34	36	20	0	2	22	36	554	134	0	134
RecordAccessDialog	2	8	20	0	1	21	8	189	32	0	32
VCreateFrom	7	17	30	0	2	32	17	275	45	3	45
VCreateFromInvoice	5	11	2	0	2	4	11	288	52	0	52
VCreateFromShipment	5	11	1	1	2	3	12	301	54	0	54
VCreateFromStatement	4	10	0	0	2	2	10	146	21	0	21
VPanel	8	14	13	0	8	21	14	297	78	0	78
VPayment	4	11	91	0	3	94	11	954	182	0	182
VSortTab	9	13	28	0	2	30	13	368	67	0	67
VTabbedPane	9	9	2	0	3	5	9	112	32	0	32
VTable	5	6	0	0	2	2	6	66	19	0	19
XLookup	7	7	1	0	1	2	7	53	22	0	22

As metricas NIM e NCM de Lorenz & Kidd foram obtidas no Metrics sem nenhuma divergência em relação aos valores obtidos pelo XSLT, exceto pelo fato do plugin não calcular estas métricas para as interfaces, como ocorrido com a interface APanelTab. Outro ponto não considerado pelo plugin é a inicialização estática. Na coleta de métricas pelo XSLT, as inicializações estáticas são tratadas como métodos de classe, devido a similaridade de suas definições. Consequentemente estas inicializações também são contabilizadas na métrica NCM.

No cálculo da métrica NM pelo Metrics encontramos reflexos das mesmas exceções encontradas na coleta das métricas NIM e NCM, ou seja, não contabilização dos métodos das interfaces e das inicializações estáticas. Isto ocorre devido a métrica NM ser obtida tomando por base o total de métodos da classe.

No cálculo da métrica NA, os valores obtidos pelo Metrics e pelo XSLT são numericamente equivalentes. No entanto, há divergências no cálculo da NIA e NCA. Estas divergências ocorrem quando utilizamos o XSLT para o cálculo do NIA, pois como descrito por Lorenz e Kidd [Lorenz & Kidd 1994], devem ser contabilizados somente os atributos *private* e *protected.* O Metrics, por sua vez, também considera os atributos *public* no cálculo do NIA. Já no cálculo do NCA somente foram contabilizados os atributos com primeiro modificador *static*. No total (NA), os dois métodos apresentam os mesmos valores.

No cálculo da métrica LOC dos métodos pelo Metrics, observamos que as aberturas e fechamentos de comandos são consideradas como linhas de código. Além disso, caso uma chamada de método ocupe *"n"* linhas de código, todas serão computadas no cálculo desta métrica, diferentemente de quando aplicamos o XSLT, onde computamos somente o comando e não suas aberturas e fechamentos. Do mesmo modo, computamos apenas um LOC para cada chamada de método, independente da quebra de linhas promovida por estilos individuais de programação. Estas diferenças na coleta justificam os valores, normalmente inferiores, obtidos pelo XSLT.

A obtenção da complexidade ciclomática pelo Metrics contabiliza todas as saídas *return* dos métodos, diferentemente do XSLT que contabiliza uma única saída *return* por método. Sendo o objetivo desta métrica contabilizar os possíveis fluxos de execução do código, optamos por totalizar os desvios e condicionais (if, for, while, case) e adicionar sempre mais uma unidade para representar o fluxo principal. A métrica WMC adota a mesma sistemática de cálculo da complexidade ciclomática total, inclusive as questões de convergência/divergência entre o plugin Metrics e o XSLT proposto.

O NOC mostrou-se com o mesmo resultado nos dois modos de cálculo, pois trata-se do total de classes filhas, que é um critério muito bem definido e sem margem para outras interpretações ou adaptações.

As demais métricas relacionadas as médias (ANIM, ANIA, ANCA e ANCM) sofrem os reflexos citados no cálculo das metricas base (NIM, NIA, NCA e NCM).

4.7. Conclusão

Este capítulo apresentou a CodeMI, uma extensão do formato XMI que é capaz de representar a estrutura do código-fonte implementado nos métodos de classes, utilizando um conjunto de doze marcadores. Em seguida, apresentamos uma série de consultas baseadas no padrão XLST que podem ser aplicadas sobre esta representação a fim de se coletar medidas da estrutura de um código-fonte.

O conjunto reduzido de marcadores da CodeMI assegura a baixa verbosidade da representação, permitindo a realização de consultas mais eficientes e manipulação menos dispendiosa. A não exposição do código-fonte também é outra característica importante da CodeMI, pois impede o *disclosure* do código-fonte da aplicação. Embora exista a possibilidade de realização de engenharia reversa para obtenção de parte dos elementos do modelo UML do software, acreditamos que a reengenharia a partir destes dados seria dificultada pela ausência de informações que exprimam as relações entre estes elementos e pela carência de representação da dinâmica de execução do software.

A utilização de transformações XSLT permitiu a coleta de métricas a partir da representação do pacote *org.compiere.grid* na CodeMI. Este mecanismo mostrou-se satisfatório para a coleta das métricas estruturais apresentadas em nível de método, classe e pacote.

CAPÍTULO 5
UM AMBIENTE DE SUPORTE A PESQUISAS SOBRE A EVOLUÇÃO DE SOFTWARES COMERCIAIS

5.1. Considerações Iniciais

Neste capítulo apresentaremos uma proposta para a construção de um ambiente de apoio a pesquisas realizadas na área de evolução de software. Este ambiente terá como principal objetivo facilitar a aquisição e distribuição de informações históricas sobre sistemas de software, necessárias para as pesquisas na linha de evolução.

O ambiente fará uso da representação CodeMI, se beneficiando da possibilidade das métricas utilizadas nas pesquisas poderem ser definidas e terem seus valores coletados a posteriori da aquisição das informações históricas que descrevem o projeto de software. O ambiente também se beneficiará da possibilidade de armazenar informações sobre projetos industriais sem quebra de direitos autorais, em decorrência da baixa verbosidade propriciada pela representação CodeMI.

Para citar exemplos de pesquisas que poderiam se beneficiar do uso de CodeMI, Barros [Barros 2008] apresentou uma abordagem baseada em simulação para estimar a distribuição de probabilidade que descreve o tamanho esperado de um projeto de software (em linhas de código) em um determinado momento no futuro, partindo de informações sobre o comportamento observado no desenvolvimento do sistema ao longo do passado recente. Para a realização destas simulações foram utilizadas informações históricas do projeto, obtidas a partir de sistemas de controle de versão, que são bem conhecidas e largamente utilizadas na indústria. Para a validação do modelo, o autor realizou um estudo de caso, aplicando a abordagem proposta para a estimativa do tamanho de um projeto de software de larga escala (cerca de 700 KLOC) em um período de tempo aproximado de três meses.

Em outro exemplo, Araújo [Araujo 2009] construiu um modelo baseado em evidências inspirado nas Leis de Evolução de Software e apoiado por simulação contínua segundo a técnica de modelagem da Dinâmica de Sistemas [Forrester 1991] - cujo objetivo é possibilitar a construção de uma infraestrutura computacional que permita predizer como sistemas de software orientados a objetos se comportam ao

longo de sucessivos ciclos de manutenção, de forma a proporcionar uma melhor compreensão do processo de decaimento. Para isto foram utilizadas métricas de tamanho, periodicidade, complexidade, esforço, confiabilidade e manutenibilidade, coletadas em cada ciclo de manutenção e analisadas ao longo do tempo. A tendência apresentada individualmente por cada uma destas medidas é aplicada sobre um modelo de formulações lógicas, que avalia a presença ou ausência de evidências sobre a aplicação das Leis de Lehman no sistema que está sendo analisado. Neste trabalho, são apresentados dois estudos experimentais, que utilizam dados de sistemas reais a fim de identificar a viabilidade e aderência do modelo proposto.

Em seguida, na seção 5.2, apresentaremos uma visão geral de um ambiente de suporte à realização de estudos em evolução de software, abordando suas principais funcionalidades, bem como potenciais benefícios proporcionados por este ambiente na realização de pesquisas de Engenharia de Software. Por fim, na seção 5.3 apresentamos nossas considerações finais.

5.2. Um Ambiente de Suporte a Estudos em Evolução de Software

Os estudos apresentados na introdução deste capítulo foram desenvolvidos a partir da coleta e análise de medidas estruturais de softwares desenvolvidos como projetos de software livre. O uso deste tipo de sistema nos estudos experimentais voltados à Evolução de Software é recorrente por conta da facilidade de acesso ao código-fonte dos sistemas, às atividades de manutenção que foram realizadas no contexto dos projetos (ainda que sem grande formalismo) e à inexistência de preocupações relacionadas com a propriedade intelectual do sistema sendo analisado.

No entanto, devido ao foco em sistemas desenvolvidos segundo a filosofia de software livre, pouco se conhece a respeito da evolução de sistemas de software desenvolvidos em ambiente industrial. Em geral, empresas desenvolvedoras e/ou detentoras de direitos autorais de sistemas de software não disponibilizam o código-fonte de suas aplicações ao domínio público e, frequentemente, são resistentes à distribuição destes mesmos artefatos a pesquisadores. Isto se deve, em grande parte, a implementações sigilosas contidas no código-fonte, que em alguns casos podem estar diretamente associadas a diferenciais estratégicos da empresa detentora ou a vantagens competitivas identificadas em seu nicho de mercado.

No capítulo 4 foi apresentada a CodeMI, uma representação XML destinada a descrever a estrutura do código-fonte de um software sem detalhar sua dinâmica de execução e sem expor o seu código-fonte original. A representação CodeMI viabiliza a coleta de métricas estruturais do código-fonte sem que seja necessário o acesso do

pesquisador aos detalhes da implementação. Assim, a CodeMI permite a distribuição dos aspectos relevantes para a análise de projetos de software no contexto de pesquisas relacionadas com Evolução de Software, sem infringir restrições impostas por direitos autorais. Com ela, acreditamos ser possível que empresas desenvolvedoras de software divulguem informações sobre sistemas desenvolvidos segundo seu modelo industrial, viabilizando pesquisas na área de Evolução de Software que tragam um melhor entendimento sobre a dinâmica do processo de desenvolvimento com que são construídos estes sistemas.

Para a realização de análises desta natureza, são necessárias coletas periódicas de métricas a partir do código-fonte ou de outros artefatos desenvolvidos ao longo de diversas versões dos sistemas sendo estudados. Diferentes tipos de pesquisa apresentarão necessidades próprias e conjuntos particulares de métricas de interesse. Armazenar apenas os valores das métricas ao longo do tempo inviabiliza pesquisas baseadas em outras métricas (desconhecidas e/ou desconsideradas no momento em que as coletas foram realizadas), exceto se os pesquisadores tiverem novamente acesso aos artefatos desenvolvidos.

Uma solução para este problema seria armazenar os próprios artefatos desenvolvidos ao longo das versões, garantindo o acesso aos artefatos para coleta dos valores das métricas necessárias a cada pesquisa no momento em que a pesquisa estiver definida. No entanto, as restrições de direitos autorais previamente citadas inviabilizam a manutenção dos artefatos componentes de sistemas industriais.

Para atender a esta necessidade, propomos o desenvolvimento de um ambiente capaz de armazenar artefatos desenvolvidos durante um projeto de software no formato CodeMI, para representar a estrutura do código-fonte de uma versão do sistema. O ambiente também deve suportar algumas das características essenciais à coleta de métricas estruturais a partir destes artefatos, facilitando a interação entre empresas fornecedoras das versões de CodeMI e pesquisadores da área de evolução de software que buscam entendimentos sobre sistemas comerciais/industriais.

5.2.1. Arquitetura do Ambiente

Um dos objetivos da Engenharia de Software é prover meios para melhorar a qualidade dos produtos de software e aumentar a produtividade do processo de desenvolvimento de sistemas. Para alcançar esta melhoria, métodos e ferramentas têm sido desenvolvidos por engenheiros de software para apoiar as atividades de desenvolvimento e manutenção de software [Pfleeger 1999].

Uma forma natural de apoiar as atividades supracitadas é prover ferramentas automatizadas para suportar a execução destas atividades. Entretanto, o uso de um grande conjunto de ferramentas independentes também traz dificuldades próprias, como integração de dados, integração de controle, homogeneidade na interface com o usuário, entre outras. Visando reduzir estas dificuldades, em especial pela integração e homogenização dos mecanismos de interação entre estas ferramentas, surgiu o conceito de Ambiente de Desenvolvimento de Software (ADS), cujo objetivo é prover um ambiente capaz de suportar todo o processo de desenvolvimento, com diversas ferramentas integradas trabalhando em conjunto.

ADSs são conjuntos de ferramentas CASE integradas que facilitam a realização de atividades de Engenharia de Software, apoiando todo o ciclo de vida de software [Harrison et al 2000]. Este mesmo conceito de ADSs pode ser aplicado a outros tipos de ambientes, como ambientes para realização de pesquisa. Neste contexto, o ambiente de suporte a pesquisas na área de Evolução de Software ora proposto deve fornecer mecanismos de armazenamento, visualização e conversão do código-fonte para a representação em CodeMI, que é o formato básico de análise deste ambiente de pesquisa. Neste ambiente também deve ser possível armazenar versões do software de diversos projetos, formando uma base histórica que permita a realização de estudos de evolução de software.

A viabilidade da coleta de métricas de quaisquer destas versões é outra característica desejável deste ambiente, permitindo o acompanhamento da evolução destas medidas. Estas métricas, coletadas durante a realização de pesquisas, devem ser armazenadas em um repositório de métricas de projetos para consultas futuras. Informações sobre artefatos de projetos de software devem ser providas por empresas desenvolvedoras de software já na representação CodeMI, evitando o *disclosure* do código-fonte e dificultando a reengenharia do software por uma outra empresa concorrente.

Pesquisadores utilizam-se destas informações para realização de estudos em evolução de software, podendo realizar pesquisas, estimativas e simulações. Além disso, novos mecanismos de coleta de métricas podem ser propostos, contribuindo para a expansão do ambiente e ampliação das séries históricas das medidas de projetos de software industriais. No entanto, a implementação deste ambiente figura como sugestão de trabalho futuro, ficando aqui realizado apenas um levantamento de requisitos.

Como proposta de arquitetura para a implementação do ambiente para a realização de estudos de evolução de software comercial/industrial baseado em representação de código-fonte CodeMI, foram idealizadas quatro camadas básicas de

componentes, conforme apresentado na Figura 5.1. São elas: (a) visualização; (b) conversão; (c) coleta; e (d) armazenamento;

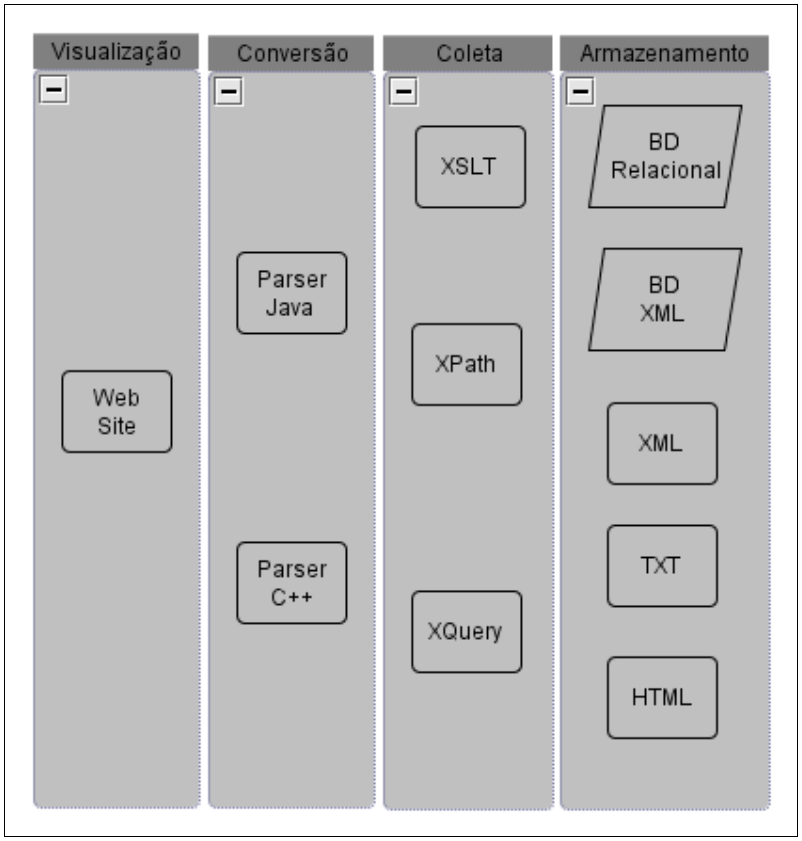

Figura 5.1. Arquitetura para ambiente de pesquisa baseado na CodeMI

A camada de visualização define a interface através da qual as empresas e os pesquisadores interagem com o ambiente e, consequentemente, com os artefatos das demais camadas. As empresas contribuem com versões de seus softwares em desenvolvimento para a realização de pesquisas, podendo beneficiar-se de métricas de sistemas de outras empresas participantes do ambiente de pesquisas.

A camada de conversão é composta por ferramentas que realizam o parser do código-fonte, transformando-o para a representação CodeMI. Para isto, o ambiente deve prover um *parser* (analisador sintático) para cada linguagem de programação orientada a objetos para a qual os estudos serão realizados. Este parser pode ser implementado na forma de um aplicativo ou de um plugin acoplado à *IDE* (*Integrated Development Environment*) utilizada pelos desenvolvedores de software.

A camada de coleta compreende ferramentas, tecnologias ou padrões de manipulação de arquivo XML (padrão subjacente ao padrão CodeMI), que permitam ao pesquisador obter as medidas estruturais que precisa utilizar em suas pesquisas.

A camada de armazenamento destina-se a guarda das representações CodeMI das versões de software em análise, bem como das métricas coletadas dos diversos projetos de software, além de toda e qualquer informação relativa às empresas e aos pesquisadores.

Além das atividades relacionadas na figura 5.2, alguns requisitos adicionais devem ser observados na confecção deste ambiente, como: (a) a construção do sistema deve adotar características baseada em web; e (b) restrições de segurança e controle de acesso. Com intuito de facilitar a integração das empresas com o ambiente e incentivá-las a contribuírem com versões de seus softwares na CodeMI, idealizamos o ambiente web como meio ideal para realizar esta interface, dada a facilidade de acesso. No entanto, questões de segurança e políticas de acesso devem ser estabelecidas a fim de evitar acessos não autorizados.

Outra vantagem que pode ser oferecida às empresas é a possibilidade de visualizar métricas coletadas de sistemas de outras empresas participantes deste ambiente. Com isto, podem ser realizadas comparações entre dois ou mais sistemas com funcionalidades semelhantes (benchmarking).

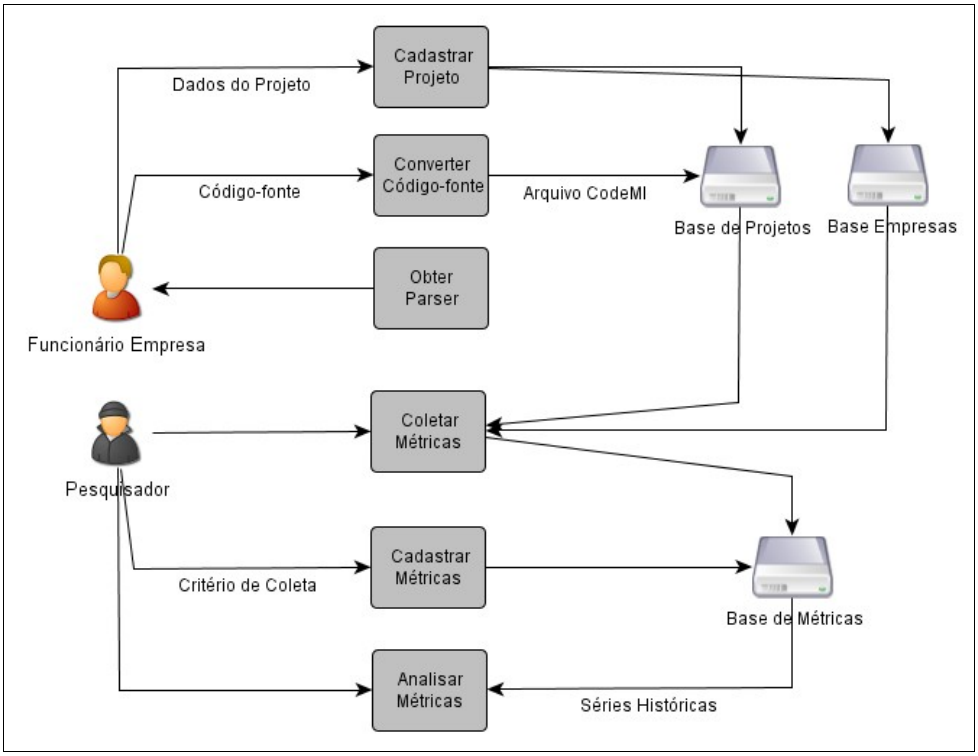

Figura 5.2. Fluxograma de Atividades do Ambiente.

Alguns destes requisitos serão abordados na próxima seção, que visa identificar os principais casos de uso que definem o ambiente proposto.

5.2.2. Casos de Uso

Para a manipulação do repositório de métricas, atendendo satisfatoriamente os requisitos mencionados na seção anterior, foram idealizados sete casos de uso:

- Cadastrar Projeto
- Cadastrar Métricas
- Obter Parser
- Converter Versão
- Submeter Versão
- Coletar Métricas da Versão
- Visualizar Métricas do Software (ao longo do tempo)

Como pode ser visto na figura 5.3, foram definidos dois atores para os casos de uso mapeados. O ator “Funcionário” é responsável pelas atividades que vão desde a recuperação do código-fonte do repositório de controle de uma versão do software da empresa até sua manipulação, conversão para CodeMI e carga no repositório de pesquisas. O ator “Pesquisador” é responsável pela coleta e análise das métricas dos softwares.

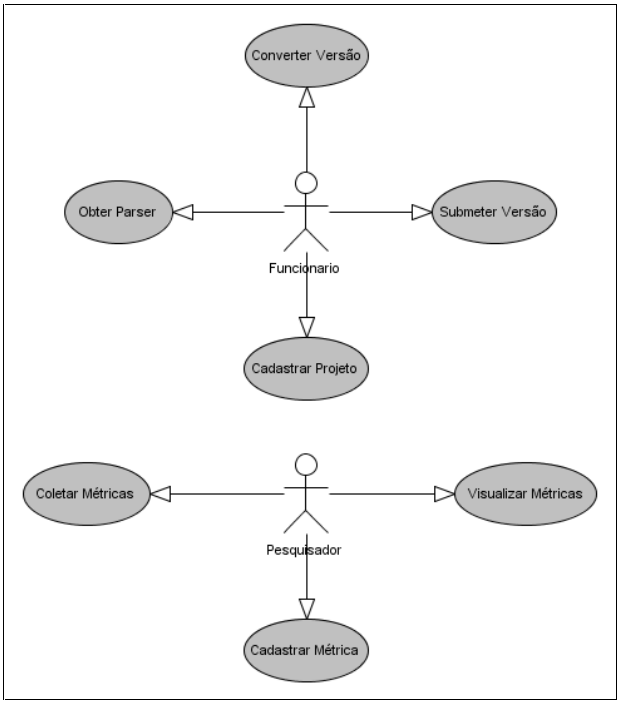

Figura 5.3. Casos de Uso do Repositório de Métricas

O caso de uso *Cadastrar Projeto* se propõe a realizar o cadastramento de um novo projeto, preparando o ambiente para o recebimento de versões do software na representação CodeMI. Este cadastramento é iniciado pelo Funcionário da empresa, que posteriormente é analisado e autorizado pelo Pesquisador.

O caso de uso *Cadastrar Métrica* destina-se a realizar o cadastramento das métricas passíveis de coleta a partir da CodeMI das versões dos softwares previamente cadastrados. Cada métrica deve possuir um método de consulta ou uma transformação associada, que defina como ela será coletada.

O caso de uso *Obter Parser* define as atividades de cópia/download do parser que será utilizado para converter uma versão do código-fonte do sistema em uma versão CodeMI passível de análise. Devendo, o Funcionário cópiar o parser adequado à linguagem de programação adotada pelo sistema em desenvolvimento.

O caso de uso *Converter Versão* especifica a conversão de uma versão do código-fonte do sistema para a representação CodeMI através da execução do parser sobre o código-fonte selecionado. Para que a conversão seja bem sucedida é necessário que o código-fonte não apresente erros de compilação.

O caso de uso *Submeter Versão* define as atividades e validações necessárias à cópia da uma versão CodeMI do software para o repositório de pesquisa.

O caso de uso *Coletar Métricas* tem por objetivo coletar métricas previamente cadastradas sobre uma versão do software previamente submetida. As métricas coletadas podem ser armazenadas visando a manutenção de séries históricas dos sistemas.

O caso de uso *Visualizar Métricas* tem a finalidade de viabilizar a visualização das métricas previamente coletadas do software e de suas séries históricas. Neste, também deve ser permitido ao Pesquisador selecionar quais métricas, sistemas e intervalo de tempo deseja-se analisar.

5.2.3. Benefícios do Ambiente na Realização de Pesquisas de ES

Com base nos benefícios de definição, coleta e armazenamento de métricas sobre a representação CodeMI oferecidos pelo ambiente, conforme abordado na seção 5.2.2, parte dos estudos de evolução de software poderiam ter sido auxiliados em sua confecção pela utilização do ambiente aqui proposto.

A realização de trabalhos similares aos realizados por Belady e Lehman [Belady e Lehman 1976], que se baseiam na observação do crescimento em tamanho e complexidade de sistemas, pode se beneficiar deste tipo de ambiente por meio da coleta de medidas estruturais, como LOCs e Complexidade Ciclomática, das versões

do software na representação CodeMI. Do mesmo modo, as observações realizadas por Cook e Roesch [Cook e Roesch 1994] poderiam se beneficiar da coleta das métricas de complexidade ciclomática de McCabe, entre outras, como ferramenta de auxílio a obtenção das series históricas que possibilitaram a sua pesquisa determinar o suporte do sistema de telefonia analisado às leis de evolução de software, principalmente às leis de Mudança Contínua, Incremento da Complexidade e Crescimento Contínuo.

A pesquisa realizada por Barros [Barros 2008] também poderia se beneficiar deste ambiente para facilitar a obtenção da distribuição de probabilidade, que descreve o tamanho de um projeto de software em um momento futuro. Deste modo seria possível realizar estimativas de tamanho de softwares comerciais/industriais.

A pesquisa realizada por Araújo [Araujo 2009] poderia se beneficiar do ambiente proposto como facilitador do processo de coleta das métricas de tamanho, periodicidade, complexidade, esforço e manutenibilidade em cada ciclo de manutenção, podendo também utilizar qualquer outra métrica estrutural para adaptar/evoluir seu modelo de formulações lógicas com intuito de avaliar a presença ou ausência de evidências sobre a aplicação das Leis de Lehman no sistema que está sendo analisado. Do mesmo modo, outros modelos lógicos poderiam ser propostos a partir das tendências apresentadas pelas métricas estruturais.

5.3. Considerações Finais

Neste capítulo apresentamos os principais trabalhos de Lehman e suas oito leis que tratam da evolução de software. Neste contexto foram identificados alguns trabalhos que utilizam métricas de software como base para proposições, análises, estimativas, entre outros resultados relacionados à dinâmica de evolução de um software em desenvolvimento ou manutenção.

Em seguida, relacionamos alguns trabalhos de destaque nesta área e idealizamos uma arquitetura para subsidiar a confecção deste tipo de experimento, ou seja, análises baseadas na coleta de métricas estruturais de código-fonte ao longo do tempo,utilizando para isto, artefatos na representação CodeMI.

A utilização da *CodeMI* como base na elaboração de pesquisas em Evolução de Software dispensa a necessidade de acessar sucessivamente os repositórios de código-fonte, bastando para isso consultar versão CodeMI previamente coletada.

Este ambiente foi descrito em termos de seus principais componentes, funcionalidades e casos de uso. Finalmente foi apresentada uma análise sobre os benefícios trazidos por este ambiente na elaboração de pesquisas em evolução de

software, sobretudo a estudos baseados na estrutura de softwares comerciais. A implementação deste ambiente é deixada como sugestão de trabalho futuro.

CAPÍTULO 6

CONCLUSÕES

6.1. Considerações Finais

O trabalho realizado focou aspectos de coleta de métricas a partir de representações de código-fonte, enfatizando métricas relacionadas à estrutura do software como forma de auxiliar a realização de pesquisas na área de evolução de softwares do segmento comercial/industrial.

Dificuldades de acesso ao código-fonte de Sistemas de Informação, impostas por empresas desenvolvedoras ou consumidoras de software em função de proteções de direitos autorais e/ou de propriedade intelectual dificultam e, em alguns casos, impossibilitam a realização de estudos de evolução de software deste segmento (comercial/industrial).

A realização de pesquisas em evolução de softwares comerciais demanda a criação de um repositório de dados onde a representação de código-fonte tenha um papel central, preservando as características necessárias para a realização de estudos de evolução de software, ao mesmo tempo em que esconda os detalhes do código-fonte, de modo a não infringir a propriedade intelectual das empresas que o desenvolveram.

Dentro deste contexto, as contribuições deste trabalho de mestrado foram as seguintes:

- A definição de um conjunto de marcadores para representar a estrutura do código-fonte de softwares orientados a objetos;
- A formulação da CodeMI – Source Code as XMI, que é uma representação de código-fonte independente de linguagem de programação e que permite a extração de métricas estruturais e que não evidencie os segredos industriais refletidos na lógica do código-fonte, até então não encontrada no material teórico pesquisado;

- A definição de métricas estruturais como sendo um tipo de métrica de produto possível de ser coletada a partir dos elementos estruturais do código-fonte;
- A criação de uma ferramenta *parser*, a título de exemplo, com a finalidade de converter um código-fonte Java para a representação CodeMI;
- A elaboração de transformações XSLT para a coleta de métricas estruturais, sobre a representação CodeMI, das suítes de Lorenz & Kidd, Chidamber & Kemerer, Complexidade Ciclomática, entre outras variações;
- A avaliação do objetivo da representação proposta (CodeMI) por meio da realização de uma prova de conceito, que consistiu em converter uma versão completa do software Compiere (vide DVD anexo) para a CodeMI e em seguida foram aplicadas transformações XSLT a fim de coletar métricas a partir da representação CodeMI do software Compiere;
- A proposta de arquitetura de um ambiente de suporte à realização de pesquisas sobre evolução de softwares comerciais, viabilizando a coleta e análise de medidas de diferentes versões de um ou mais sistemas.

A representação CodeMI foi implementada em conformidade com os quatro critérios estabelecidos como essenciais à extração de métricas do código-fonte de sistemas comerciais/industriais. A aderência a estes critérios possibilitou à CodeMI:

- Ser uma representação genérica, ou seja, independente de linguagem de programação, porém destinada a representar somente linguagens de programação orientada a objetos;
- Possuir granularidade em nível de pacote, ou seja, após o processo de conversão é gerado um arquivo para cada pacote do sistema;
- Apresentar baixa verbosidade devido ao seu enxuto conjunto de elementos e por associar cada linha do código-fonte a um único marcador;
- Apresentar baixa exposição do código-fonte tendo em vista a impossibilidade de extração do código-fonte original do sistema a partir de um arquivo na CodeMI.

A utilização de transformações XSLT permitiu a coleta de métricas a partir da representação CodeMI, conforme a prova de conceito realizada no capítulo 4. Este mecanismo mostrou-se satisfatório para a coleta das métricas estruturais apresentadas em nível de método, classe e pacote. No entanto, este mecanismo não viabiliza a coleta de métricas em nível de projeto, visto que os arquivos da CodeMI são

gerados por pacotes e que a XSLT não fornece suporte a realização da operação de *junção* típica dos bancos de dados para a coleta de métricas em vários pacotes [Santos & Barros 2009].

A utilização de transformações XSLT que realizem operações muito complexas para coletar determinada métrica pode dificultar a realização deste processo em softwares com grandes volumes de código-fonte. Podendo ser agravado, caso sejam utilizadas em estudos de evolução de software, onde é necessário realizar extração medidas das diferentes versões do software ao longo do tempo, aumentando significativamente o volume de código-fonte a ser analisado e o tempo de processamento.

6.2. Perspectivas Futuras

Como perspectivas de trabalhos futuros, destacamos a realização de atividades que subsidiem a realização de trabalhos em evolução de software com base em dados obtidos a partir de repositórios de softwares industriais.

Inicialmente, a principal atividade a ser desempenhada futuramente é implementar o ambiente proposto no capítulo 5, de modo a facilitar a interação entre pesquisadores e empresas, permitindo que estas contribuam com versões de seus software na CodeMI, possibilitando aos pesquisadores selecionarem entre as métricas disponíveis, quais serão coletadas e analisadas para o software ao longo do tempo.

A partir deste ambiente, torna-se viável a manutenção das séries históricas de medidas coletadas de diversos sistemas, que podem contribuir na confecção de diversos tipos de estudos de evolução de software, como realização de estimativas e projeções de métricas.

Neste contexto, a construção de plugins para integração em Ambientes de Desenvolvimento de Software, como o Eclipse, figura como sugestão de ferramenta a ser implementada futuramente para facilitar o processo de conversão do código-fonte pelas empresas. Podendo também ser desenvolvido para outras linguagens de programação orientada a objetos como Delphi, C++, entre outras.

Embora tenham sido utilizadas consultas XSLT na coleta de métricas a partir da CodeMI realizada no capítulo 4, outras formas de coleta de métricas podem ser mais exploradas como, a utilização da XQuery. O mesmo aplica-se às formas de armazenamento, podendo-se explorar um pouco mais a utilização de bancos de dados XML.

Junto a estas perspectivas, posso dizer que do ponto de vista pessoal, a realização deste trabalho me fez adquirir experiência e grande aprendizado sobre

representações de código-fonte baseado no formato XML, focando na extração de métricas para a realização de pesquisas em Evolução de Software. Além disso, o interesse despertado pela área me motiva a continuar buscando novos conhecimentos sobre o assunto, mantendo o engajamento nas atividades acadêmicas.

Para o nosso grupo de pesquisa, deixamos concretizada uma pequena parcela rumo à construção de um laboratório/ambiente de análises temporais de software, idealizado para a realização de novas pesquisas e replicação de estudos já realizados. Do mesmo modo, disponibilizamos à comunidade este ferramental construído para subsidiar a realização de pesquisas baseadas em análises de medidas de código-fonte de sistemas ao longo de intervalo de tempos.

Referências Bibliográficas

[Aguiar 2004] Aguiar A., David G., Badros G. J., “JavaML 2.0: Enriching the Markup Language for Java Source Code”, in XML: Aplicações e Tecnologias Associadas (XATA’2004), Porto, Portugal, Fevereiro de 2004.

[Araujo 2009] Araújo, Marco Antônio Pereira Um Modelo para Predição de Decaimento em Software Orientado a Objetos / Marco Antônio Pereira Araújo. – Rio de Janeiro: UFRJ/COPPE, 2009.

[Badros 2000] Badros G. J., “JavaML: A Markup Language for Java Source Code”, In Proceedings of the 9th Int. Conf. On the World Wide Web (WWW9), Amsterdam, Netherlands, May 2000.

[Barros 2008] Barros, M. O., Predicting Software Project Size Using Project Generated Information. SEKE 2008.

[Booch 1993] Booch Grady. Object oriented Analysis and Design with Applications (2nd ed. Ed). Redwood City: Benjamin Cummings. 1993.

[BoUML 2002] BoUML. A UML 2 tool box 2002. Home Page disponível em: http://bouml.free.fr. Último acesso em: 08/02/2009.

[Chidamber & Kemerer 1994] Chidamber, S. e Kemerer, C. *A Metrics Suite for Object Oriented Design*. In IEEE Transaction on Software Engineering, Volume 20, N. 6 pp. 476-493, June 1994.

[Compiere 2009] Compiere Open Source ERP and CRM Business Solution. Home Page disponível em: http://www.compiere.com

[CWM 2005] Common Warehouse Metamodel (CWM) Specification, version 1.1. Object Management Group.

[Dreger 1989] Dreger, J. Brian, Function Point Analysis, Prentice Hall, Englewood Cliffs, NJ, 1989.

[Fenton & Pfleeger 1997] Fenton, N.; Pfleeger, S. Software Metrics: A Rigorous and Practical Approach. 2.ed. London: PWS, 1997.

[Forrester 1991] Forrester, J.W., 1991, System Dynamics and the Lessons of 35 Years, Technical Report D-4224-4, MIT System Dynamics Group, Cambridge, MA.

[Gall et al 1998] Harald Gall, Karin Hajek and Mehdi Jazayeri, “Detection of Logical Coupling Based on Product Release History”, presented at the Proceedings of the International Conference on Software Maintenance (ICSM), September 1998.

[Godfrey & Tu 2000] M. W. Godfrey and Q. Tu, “Evolution in Open Source Software: A Case Study,” Proceedings of International Conference on Software Maintenance (ICSM'00), 2000.

[Halstead 1977] M. H. Halstead, “Elements of Software Science,” Elsevier North-Holland, New York, 1977.

[Harrison et al 2000] W. Harrison, H. Ossher, P. Tarr, “Software Engineering Tools and Environments: A Roadmap”, in Proc. of the Future of Software Engineering, ICSE’2000, Ireland, 2000.

[Holt 2000] Holt, R. C., Winter, A., Schürr, A. “GXL: Toward a Standard Exchange Format”, in Proc. of the 7th Working Conf. on Reverse Engineering (WCRE’00), Brisbane, Australia, pp. 162–171, November 2000.

[Howison & Crowston 2004] J. Howison and K. Crowston, “The perils and pitfalls of mining sourceforge,” in Proc. of Workshop on Mining Software Repositories at the International Conference on Software Engineering ICSE, 2004.

[IBM 1998] IBM Jikes Compiler 1998. Home Page disponível em: http://jikes.sourceforge.net. Último acesso em 24/01/2009.

[Jacobson 1992] Jacobson, Ivar. Object Oriented Software Engineering: A Case Use Driven Approach. Julho, 1992.

[JavaCC 2003] JavaCC, Java Compiler Compiler Home Page. Disponível em: http://javacc.dev.java.net. Último acesso em 23/01/2009.

[Khun 2001] Khun Yeee Fung. XSLT Interagindo com XML e HTML, 1ed, Ciência Moderna, 409p, 2001.

[Kitchenham et al 1995] Kitchenham, B.; Pfleeger, S.; Fenton, N. Towards a Framework for Software Measurement Validation. IEEE Transactions on Software Engineering. December 1995, 21 (12), pp. 929-943.

[LangPop 2009] LangPop. Programming Language Popularity Home Page. Disponível em: http://www.langpop.com. Último acesso em: 03/01/2009.

[Levine 1992] Levine J. R., Lex & YACC. O’Reilly & Associates, Inc., Sebastopol, California, 2nd edition, 1992.

[Lisboa Filho et al. 1999] Lisboa Filho, J. and Lochpe, C. (1999), “Specifying analysis patterns for geographic databases on the basis of a conceptual framework”. In Proc.7th ACM GIS, Kansas City.

[Lorenz & Kidd 1994] Lorenz, M. e Kidd J. *Object-Oriented Software Metrics, A Practical Guide*. Englewood Cliffs, N.J.: PTR Prentice-Hall, 1994.

[MagicDraw 2009] UML 2 diagramming, OO software modeling, Source code engineering tool Magic Draw UML from No Magic. Home Page disponível em: http://www.magicdraw.com. Último acesso em: 12/04/2009.

[Maletic 2002] Maletic, J.I., Collard, M.L. and Marcus, A., “Source Code Files as Structured Documents”, in Proc. of the10th Int. Workshop on Program Comprehension (IWPC ’02), Paris, France, pp. 289–292, June, 2002.

[Mamas & Kontogiannis 2000] Mamas, E. and Kontogiannis, C., “Towards Portable Source Code Representations Using XML”, in Proc, of the 7th Working Conf. on Reverse Engineering (WCRE’00), Brisbane, Australia, pp. 172-18, November 2000.

[Matula 2005] Matula, M., 2005, “NetBeans Metadata Repository”. Whitepaper. In: http://mdr.netbeans.org/ ,acessado em 06/04/2005.

[McCabe 1976] McCabe, T.J., “A Complexity Measure”, IEEE Transactions on Software Engineering, 1976.

[Mendonça et al 2004] Mendonça, N. C., Maia, P. H. M., Fonseca, L. A., and Andrade, R. M. C. (2004), “RefaX: A Refactoring Framework Based on XML”, In 20th. IEEE International Conference on Software Maintenance (ICSM 2004), pages 147-156, IEEE Computer Society.

[Metrics 2009] Eclipse Metrics Plugin. Home Page disponível em: http://www.metrics.sourceforge.net. Último acesso em: 30/10/2009.

[MOF 2002] Meta Object Facility Specification, Version 1.4. Object Management Group 2002a.

[Myers 1977] Glenford J. Myers, “An Extension to the Cyclomatic Measure of Program Complexity”. Sigplan Notices, vol 12, no. 10, pp. 61-64, 1977.

[NetBeans 2008] Metamodel for Java Language. In: http://java.netbeans.org/models/java/java-model.html, acessado em 06/04/2008.

[Notepad 2007] XML Notepad 2007. XML Developer Center in: http://msdn.microsoft.com/pt-br/xml/default(en-us).aspx.

[Pressman 2006] Pressman, Roger S., Engenharia de Software, 6 ed., McGraw-Hill, 2006.

[Pfleeger 1999] Pfleeger, S.L. (1999) “Albert Einstein and Empirical Software Engineering”, IEEE Computer, October.

[Raymond 1999] E. S. Raymond. The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary. O’Reilly and Associates, Oct 1999.

[Rumbaugh 1997] Rumbaugh, James. OMT Insights: Perspective on Modeling from the Journal of Object-Oriented Programming. Dezembro, 1997.

[Santos & Barros 2009] João Paulo O. Santos, Márcio de O. Barros. Source Code as XMI. Uma Representação Estrutural de Código-fonte para Coleta de Métricas.SBSI 2009.

[Seguin 2000] Seguin, C., JRefactory Home. ACM, 2000. Disponível em: http://jrefactory.sourceforge.net

[SourceForge 2009] SorceForge Web Site , http://sourceforge.net

[XJava 2008] XJava 1.1. BeautyJ Home Page. Disponível em: http://beautyj.berlios.de/beautyJ.html. Acessado em: 11/06/2008.

[Toffano et al 2005] Angélica Toffano Seidel Calazans, Marcelo Antonio Lopes de Oliveira. Avaliação de Estimativa de Tamanho para Projetos de Manutenção de Software. Proc. of Argentine Symposium on Software Engineering, 2005.

[TortoiseSVN 2009] TortoiseSVN Home Page. Disponível em: http://tortoisesvn.tigris.org

[UML 2009] UML 2.2. Unified Modeling Language. Version 2.2. OMG Document Number formal/2009-02-04. Disponível em: http://www.omg.org/spec/UML/2.2/

[W3C 2009] World Wide Web Consortium – Web standards. Acesível em: http://www.w3.org/

[XMI 2007] XMI 2.1. XML Metadata Interchange Specification, Version 2.1. OMG Document Number: formal/2007-12-01. Disponível em: http://www.omg.org/spec/XMI/2.1/PDF

[XML 2006] XML. Extensible Markup Language (XML). W3C Recommendation 16 august 2006. Disponível em: http://www.w3.org/TR/REC-xml/

[XPath 1999] XPath 1.0: XML Path Language. W3C Recommendation 16, November 1999. Disponível em: http://www.w3.org/TR/xpath

[XQuery 2007] XQuery 1.0: An XML Query Language. W3C Recommendation 23, January 2007. Disponível em: http://www.w3.org/TR/xquery/

[XSLT 2001] XSLT 2.0: Extensible Stylesheet Language for Transformation. W3C Working Draft 14 February 2001. Disponivel em: http://w3.org/TR/xslt20req

[Zimmermann & WeiBgerber 2004] Thomas Zimmermann and Peter WeiBgerber, “Processing CVS data for Fine- Grained Analysis”, presented at International Workshop on Mining Software Repositories (MSR), May 2004

Apêndice A

Este apêndice tem como objetivo descrever a transformação XSLT aplicada na obtenção das métricas da suíte de Lorenz & Kidd. O conteúdo pode ser encontrado no arquivo Lorenz_Kidd.xsl, armazenado no DVD em anexo.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns:fo="http://www.w3.org/1999/XSL/Format"
   xmlns:uml="http://schema.omg.org/spec/UML/2.1"
   xmlns:xmi="http://schema.omg.org/spec/XMI/2.1">
      <xsl:template match="uml:Model">
         <html>
         <body>
         <h1>Suite de Métricas de Lorenz e Kidd </h1>
         <p> LOC  - Linhas de Código</p>
         <p> NCM  - Métodos de Classe numa Classe</p>
         <p> NCA  - Atributos de Classe numa Classe</p>
         <p> ANCA - Média dos Atributos de Classe numa Classe</p>
         <p> NIM  - Métodos de Instância numa Classe</p>
         <p> ANIM - Média dos Métodos de Instância numa Classe</p>
         <p> NIA  - Atributos de Instância numa Classe</p>
         <p> ANIA - Média dos Atributos de Instância numa Classe</p>
         <p> NV   - Atributos da Classe (NCA + NIA)</p>
         <p> NM   - Métodos da Classe (NCM + NIM)</p>
         <p> PIM  - Métodos Públicos de Instância numa Classe </p>
         <p> PPM  - Parâmetros por Método </p>

         <xsl:for-each select="packagedElement">
            <h2>Metricas do Pacote: <xsl:value-of select="./@name"/>
            </h2>
            <table border="1">
               <tr bgcolor="lightgreen">
                  <th align="left">Total de Classes</th>
                  <th align="left">Total de Metodos</th>
                  <th align="left">Total de Comandos</th>
                  <th align="left">ANIM</th>
                  <th align="left">ANIA</th>
                  <th align="left">ANCA</th>
                  <th align="left">ANCM</th>
               </tr>
               <tr>
                  <td>
                     <xsl:value-of select="count(packagedElement)"/>
                  </td>
                  <td>
                     <xsl:value-of
                        select="count(packagedElement/ownedOperation)
                        "/>
                  </td>
                  <td>
                     <xsl:value-of select="count(//statement)"/>
                  </td>
                  <td>
                     <xsl:value-of
```

```
                select="round((count(packagedElement/
                ownedOperation[@visibility!='static']) div
                count(packagedElement)) * 100) div 100"/>
            </td>
            <td>
               <xsl:value-of
                select="round((count(packagedElement/
                ownedAttribute[@visibility='private' or
                @visibility='protected']) div
                count(packagedElement)) * 100) div 100 "/>
            </td>
            <td>
               <xsl:value-of
                select="round((count(packagedElement/
                ownedAttribute[@visibility='static']) div
                count(packagedElement)) * 100) div 100"/>
            </td>
            <td>
               <xsl:value-of
                select="round((count(packagedElement/
                ownedOperation[@visibility='static']) div
                count(packagedElement)) * 100) div 100"/>
            </td>
         </tr>
      </table>
      </xsl:for-each>
         <h2>Classes do Pacote</h2>
         <table border="1">
            <tr bgcolor="lightgreen">
               <th align="left">Nome da Classe</th>
               <th align="left">LOC/NM</th>
               <th align="left">PIM</th>
               <th align="left">NIM</th>
               <th align="left">NIA</th>
               <th align="left">NCM</th>
               <th align="left">NCA</th>
               <th align="left">NV</th>
               <th align="left">NM</th>
               <th align="left">LOC</th>
            </tr>
            <xsl:for-each
               select="packagedElement/packagedElement">
               <tr>
                  <td>
                     <xsl:value-of select="./@name"/>
                  </td>
                  <td>
                     <xsl:value-of
                        select="round((count(ownedOperation/
                        xmi:extension//*) div
                        count(ownedOperation)) * 100)
                        div 100 "/>
                  </td>
                  <td>
                     <xsl:value-of
                        select="count(ownedOperation[@visibility
                        ='public'])"/>
                  </td>
                  <td>
                     <xsl:value-of
                        select="count(ownedOperation[@visibility
```

```
                        !='static'])"/>
                    </td>
                    <td>
                       <xsl:value-of
                          select="count(ownedAttribute[
                          @visibility='private' or
                          @visibility='protected'])"/>
                    </td>
                    <td>
                       <xsl:value-of
                          select="count(ownedOperation[
                          @visibility='static'])"/>
                    </td>
                    <td>
                       <xsl:value-of
                          select="count(ownedAttribute[
                          @visibility='static'])"/>
                    </td>
                    <td>
                       <xsl:value-of
                          select="count(ownedAttribute)"/>
                    </td>
                    <td>
                       <xsl:value-of
                          select="count(ownedOperation)"/>
                    </td>
                    <td>
                       <xsl:value-of
                          select="count(ownedOperation/
                          xmi:extension//*)"/>
                    </td>
                 </tr>
              </xsl:for-each>
           </table>

           <xsl:for-each select="packagedElement/packagedElement">
           <h2>Classe: <xsl:value-of select="./@name"/></h2>
           <table border="1">
              <tr bgcolor="lightgreen">
                 <th align="left">Nome do MÃ©todo</th>
                 <th align="left">LOC</th>
                 <th align="left">PPM</th>
              </tr>
              <xsl:for-each select="ownedOperation">
                 <tr>
                    <td>
                       <xsl:value-of select="./@name"/>
                    </td>
                    <td>
                       <xsl:value-of
                          select="count(xmi:extension//*)"/>
                    </td>
                    <td>
                       <xsl:value-of
                          select="count(ownedParameter)"/>
                    </td>
                 </tr>
              </xsl:for-each>
           </table>
        </xsl:for-each>
     </body>
```

```
        </html>
      </xsl:template>
</xsl:stylesheet>
```

Apêndice B

Este apêndice tem como objetivo descrever a transformação XSLT aplicada na obtenção das métricas da suíte de Chidamber & Kemerer. O conteúdo pode ser encontrado no arquivo Chidamber_Kemerer.xsl, armazenado no DVD em anexo.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns:fo="http://www.w3.org/1999/XSL/Format"
   xmlns:uml="http://schema.omg.org/spec/UML/2.1"
   xmlns:xmi="http://schema.omg.org/spec/XMI/2.1">
      <xsl:template match="uml:Model">
         <html>
         <body>
         <h1>Suite de Metricas de Chidamber e Kemerer</h1>
         <p> WMC - Métodos Ponderados por Classe</p>
         <p> NOC - Número de Filhos</p>

         <xsl:for-each select="packagedElement">
            <h2>Metricas do Pacote: <xsl:value-of select="./@name"/>
            </h2>
            <table border="1">
               <tr bgcolor="lightgreen">
                  <th align="left">Total de Classes</th>
                  <th align="left">Total de Metodos</th>
               </tr>
               <tr>
                  <td>
                     <xsl:value-of select="count(packagedElement)"/>
                  </td>
                  <td>
                     <xsl:value-of
                        select="count(packagedElement/ownedOperation
                        )"/>
                  </td>
               </tr>
            </table>
         </xsl:for-each>
         <h2>Classes do Pacote</h2>
         <table border="1">
            <tr bgcolor="lightgreen">
               <th align="left">Nome da Classe</th>
               <th align="left">WMC</th>
               <th align="left">NOC</th>
            </tr>
            <xsl:for-each select="packagedElement/packagedElement">
               <tr>
                  <td>
                     <xsl:value-of select="./@name"/>
                  </td>
                  <td>
                     <xsl:value-of
                        select="(count(ownedOperation/xmi:extension//
                        if) +
                        count(ownedOperation/xmi:extension//for) +
```

```
                        count(ownedOperation/xmi:extension//while) +
                        count(ownedOperation/xmi:extension//case) +
                        count(ownedOperation))"/>
                  </td>
                  <td>
                     <xsl:variable name="id" select="./@xmi:id" />
                     <xsl:value-of
                        select="count(//packagedElement/
                        packagedElement/generalization[
                        @general = $id])"/>
                  </td>
               </tr>
            </xsl:for-each>
         </table>
         </body>
         </html>
      </xsl:template>
</xsl:stylesheet>
```

Apêndice C

Este apêndice tem como objetivo descrever a transformação XSLT aplicada na obtenção das métricas de Complexidade Ciclomática. O conteúdo pode ser encontrado no arquivo Complexidade_Ciclomatica.xsl, armazenado no DVD em anexo.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0"
   xmlns:xsl="http://www.w3.org/1999/XSL/Transform"
   xmlns:fo="http://www.w3.org/1999/XSL/Format"
   xmlns:uml="http://schema.omg.org/spec/UML/2.1"
   xmlns:xmi="http://schema.omg.org/spec/XMI/2.1">
      <xsl:template match="uml:Model">
         <html>
         <body>
         <h1>Suite de MÃ©tricas de Complexidade CiclomÃ¡tica</h1>
         <p> CC  - Complexidade CiclomÃ¡tica de McCabe</p>
         <p> CCM - Complexidade CiclomÃ¡tica de Myers</p>

         <xsl:for-each select="packagedElement">
            <h2>Metricas do Pacote: <xsl:value-of select="./@name"/>
            </h2>
            <table border="1">
               <tr bgcolor="lightgreen">
                  <th align="left">Total de Classes</th>
                  <th align="left">Total de Metodos</th>
                  <th align="left">Total de Comandos</th>
                  <th align="left">C.C. MÃ©dia</th>
                  <th align="left">C.C. Total</th>
               </tr>
               <tr>
                  <td>
                     <xsl:value-of select="count(packagedElement)"/>
                  </td>
                  <td>
                     <xsl:value-of
                        select="count(packagedElement/
                        ownedOperation)"/>
                  </td>
                  <td>
                     <xsl:value-of select="count(//statement)"/>
                  </td>
                  <td>
                     <xsl:value-of
                        select="round(((count(packagedElement/
                        ownedOperation/xmi:extension//if)  +
                        count(packagedElement/ownedOperation/
                        xmi:extension//for) +
                        count(packagedElement/ownedOperation/
                        xmi:extension//while)  +
                        count(packagedElement/ownedOperation/
                        xmi:extension//case) +
                        count(packagedElement/ownedOperation)) div
                        count(packagedElement/ownedOperation)) * 100)
```

```
                div 100"/>
            </td>
            <td>
               <xsl:value-of
                  select="(count(packagedElement/ownedOperation/
                  xmi:extension//if) +
                  count(packagedElement/ownedOperation/
                  xmi:extension//for) +
                  count(packagedElement/ownedOperation/
                  xmi:extension//while) +
                  count(packagedElement/ownedOperation/
                  xmi:extension//case) +
                  count(packagedElement/ownedOperation))"/>
            </td>
         </tr>
      </table>
   </xsl:for-each>
   <h2>Classes do Pacote</h2>
   <table border="1">
      <tr bgcolor="lightgreen">
         <th align="left">Nome da Classe</th>
         <th align="left">CC MÃ©dia</th>
         <th align="left">CC Total</th>
         <th align="left">CCM MÃ©dia</th>
         <th align="left">CCM Total</th>
      </tr>
      <xsl:for-each select="packagedElement/packagedElement">
         <tr>
            <td>
               <xsl:value-of select="./@name"/>
            </td>
            <td>
               <xsl:value-of
                  select="round(((count(ownedOperation
                  /xmi:extension//if) +
                  count(ownedOperation/xmi:extension//for) +
                  count(ownedOperation/xmi:extension//while) +
                  count(ownedOperation/xmi:extension//case) +
                  count(ownedOperation)) div
                  count(ownedOperation)) * 100) div 100"/>
            </td>
            <td>
               <xsl:value-of
                  select="(count(ownedOperation/
                  xmi:extension//if) +
                  count(ownedOperation/xmi:extension//for) +
                  count(ownedOperation/xmi:extension//while) +
                  count(ownedOperation/xmi:extension//case) +
                  count(ownedOperation))"/>
            </td>
            <td>
               <xsl:variable name="cc"
                  select="round((count(ownedOperation/
                  xmi:extension//if) +
                  count(ownedOperation/xmi:extension//for) +
                  count(ownedOperation/xmi:extension//while) +
                  count(ownedOperation/xmi:extension//case) +
                  count(ownedOperation)) div
                  count(ownedOperation) * 100) div 100" />
               <xsl:variable name="ccm"
                  select="round((sum(ownedOperation/
```

```
                  xmi:extension//if/@conditions) +
                  sum(ownedOperation/
                  xmi:extension//for/@conditions) +
                  sum(ownedOperation/
                  xmi:extension//while/@conditions) +
                  sum(ownedOperation/
                  xmi:extension//case/@conditions) +
                  count(ownedOperation)) div
                  count(ownedOperation) * 100) div 100" />
               <xsl:value-of select="concat($cc,':',$ccm)"/>
            </td>
            <td>
               <xsl:variable name="cc"
                  select="count(ownedOperation/xmi:extension
                  //if) +
                  count(ownedOperation/xmi:extension//for) +
                  count(ownedOperation/xmi:extension//while) +
                  count(ownedOperation/xmi:extension//case) +
                  count(ownedOperation)" />
               <xsl:variable name="ccm"
                  select="sum(ownedOperation/
                  xmi:extension//if/@conditions) +
                  sum(ownedOperation/
                  xmi:extension//for/@conditions) +
                  sum(ownedOperation/
                  xmi:extension//while/@conditions) +
                  sum(ownedOperation/
                  xmi:extension//case/@conditions)  +
                  count(ownedOperation)" />
               <xsl:value-of select="concat($cc,':',$ccm)"/>
            </td>
         </tr>
      </xsl:for-each>
   </table>
   <xsl:for-each select="packagedElement/packagedElement">
      <h2>Classe: <xsl:value-of select="./@name"/></h2>
      <table border="1">
         <tr bgcolor="lightgreen">
            <th align="left">Nome do MÃ©todo</th>
            <th align="left">CC</th>
            <th align="left">CCM</th>
         </tr>
         <xsl:for-each select="ownedOperation">
            <tr>
               <td>
                  <xsl:value-of select="./@name"/>
               </td>
               <td>
                  <xsl:value-of
                     select="count(xmi:extension//if) +
                             count(xmi:extension//for) +
                             count(xmi:extension//while) +
                             count(xmi:extension//case) + 1" />
               </td>
               <td>
                  <xsl:variable name="cc"
                     select="count(xmi:extension//if) +
                     count(xmi:extension//for) +
                     count(xmi:extension//while) +
                     count(xmi:extension//case) + 1" />
                  <xsl:variable name="ccm"
```

```
                              select="sum(xmi:extension//
                              if/@conditions) +
                              sum(xmi:extension//for/@conditions) +
                              sum(xmi:extension//while/@conditions) +
                              sum(xmi:extension//case/@conditions) + 1"
                              />
                           <xsl:value-of select="concat($cc,':',$ccm)"/>
                        </td>
                     </tr>
                  </xsl:for-each>
               </table>
            </xsl:for-each>
            </body>
            </html>
......</xsl:template>
</xsl:stylesheet>
```