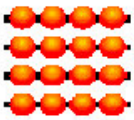
UNIVERSIDADE FEDERAL DO CEARÁ
DEPARTAMENTO DE COMPUTAÇÃO
MESTRADO EM CIÊNCIA DA COMPUTAÇÃO



PAULO HENRIQUE MENDES MAIA

REFAX: UM ARCABOUÇO PARA DESENVOLVIMENTO DE FERRAMENTAS DE REFATORAÇÃO BASEADO EM XML

FORTALEZA
2004

PAULO HENRIQUE MENDES MAIA

REFAX: UM ARCABOUÇO PARA DESENVOLVIMENTO DE FERRAMENTAS DE REFATORAÇÃO BASEADO EM XML

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal do Ceará, como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

Orientador: Prof. Nabor das Chagas Mendonça
Co-Orientadora: Profa. Rossana Maria de Castro Andrade

FORTALEZA
2004

PAULO HENRIQUE MENDES MAIA

REFAX: UM ARCABOUÇO PARA DESENVOLVIMENTO DE FERRAMENTAS DE REFATORAÇÃO BASEADO EM XML

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Ciência da Computação da Universidade Federal do Ceará, como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

APROVADA EM __/__/_____

BANCA EXAMINADORA

Prof. Dr. Nabor das Chagas Mendonça (Orientador)
Universidade de Fortaleza – UNIFOR

Profa. Dra. Rossana Maria de Castro Andrade (Co-Orientadora)
Universidade Federal do Ceará – UFC

Prof. Dr. Javam de Castro Machado
Universidade Federal do Ceará – UFC

Prof. Dr. Paulo César Masiero
Universidade de São Paulo – ICMC – USP

Aos meus pais, Paulo de Társio e Valquíria.

AGRADECIMENTOS

Primeiramente, a Deus, por estar sempre presente em minha vida.

Á toda minha família, em especial aos meus pais, Paulo de Társio e Valquíria, e aos meus irmãos Paula Virgínia e Paulo Emílio, por todo amor e apoio que me deram ao longo da minha vida, principalmente durante o Mestrado, vibrando com minhas conquistas e se orgulhando com meus méritos. Amo vocês.

À Rachel, por todo amor, carinho e compreensão para comigo durante esses dois últimos anos, e por entender o quão importante pra mim é conseguir esse título. Também à Cecília, por ser minha razão para ir cada vez mais. Amo vocês também.

Ao professor Nabor das Chagas Mendonça e à professora Rossana Andrade, pela competência demonstrada na orientação deste trabalho e confiança depositada em mim, e pela certeza que as madrugadas adentradas escrevendo artigos valeriam a pena. Hoje sabemos que o esforço foi compensado. Obrigado!

Ao amigo Leonardo Fonseca, por toda ajuda e aprendizado que me foi dado desde o começo do trabalho. Boa sorte!

A Elizardo Medeiros, pelo apoio e compreensão demonstrados desde que entrei para o Sistema Verdes Mares, e aos meus colegas de trabalho, por me ajudarem nessa conquista. Valeu galera!

Aos companheiros da Turma 2002 do Mestrado em Ciência da Computação da UFC, pela amizade verdadeira que fizemos e a solidariedade comprovada nos momentos difíceis. Sucesso a todos.

Á Fundação Cearense de Apoio à Pesquisa (FUNCAP), pelo apoio financeiro.

RESUMO

Refatoração, isto é, o processo de alterar um software para melhorar sua qualidade interna preservando seu comportamento externo, está ganhando cada vez mais adeptos entre os desenvolvedores de software. Embora existam muitas ferramentas de refatoração disponíveis para uma variedade de linguagens de programação, a maioria delas é baseada em mecanismos próprios, ou seja, fechados, para representar e manipular informações de código fonte, o que as torna de difícil customização, extensão e reuso.

Neste trabalho fornecemos três importantes contribuições para o desenvolvimento de ferramentas de refatoração mais flexíveis. Primeiro, propomos um processo de refatoração baseado em XML no qual XML é utilizado como formato padrão para representar, analisar e modificar informações de código fonte. Segundo, apresentamos uma concretização desse processo na forma de um arcabouço para refatoração, denominado RefaX, o qual utiliza tecnologias de processamento e representações de código baseadas em XML para facilitar o desenvolvimento, extensão e reuso de ferramentas de refatoração. Por fim, demonstramos a aplicabilidade do processo e do arcabouço propostos através de dois protótipos de refatoração para as linguagens Java e C++, respectivamente.

Palavras-chave: Refatoração, Arcabouço, Manutenção de Software, XML, Java, C++.

ABSTRACT

Refactoring, i.e., the process of changing a software system to improve its internal quality while preserving its external behavior, is gaining increasing acceptance among software developers. Even though many refactoring tools are now available for a variety of programming languages, most of them rely on their own, i.e. closed, mechanisms for representing and manipulating source code information, which makes them difficult to customize, extend and reuse.

This work makes three contributions towards the development of more flexible refactoring tools. Firstly, it proposes an XML-centric refactoring process in which XML is used as a standard way to represent, analyze and modify source code information. Secondly, it presents a concrete realization of that process, in the form of a refactoring framework, called RefaX, which builds on existing XML-based source code models and processing technologies to facilitate the development, extension and reuse of refactoring tools. Finally, it demonstrates the applicability of the proposed process and framework through two XML-based refactoring prototypes for Java and C++, respectively.

Keywords: Refactoring, Framework, Software Maintenance, XML, Java, C++.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS E SIGLAS

AF	*Analysis Functions*
API	*Application Programming Interface*
AST	*Abstract Syntax Tree*
CAF	*Code Analysis Function*
CASE	*Computer Aided Software Environment*
IDE	*Integrated Development Environment*
OCL	*Object Constraint Language*
OO	*Orientado a Objeto*
RO	*Refactoring Operations*
XF	*XQuery Functions*
XML	*Extensible Markup Language*
XSLT	*XSL Transformations*
W3C	*World Wide Web Consortium*

SUMÁRIO

CAPÍTULO 1

INTRODUÇÃO

O custo e a complexidade de se manter um software é amplamente reconhecido. Estima-se que cerca de 50% do tempo de um engenheiro de software é gasto com tarefas de manutenção e compreensão de código [37] e que, ao longo das ultimas três décadas, mais de 60% dos custos de desenvolvimento de software das organizações foram gastos com manutenção [46].

Com o intuito de reduzir os custos e a complexidade da manutenção de software, várias técnicas têm sido propostas, entre elas métricas, teste e verificação, engenharia reversa, e reengenharia. Uma técnica mais recente, denominada de *refatoração* (*refactoring*), e definida como o processo de mudar um software para melhorar sua qualidade interna preservando seu comportamento externo [19], está sendo cada vez mais utilizada entre os desenvolvedores de software. Seu principal objetivo é reduzir a complexidade do código, que cresce rapidamente à medida que o software evolui, tornando-o mais extensível, modular, reutilizável e manutenível [40]. A utilização de refatoração tem sido propagada por ser um dos pilares da *Programação Extrema* (*Extreme Programming*), uma metodologia de desenvolvimento de software emergente e cada vez mais popular, e pela grande disseminação do catálogo de refatorações criado por Fowler [19].

1.1 MOTIVAÇÃO

Como qualquer outra técnica de alteração de código, a refatoração é mais eficientemente realizada através de ferramentas automatizadas. Isto diminui o risco de se introduzir novos erros devido a intervenções manuais, e minimiza a quantidade de testes realizados toda vez que o código é alterado. Um considerável número de ferramentas de refatoração pode ser encontrado para uma variedade de linguagens de programação, incluindo ferramentas para Smalltalk [52], Java [26][24][29] e C# [9]. Entretanto, é praticamente impossível encontrar uma ferramenta de refatoração, ou um conjunto delas, que satisfaça totalmente as necessidades de cada desenvolvedor, haja vista não só a enorme variabilidade

dos softwares em termos de linguagens de programação, sistemas operacionais e estilos de arquitetura, mas principalmente porque essas ferramentas costumam oferecer apenas um conjunto fixo de refatorações que não pode ser facilmente adequado pelo desenvolvedor para suas tarefas de manutenção específicas.

A maioria das ferramentas de refatoração existentes pode ser considerada "fechada", no sentido em que tanto as suas estruturas de dados internas quanto as suas rotinas para manipulação desses dados não estão visíveis para seus usuários. Em geral, os usuários apenas invocam as rotinas de manipulação de código disponibilizadas pelas ferramentas, através de uma interface gráfica ou de programação, mas não têm acesso aos programas que implementam essas rotinas. Claramente, isto limita sobremaneira a capacidade de se customizar essas ferramentas para usos e domínios específicos, além de dificultar o seu reuso no contexto de diferentes ambientes de desenvolvimento.

Duas alternativas têm sido tradicionalmente consideradas na busca por ferramentas de manutenção mais "abertas": a primeira envolve o desenvolvimento de ferramentas de código aberto, isto é, ferramentas cujo código fonte esteja livre para ser acessado e modificado por qualquer programador ou usuário interessado. A maior desvantagem, nesse caso, está na necessidade do usuário ter um conhecimento detalhado tanto da linguagem de programação da ferramenta quanto de suas rotinas e estruturas internas. Além disso, a maioria das ferramentas de código aberto não foi desenvolvida tendo em vista facilitar a sua reutilização ou extensão. Conseqüentemente, pode ser extremamente difícil para um desenvolvedor reutilizar qualquer funcionalidade dessas ferramentas fora de seu ambiente de desenvolvimento original.

A segunda alternativa baseia-se no desenvolvimento de ferramentas de manutenção que sejam configuráveis pelo usuário final, ou seja, ferramentas cujo comportamento possa ser alterado sem a necessidade de que o usuário responsável pela alteração conheça os detalhes de sua implementação. O problema dessa solução é que, em geral, cada ferramenta define sua própria representação interna do código fonte e seu próprio mecanismo de manipulação. Com isso, as reestruturações disponíveis nessas ferramentas, mesmo que de fácil configuração pelo usuário final, são virtualmente impossíveis de serem utilizadas fora do seu próprio ambiente de execução. Além disso, as possibilidades de configuração disponíveis para o usuário dessas ferramentas costumam ser bastante restritas, limitando-se, na maioria dos casos, a mudanças de pouco efeito que não afetam a estrutura de

representação do código fonte interna da ferramenta nem as suas rotinas de manipulação dessa estrutura.

Uma abordagem mais recente, e que está cada vez mais ganhando o interesse da comunidade de engenharia de software, é focada no uso do padrão XML [62] e suas tecnologias correlatas como uma forma de facilitar o desenvolvimento, reuso e interoperabilidade de ferramentas de suporte à manutenção. Esta abordagem é particularmente atrativa para os desenvolvedores de ferramentas, pois estes podem tirar proveito do fato de XML, por ser um padrão gratuito e amplamente utilizado no mundo todo, ter disponível uma abundância de tecnologias para processamento [54][61], consulta [63][64], atualização [69], transformação [65] e armazenamento [17][68][55]. Contudo, a maior parte dos trabalhos realizados nessa direção tem se concentrado apenas na questão da interoperabilidade, principalmente com a definição de novas representações de código fonte no formato XML, como por exemplo JavaML [4], GXL [22] e CppML[36], e com o desenvolvimento de ferramentas de apoio à extração dessas representações a partir do código fonte, como Jikes [4], Columbus CAN [15] e srcML [35].

Embora algumas ferramentas de manutenção já estejam se beneficiando de XML para representar e trocar informações de código fonte, essas ferramentas ainda não exploram todo o potencial de XML e suas tecnologias para melhorar sua capacidade de customização, reuso e extensão. Em especial, não foi encontrada na literatura nenhuma ferramenta de refatoração que utilize tecnologias XML como uma forma mais flexível de representação e manipulação de estruturas de código fonte. A necessidade de ferramentas de refatoração mais abertas e flexíveis também é reconhecida em [40].

1.2 CONTRIBUIÇÕES

Com o intuito de preencher a lacuna mostrada na seção anterior, este trabalho propõe um arcabouço de refatoração de código baseado em XML, denominado RefaX [38][39], cujo principal objetivo é fornecer uma infra-estrutura aberta, baseada em padrões e tecnologias XML, para facilitar a implementação, manutenção, customização e reutilização de ferramentas de refatoração de código.

Neste sentido, este trabalho oferece três importantes contribuições para o desenvolvimento de ferramentas de refatoração mais flexíveis. Como primeira contribuição,

propomos um processo de refatoração no qual XML é utilizado como um mecanismo padrão para representar, analisar e modificar informações de código fonte.

A segunda contribuição é a concretização desse processo, na forma de um arcabouço de refatoração (RefaX), que utiliza tecnologias de processamento e representações de código fonte baseados em XML para facilitar o desenvolvimento, extensão e reuso de ferramentas de refatoração. Em especial, RefaX torna possível implementar operações de refatoração que são independentes de representação de código fonte, linguagem de programação e mecanismo de manipulação. Isso significa que uma ferramenta construída a partir de uma instância de RefaX pode ser mais facilmente reutilizada para diferentes representações de código fonte e tecnologias de processamento baseados em XML.

A terceira e última contribuição é uma demonstração da aplicabilidade do arcabouço e, conseqüentemente, do processo de refatoração proposto, através da instanciação de RefaX na forma de dois protótipos de refatoração baseados em XML para as linguagens de programação Java e C++, respectivamente.

Acreditamos que uma abordagem aberta e flexível para a construção de ferramentas de refatoração, como proposta neste trabalho, pode ser benéfica não apenas para desenvolvedores de ferramentas de refatoração, que terão um poderoso ambiente a partir do qual possam construir, estender e reutilizar operações de refatoração, mas também para os próprios usuários, cujas ferramentas poderão ser mais facilmente customizadas para suas necessidades de manutenção específicas.

1.3 TRABALHOS RELACIONADOS

Embora técnicas de refatoração sejam relativamente recentes, pelo menos quando comparadas a outras técnicas de manutenção mais tradicionais, já existe um considerável número de trabalhos nessa área. Um excelente *survey* sobre o assunto pode ser encontrado em [40]. Nesta seção, compararemos RefaX a quatro trabalhos com os quais compartilhamos algumas características e/ou objetivos comuns.

Leite [34] descreve Draco-PUC, um gerador de meta-programas que produz artefatos de software através da reutilização de linguagens de domínios. Através dele, analistas de sistemas podem modelar aplicações específicas em um alto nível de abstração de forma que o conhecimento gerado por essas aplicações possa ser reutilizado em futuros sistemas. Uma linguagem de domínio tem sua sintaxe expressada através de uma gramática e

sua semântica expressada por transformações, que por sua vez podem ser divididas em duas categorias: horizontais, que são projetadas para manipularem a AST do código interno de programas Draco, e verticais, que mapeiam estruturas sintáticas entre domínios diferentes. A principal diferença do trabalho dele para o nosso é que Draco-PUC utiliza uma representação do código e rotinas de manipulação próprias, além de uma *engine* de transformação fixa, o que implica que transformações criadas para ele não podem ser facilmente reutilizadas no contexto de outros ambientes de manutenção, enquanto RefaX utiliza padrões e tecnologias XML como forma de facilitar a customização de ferramentas de refatoração.

Mamas e Kontrogiannis [36] descrevem um ambiente de manutenção de software integrado, chamado ISME, sobre o qual desenvolvedores podem construir novas ferramentas de manutenção via integração de ferramentas CASE existentes. O ambiente ISME pode ser visto como uma camada extra de abstração entre o código fonte no qual está sendo feita a manutenção e as ferramentas de manutenção em si. Embora compartilhemos algumas técnicas e objetivos com Mamas e Kontogiannis, principalmente o uso de representações de código em XML para aumentar o reuso e customização das ferramentas de manutenção, o trabalho deles difere do nosso no fato que eles não aplicam tecnologias XML para manipular as informações do código fonte diretamente, mas apenas como um formato padrão de troca de informações para facilitar a integração entre as ferramentas constituintes do ISME. Para manipulação do código fonte, o ISME utiliza ferramentas externas tradicionais, cujas rotinas de manipulação, conseqüentemente, não estão disponíveis para customização.

Tichelaar *et al*. [57] descrevem FAMIX, um meta-modelo de código fonte independente de linguagem. FAMIX foi usado para desenvolver uma variedade de ferramentas de manutenção independentes de linguagem, incluindo uma ferramenta de refatoração, chamada Moose, para programas escritos em Java e Smalltalk. FAMIX difere do nosso trabalho por focar exclusivamente no requisito independência de linguagem. Em especial, Moose foi desenvolvida utilizando um conjunto de mecanismos de manipulação e modelos de código fonte próprios, ou seja, fechados, o que compromete a capacidade de customização e reuso de suas funcionalidades fora do seu ambiente de execução original.

Outro trabalho similar ao nosso foi recentemente descrito por Collard [14], onde é proposta uma infra-estrutura baseada em XML para permitir a recuperação (semi) automatizada de refatorações via uma abordagem de diferenciação de código em nível sintático. A abordagem é baseada numa representação XML do código fonte, especificamente

srcML [35]. A idéia principal é que, comparando as diferenças sintáticas entre duas versões em srcML do mesmo código, onde uma versão representa o código no seu estado original e a outra representa o código depois de ter sido alterado manualmente pelo desenvolvedor, é possível inferir que operações de refatoração foram aplicadas ao código e de que modo. Collard sugere usar a tecnologia de transformação XSLT [65] para capturar tais informações, pois assim as refatorações recuperadas podem ser facilmente checadas e re-executadas. O trabalho de Collard difere do nosso em dois pontos principais. Primeiro, Collard usa um conjunto fixo de tecnologias de transformação e representação de código (no caso, srcML e XSLT), enquanto nós deixamos essa decisão aberta para que o desenvolvedor escolha a representação e tecnologia que melhor se adequam às suas necessidades. Segundo, a abordagem de Collard foca em derivar novas refatorações a partir das diferenças entre duas ou mais representações XML do mesmo código fonte, antes e depois de serem aplicadas transformações, enquanto no nosso trabalho buscamos facilitar a implementação e reuso de refatorações já consolidadas na literatura.

1.4 ESTRUTURA DA DISSERTAÇÃO

Além desta Introdução, a dissertação está organizada em mais seis capítulos, descritos a seguir.

No Capítulo 2, fazemos uma revisão da área de refatoração, apresentando um exemplo de uso, no qual várias refatorações conhecidas são aplicadas a um sistema orientado a objeto fictício para a introdução de um padrão de projeto. Também mostramos como refatorações podem ser aplicadas a diferentes níveis de abstração; em seguida, descrevemos detalhadamente o processo de refatoração e, por fim, discutimos algumas das principais ferramentas de refatoração existentes.

No Capítulo 3, apresentamos alguns dos principais padrões para representação de código fonte em XML, ilustrando a estrutura do código XML utilizada e discutindo os pontos positivos e negativos de cada um com relação à sua adequação para a aplicação de refatorações.

No Capítulo 4, descrevemos as principais tecnologias de consulta e atualização de dados XML, mostrando exemplos de uso para cada uma e discutindo as vantagens e desvantagens de se utilizar cada tecnologia no contexto do desenvolvimento de ferramentas de refatoração.

No Capítulo 5, introduzimos o arcabouço RefaX, proposto neste trabalho, descrevendo seus requisitos, sua arquitetura e algumas decisões importantes de projeto. Os principais detalhes de implementação também são discutidos, juntamente com as diretrizes que um desenvolvedor terá que seguir para instanciar o arcabouço.

No Capítulo 6, apresentamos dois protótipos de ferramentas de refatoração desenvolvidos a partir de RefaX, para as linguagens Java e C++, respectivamente, discutindo as facilidades e dificuldades encontradas no desenvolvimento de cada um. Também mostramos como foi desenvolvido um *plugin* para o ambiente de desenvolvimento Eclipse e que serve como uma interface de interação de alto nível para o usuário utilizar os serviços dessas ferramentas.

Por fim, no Capítulo 7, concluímos a dissertação com um resumo dos principais resultados obtidos e algumas sugestões para trabalhos futuros.

Um aspecto da dissertação que vale a pena ressaltar é o padrão de escrita adotado, o qual mostraremos a seguir:

- o texto da dissertação foi escrito utilizando-se a fonte Times New Roman, tamanho 12, como espaçamento de 1 linha e meia entre frases e cada parágrafo começando 2 centímetros à direita da margem;
- os termos em *itálico* representam palavras estrangeiras, palavras ou expressões que queremos enfatizar ou destacar, e nome de padrões de projeto, refatorações, funções de análise e funções de acesso ao código;
- a fonte `Courier New` foi utilizada nos seguintes casos: para os códigos fonte escritos em Java e C++ mostrados nas figuras, onde utilizamos o `tamanho 10`; para palavras no texto da dissertação que representem elementos de código ilustrados nas figuras e para exemplos de implementação de linguagens de consulta e atualização XML, aplicamos a fonte no `tamanho 11`; e para os exemplos de representações de código em XML, para os quais utilizamos o `tamanho 9`.

CAPÍTULO 2

REFATORAÇÃO DE CÓDIGO

Refatoração é o processo de mudar um software de tal forma que melhore a sua estrutura interna sem, contudo, alterar o seu comportamento externo. É, portanto, uma forma disciplinada de re-organizar o código, minimizando as chances de introduzir erros [19]. Refatorar tem a vantagem de melhorar a estrutura do código, facilitando o seu reuso e diminuindo o tempo gasto com tarefas de manutenção. Refatoração é um termo aplicado a sistemas orientados a objeto; para outros paradigmas de programação, esse mesmo processo é descrito como *reestruturação*. Como o contexto do nosso trabalho é refatoração de código, a reestruturação não será tão enfatizada nesta dissertação.

Aplicada à manutenção de software, a refatoração ajuda a tornar o código mais legível e resolver problemas de códigos mal escritos (*bad smells*) [11]. Já no contexto da evolução de software, a refatoração é usada para melhorar os atributos de qualidade do software, como extensibilidade, modularidade e reusabilidade, entre outros [40]. A refatoração também pode ser usada no contexto da reengenharia, ou seja, para alterar um sistema específico visando reconstruí-lo em um novo formato. Nesse contexto, refatorar é necessário para converter código legado ou deteriorado em um formato mais estruturado ou modular [48], ou para migrar o código para uma diferente linguagem de programação, ou mesmo um diferente paradigma de linguagem [49].

Este capítulo é baseado, em parte, no trabalho descrito por Mens e Tourwé [40], e está organizado da seguinte maneira: na seção 2.1, mostramos um exemplo de utilização de refatorações para aplicar um padrão de projeto a um código de um sistema orientado a objeto; na seção 2.2, descrevemos como a pesquisa sobre refatoração evolui ao longo do tempo e como suas aplicações se diversificaram; na seção 2.3, detalhamos os níveis de abstração nos quais podemos aplicar refatorações; na seção 2.4, descrevemos o processo geral de refatoração; por fim, na seção 2.5, apresentamos algumas das principais ferramentas de refatoração que podem ser encontradas e, na seção 2.6, discutimos os seus problemas mais relevantes.

2.1. UM EXEMPLO DE REFATORAÇÃO

Como exemplo de uma aplicação de refatoração, vamos considerar um determinado conjunto de classes de um sistema fictício orientado a objeto escrito na linguagem Java, criado de tal forma que facilite a compreensão das refatorações, e que pode ser visto na Figura 1. Ela mostra a classe abstrata `BancoDeDados`, com suas operações `Conectar`, `Consultar` e `Executar`, e as classes `BDSQLServer`, `BDOracle` e `BDMySQL`, `que` estendem a classe `BancoDeDados` e implementam os métodos acima descritos, além de definir o atributo `conexao` e os métodos de devolução e configuração (*get* e *set*) desse atributo. A classe `Cliente` representa uma classe de negócio que, em algum momento, necessita se conectar ao banco de dados para realizar uma operação de inserção, remoção, atualização ou consulta.

O sistema, propositadamente, não está bem projetado, pois caso as classes de banco de dados sejam alteradas, toda classe que faça referência a esse banco deverá ser alterada. O desenvolvedor percebe que o projeto ficaria mais modular se ele utilizasse o padrão de projeto *Factory Method* [20], que define uma interface para criar um objeto, mas deixa para as subclasses decidirem que classe instanciar. A vantagem de utilizar esse padrão é

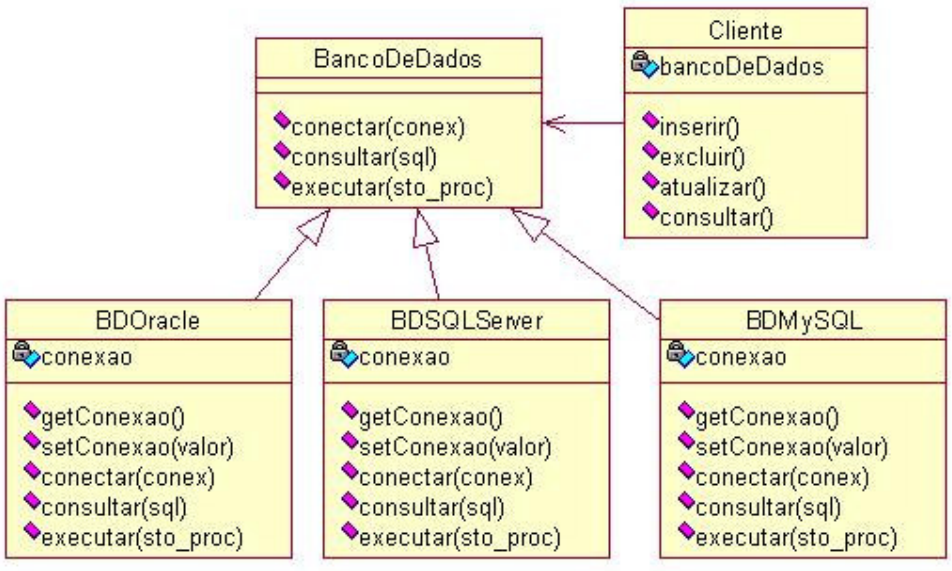


Figura 1: Classes de um sistema OO fictício antes da aplicação das refatorações.

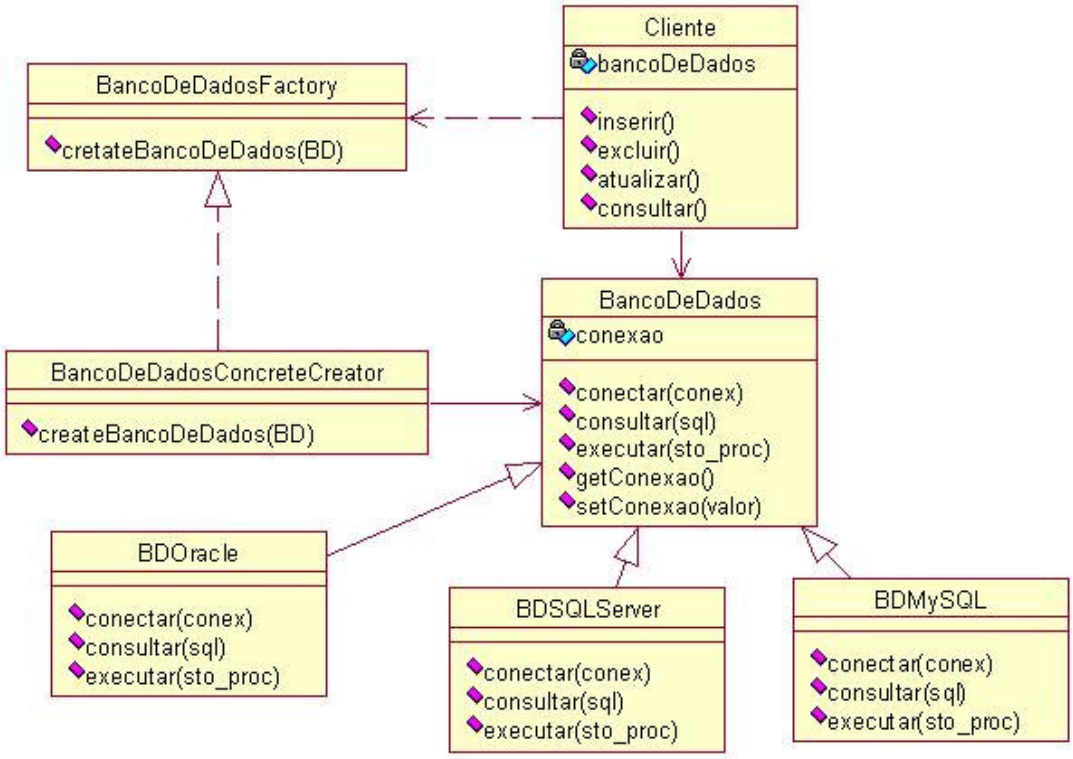


Figura 2: Nova estrutura de classes após a aplicação das refatorações.

que ele torna o projeto mais customizável, apesar de um pouco mais complicado, diminuindo o acoplamento entre classes. Para aplicar esse padrão, serão necessárias diversas refatorações.

A Figura 2 mostra a nova estrutura de classes após a aplicação do padrão *Factory Method*. A primeira alteração é criar a interface `BancoDeDadosFactory`, juntamente com o método `createBancoDeDados(BD)`. O próximo passo é criar a classe que será a criadora dos objetos `BancoDeDados`, chamada `BancoDeDadosConcreteConcreator`. Ela implementa a interface e seu método, que instanciará um objeto do tipo `BancoDeDados` correspondente ao parâmetro passado.

Além disso, percebemos que o atributo `conexao` e os métodos de devolução e configuração desse atributo se repetem em toda subclasse de `BancoDeDados`, o que nos levou a deslocar o atributo e métodos para a superclasse.

Apesar do exemplo parecer simples, várias refatorações primitivas são necessárias para introduzir o padrão *Factory Method* e fazer as alterações indicadas a seguir:

1. A interface `BancoDeDadosFactory` é criada através da refatoração *Add Interface*.
2. O método `createBancoDeDados()` é criado na interface usando a refatoração *Add Method*.
3. O parâmetro `BD` do método da interface é inserido através da refatoração *Add Parameter*.
4. A classe `BancoDeDadosConcreteCreator` é criada através da refatoração *Add Class*.
5. O método `createBancoDeDados()` é criado na nova classe usando a refatoração *Add Method*.
6. O parâmetro `BD` do método da nova classe é inserido através da refatoração *Add Parameter*.
7. O atributo `conexao` é movido de uma das subclasses para a superclasse `BancoDeDados` através da refatoração *Pull Up Attribute*.
8. Nas subclasses que não enviaram o atributo `conexao` para a superclasse, remover o atributo através da refatoração *Remove Attribute*.
9. Os métodos `getConexao` e `setConexao` são movidos de uma das subclasses para a nova superclasse usando a refatoração *Pull Up Method*.
10. Nas subclasses que não enviaram os métodos `getConexao` e `setConexao` para a superclasse, remover os métodos através da refatoração *Remove Method*.

Essas refatorações são consideradas *primitivas*, ou seja, são transformações elementares de preservação do comportamento que podem ser combinadas para criar novas refatorações, chamadas *refatorações compostas* [42]. Essas refatorações compostas são definidas como uma seqüência de primitivas que formam operações de preservação do comportamento mais complexas e têm um significado mais interessante para o usuário. Por exemplo, a refatoração primitiva *Add Method* não possui muito sentido se usada individualmente, mas combinada com outras refatorações, formando uma refatoração composta, ela pode ser muito útil. Um exemplo de refatoração composta pode ser a criação da refatoração *Apply Factory Pattern*, que combina as refatorações 1 a 6 na lista acima.

2.2. HISTÓRICO

O primeiro trabalho mais relevante publicado nessa área foi a tese de William Opdyke [42], a qual trata de refatorações para programas escritos em C++. Ele definiu vinte e três refatorações primitivas e mostrou três exemplos de refatorações compostas, formadas por duas ou mais primitivas. Para cada refatoração primitiva, o trabalho introduz um conjunto de pré-condições que fornecem a noção de preservação do comportamento do sistema. Assim, se cada refatoração primitiva preserva o comportamento, uma refatoração composta a partir dessas primitivas também preservará. As refatorações primitivas definidas por Opdyke foram classificadas nas quatro categorias abaixo:

- **Criação de entidades**: criar uma classe vazia, uma variável ou uma função.
- **Remoção de entidades**: remover classe, variável ou função não referenciada.
- **Alterar entidade**: alterar o nome da classe, alterar o nome da variável, alterar o nome da função, alterar tipo de variáveis e retorno de funções, alterar modo de controle de acesso, adicionar parâmetro à função, remover parâmetro de função, reordenar parâmetros de função, adicionar corpo de função, remover corpo de função, converter variável para ponteiro, converter referência à variável para chamada de função, substituir lista de declarações por chamada de função, extrair chamada de função e alterar superclasse de uma classe.
- **Mover variável**: mover variável para superclasse e mover variável para subclasse.

As refatorações compostas foram:

- **Abstrair acesso a uma variável**.
- **Converter segmento de código para função**.
- **Mover classe**.

Opdyke também identificou sete propriedades que uma refatoração deve obedecer para preservar o comportamento, também chamadas de *invariantes*: cada classe deve ter uma única superclasse; cada classe deve ter nome distinto das outras classes; os membros de uma classe (variáveis e métodos) devem ter nomes distintos; variáveis herdadas não podem ser redefinidas; funções herdadas devem ter a mesma assinatura da função original; atribuições de tipo seguras, que consiste em forçar que o tipo da expressão do lado esquerdo de uma atribuição seja igual ou um subtipo da expressão contida no lado direito; equivalência

semântica de operações e referências, isto é, um programa deve produzir a mesma saída para uma dada entrada antes e depois de se aplicar a refatoração.

Tokuda [58] se baseou no trabalho de Opdyke para ir mais além e avaliar projetos orientados a objeto com refatorações. Primeiramente, ele definiu mais quatro propriedades (invariantes) que uma refatoração deve obedecer para preservar o comportamento e que não foram previstas por Opdyke, a seguir: implementação de funções puramente virtuais; manter objetos agregados; nenhum efeito colateral de instanciação; e independência de layout ou tamanho. Sua pesquisa também mostrou que todos os tipos de evolução de projeto, isto é, transformações de esquema, micro arquiteturas de padrões de projeto e abordagem dirigida a *hot-spot*, são automatizáveis com refatorações, e propôs, então, uma série de novas refatorações para cada tipo de evolução.

Tokuda também fez experimentos para avaliar se a aplicação de refatorações para programas C++ já existentes, cujo código ultrapasse mais de 100.000 linhas, é viável. Suas conclusões mostram que a aplicação de refatorações traz muitos benefícios, entre os quais estão automatização das mudanças no projeto, redução de testes, e criação de projetos mais simples.

Roberts [52] se concentrou em propor um arcabouço para linguagens orientadas a objeto, que permite a criação de ferramentas de refatoração confiáveis e rápidas o suficiente para serem usadas na prática por desenvolvedores. Como resultado, desenvolveu uma das primeiras ferramentas de refatoração, o *Refactoring Browser* [52], para a linguagem Smalltalk.

Roberts também estendeu o conceito de refatoração de Opdyke acrescentando pós-condições que cada refatoração deve obedecer. Essas pós-condições descrevem o que deve ou não haver depois da aplicação da refatoração e podem ser usadas para derivar pré-condições de refatorações compostas, calcular dependência entre refatorações e reduzir a quantidade de análises que refatorações posteriores em uma seqüência devem realizar para garantir que preservam o comportamento.

Para implementar as pré e pós-condições, Roberts definiu o conceito de funções de análise, que descrevem o relacionamento entre componentes do programa, isto é, classes, métodos e variáveis. As funções de análise foram divididas em duas categorias: primitivas e derivadas. As primitivas são usadas tanto nas pré como nas pós-condições, enquanto as derivadas são utilizadas apenas nas pré-condições.

Baseado nesse novo conceito, ele redefiniu algumas refatorações propostas por Opdyke, e as dividiu em três grupos:

- **Refatorações de classe**: criar nova classe, renomear e remover uma classe.
- **Refatorações de métodos**: criar novo método, renomear, remover e mover um método.
- **Refatorações de variáveis**: criar nova variável, remover e renomear variável, mover uma variável para a superclasse, subclasse ou outra classe qualquer.

Tichelaar et al. [57] implementaram 15 refatorações primitivas independentes de linguagem de programação, definindo um conjunto de pré-condições que podem ser aplicadas a qualquer linguagem. Porém, para algumas refatorações, é necessário acrescentar pré-condições específicas da linguagem para que o comportamento realmente seja preservado. Como exemplo, mostraram pré-condições específicas para Java e Smalltalk, que devem ser verificadas na mesma refatoração.

Fowler [19] se focou em outra direção, diferente de Opdyke e Roberts: o processo de refatoração. Ele explica princípios e melhores práticas de refatoração, além de fornecer um guia sobre o processo de refatoração (onde começar, quando começar, quando parar) e um extenso catálogo de refatorações. Contudo, ele não define as condições que garantem que as refatorações preservam o comportamento.

Com o crescimento do uso de padrões de projetos, muitos pesquisadores passaram a estudar as oportunidades de refatorações inerentes a cada padrão. Cinnéide [12] desenvolveu uma metodologia para desenvolvimento de transformações de padrões de projeto baseadas em refatorações. Essa metodologia tem sido aplicada para sete padrões e é implementada através de uma ferramenta para código Java.

Outra metodologia que foi responsável pelo crescimento da visibilidade de aplicar refatoração foi *Extreme Programming* (XP) [6], pois uma de suas principais idéias é que o desenvolvedor deve trabalhar em apenas um caso de uso por vez e assim deve projetar o software para que fique coerente com o caso de uso em questão. Se um determinado caso de uso não ficou bem projetado, deve-se aplicar refatorações até que o caso de uso possa ser implementado de uma maneira coerente. Ao contrário de tentar evitar mudanças, XP é uma metodologia que é baseada em mudanças. Um dos principais pilares de XP é a contínua e agressiva aplicação de refatorações. Sem elas, XP não funcionaria.

2.3. NÍVEIS DE ABSTRAÇÃO PARA REFATORAÇÃO

Refatorações podem ser aplicadas em diferentes níveis de abstração e tipos de artefatos de software. Por exemplo, é possível aplicar refatorações a modelos de projeto, esquemas de banco de dados, requisitos, e arquitetura de software, além de estruturas próprias das linguagens [40]. Essa diversidade possibilita ao desenvolvedor realizar mudanças estruturais que não necessariamente se refletem no código fonte e, portanto, introduzem a necessidade de manter todos os artefatos em sincronia. Nas subseções seguintes detalhamos os três níveis de abstração nos quais podemos aplicar refatorações.

2.3.1. NÍVEL DE ANÁLISE

Nesse nível, as alterações são consideradas reestruturações, pois não são descritas em termos de código fonte. Um exemplo de reestruturações nesse nível são mudanças na especificação dos requisitos de software. Russo *et al*. [53] sugere reestruturar essas especificações decompondo-as em uma estrutura de *pontos de vista* (*view points*), que encapsulam requisitos parciais de algum componente do sistema. Essa abordagem aumenta o entendimento dos requisitos, facilita a detecção de inconsistências e permite um melhor gerenciamento da evolução dos requisitos.

2.3.2. NÍVEL DE PROJETO

Novas pesquisas em refatorações se direcionam para artefatos de projeto de software. Para arquiteturas de software, podemos destacar duas abordagens: a primeira propõe pré-condições de refatorações, que preservam o comportamento do sistema através da relação de causa entre os componentes, baseadas diretamente em representações gráficas da arquitetura do sistema [44]. A segunda é mais pragmática e propõe que mudanças na arquitetura possam ser feitas através uma seqüência de refatorações primitivas [58].

Outro tipo de artefato de projeto são modelos de projeto de software. Em particular, modelos UML têm sido largamente utilizados como mecanismo de projeto de software. Uma integração entre refatorações e ferramentas de modelagem UML é mostrada em [8]. A ferramenta suporta refatorações de diagramas de classes, estado e atividades. Para cada diagrama, usuários podem aplicar refatorações que não são facilmente ou naturalmente expressas em outros diagramas ou no código fonte. Uma extensão de UML para expressar pré

e pós-condições de refatorações de código fonte usando *Object Constraint Language* (OCL) [66] é mostrada em [21]. A extensão proposta permite uma ferramenta CASE que implemente OCL verificar pré e pós-condições não triviais, compor seqüências de refatorações e usar a ferramenta de consulta OCL para detectar códigos mal escritos. Tais abordagens são desejáveis como uma forma para refatorar artefatos de projeto independentemente da linguagem de programação do software.

Padrões de projeto [20] estão cada vez mais sendo usados por desenvolvedores. Eles fornecem uma maneira de escrever o programa em um nível de abstração mais alto. Padrões de projeto criam várias oportunidades de aplicação de refatoração. Um exemplo disso foi mostrado na seção 2.1, onde aplicamos o padrão *Factory Method* para um conjunto de classes existentes através de refatorações. Embora as mudanças sejam feitas diretamente no código fonte, a decisão de se aplicar padrões de projeto está no nível de projeto.

2.3.3. NÍVEL DE CÓDIGO

Essa é forma mais comum de se aplicar refatoração a um sistema. Refatorações podem ser aplicadas a diversas entidades do código, como mostrado na seção 2.2, e às mais variadas linguagens de programação e paradigmas de linguagem, como por exemplo, linguagens imperativas (Fortran, Cobol), funcionais (Liso, Haskell), lógicas (Prolog), orientadas a objeto (Smalltalk, C++, Java) e orientadas a aspectos (AspectJ) [40].

Linguagens não orientadas a objeto são mais difíceis de se reestruturar, pois fluxos de controle e de dados são fortemente interligados e, por causa disso, as reestruturações são limitadas ao nível de função ou bloco de código [32]. Contudo, linguagens puramente orientadas a objeto apresentam características que tornam algumas refatorações extremamente trabalhosas para serem implementadas, como é o caso das refatorações que lidam com herança, polimorfismo, ligação dinâmica e interfaces. Além disso, quanto mais complexa a linguagem, maior a dificuldade de implementar refatorações. Esse é o caso de programas escritos em C++, que após serem pré-processados perdem muitas informações estruturais, impossibilitando ou reduzindo o escopo de aplicação de refatorações.

2.4. PROCESSO DE REFATORAÇÃO

A maioria das ferramentas de refatoração segue um processo comum, no qual operações de refatoração são aplicadas ao código-fonte dos programas de acordo com passos bem definidos [40]. Em geral, esses passos incluem:

1. Detectar trechos do código com oportunidades de refatorações.
2. Determinar que refatorações aplicar a cada trecho do código selecionado.
3. Garantir que as refatorações escolhidas preservem o comportamento.
4. Aplicar as refatorações escolhidas aos seus respectivos locais
5. Verificar que o comportamento do programa foi preservado após as refatorações terem sido aplicadas.

Nas subseções seguintes discutiremos um pouco mais cada um dos passos acima.

2.4.1. DETECÇÃO DO LOCAL DE APLICAÇÃO DAS REFATORAÇÕES

A primeira tarefa para aplicar uma refatoração é identificar em que nível de abstração a refatoração se enquadra. Os três níveis foram discutidos na seção anterior, por isso nossa análise irá se restringir apenas ao nível de código, já que é o mais utilizado pelos desenvolvedores e é o foco do nosso trabalho. Para esse nível, os passos de detecção e determinação, 1 e 2, respectivamente, são geralmente aplicados juntos.

Existem diversas abordagens para detecção de oportunidades de refatoração. Kataoka *et al*. implementaram a ferramenta Daikon para indicar onde refatorações poderiam ser aplicáveis através da detecção automática de invariantes de programas, como por exemplo, um dos invariantes propostos por Opdyke, mostrados na seção 2.2. Outra abordagem é baseada em meta-programação declarativa, como proposto por Tourwé *et al*. [59], usada para formalmente especificar e detectar códigos mal escritos e propor refatorações que removam esses trechos de códigos. Carneiro e Neto [11] relacionam métricas a oportunidades de refatoração através de duas abordagens: a primeira almeja identificar analiticamente métricas adequadas à avaliação de códigos mal escritos, e a segunda utiliza, empiricamente, mensuração de um grande conjunto de métricas para verificar o seu relacionamento com refatorações e códigos mal escritos.

2.4.2. GARANTIA DE PRESERVAÇÃO DO COMPORTAMENTO

Depois de determinados onde e que refatorações aplicar, o próximo passo consiste em garantir que as refatorações preservarão o comportamento do sistema. Preservar o comportamento é uma definição vaga, pois diz muito respeito ao domínio em que o sistema está inserido. Por exemplo, se o sistema é um software de tempo real, o fator tempo de execução é extremamente importante, e toda refatoração deve preservar esse fator; já se é um software embarcado, restrições de memória devem ser respeitadas após as refatorações.

A idéia original de preservação do comportamento, introduzida por Opdyke, dizia que, para um mesmo conjunto de dados de entrada, o conjunto de dados de saída deve ser o mesmo, antes e depois da aplicação da refatoração. Com isso, ele propôs o conceito de pré-condições, que são um conjunto de verificações que devem ser verdadeiras para que a refatoração possa ser aplicada. Cada refatoração tem um conjunto de pré-condições. Por exemplo, para a aplicar refatoração primitiva *Add Class (className, packageName, superclasses, subclasses)*, mostrada na seção 2.1, é necessário que o sistema obedeça às seguintes regras: nenhuma classe com o nome *className* deve existir, nenhuma variável global com o nome *className* deve existir no mesmo escopo, *subclasses* são subclasses de todas as superclasses.

Outras formas de garantir a preservação do comportamento são: através de um rigoroso conjunto de casos de testes, que deve ser satisfeito após a aplicação da refatoração [45]; através da redução do escopo da preservação do comportamento, por exemplo, garantir que todas as chamadas a métodos são preservadas após a refatoração [41]; através de uma prova formal que a refatoração preserva toda a semântica do programa, o que seria mais fácil de fazer em uma linguagem de programação com uma semântica simples e formalmente definida, como Prolog [47].

2.4.3. APLICAÇÃO DAS REFATORAÇÕES

O quarto passo no processo é executar as operações necessárias para alterar o código fonte. Essa etapa está ligada à forma como o código fonte é representado internamente em cada ferramenta de refatoração. Geralmente, as tais ferramentas possuem um formato próprio para representar o código, seja em forma de *Árvore Sintática Abstrata* (*Abstract Syntax Tree – AST*) ou XML. Neste trabalho, propomos um arcabouço para construção de

ferramentas de refatoração que se baseia na utilização de XML não só como mecanismo para representação de código, mas também para validação e aplicação das refatorações. O arcabouço será descrito em mais detalhes no Capítulo 5.

2.4.4. VERIFICAÇÃO DA PRESERVAÇÃO DO COMPORTAMENTO

Como forma de garantir que o comportamento realmente foi preservado, é necessário verificar se todas as operações de refatorações foram realizadas com sucesso. Para isso, aplica-se um conjunto de pós-condições que devem ser satisfeitas após a aplicação da refatoração, como as definidas por Roberts [52] e Tichelaar et al. [57]. Por exemplo, a refatoração *Add Class(className, packageName, superclasses, subclasses)* usada na seção 2.1, deve satisfazer as seguintes pós-condições: a nova classe é inserida na hierarquia tendo *superclasses* como *superclasses* e subclasses como subclasses, nova classe deve se chamar *className*, *subclasses* não devem ser mais subclasses de *superclasses*.

2.5. FERRAMENTAS DE REFATORAÇÃO EXISTENTES

Apesar de podermos aplicar refatorações manualmente, ferramentas que automatizem o processo diminuem o risco de erros e inconsistência no código, além de poupar um grande trabalho em se tratando de sistemas com centenas ou milhares de linhas de código.

A primeira ferramenta de refatoração desenvolvida foi o Smalltalk Refactoring Browser [52]. Projetada para auxiliar programadores da linguagem Smalltalk, esta ferramenta tornou o processo de refatoração mais rápido e satisfatório. Porém, como Smalltalk não é uma linguagem comercial, seu uso ficou restrito, já que poucas pessoas utilizam essa linguagem hoje em dia, embora a ferramenta tenha muitas funcionalidades.

Desde então, muitas ferramentas de refatoração surgiram, sejam comerciais ou acadêmicas, para diferentes linguagens, como por exemplo, C# Refactory [9] e C# Refactoring Tool [10], para códigos escritos em C#, Project Analyzer [3], para programas em Visual Basic, e Bicycle Repair Man [7], que é uma tentativa de criar uma ferramenta similar a Refactoring Browser, porém para a linguagem Python.

Com a expansão de Java, muitas ferramentas de refatoração para essa linguagem surgiram, sendo X-Refactoring [70] e IntelliJ IDEA [24] as pioneiras. Com o crescimento do número de IDE's para Java, algumas ferramentas de refatoração foram criadas como *plugins*

para esses ambientes, dentre as quais podemos citar JFactor [27], RefactorIt [50], JRefactor [29], Transmogrify [60] e JavaRefactor [26]. Outras apresentam-se como ferramentas independentes de IDE e trazem algumas funcionalidades de engenharia reversa, como é caso de OptimalJ [43].

No meio acadêmico, Elbereth [31] foi desenvolvida como uma ferramenta de refatoração para código Java baseada no conceito de Diagramas Estrela, que fornece uma visão gráfica das estruturas do código, como forma de melhor representá-lo e manipulá-lo.

A grande maioria dessas ferramentas implementa os mesmos tipos de refatoração, dentre os quais estão: extrair métodos e superclasses, renomear variáveis, métodos e atributos, mover para cima e para baixo atributos e métodos na hierarquia de classes. O problema é que elas não possuem o código fonte aberto que possibilite um desenvolvedor estender a ferramenta, ou então, quando possuem, usam representações próprias do código, o que também dificulta a sua extensão, dado que o desenvolvedor deve ter um domínio sobre a manipulação dessa representação. Utilizar padrões abertos e difundidos em todo o mundo, como XML, que dispõe de uma gama de ferramentas para manipular essa representação, abre caminho para a implementação de ferramentas de refatoração mais extensíveis, reutilizáveis e customizáveis. Essa abordagem foi utilizada por nós para o desenvolvimento de RefaX.

2.6. CONCLUSÃO

Refatoração de código, como atividade de manutenção, não é um processo recente. Ela vem sendo realizada há muito tempo pelos desenvolvedores, os quais não a chamavam de nenhum nome específico, e hoje está cada vez mais presente, seja como mecanismo de manutenção de software ou como um artefato para desenvolvimento e evolução de software.

Nosso trabalho mostra como técnicas modernas de refatoração podem ser realizadas a partir de modelos de linguagem representados em XML, visando suprir a carência deixada pelas ferramentas de refatoração existentes, muitas das quais foram descritas na seção anterior. No próximo capítulo apresentaremos diversas representações de código baseadas em XML, e discutiremos como elas podem ser utilizadas para a implementação de ferramentas de refatoração.

CAPÍTULO 3

REPRESENTAÇÕES DE CÓDIGO EM XML

Como forma de deixar o código fonte independente de estilo de programação e mais fácil de ser manipulado e analisado por diferentes ferramentas de manutenção, surgiu a necessidade de uma representação padrão para o mesmo. Muitas representações foram propostas, sendo XML [62] o modelo mais comum.

A representação em XML do código traz uma série de benefícios: estrutura o código de forma explícita, o que facilita sua manipulação; possibilita criar consultas mais poderosas, pois não será mais feita uma pesquisa textual através de expressões regulares, e sim uma busca em marcadores onde se pode utilizar linguagens padronizadas e ferramentas apropriadas; possui representação extensível, o que facilita criar extensões no código, como por exemplo anotações, meta-informações, revisões, documentação e código condicional, características que o formato textual não permitia; possibilita fazer referência cruzada entre elementos do código, atividade que não é possível em representações textuais, pois os elementos são referenciados através de linhas e colunas; e suporte amplo, uma vez que XML é suportado por várias plataformas [4].

Podemos classificar os formatos de representação de código em XML existentes em duas categorias: representações específicas de linguagem e representações genéricas. As seções 3.1 e 3.2 mostram, respectivamente, exemplos de representações de código para essas categorias. A adequação desses modelos para implementar refatorações é discutida na seção 3.3.

3.1. REPRESENTAÇÕES ESPECÍFICAS DE LINGUAGEM

Apesar de existirem diversas linguagens de programação, representações de código em XML só foram encontradas para duas delas: Java e C++. Nas próximas subseções, apresentamos as principais representações para ambas as linguagens, discutindo as características de cada uma.

3.1.1. REPRESENTAÇÕES PARA JAVA

Talvez por ter uma gramática relativamente simples, pelo menos se comparada a linguagens tradicionais como C e C++, e ser cada vez mais utilizada pelos desenvolvedores em todo o mundo, Java é a linguagem para a qual surgiram mais representações baseadas em XML, dentre as quais vale destacar JavaML de Mamas e Kontogiannis [36], JavaML de Badros [4] e XJava[5]. Para cada uma delas iremos mostrar a representação em XML gerada para a classe `Funcionario` mostrada na Figura 3 e discutir os seus principais aspectos.

3.1.1.1. JAVAML DE MAMAS E KONTOGIANNIS

Mamas e Kontogiannis [36] propõem uma padrão chamado JavaML que representa as informações do código em forma de árvore sintática abstrata (*Abstract Syntax Tree - AST*), sendo esta estruturada em um formato XML. Os elementos do código são especificados em marcadores, enquanto seus valores estão contidos nos atributos dos marcadores, isto é, sem o uso do corpo do marcador para armazenar informação de código. Um exemplo pode ser visto na Figura 4.

`CompilationUnit` é o elemento raiz do modelo e representa o arquivo java que está sendo transformado. Ele possui três elementos filhos: `PackageDeclaration`, `ImportDeclaration` e `ClassDeclaration`, que representam o pacote no qual a classe está contida, uma declaração de importação e a classe, respectivamente. O arquivo XML possuirá tantos elementos `ImportDeclaration` quantas forem as declarações *import* no

```
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 3: Exemplo de classe Java.

formato texto. O elemento `ClassDeclaration` possui vários sub-elementos `FieldDeclaration`, `ConstructorDeclaration` e `MethodDeclararion`, representando os atributos, construtores e métodos declarados na classe, respectivamente. A representação JavaML para parâmetros, variáveis e outros elementos presentes em métodos e construtores pode ser vista na Figura 4.

O nome de cada elemento do código é armazenado no atributo `Identifier`, presente no marcador localizado pelo menos um nível abaixo do que representa o elemento. Por exemplo, para encontrar o nome de uma classe basta procurar em `ClassDeclaration/UnmodifiedClassDeclaration[@Identifier]`, enquanto o nome de um método é encontrado em `MethodDeclaration/MethodDeclarator[@Identifier]`.

Para representar atributos, parâmetros e variáveis, utiliza-se uma estrutura comum aos três: abaixo do marcador que representa o elemento existem mais dois marcadores, `Type` e `VariableDeclarator`. O primeiro possui um marcador interno que contém o tipo de dados do elemento do código, e 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 possui o atributo definindo o nome do elemento do código. O elemento `Type` também pode ser encontrado interno 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` fica vazio, isto é, sem atributos nem sub-elementos.

A vantagem dessa representação é que ela representa os elementos essenciais do código, como classes, métodos, atributos e a interação entre eles, já que é baseada na AST da linguagem, o que permite a especificação de qualquer consulta para analisar o código. Porém, apresenta como desvantagens o fato de que não guarda informações estruturais, como linha, coluna, início e fim de blocos, formatação e comentários, além de ser um padrão muito verboso, o que aumenta bastante o tamanho da representação em XML e diminui sua legibilidade. Por exemplo, para representar a visibilidade de um elemento, todo marcador, ao invés de definir apenas um atributo, define sempre três (um para a visibilidade pública, outro para a protegida e outro para a privada), como pode ser visto nos marcadores `MethodDeclaration` e `FieldDeclaration`. Outro exemplo de quão verbosa é essa representação é que, para representar uma simples atribuição, como a que existe no corpo do método da Figura 3, são necessários 19 marcadores, enquanto em outras representações utilizam-se apenas 5 ou 6 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 4: Representação em JavaML de Mamas e Kontogiannis da classe `Funcionário`.

3.1.1.2. JAVAML DE BADROS

O padrão JavaML de Badros [4] se assemelha, em alguns aspectos, ao de Mamas e Kontogiannis: ambos preocupam-se em manter informações da AST do código fonte no

```
<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 5: Representação em JavaML de Badros da classe `Funcionario`.

formato XML e representam cada elemento do código através de marcadores, e seus valores através de atributos, sem usar assim o corpo das marcadores, como pode ser visto na Figura 5.

Esse formato, em comparação especificamente ao seu homônimo, traz muitos benefícios. Primeiro, ele representa de forma mais completa e simples o código, incluindo algumas informações estruturais ausentes no outro padrão, como número das linhas e colunas iniciais e finais de cada elemento. Com isso, a especificação de várias funções de análise de engenharia reversa, métricas, críticas e refatoração, por parte das ferramentas e técnicas XML existentes, torna-se mais fácil. Além disso, é uma representação menos verbosa, ou seja, utiliza uma quantidade menor de marcadores para representar um elemento do código e tais marcadores possuem nomes menores e mais intuitivos, melhorando a legibilidade da representação. Outra vantagem é que JavaML de Badros atribui um *id* para cada método e variável e um *idref* para todo relacionamento que utilize essas entidades, como invocação de método e atribuição de variáveis, por exemplo, que ajuda no cruzamento de informações para análises.

A estrutura do código JavaML de Badros inicia com o elemento `java-source-program`, seguindo pela sub-elemento `java-class-file` representando o arquivo que está

sendo transformado. Dentro desse elemento estão `package`, `import` e `class`, representando o pacote que contém a classe, os arquivos importados e a classe do arquivo, respectivamente. Os elementos internos a `class` que representam construtores, atributos, métodos e variáveis são, nessa ordem, `constructor`, `field`, `method` e `local-variable`.

Os nomes dos elementos do código estão sempre nos mesmos marcadores que os representam, contidos no atributo *name*. 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`, interno ao que representa o elemento do código correspondente.

Apesar desse padrão ser mais legível e bem menos verboso que JavaML de Mamas e Kontogiannis, algumas desvantagens também são compartilhadas por ambos, como o fato de não preservar informações de formatação, linhas em branco e comentários, que prejudica a transformação do código em XML para seu formato textual. Com intuito de suprir essas deficiências, Aguimar *et al.* propuseram a versão 2 do padrão JavaML [1], que inclui esses e outros tipos de informações que estavam ausentes na proposta original. O preço das informações adicionais é que o padrão se tornou mais verboso, gerando representações mais extensas.

3.1.1.3. XJAVA

XJava [5] é uma transformação para XML de um *parser* feito em código Java pela ferramenta BeaultyJ [5]. Diferentemente das outras representações mostradas anteriormente, XJava não representa a AST do código; mesmo assim, sua estrutura de marcadores e atributos é bastante simples, sendo bem parecida com a de JavaML de Badros, como pode ser visto na Figura 6.

Assim como JavaML de Badros, esse padrão também representa cada elemento do código através de um marcador, com o nome dos elementos estando contido no atributo `name`. O elemento inicial que representa o código transformado é `xjava`. Os marcadores que representam pacotes, importações, classes, atributos, construtores e métodos possuem os mesmos nomes dos marcadores correspondentes em JavaML de Badros. Outra similaridade é que o tipo de retorno de métodos, bem como o tipo de dados de atributos e parâmetros, também são representados pelo marcador `Type` e seu valor armazenado no atributo `name`.

XJava, diferentemente das duas representações JavaML, armazena todos os arquivos Java transformados em um só arquivo XML, sendo organizados internamente por

```
<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 6: Representação em XJava da classe `Funcionário`.

pacotes. Para sistemas com poucos arquivos, essa estrutura não representa nenhum obstáculo do ponto de vista de análise e manipulação; no entanto, para sistemas com centenas de arquivos, esse tipo de representação pode não ser adequada, na medida que pode gerar um documento XML extremamente grande e que pode trazer problemas de processamento para as ferramentas de consulta e manipulação.

Um ponto positivo dessa representação é que ela armazena em XML os comentários feitos no código pelo programador. Além disso, esse padrão traz como vantagem a facilidade de especificar *alguns* tipos de consulta no código em XML, devido ao seu esquema XML simplificado. Enfatizamos que não são todos os tipos de consulta porque sua maior desvantagem é não representar estruturadamente o conteúdo dos construtores e métodos, e sim apenas transcrever o conteúdo do formato texto para dentro do marcador `code`, perdendo informações importantes, como definições de variáveis, invocação de métodos e instanciação de outras classes. Isso obriga a utilização de pesquisa em texto, o que vai de encontro aos benefícios oferecidos pela linguagem XML. Outra desvantagem de Xjava é que ela também não armazena informações estruturais do código, como linhas, colunas, espaços em branco, e formatação em geral, prejudicando a conversão desse formato para a representação textual.

3.1.2. REPRESENTAÇÕES PARA C++

Ao contrário de Java, C++ possui apenas duas representações de código em XML, ambas denominadas CppML, sendo uma proposta por Mamas e Kontogiannis [36] e a outra pela empresa Columbus [15]. Para cada uma delas iremos mostrar a representação XML gerada para a classe `Quadrado` mostrada na Figura 7 e discutir seus principais aspectos.

3.1.2.1. CPPML DE MAMAS E KONTOGIANNIS

A primeira representação CppML foi proposta por Mamas e Kontogiannis [36] e se assemelha ao padrão JavaML, também proposto pelos autores, pois representa informações de código no nível de AST em forma de XML. Outra semelhança entre as representações é o fato de representar elementos do código em marcadores e seus valores correspondentes em atributos, sem utilizar o corpo dos marcadores, como pode ser visto na Figura 8.

O arquivo XML gerado engloba todos os arquivos de cabeçalho (*.h*) e de código (*.cpp*) que foram transformados. Seu elemento inicial é `Program`, que armazena no seu atributo `name` o nome do projeto e possui o sub-elemento `IncludeSource` representando cada arquivo que foi transformado para a representação XML, este possuindo o atributo `name` para identificar o nome do arquivo correspondente. Cada elemento `IncludeSource` possui dois sub-elementos: `Include`, que como o próprio nome indica, representa um arquivo cabeçalho incluído no dado arquivo, e `Class`, que traz todas as informações da classe que representa. Atributos e variáveis são representados pelo marcador `Variable`, assim como funções e construtores são representados pelo marcador `Function`. A diferença está no atributo `kind` de cada marcador, que indica se ele é uma variável ou atributo, ou uma função

```
class Quadrado {
  private:
     int lado;

  public:

     void setLado(int iLado) {

         lado = iLado;

    };
```

Figura 7: Exemplo de classe C++.

```
<Program name="..\avl.icc">
   <IncludedSource name="quadrado.h">
      <Class name="Quadrado" line="37" column="7" accessKind="public"
         kind="Class" abstract="False" anonymous="False"
         unnamed="False">
         <Variable name="Quadrado::lado" line="41" column="17"
            accessKind="private" kind="RegularMember" mutable="False"
            UnnamedBitfield="False" UnboundedArray="False">
            <TypeDescriptor name="int" kind="PredefinedType" constant="False"
               volatile="False" />
         </Variable>
         <Function name="Quadrado::setLado" line="55" column="11"
            accessKind="public" kind="RegularMember" virtual="False"
            purevirtual="False" inline="True" explicit="False" operator="New">
            <TypeDescriptor name="void (int)" kind="Function" constant="False"
               volatile="False">
            <FunctionParameter identifier="iLado" register="False">
               <TypeDescriptor name="int" kind="PredefinedType"
                  constant="False" volatile="False" />
            </FunctionParameter>
            <TypeDescriptor name="void" kind="PredefinedType" constant="False"
               volatile="False" />
            </TypeDescriptor>
            <FunctionBody>
               . . .
            </FunctionBody>
         </Function>
      </Class>
   </IncludedSource>
</Program>
```

Figura 8: Representação da classe Quadrado para o formato CppML de Mamas e Kontogiannis.

ou construtor.

Os nomes dos elementos do código são representados através do atributo `name`, presente em cada marcador que representa um elemento, à exceção dos parâmetros de métodos e construtores, cujo nome é armazenado no atributo `identifier`. Tipos de dados de atributos, variáveis e parâmetros, bem como o valor de retorno de métodos, são sempre representados pelo atributo `name` do marcador `TypeDescriptor`, interna à que representa o elemento do código correspondente.

Ao contrário do padrão para Java proposto pelos mesmos autores, CppML não é uma representação verbosa, facilitando a especificação de consultas. Sua principal vantagem é que a transformação é feita no código antes do pré-processamento, ou seja, ele representa tanto o código fonte quanto os arquivos de cabeçalho, permitindo especificar consultas a elementos que estejam presentes em apenas um arquivo. As desvantagens também são as mesmas de JavaML de Mamas e Kontogiannis, ou seja, ausência de representação para comentários, formatação e linhas em branco.

3.1.2.2. CPPML DE COLUMBUS

Um outra representação CppML foi proposta pela empresa Columbus [15]. Assim como a representação anterior, CppML representa elementos do código em marcadores e seus valores correspondentes em atributos, sem utilizar o corpo dos marcadores para armazenar informação. Também representa informações estruturais como linhas e colunas iniciais e finais para cada elemento do código. Esse padrão traz como novidade o uso de *namespaces* na nomenclatura dos marcadores, como pode ser visto na Figura 9, e a representação de comentários feitos no código original.

O elemento inicial da representação é `Project`. Ele possui três sub-elementos: `struc:Namespace`, que por sua vez engloba os elementos `struc:Class`, contendo a definição de cada classe, `type:SimpleType`, que representa cada tipo de dados utilizados

```
<Project name="avl">
<struc:Namespace id="id100" name="global namespace">
 <struc:Class id="id123" path="C:\Quadrado.h" line="37" col="-1"
    endLine="59" endCol="-1" comment="" name="Quadrado"
    accessibility="ackNone" storageClass="sckNone" nonISOSpec=""
    kind="clkClass" isAbstract="false" isDefined="true">
   <struc:Object id="id1927" path="C:\Quadrado.h" line="41" col="-1"
      endLine="41" endCol="-1" comment="" name="lado"
      accessibility="ackPrivate" storageClass="sckNone" nonISOSpec="">
      <struc:hasTypeRep ref="id104"/>
  </struc:Object>
  <struc:Function id="id1963" path="C:\Quadrado.h" line="55" col="-1"
     endLine="55" endCol="-1" comment="" name="setLado"
     accessibility="ackPublic" storageClass="sckNone" nonISOSpec=""
     mangledName="setLado@void@(int)" kind="fnkNormal"
     isVirtual="false" isPureVirtual="false" isInline="false"
     isExplicit="false">
     <struc:hasTypeRep ref="id1394"/>
     <struc:Parameter id="id1964" path="C:\Quadrado.h" line="55" col="-1"
        endLine="55" endCol="-1" comment="" name="iLado"
        isEllipsis="false">
        <struc:hasTypeRep ref="id104" />
     </struc:Parameter>
     <struc:hasBody>
        <statm:Block id="id1965" path="C:\Quadrado.h" line="55" col="-1"
          . . .
        </statm:Block>
     </struc:hasBody>
  </struc:Function>
  </struc:Class>
 </struc:Namespace>
</Project>
```

Figura 9: Representação da classe Quadrado para o formato CppML de Columbus.

nas classes; e `type:TypeRep`, que é uma estrutura que armazena toda referência a um tipo de dado no código. Dentro de `struc:Class` existem os elementos `struc:Object`, representando os atributos, e `struc:Function`, representando as funções e construtores. Assim como em CppML de Mamas e Kontogiannis, a diferença está no atributo `kind`, que indica o tipo correto daquele marcador.

Os nomes dos elementos do código são armazenados no atributo `name` do marcador que o representa. Já o tipo de dados de atributos, variáveis e parâmetros, além do tipo de retorno de métodos, não é acessível nesse mesmo marcador. É necessário pesquisar no marcador interno `hasTypeRep` o valor do atributo `ref`, que é representa um "código" para o tipo de dados, e cruzá-lo com uma lista dos tipos de dados utilizados na representação, que encontra-se no marcador `type:TypeRep`.

Um ponto positivo da representação é que todo elemento de código possui um atributo `id` no marcador que o representa, o que facilita a identificação do elemento em outros trechos do código. Contudo, a grande desvantagem de CppML de Columbus é que, diferentemente do seu padrão homônimo, este não representa a AST do código em formato XML, mas sim representa em XML o código já pré-processado, ou seja, uma mistura do código fonte com as definições do arquivo de cabeçalho. Com isso, muitas informações são perdidas, como por exemplo, a superclasse de uma determinada classe, dificultando a especificação de consultas.

3.2. REPRESENTAÇÕES GENÉRICAS

Esse tipo de modelo não se destina a representar uma linguagem de programação específica, mas sim as características comuns de linguagens orientadas a objeto num formato XML. Utilizar esse tipo de representação pode ser útil para desenvolver ferramentas de manutenção genéricas, como por exemplo calcular métricas de sistemas orientados a objeto ou mesmo extrair informações para análise de código OO, como dependência de classes. Entre essas representações podemos destacar três: OOML [36], GXL [22] e srcML. [35]. A seguir discutiremos um pouco mais sobre cada representação.

3.2.1. OOML

OOML [36] também foi proposta por Mamas e Kontogiannis e é um padrão no qual estão presentes as características comuns utilizadas pelas linguagens Java e C++. Pode ser obtido não só diretamente através do código fonte original, mas também como um mapeamento a partir de JavaML ou CppML propostos por eles, como mostra a Tabela 1 [36]. Apesar de ser uma representação que serviria para muitas linguagens, sua desvantagem é a perda de características próprias de cada linguagem, já que nem todas possuem as mesmas estruturas.

3.2.2. GXL

GXL [22] é uma padrão de representação de estruturas de código em forma de grafos direcionados. Entidades são representadas como nós, enquanto arestas representam o relacionamento entre elas. Apesar de também representar o código fonte, GXL foi originalmente projetada para ser um formato padrão de troca de artefatos de software em diferentes níveis de abstração; por isso, informações relativas à estrutura do código, como comentários e formatação, não são preservadas.

3.2.3. SRCML

Uma outra abordagem que visa representar código fonte por completo, sem perda de informações, é srcML [35]. Porém, diferentemente das representações específicas, srcML

JavaML	CppML	OOML
CompilationUnit	Program	Program
ImportDeclaration	Include	Include
ClassDeclaration	Class	Class
MethodDeclaration	Function	Method
FieldDeclaration	Variable	VariableDeclaration
Block	LexicalBlockStatement	Body
Name	NameExprtession	VariableUse
Type	TypeDescriptor	Type
FormalParameter	-	Parameter

Tabela 1: Mapeamento de alguns elementos de JavaML e CppML, ambos de Mamas e Kontogiannis, para OOML.

não representa a AST do código, e sim o "anota" com marcadores XML, preservando, assim, informações como espaços em branco, comentários e a formatação do texto. Sua vantagem é que não é necessário fazer um *parser* do código para gerar a AST. Contudo, como srcML apenas adiciona marcadores ao código, ela torna-o muito extenso e com uma abundância de informações não essenciais para sua manipulação.

3.3. ADEQUAÇÃO DOS MODELOS PARA REFATORAÇÃO

Nem todos as representações acima são adequadas para aplicar refatoração, ou então são úteis apenas para certos tipos de refatorações. Isso porque elas não trazem todas as informações contidas no código fonte necessárias para testar as pré e pós-condições de alguns tipos de refatorações, ou mesmo para poder aplicar as próprias operações de refatoração.

As representações genéricas OOML e GXL, como não guardam todas as informações contidas no corpo de métodos, não são adequadas para se utilizar em refatorações como *Pull Up Method* e *Pull Down Method*, já que ambas se destinam a mover um método para a superclasse e para a subclasse, respectivamente. Como a transformação das linguagens Java e C++ para OOML pode deixar de transformar todo o conteúdo dos métodos, pois pode haver nesse conteúdo detalhes específicos da linguagem, aplicar essas refatorações podem introduzir erros no código. Elas são mais indicadas para se utilizar em refatorações de criação, renomeação e remoção de elementos de código, pois são informações presentes nessas representações.

O uso de XJava pode ser proibitivo do ponto de vista de custo de processamento, pois como armazena o conteúdo de métodos e construtores em um único marcador, obriga a utilização de pesquisa em texto para encontrar elementos de código desejados.

O padrão CppML de Columbus não é adequado para a aplicação de operações de refatoração pois, como já foi mencionado, este não traz todas as informações do código fonte original, já que o código em XML representa o estado do código após o pré-processamento. Por exemplo, caso se deseje alterar o nome de uma classe, a refatoração não será completa, pois não há como sincronizar o arquivo de código com o arquivo cabeçalho.

As representações JavaML permitem especificar todos os tipos de refatoração, porém JavaML de Badros leva vantagem em relação ao de Mamas e Kontogiannis pois apresenta características que facilitam a especificação de pré-condições e operações de refatoração, como mostradas na seção 3.1.1. O fato de armazenar o nome dos elementos do

código em marcadores mais internos aos que representam esses elementos, faz com que determinadas operações de refatoração sejam mais trabalhosas de especificar para JavaML de M&K do que para o de Badros, como por exemplo operações de remoção de elementos. Contudo, ambos padrões JavaML apresentam limitações na transformação do XML para a representação textual, pois ambas não guardam informações de comentários, formatação e linhas em branco. Assim, sempre que uma refatoração for aplicada, o usuário irá perder a formatação do seu código original, o que não é desejável. Porém a versão 2.0 de JavaML de Badros já implementa as limitações da primeira versão, o que a torna a representação mais completa para especificar operações de refatoração e pré e pós-condições entre as pesquisadas.

3.4. CONCLUSÃO

O número de representações de linguagens de programação em XML vem crescendo cada vez mais. À medida que essas representações vão sendo utilizadas por ferramentas de manutenção ou reengenharia, seus pontos positivos e negativos poderão ser melhor identificados, e seu uso explorado no contexto que melhor se enquadrem.

No contexto de refatoração, é importante criar um mecanismo para que o desenvolvedor possa ficar livre para escolher a representação que melhor atenda a suas necessidades, facilitando a manipulação do código. RefaX foi desenvolvido tendo em vista esse pensamento, tanto para representar quanto para manipular o código em XML, ou seja, ele deixa o desenvolvedor decidir que representação e padrão de manipulação de dados XML são mais apropriados para a sua ferramenta. Alguns dos padrões de consultas e manipulação de dados XML mais utilizados atualmente são discutidos no próximo capítulo.

CAPÍTULO 4

TECNOLOGIAS DE MANIPULAÇÃO DE DADOS XML

Com a transformação do código para o formato XML, surge a necessidade de mecanismos que permitam manipulá-lo nessa nova representação. A seção 4.1 descreve as duas linguagens de consultas a dados XML padronizadas pelo World Wide Web Consortium (W3C) [67], enquanto as principais tecnologias de atualização de dados XML são descritas na seção 4.2. Os exemplos mostrados neste capítulo são referentes à representação da classe `Funcionario` no formato JavaML de Badros, apresentado na Figura 5 no Capítulo 3.

4.1. LINGUAGENS DE CONSULTA

O aumento da quantidade de informação que é armazenada, trocada e apresentada utilizando XML tornou a habilidade de consultar dados XML uma atividade cada vez mais importante. Como XML pode armazenar diferentes tipos de informação de diversas fontes, uma linguagem de consulta XML deve fornecer características para recuperar e interpretar informações provenientes dessas diferentes fontes. Nas subseções a seguir discutimos as duas principais linguagens de consulta para dados XML: XPath [63] e XQuery [64].

4.1.1. XPATH

XPath [63] é uma linguagem de consulta que acessa partes de documentos XML através de expressões de caminho. Além de descrever um caminho para percorrer a árvore XML, expressões XPath também oferecem manipulação com *strings*, números e expressões booleanas.

XPath geralmente é utilizado dentro de outras tecnologias, como XSLT [65], ou usado com uma API para extrair partes de documentos XML para futuros processamentos. Sua maior desvantagem é realizar apenas consultas simples, não permitindo ao usuário fazer consultas mais complexas, como consultas aninhadas ou condicionais.

As Figuras 10 e 11 mostram, respectivamente, exemplos de consultas XPath para acessar todos os métodos de todas as classes e retornar o nome do primeiro atributo da classe `Funcionario`.

4.1.2. XQUERY

XML Query Language – XQuery [64] é uma linguagem de consulta projetada para ser concisa e ter uma sintaxe de fácil entendimento por parte dos usuários. É também flexível o bastante para consultar diferentes fontes de dados XML, estejam eles armazenados em arquivos ou em bancos de dados nativos.

XQuery é uma extensão de XPath. Sendo assim, também utiliza o conceito de expressão de caminho para navegar em árvores XML. Contudo, XQuery se diferencia por suas consultas serem compostas por expressões baseadas nas cláusulas FOR, LET, ORDER BY, WHERE e RETURN (FLOWR, lê-se *flower*), que permitem que várias expressões sejam aninhadas dentro da mesma consulta, tornando-a mais legível, simples, e poderosa.

Além disso, XQuery oferece operadores para ordenar os nós resultantes de uma consulta, operadores matemáticos como COUNT e AVG, permite realizar consultas condicionais através dos operadores IF/THEN/ELSE, e encapsular consultas em funções, o que possibilita a especificação de consultas recursivas.

Com XQuery é possível formatar o resultado de uma consulta de maneira que ele seja um documento XML totalmente diferente do de entrada. O mesmo não ocorre com XPath, já que o retorno de uma consulta é sempre um conjunto de nós da árvore XML ou o valor desses nós ou de algum de seus atributos, não permitindo que os mesmos sejam manipulados na própria linguagem.

```
/java-source-program/java-class-file/class//method
```

Figura 10: Consulta XPath que retorna todos os métodos de todas as classes.

```
/java-source-program/java-class-file/ class[@name =
"Funcionario"]//field[1]/@name
```

Figura 11: Consulta XPath que retorna o nome do primeiro atributo da classe Funcionário.

As Figuras 12 e 13 mostram exemplos de consultas XQuery e que seriam extremamente complexas de serem especificadas caso fossem expressas usando somente XPath. A primeira retorna os métodos e os atributos da classe `Funcionario` que sejam do mesmo tipo, enquanto a segunda mostra o nome de todos os atributos da classe `Funcionario` que são usados em algum método. Nesses exemplos, as características não suportadas por XPath são a capacidade de comparar o valor do atributo de um nó ao valor do atributo de um nó mais interno, como por exemplo `$f/@name = $m/@name`, e de formatar o retorno da consulta, como demonstrada pelos marcadores `par`, na Figura 12, e `atributo`, na Figura 13.

```
<consulta>
{
  LET $doc := document("Funcionario.xml")
  FOR $c in $doc/java-source-program/java-class-
file/class[@name='Funcionario'],
      $f in $c//field/:type,
      $m in $c//method/:type
  WHERE $f/@name = $m/@name
  RETURN
   <par>
     <metodo nome="{$m/../@name}">
     <atributo nome ="{$f/../@name}">
   </par>
}
</consulta>
```

Figura 12: Consulta XQUERY que retorna pares com os nomes dos métodos e atributos da classe `Funcionario` que sejam do mesmo tipo.

```
<consulta>
{
  LET $doc := document("Funcionario.xml")
  FOR $ c in $doc/java-source-program/java-class-
file/class,
      $f in $c//field,
      $var in $c//method//block//var-ref
  WHERE $f/@name = $var/@name
  RETURN
    <atributo name= "{$f/@name}"></atributo>
}
</consulta>
```

Figura 13: Consulta XQUERY que retorna o nome de todos os atributos da classe `Funcionario` que são utilizados dentro de algum método.

4.2. TECNOLOGIAS DE ATUALIZAÇÃO

Além da consulta aos dados em armazenados em XML, outra necessidade inerente à utilização dessa nova representação é a capacidade de atualizar esses dados. Para isso, utiliza-se uma linguagem que atualize diretamente os dados no documento XML ou transforme um documento XML em outro. Como até o momento o W3C não definiu uma linguagem padrão para atualização de dados XML, algumas empresas e pesquisadores desenvolveram especificações próprias para esse tipo de tecnologia.

Mostraremos a seguir as principais linguagens de atualização propostas, quais sejam, XQuery Update [33], o padrão proposto por Tatarinov *et al*. [56], e XUpdate [69], juntamente com dois dos padrões de transformação de documentos XML mais utilizados, que são XSLT [65] e XML TreeDiff [23].

4.2.1. XQUERY UPDATE

XQuery Update [33] é uma extensão de XQuery para atualização de documentos XML baseado na especificação de atualizações feita por membros do XQuery Working Group. Expressões de consulta FLOWR de XQuery foram modificadas para permitirem atualizações mais sofisticadas, tornando-se expressões do tipo UPDATE ... FOR ... LET ... WHERE ... *Op*, onde *Op* representa uma ou mais operações de atualização.

Esse padrão fornece funcionalidades para inserção, exclusão, atualização de valor e renomeio de marcadores e atributos, através dos operadores INSERT AFTER/BEFORE, DELETE, REPLACE, RENAME, respectivamente, além de permitir consultas condicionais com cláusulas IF/THEN/ELSE e atualizações múltiplas, isto é, várias operações de atualização dentro de uma mesma consulta.

Sua grande vantagem é a facilidade de especificar consultas, tornando-as mais legíveis. Porém, até o momento XQuery Update é suportada apenas por uma ferramenta, cuja implementação ainda não está estabilizada (nem sempre retorna os resultados corretos). A Figura 14 mostra a inserção de um novo atributo à classe `Funcionario`, enquanto a Figura 15 mostra a renomeação de todas as variáveis denominadas `salarioBase` para `valorSalario`, ambas utilizando XQuery Update.

```
   UPDATE
   LET $doc := document ("Funcionario.xml")
   FOR    $c    in    $doc/java-source-program/java-class-
file/class
   WHERE $c/$name="Funcionario"
  INSERT
     <field name="novo_atributo">
         <type name="long" primitive="true" />
     </field>
  AFTER $c/field[1]
```

Figura 14: Inserção de um novo atributo na classe `Funcionario` usando XQuery Update.

```
   UPDATE
   LET $doc := document ("Funcionario.xml")
   FOR   $c   in   $doc/java-source-program/java-class-file/
class,
        $m in $c//method,
        $v in $m//local-variable[@name='salarioBase']
   WHERE $c/$name="Car"
   REPLACE $v/@name with "valorSalario"
```

Figura 15: Renomeação da variável `salarioBase` para `valorSalario` na classe `Funcionario` usando XQuery Update.

4.2.2. PADRÃO PROPOSTO POR TATARINOV ET AL

Outro padrão que visa complementar a estrutura atual de XQuery, provendo-lhe a capacidade de atualizar documentos XML, é proposto por Tatarinov et al [56]. Assim como no padrão anterior, também são propostas aqui extensões que abrangem as operações de inserção, remoção, substituição e modificação de valor de nós e atributos XML, utilizando os mesmos operadores descritos por XQuery Update.

Esse padrão também é capaz de realizar atualizações múltiplas, porém não especifica consulta com operadores condicionais. Outra diferença está na sintaxe: o padrão de Tatarinov *et al* estendeu as expressões FLOWR de XQuery com a seguinte estrutura para atualizações: FOR...LET...WHERE...UPDATE...*Op*, onde *Op* representa uma ou mais operações de atualização.

Assim como XQuery Update, sua vantagem está na facilidade de criar consultas dada sua alta legibilidade. Contudo, o grande empecilho de utilizar esse padrão é a falta de

ferramentas de suporte, já que o protótipo desenvolvido pelos autores visava traduzir consultas XML em consultas SQL para acessar um banco de dados relacional.

As Figuras 16 e 17 mostram as mesmas consultas das Figuras 14 e 15, porém escritas usando o padrão de Tatarinov *et al*.

4.2.3. XSLT

XSL Transformations (XSLT) [65], é a mais usada linguagem de transformação de documentos XML. Uma transformação XSLT é expressa como um documento XML bem formado que inclui elementos definidos pela linguagem e elementos externos definidos pelo usuário. Uma de suas principais aplicações é a transformação de dados XML em HTML.

Uma transformação expressa em XSLT descreve regras de transformação de uma árvore XML em outra 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-o a cada padrão do modelo, e caso eles casem, a regra é aplicada.

```
   LET $doc := document ("Funcionario.xml")
   FOR   $c   in   $doc/java-source-program/java-class-
file/class
   WHERE $c/$name="Funcionario"
   UPDATE
      INSERT
       <field name="novo_atributo">
         <type name="long" primitive="true" />
        </field>
       AFTER $c/field[1]
```

Figura 16: Inserção de um novo atributo na classe `Funcionario` usando o padrão proposto por Tatarinov et al.

```
   LET $doc := document ("Funcionario.xml")
   FOR $c in $doc/java-source-program/java-class-file/
class,
       $m in $c//method,
       $v  in  $m//local-variable[@name='salarioBase']

   WHERE $c/$name="Funcionario"
   UPDATE
      REPLACE $v/@name WITH "valorSalario"
```

Figura 17: Renomeação da variável `salarioBase` para `valorSalario` na classe `Funcionario` usando o padrão proposto por Tatarinov et al.

Uma transformação XSLT é chamada de folha de estilo (*stylesheet*), e usa a linguagem de expressão definida por XPath para selecionar elementos do documento. Uma vantagem de utilizar XSLT é que, por ser bastante difundida, existem diversos processadores que implementam essa linguagem. Entretanto, sua sintaxe não é muito legível quando comparada à de outras linguagens de atualização. Além disso, por não ser projetada para atualizar documentos, alguns tipos de transformações que seriam simples de especificar numa linguagem própria de atualização tornam-se relativamente complexas, como por exemplo, as consultas mostradas nas Figuras 18 e 19, que mostram as mesmas atualização da Figuras 14 e

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template
     match="*|@*|processing-instruction()|text()">
    <xsl:copy>
      <xsl:apply-templates
        select="*|@*|processing-instruction()|text()"/>
    </xsl:copy>
  </xsl:template>
  <xsl:template match="//class[@name='Funcionario']//field”>
    <xsl:element name=”field”>
      <xsl:attribute name=”name”>novo_atributo
      </xsl:attribute>
      <type name=”long” primitive=”true”/>
    </xsl:element>
  </xsl:template>
</xsl:stylesheet>
```

Figura 18: Inserção de um novo atributo na classe Funcionario usando XSLT

```
<?xml version="1.0"?>
<xsl:stylesheet version="1.0"
  xmlns:xsl="http://www.w3.org/1999/XSL/Transform">
  <xsl:template
     match="*|@*|processing-instruction()|text()">
    <xsl:copy>
      <xsl:apply-templates
        select="*|@*|processing-instruction()|text()"/>
    </xsl:copy>
  </xsl:template>
  <xsl:template match="//class[@name='Funcionario']//method// local-
variable[@name='salarioBase']”>
    valorSalario
  </xsl:template>
</xsl:stylesheet>
```

Figura 19: Renomeação da variável `salarioBase` para `valorSalario` na classe `Funcionario` usando XSLT.

15, porém escritas em XSLT.

4.2.4. XUPDATE

Diferentemente de XQuery Update e o padrão proposto por Tatarinov et al, que complementam a estrutura de XQuery, XUpdate [69] é uma linguagem de atualização na qual uma operação de atualização é expressada como um documento XML bem formado. Para tanto, esta linguagem utiliza elementos que definem que tipo de operação será realizado no documento, que variam entre criar, inserir antes/depois, remover, renomear e atualizar o valor de nós.

XUpdate utiliza XPath para selecionar nós e é baseada em definições como as de XSLT. Contudo, sua sintaxe XML é mais legível, o que gera atualizações mais simples e fáceis de especificar. A grande vantagem de XUpdate é ser a linguagem escolhida pelo consórcio Apache [2] para atualização de dados XML. Isso mostra a confiabilidade na linguagem e impulsiona novas pesquisas para melhorá-la. Boa parte dos bancos de dados nativos XML implementa essa linguagem como padrão para atualização de documentos.

A desvantagem é a ausência de atualizações condicionais na atual versão da linguagem. As Figuras 20 e 21 descrevem em XUpdate as mesmas modificações mostradas nas Figuras 14 e 15.

```
<xupdate:modifications version="1.0"
   xmlns:xupdate="http://www.xmldb.org/xupdate">
<xupdate:append select="/java-source-program / java-class-
file/class[@name=”Funcionario”]/field " child="last()">
   <xupdate:element name="field">
     <xupdate:attribute name="name"> novo_atributo
     </xupdate:attribute>
     <xupdate:element name="type">
        <xupdate:attribute name="name"> long
         </xupdate:attribute>
           <xupdate:attribute name="primitive"> true
         </xupdate:attribute>
     </xupdate:element>
   </xupdate:element>
 </xupdate:append>
</xupdate:modifications>
```

Figura 20: Inserção de um novo atributo na classe `Funcionario` usando XUpdate.

```
<xupdate:modifications version="1.0"
   xmlns:xupdate="http://www.xmldb.org/xupdate">
   <xupdate:update select="/java-source-program / java-
class-file/class[@name=”Funcionario”]//method//local-
variable [@name='salarioBase']">
      valorSalario
   </xupdate:append>
</xupdate:modifications>
```

Figura 21: Renomeação de todas as variáveis `salarioBase` para `valorSalario` na classe `Funcionario` usando XUpdate.

4.2.5. XML TREE DIFF

XML TreeDiff [23] é uma ferramenta de diferenciação e atualização de documentos XML desenvolvida pela IBM, que utiliza a árvore DOM de cada documento, ao invés de trabalhar com suas representações textuais. Ele computa a diferença entre dois documentos XML como uma seqüência de operações básicas que transformam o primeiro no segundo e mostra essa seqüência como um documento XML de resultado.

Como XML TreeDiff interpetra os documentos como árvores DOM, essas operações são descritas em termos de operações nativas de inserção, remoção e substituição de nós em árvores. Além disso, a ferramenta também oferece operações agregadas (podar, enxertar e mover um ramo da árvore) com intuito de aumentar a concisão e legibilidade do resultado, podendo todas ser descritas em termos das três operações básicas.

Essas operações são representadas por uma das duas linguagens de atualização: FUL (Flat Update Language) ou XUL (XML Update Language). FUL codifica a saída como uma lista não ordenada de operações, onde cada operação é representada por um elemento da linguagem, e as informações necessárias são geralmente codificadas nos atributos desses elementos. Ao contrário de FUL, XUL fornece uma visão estruturada da diferença dos documentos. Isto pode implicar em um documento de resultado mais extenso, em compensação, mais legível. Ambas utilizam XPath para acessar os nós da árvore XML.

Embora seja possível criar consultas aninhadas, essas linguagens não permitem criar consultas mais complexas que façam uso de expressões condicionais. Além disso, assim como XSLT, a sintaxe das linguagens de atualização é menos legível do que a de outras linguagens de atualização. As Figuras 22 e 23 mostram as consultas das Figuras 14 e 15 escritas em FUL e XUL, respectivamente.

```
<diff>
<add match="field" type="1" parent="/java-source-program/java-
class-file/class[@name='Funcionario']"  psib="="/java-source-
program/java-class-file/class[@name='Funcionario']/field[1]"
name="field" />
   <children>
           <child>
                    <type name="int" primitive="true"\>
           </child>
   </children>
</add>
```

Figura 22: Inserção de um novo atributo na classe `Funcionario` usando FUL.

```
<node id="/ java-source-program/java-class-file/class[@name='Funcionario']/
/method//local-variable[@name='salarioBase'] " op="replace" type="2"
name="name" />
<value>valorSalario</value>
```

Figura 23: Renomeação da variável `salarioBase` para `valorSalario` na classe `Funcionario` usando XUL.

4.3. CONCLUSÃO

Existem muitas opções de linguagens de manipulação de dados XML, e com certeza muitas outras surgirão. Deixar o desenvolvedor escolher com qual linguagem deseja trabalhar é um fator que aumenta a customização das ferramentas que utilizam essas tecnologias.

O arcabouço RefaX foi projetado visando atender essa necessidade, bem como outras discutidas nos capítulos anteriores, como independência de tecnologia e de esquema XML para representação de código. No próximo capítulo, descreveremos RefaX em mais detalhes, e discutiremos suas principais características.

CAPÍTULO 5

REFAX: UM ARCABOUÇO PARA REFATORAÇÃO BASEADO EM XML

Este capítulo apresenta RefaX [38][39], um arcabouço que utiliza tecnologias XML como mecanismo de representação e manipulação do código, a fim de auxiliar a construção de ferramentas de refatoração mais facilmente reutilizáveis, customizáveis e extensíveis. RefaX segue um processo de refatoração de código baseado em XML, também proposto por nós, através do qual o desenvolvedor pode usufruir da grande variedade de ferramentas e padrões XML disponíveis.

Na sua versão mais recente, RefaX disponibiliza seis refatorações primitivas que abrangem as categorias de inserção, remoção e renomeação de elementos de código. São elas: *Add Class*, *Add Attribute*, *Add Method*, *Rename Class*, *Rename Attribute* e *Remove Attribute*. RefaX também possibilita ao desenvolvedor criar novas refatorações, sejam elas a partir da junção de duas ou mais já disponíveis (o que gera uma refatoração composta), ou pela implementação de refatorações independentes das existentes.

O restante deste capítulo está organizado da seguinte forma: a seção 5.1 descreve o processo de refatoração baseado em XML no qual RefaX se baseia; a seção 5.2 discute os requisitos de RefaX; a arquitetura do arcabouço é apresentada na seção 5.3, enquanto as principais decisões de projeto são discutidas na seção 5.4; alguns aspectos de implementação de RefaX são descritos na seção 5.5; a seção 5.6 indica diretrizes para a instanciação do arcabouço, e a seção 5.7 encerra o capítulo.

5.1. PROCESSO DE REFATORAÇÃO DE CÓDIGO BASEADO EM XML

O processo de refatoração de código baseado em XML que propomos segue os mesmos passos do processo de refatoração descritos na seção 2.4, porém, como sua ênfase é a aplicação das refatorações em si, foca nos passos 3 a 5 (garantia da preservação do comportamento, aplicação das refatorações, e verificação da preservação do comportamento). Partimos do pressuposto de que o usuário é responsável pela detecção e determinação de qual

refatoração utilizar, seja manualmente, com o mantenedor explorando sua própria experiência e conhecimento do código para selecionar a melhor refatoração a ser aplicado no local apropriado do código fonte, ou com a ajuda de soluções semi-automatizadas, como as oferecidas pelas ferramentas de métricas e engenharia reversa.

Em nosso processo, os passos 3 a 5 foram redefinidos para se adaptarem ao conjunto de modelos de dados e tecnologias de processamento XML utilizados. O processo resultante foi então reorganizado dentro dos seguintes novos passos:

1. Conversão do código fonte para XML.
2. Armazenamento dos dados XML gerados.
3. Aplicação de refatoração via manipulação de dados XML
4. Conversão de XML para código fonte

A Figura 24 ilustra os relacionamentos entre esses 4 passos como parte do processo de refatoração baseado em XML. As seções seguintes descrevem cada um dos passos em mais detalhes.

5.1.1. CONVERSÃO DO CÓDIGO FONTE PARA XML

O primeiro passo inicia quando o usuário seleciona os arquivos do código fonte que serão convertidos para a representação XML escolhida. O processo de conversão é geralmente realizado através de ferramentas automatizadas, como *parsers* de linguagem ou ferramentas de processamento de *strings*. Nos referimos à ferramenta responsável por esse passo como *conversor XML*. Na Figura 24, o elemento *Código para XML* representa o conversor utilizado no processo.

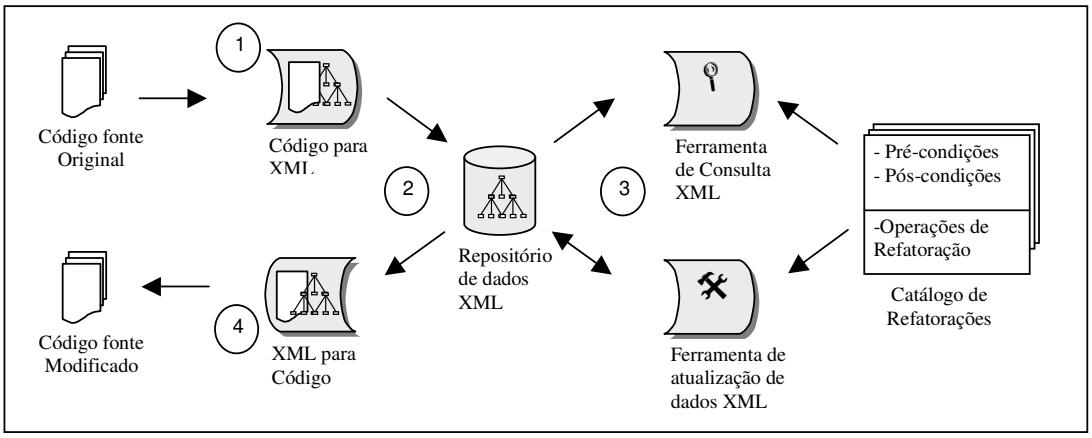


Figure 24: Processo de refatoração de código baseado em XML.

5.1.2. ARMAZENAMENTO DOS DADOS XML GERADOS

Uma vez que os arquivos do código fonte foram convertidos para a correspondente representação XML, esses dados devem ser armazenados em algum tipo de repositório, representado na Figura 24 pelo elemento *Repositório de Dados XML*, para que eles possam ser eficientemente acessados nos passos subseqüentes.

Esse repositório pode variar desde um simples diretório de arquivos até a um robusto banco de dados nativo XML. A escolha dependerá do tipo de tecnologia de processamento XML utilizada nos passos de refatoração, por exemplo, se tais tecnologias são integradas com bancos de dados ou acessam os arquivos individualmente, colocando-os em memória, bem como vários outros fatores, como a quantidade de informação de código que será manipulada e o nível de performance e/ou escalabilidade esperado da ferramenta de refatoração.

5.1.3. APLICAÇÃO DE REFATORAÇÃO VIA MANIPULAÇÃO DE DADOS XML

Nesse passo, o usuário escolhe a refatoração que será aplicada ao código fonte em XML convertido no passo 1. Esse passo é subdividido em três etapas: teste de pré-condições, aplicação das operações de refatoração, e verificação da preservação do comportamento. Essas etapas são descritas a seguir.

a) Teste de pré-condições

As pré-condições são formadas por combinações de *Funções de Análise*, como proposto por Roberts [52]. Elas são divididas em duas categorias: primitivas e derivadas. Como exemplo, considere as funções de análise primitivas *IsClass(className)*, que checa se já existe uma classe denominada *className* no código fonte, e *Superclass(className)*, que retorna a superclasse de *className*, caso essa *className* estenda outra classe, ou Ø, caso contrário. Uma função de análise derivada pode ser descrita em termos de uma ou mais primitivas, como por exemplo, *AllSuperclasses(className)*, que retorna o conjunto de todas as superclasses de *className*, e pode ser descrita da seguinte forma:

- Ø, caso *Superclass(className)* = Ø; ou

- *Superclass(className)* U *AllSuperclasses(Superclass(className))*, caso contrário.

Tichelaar [57] também definiu várias pré-condições para refatorações independentes de linguagens e que foram transformadas em funções de análise. Por acessarem informações do código fonte, sem efetivamente alterá-las, em nosso processo de refatoração as funções de análise são expressas através de alguma linguagem de consulta XML apropriada. Dessa forma, é necessária uma ferramenta de consulta XML que suporte a linguagem escolhida para que as funções de análise possam ser aplicadas ao código representado em XML. Na Figura 24, essa etapa é representada pelo fluxo indicado entre Repositório de Dados XML, Ferramenta de Consulta XML e Pré-condições.

b) Aplicação das operações de refatoração

Uma vez que todas as pré-condições foram validadas, a próxima atividade consiste em aplicar as operações de refatoração propriamente ditas. Uma única refatoração pode ser especificada em termos de múltiplas operações. Essas características não apenas incrementam a modularidade e o reuso das refatorações, mas também fornecem um poderoso mecanismo de abstração que permite descrever refatorações em um nível mais alto, de forma que sejam independente de linguagem de programação e modelo de código fonte. Como cada operação de refatoração altera diretamente o código, em nosso processo de refatoração baseado em XML elas devem ser expressas usando alguma linguagem de transformação ou atualização de dados XML apropriada, e devem ser aplicadas utilizando uma ferramenta de atualização apropriada. Na figura 24 essa etapa consiste no fluxo de informações entre os elementos *Repositório de Dados XML*, *Ferramenta de Atualização de Dados XML*, e *Operações de Refatoração*.

c) Verificação da preservação do comportamento

Após todas as operações de refatoração terem sido executadas, é necessário verificar se o código alterado ainda preserva seu comportamento externo original. Para isso, usamos as pós-condições propostas por Roberts [52] e Tichelaar *et al.* [57]. Assim como as pré-condições, as pós-condições variam de acordo com o tipo e o propósito de cada refatoração. Como as pós-condições são limitadas a acessar apenas informações do código fonte, em nosso processo de refatoração baseado em XML elas também são expressas em

termos de funções de análise, usando uma linguagem de consulta XML e uma ferramenta de consulta XML apropriadas. Essa etapa é representada na Figura 24 pelo fluxo de informações entre os elementos *Repositório de Dados XML*, *Ferramenta de Consulta XML*, e *Pós-condições.*

5.1.4. CONVERSÃO DE XML PARA CÓDIGO FONTE

Uma vez que todas as refatorações foram aplicadas aos devidos elementos do código fonte no repositório, seus resultados devem ser refletidos nos artefatos do código fonte original. Isso é feito geralmente utilizando uma ferramenta de transformação XML, que processa os elementos selecionados do código fonte de acordo com o esquema XML subjacente, gerando uma representação textual equivalente para esses elementos na linguagem de programação original. Nos referimos a esse segundo passo de conversão (de XML para o código fonte) como *reversão*, e à ferramenta de conversão utilizada como *reversor XML*. Na Figura 24 o elemento *XML para código* representa o reversor utilizado no processo.

5.2. REQUISITOS

Os requisitos de RefaX foram, em grande parte, influenciados pelos requisitos de FAMIX [57], um meta-modelo para representar entidades de softwares orientados a objetos e seus relacionamentos de uma forma independente de linguagem.

Além da independência de linguagem, nós também consideramos outros requisitos específicos para XML, como independência de modelo de dados (ou esquema) e independência de tecnologia de processamento. Outros requisitos importantes, como fácil customização e extensão, são intrínsecos do projeto de arcabouços orientados a objetos. Esses e outros requisitos são descritos mais detalhadamente a seguir.

5.2.1. INDEPENDÊNCIA DE ESQUEMA XML

RefaX suporta diferentes modelos de dados XML para uma mesma linguagem de programação. Isso significa que qualquer operação de refatoração escrita para uma determinada linguagem, digamos Java, usando um modelo de dados específico, como XJava, deve ser facilmente aplicada aos elementos do código fonte da mesma linguagem representada em outro modelo de dados, como JavaML. Isso é possível através do uso de funções XQuery que encapsulam e localizem o acesso ao código em XML

5.2.2. INDEPENDÊNCIA DE LINGUAGEM

RefaX suporta a especificação de refatorações em vários níveis de abstração. Isso torna mais fácil para desenvolvedores expressarem operações de refatoração abstratas o suficiente para serem aplicáveis para diferentes linguagens de programação. Este requisito é particularmente importante, já que diferentes linguagens pertencentes a um mesmo paradigma geralmente têm muitas entidades de código em comum, daí oferecendo um número de oportunidades de reuso de refatorações através dessas linguagens. Em geral, quanto mais significativas forem as diferenças entre duas linguagens, mais alto será o nível de abstração necessário para especificar refatorações comuns a elas.

5.2.3. INDEPENDÊNCIA DE TECNOLOGIA

RefaX foi projetado seguindo um sólido projeto de arquitetura, no qual vários pontos de extensão (*hot spots*) são fornecidos para que o desenvolvedor possa facilmente desacoplar a implementação das refatorações do conjunto de tecnologias XML (conversor, repositório, ferramenta de consulta XML, ferramenta de atualização XML e reversor) necessário para executá-los. Isso deixa o desenvolvedor livre para usar qualquer tecnologia no processo de desenvolvimento, e também torna mais fácil a substituição de uma ou mais tecnologias por outras tecnologias alternativas em estágios subseqüentes do ciclo de desenvolvimento.

5.2.4. CONFIABILIDADE

Através da verificação de pré e pós-condições, as refatorações de RefaX garantem a preservação do comportamento do código – desde, é claro, que elas tenham sido propriamente definidas. Em termos práticos, isso significa que novas refatorações devem ser implementadas com cautela e rigorosamente testadas, especialmente com relação às suas pré e pós-condições.

5.2.5. ESCALABILIDADE

Uma refatoração de RefaX deve poder ser efetivamente aplicada a sistemas de vários tamanhos, com poucas ou milhares de linhas de código. Na prática, a performance de qualquer ferramenta baseada em RefaX dependerá fortemente da escalabilidade das tecnologias de processamento XML que foram utilizadas na sua implementação. Como RefaX

foi projetado de uma forma independente de tecnologia, desenvolvedores podem usufruir do arcabouço para experimentar novas tecnologias mais escaláveis, à medida que elas se tornem disponíveis.

5.3. ARQUITETURA

RefaX foi construído seguindo uma arquitetura que visa atender aos requisitos descritos na seção anterior. Essa arquitetura pode ser vista na Figura 25. A seguir detalharemos a funcionalidade de cada elemento da arquitetura e como eles contribuem para atender aos requisitos acima.

5.3.1. FERRAMENTAS REFAX

Esta camada representa qualquer ferramenta de refatoração desenvolvida através da extensão dos serviços oferecidos pelo arcabouço, e com a qual o usuário irá interagir para aplicar as refatorações disponíveis ao código fonte. Esta interação poderá acontecer através de uma interface própria, seja gráfica ou de programação, ou na forma de *plugins* para algum ambiente de desenvolvimento (IDE) existente.

5.3.2. REFAX FACADE

Esta camada é responsável por fornecer uma interface unificada para o conjunto de serviços implementados nas camadas mais internas. Como o próprio nome indica, utilizamos

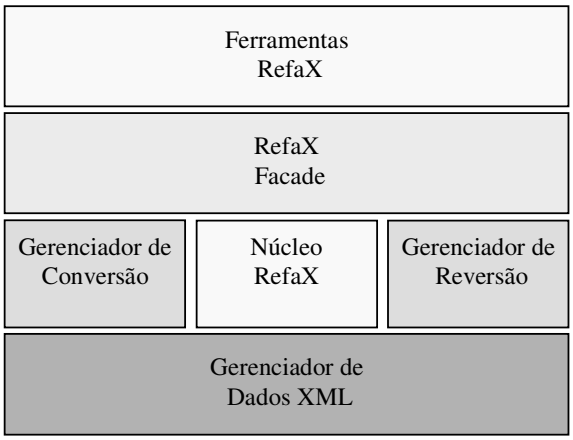


Figure 25: Arquitetura geral de RefaX.

o padrão de projeto *Facade* [20] para oferecer uma única forma de acesso às classes mais internas do arcabouço.

5.3.3. GERENCIADOR DE CONVERSÃO

Esta camada é responsável por oferecer as operações necessárias para converter os arquivos originais do código fonte para a representação XML escolhida pelo desenvolvedor, através da utilização de um conversor.

5.3.4. GERENCIADOR DE DADOS XML

Esta camada tem como função armazenar o código em XML no repositório de dados, bem como fornecer mecanismos para as outras camadas manipularem os dados armazenados.

5.3.5. GERENCIADOR DE REVERSÃO

Esta camada fornece as operações que o desenvolvedor irá utilizar para transformar o código em XML de volta para a sua representação original.

5.3.6. NÚCLEO REFAX

A camada representada na Figura 25 como *Núcleo RefaX* é composta por 6 subcamadas: Refatorações, Funções de Análise, Funções de Acesso ao Código, Gerenciador de aplicação de consulta, Operações de Refatoração, e Gerenciador de aplicação de atualização. Em conjunto, elas são responsáveis pelas refatorações propriamente ditas, testando antes as pré-condições, aplicando as operações de refatoração e verificando as pós-condições. A Figura 26 mostra como essas subcamadas estão estruturadas. Abaixo descrevemos o papel de cada uma na arquitetura.

- **Refatorações**: como o próprio nome indica, ela representa as refatorações disponíveis em RefaX.
- **Funções de análise**: representa as funções de análise (*Analysis Functions - AF*) utilizadas para compor as pré e pós-condições das refatorações.
- **Funções de acesso ao código:** representa o conjunto de funções responsáveis por acessarem cada elemento do código em XML. É ela que implementa o

requisito de independência de esquema XML, pois sempre que um desenvolvedor for instanciar RefaX para uma determinada representação XML, cada função de acesso ao código (*Code Access Functions - CAF*) deve ser especificada para essa representação.

- **Gerenciador de aplicação de consulta**: fornece as operações necessárias para que um processador de consulta possa executar as funções de acesso ao código e funções de análise.
- **Operações de refatoração**: representa cada operação que vá alterar o código XML a fim de aplicar a refatoração desejada. As operações de refatoração (*Refactoring Operation - RO*) não utilizam as funções de acesso ao código pois cada uma utiliza uma tecnologia XML diferente: as funções de acesso são especificadas usando uma linguagem de consulta, pois seu objetivo é pesquisar elementos de código, enquanto que para especificar as operações de refatoração utiliza-se uma linguagem de atualização ou transformação, já que têm a função de alterar o código.
- **Gerenciador de aplicação de atualização**: tem como função oferecer as operações necessárias para a aplicação das operações de refatoração.

5.4. DECISÕES DE PROJETO

Nesta seção, discutiremos algumas decisões de projeto que foram tomadas antes da implementação do arcabouço. A primeira decisão diz respeito à linguagem na qual seria implementado o arcabouço. Era necessário escolher uma linguagem bastante utilizada, para que o arcabouço possa ser facilmente disseminado entre desenvolvedores e pesquisadores, e para a qual existissem APIs de acesso às tecnologias envolvidas, ou seja, conversores,

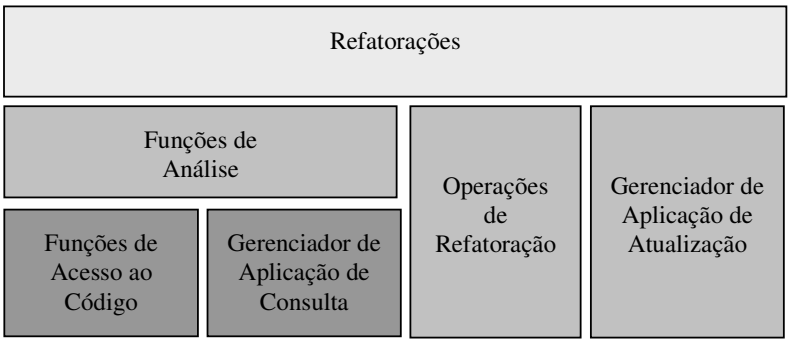


Figure 26: Visão ampliada da camada Núcleo RefaX.

reversores, processadores de consulta, repositórios e ferramentas de atualização. A única linguagem que preenche todos esses requisitos é Java e, por isso, o arcabouço foi implementado nessa linguagem.

O próximo passo foi criar mecanismos para que a implementação do arcabouço atendesse aos requisitos. A independência de tecnologia é garantida através da utilização do padrão *Factory Method* [20], que estrutura o código de tal forma que permite as instâncias de RefaX decidirem que tecnologias utilizar como conversor, reversor, repositório, processador de consulta e ferramenta de atualização. Esse padrão é utilizado nas camadas Gerenciador de Conversão, Gerenciador de Reversão, Gerenciador de Dados XML, Gerenciador de aplicação de consultas e Gerenciador de aplicação de atualização.

A independência de linguagem é obtida através da utilização de refatorações independentes de linguagem propostas por Tichelaar *et al* [57]. As pré e pós-condições foram implementadas em termos de funções de análise, enquanto as operações necessárias para alterar o código foram descritas através das operações de refatoração.

Para implementar a independência de esquema XML, decidimos encapsular todo acesso ao código XML em funções XQuery, chamadas de funções de acesso ao código, que podem ser vistas na Tabela 2. Dessa forma, a especificação dessas funções está ligada à representação XML da linguagem escolhida pelo desenvolvedor. Elas sempre recebem como parâmetro um documento ou nó XML e retornam outros nós XML ou *strings*. Vale ressaltar que foi feita uma distinção entre classes e interfaces, e entre construtores e métodos, com o intuito de facilitar a implementação de funções de análise e, conseqüentemente, as pré e pós-condições das refatorações.

Ainda com relação às funções de acesso ao código e também às funções de análise, outra decisão tomada foi em relação à forma da implementação dessas funções. A idéia original era criar um só arquivo do tipo XQuery, contendo todas as definições das CAFs e AFs. Essa abordagem teria como vantagem o fato de estar concentrado em um só arquivo todas as funções XQuery. Porém, a desvantagem seria a baixa legibilidade que esse arquivo teria, na medida que ele possuiria várias CAFs e AFs, dificultando a redefinição dessas funções e principalmente a criação de novas.

Outra alternativa considerada foi armazenar as CAFs e AFs em um arquivo XML. Contudo, essa idéia traria as mesmas desvantagens de se utilizar um arquivo XQuery. A solução encontrada foi representar cada função de acesso ao código e função de análise com

Funções de acesso ao código	Descrição
getRootNode($doc)	Retorna o nó inicial da representação XML do código
getClass($doc)	Retorna todas as classes contidas no documento XML $doc
getClassName($class)	Retorna o nome da classe contida no nó XML $class
getPackageName($doc)	Retorna o nome do pacote ao qual a documento XML $doc pertence
getImport($doc)	Retorna todas as declarações de importação contidas no documento XML $doc
getImportName($import)	Retorna o nome dos pacotes ou arquivos importados contidos no nó XML $import
getSuperclass($class)	Retorna todas as superclasses da classe contida no nó $class
getSuperclassName($superclass)	Retorna o nome da superclasse contida no nó $superclasse
getConstructor($class)	Retorna todos os construtores da classe contida no nó $class
getAttribute($class)	Retorna todos os atributos da classe contida no nó $class
getAttributeName($attribute)	Retorna o nome do atributo contido no nó $attribute
getAttributeType($attribute)	Retorna o tipo do atributo contido no nó $attribute
getMethod($class)	Retorna todos os métodos contidos no nó $class
getMethodName($method)	Retorna o nome do método contido no nó $method
getMethodReturnType($method)	Retorna o tipo de retorno do método contido no nó $method
getParameter($method)	Retorna todos os parâmetros do método contido no nó $method
getParameterName($parameter)	Retorna o nome do parâmetro contido no nó $parameter
getParameterType($parameter)	Retorna o tipo do parâmetro contido no nó $parameter
getVariable($method)	Retorna todas as variáveis do método contido no nó $method
getVariableName($variable)	Retorna o nome da variável contida no nó $variable
getVariableType($variable)	Retorna o tipo da variável contida no nó $variable

Tabela 2: Funções de acesso ao código atualmente disponíveis em RefaX.

uma classe Java distinta, na qual o desenvolvedor encontra métodos e atributos para descrever as operações XQuery. Essa decisão deu maior flexibilidade para a especificação e o reuso de AFs e CAFs, facilitando, assim, a instanciação do arcabouço. Em contrapartida, essa organização de CAFs e e AFs acarretou a criação de muitas classes no arcabouço. Da mesma forma, cada operação de refatoração também foi implementada por uma classe Java em separado.

Outra decisão importante foi qual tipo de dados utilizar no arcabouço para representar os documentos XML obtidos como resultado da transformação do código fonte original. Apesar de podermos utilizar elementos do tipo DOM, uma opção mais simples e

também eficiente foi representá-los através de *strings*, o que facilitou a manipulação desses documentos.

5.5. ASPECTOS DE IMPLEMENTAÇÃO

A implementação de RefaX segue as normas e orientações da Sun para codificação Java. As classes do arcabouço são agrupadas em pacotes de acordo com a camada que representam na arquitetura. A Figura 27 mostra um diagrama de classes de RefaX em nível de projeto, no qual são mostradas as principais classes do arcabouço, sem, contudo, apresentar seus atributos e métodos. O principal objetivo desse diagrama é mostrar como é o relacionamento de instanciação, representada pelas setas pontilhadas, e herança entre as classes.

Algumas observações são importantes sobre o que o arcabouço *não* implementa: tratamento de exceções geradas pelas tecnologias XML utilizadas é uma tarefa que deve ser feita em conjunto entre desenvolvedor e arcabouço, onde a missão deste é capturar qualquer exceção que por ventura ocorra, mas tratar tal exceção é deixado para o desenvolvedor, para que ele o faça da maneira que melhor considerar; outra atividade não implementada pelo

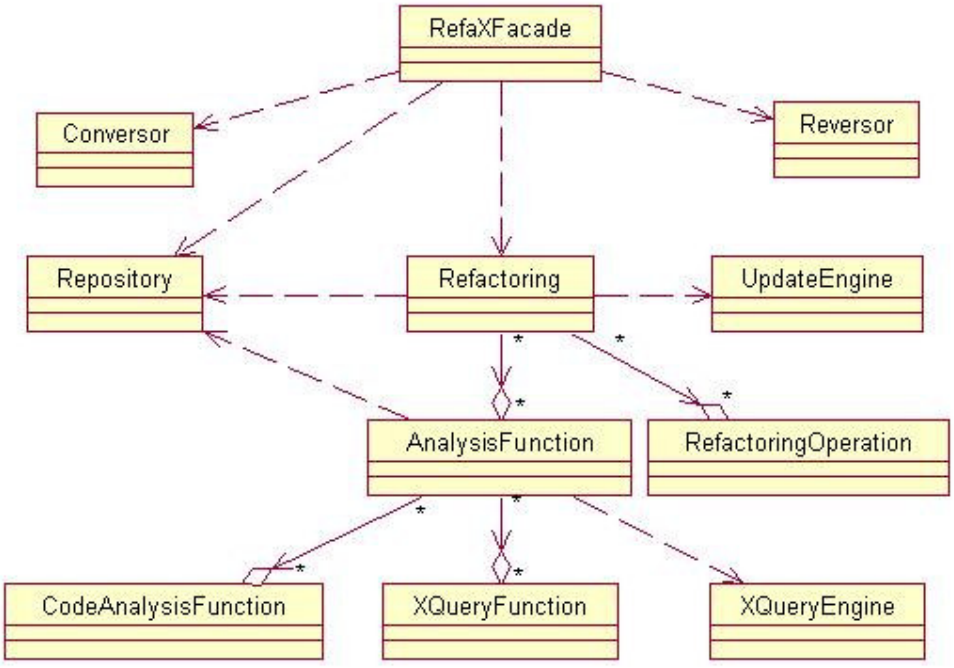


Figura 27: Diagrama de classes simplificado de RefaX.

arcabouço é um esquema de *log*, já que isso diz muito respeito à implementação da ferramenta instanciada.

Um aspecto importante de implementação é que as classes que são responsáveis pela independência de tecnologia implementam uma interface, próprio do padrão *Factory Method*, enquanto as que implementam a independência de esquema e linguagem estendem uma superclasse.

A seguir apresentamos como cada componente do arcabouço foi implementado em seus devidos pacotes e classes, descrevendo seus atributos e métodos.

5.5.1. CONVERSOR

No pacote `conversor` existe apenas a interface `ConversorInterface`, que deve ser implementada pela classe que representará o conversor. Ela disponibiliza apenas o método `convert` para implementação, que recebe como parâmetro um arquivo do tipo `File` e devolve uma *string* contendo o resultado da conversão desse arquivo para a representação XML da linguagem escolhida. Outro método que seja importante para que o conversor funcione deve ser declarado e implementado pelo desenvolvedor na classe concreta.

5.5.2. REVERSOR

Assim como o pacote `conversor`, o pacote `reversor` contém apenas a interface `ReversorInterface` que deve ser implementada pela classe que representará o reversor. Esta interface declara apenas o método `revert`, que recebe como parâmetro uma string representando um documento XML e devolve um arquivo do tipo File contendo o código fonte na sua linguagem original. Outro método que seja importante para que o reversor funcione deve ser declarado e implementado pelo desenvolvedor na classe concreta.

5.5.3. PROCESSADOR DE CONSULTA

No pacote `xqueryEngine` existe apenas a interface `XqueryEngineInterface` que deve ser implementada pela classe que representará o processador de consulta XQuery. Ela declara os seguintes métodos: `execute`, que recebe a consulta XQuery a ser executada como um parâmetro do tipo *string*, a executa e não retorna nenhum valor; `getResultAsString`, que retorna o resultado da consulta realizada como uma string. É utilizada para funções de análise como *Subclasses(className)*; `getResultAsBoolean`, que

retorna o resultado da consulta realizada como um valor verdadeiro ou falso. É utilizado por funções de análise como *IsClass(className)*, e `getResultAsNode`, que retorna o resultado da consulta como conjunto de nós XML. É utilizada por funções de análise como *MethodsThatAccessAttribute(attributeName)*. Outro método que seja importante para que o processador de consulta XQuery funcione deve ser declarado e implementado pelo desenvolvedor na classe concreta.

5.5.4. FERRAMENTA DE ATUALIZAÇÃO

O pacote `updateEngine` possui apenas a interface `UpdateEngineInterface`, que deve ser implementada pela classe que for representar a ferramenta de atualização. Ela declara os seguintes métodos: `applyUpdate`, passando como parâmetro uma *string* representando a operação de atualização na linguagem escolhida pelo desenvolvedor, onde essa operação será aplicada a todos os arquivos do repositório, e `applyUpdateToFile`, passando como parâmetro o id do arquivo que deve ser atualizado e a *string* contendo a devida operação de atualização. Outro método que seja importante para que a ferramenta de atualização funcione deve ser declarado e implementado pelo desenvolvedor na classe concreta.

5.5.5. REPOSITÓRIO

A classe que representará o repositório deve implementar a interface `RepositoryInterface`, contida no pacote `repository`. Os métodos declarados na interface são: `createNewCollection`, que recebe uma *string* representando o nome da nova coleção e a cria no repositório; `connect`, que recebe o nome da coleção que irá acessar; `insertDocument`, passando como parâmetro a string representando o documento que será inserido na coleção, juntamente com o *id* que o representará; `selectDocument`, que seleciona o documento que possua o *id* especificado no parâmetro, `deleteDocument`, que apaga o documento que possui o *id* especificado no parâmetro, `disconnet`, que fecha a conexão com a coleção, `getAllDocuments`, que retorna todos os documentos da coleção em uma lista de *strings*; `getNumberOfDocuments`, que retorna o número total de documentos da coleção; por fim, `isConnected`, que testa se a coleção está conectada. Outro método que

seja importante para que o repositório funcione deve ser declarado e implementado pelo desenvolvedor na classe concreta.

5.5.6. FUNÇÕES DE ACESSO AO CÓDIGO

As classes que representam as funções de acesso ao código estão contidas no pacote `codeAnalysisFunction` e estendem a superclasse `CodeAnalysisFunction`. Esta classe define apenas o atributo `xqueryDescription` e os métodos de configuração e devolução (*set* e *get*) que permitem que as demais classes o acessem.

As subclasses são nomeadas juntando-se a abreviação "CAF" mais o nome da função de acesso ao código que representam sem espaços. Por exemplo, o nome da classe que implementa a função de acesso ao código *getClassName* é `CAFGetClassName`. Elas possuem apenas um construtor, que recebe como parâmetro uma expressão XPath indicando o caminho para acessar o elemento do código desejado na representação XML específica. No corpo do construtor, é especificado o valor do atributo `xqueryDescription`, que conterá a declaração da função XQuery, genérica para qualquer linguagem, mais o valor do parâmetro.

Uma dessas subclasses é especial, pois ela especifica o nó inicial da representação XML do código escolhida. Seu nome é `CAFGetRootNodeName`, e ela é acessada por todas as funções de análise, para comporem as chamadas das funções.

5.5.7. OPERAÇÕES DE REFATORAÇÃO

As operações de refatoração são implementadas pelas classes contidas no pacote `refactorinOperation`. Essas classes devem estender a classe `RefactoringOperation`, que por sua vez define somente um atributo, `roDescription`, e seus métodos de atribuição e devolução.

A nomenclatura das subclasses segue o padrão utilizado nas funções de acesso ao código: o nome da classe começa pela abreviação "RO", seguido pelo nome da operação de refatoração sem espaços. Por exemplo, para implementar a operação *UpdateClassName*, criamos uma classe denominada `ROUpdateClassName`.

Diferentemente das funções de acesso ao código, as operações de refatoração são um pouco mais complexas para especificar. Cada uma deve ser especificada na linguagem de atualização ou transformação XML escolhida, e não apenas configurar os elementos típicos da representação XML, como acontece nas funções de acesso ao código. Elas possuem apenas

um construtor, que recebe como parâmetro os elementos do código e as informações necessárias para realizar as alterações. No corpo do construtor é especificada a alteração na linguagem escolhida e então configurado o atributo `roDescription`.

5.5.8. FUNÇÕES DE ANÁLISE

As classes que representam uma função de análise encontram-se no pacote `analysisFunction`. Elas estendem a classe abstrata `AnalysisFunction`, que por sua vez define os seguintes atributos: `xqueryDescription`, que armazena a descrição em XQuery da função de análise; `AFList` e `CAFList`, que representam as listas das outras funções de análise e funções de acesso ao código, respectivamente, que são utilizadas para compor a função de análise. Além desses atributos, a classe define os métodos `ListAllCAFs` e `ListAllAFs`, que retornam o conteúdo das funções de acesso ao código e funções de análise armazenadas nos atributos `AFList` e `CAFList`, respectivamente. Ela também declara os métodos abstratos `testAsString` e `testAsBoolean`, que serão definidos pelas subclasses.

As subclasses têm nomes formados a partir da união da abreviação "AF" e do nome da função de análise sem espaços. Por exemplo, o nome da classe que representa a função de acesso ao código *All Superclasses* é `AFAllSuperclasses`. No seu construtor é atribuído o valor ao atributo `xqueryDescription`, e adicionado às listas `AFList` e `CAFList` as demais funções de análise e funções de acesso ao código, respectivamente, que a função de análise implementada pela classe necessita.

Cada subclasse implementa um método chamado `FunctionCall`, que tem como objetivo definir a chamada da função de análise que está sendo implementada. Por exemplo, para a classe `AFAllSuperclasses`, o método `FunctionCall` recebe os parâmetros `doc`, representando um documento XML, `className`, o nome da classe que se deseja recuperar todas suas superclasses, e `packageName`, contendo o nome do pacote desta classe. Além desses parâmetros, ele utiliza o método `getXqueryDescription`, do atributo do tipo `CAFGetRootNode`, que todas declaram, para formar a chamada da função.

Como cada função de análise tem um tipo de retorno, que pode ser uma string ou um valor lógico, as classes devem especificar o método de teste que lhe é conveniente. Por exemplo, a função de análise *AllSuperclasses* retorna uma string, então a classe `AFAllSuperclasses` deve implementar o método `testAsString`. Já a função de análise

IsClass retorna um valor verdadeiro ou falso; logo, a classe `AFIsClass` deve implementar o método `testAsBoolean`.

No método de teste é feito o seguinte procedimento: são instanciados objetos do tipo repositório, processador de consulta e funções XQuery auxiliares, que descreveremos na seção 5.5.10. A consulta é então montada concatenando-se os métodos `ListAllXQueyFunctions`, `ListAllCAFs`, `ListAllAFs` e `FunctionCall`. Então, para cada documento do repositório a consulta é aplicada. Ao final, o valor é retornado para a classe que executou o método de teste.

5.5.9. REFATORAÇÕES

Toda classe que represente uma refatoração disponível no arcabouço está contida no pacote `refactoring`. Elas implementam a interface `RefactoringInterface`, que por sua vez define três métodos: `testPreConditions`, `apply` e `testPostConditions`.

O nome da classe é composto pela palavra "refactoring" mais o nome da refatoração sem espaços. Por exemplo, para representar a refatoração *Rename Attribuute*, criamos uma classe denominada `RefactoringRenameAttribute`. Cada classe de refatoração possui seus próprios atributos, muitas vezes totalmente distintos dos atributos de outra refatoração, pois representam os elementos que a refatoração irá alterar ou manipular. Por exemplo, a refatoração *Rename Attribute* possui três atributos: *attributeName*, representando o nome do atributo que se quer alterar, *className*, representando a classe que define o atributo, *classPackage*, que indica o nome do pacote ao qual a classe pertence, e *newAttributeName*, que indica o novo nome do atributo. Seu único construtor recebe os valores dos atributos como parâmetros e atribui esses valores a seus respectivos atributos.

Na definição dos métodos da interface `testPreConditions` e `testPostConditions` são instanciadas todas as classes que representam as funções de análise usadas para compor as pré e pós-condições da refatoração, respectivamente. Para cada função de análise é executado ou seu método `testAsString` ou `testAsBoolean`, dependendo do tipo de função de análise. Tanto `testPreConditions` como `testPostConditions` retornam verdadeiro caso todas as pré ou pós-condições tenham sido satisfeitas, ou falso, caso pelo menos uma tenha dado errada. Neste caso, a classe informa qual pré ou pós-condição não foi satisfeita.

Na definição do método da interface `apply` são instanciadas todas as classes que representam as operações de atualização utilizadas pela própria refatoração. É instanciada também a classe que representa a ferramenta de atualização, e então se executa o método `applyUpdate` ou `applyUpdateToFile` para cada operação. O método `apply` não retorna nenhum valor.

Como exemplo de implementação de uma refatoração, a Figura 28 mostra as classes relacionadas à refatoração *Add Class*, incluindo as funções de acesso ao código, funções de análise e operações de refatoração.

5.5.10. FUNÇÕES XQUERY AUXILIARES

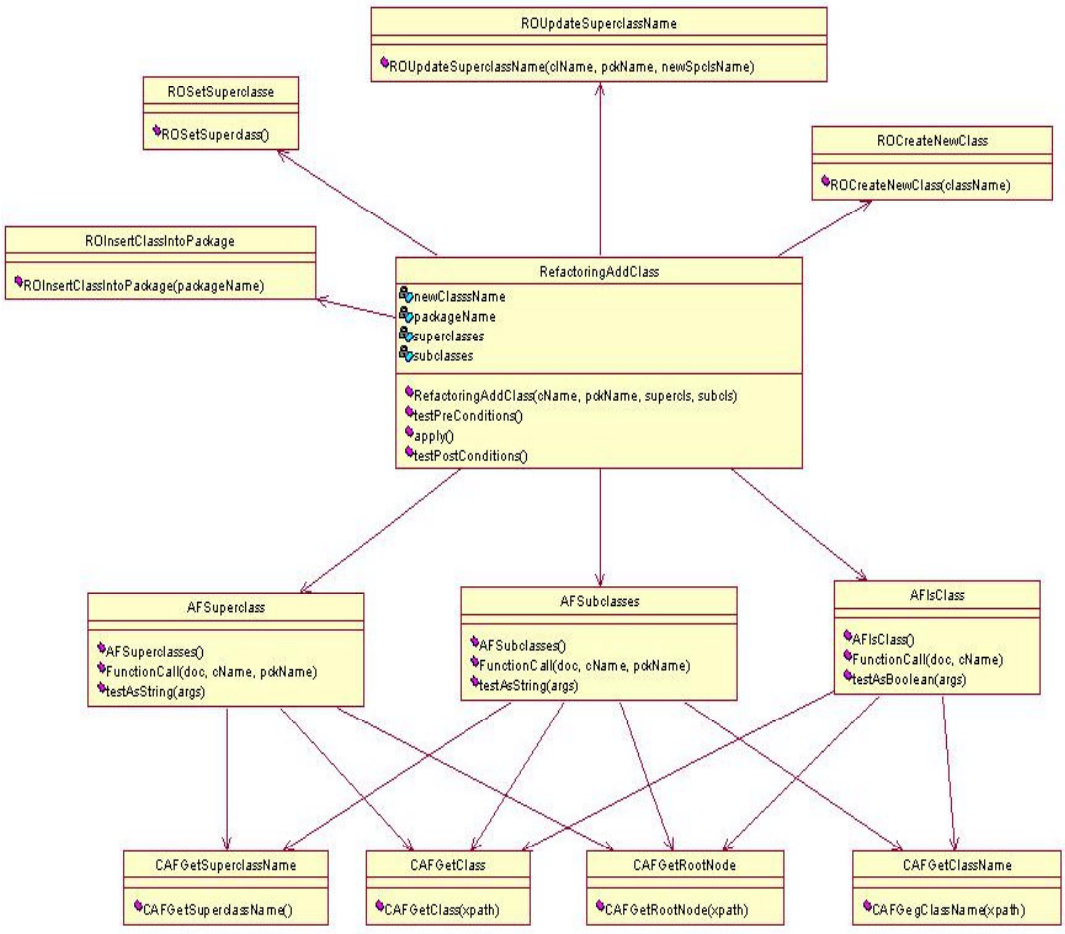


Figura 28: Diagrama de classes para a refatoração *Add Class*.

O pacote `xqueryFunction` contém todas as classes que representam uma determinada função XQuery que não seja nem função de acesso ao código, nem função de análise. Qualquer função desse tipo deve estender a classe `XQueryFunction`, que define o atributo `xqueryDescription` e seus métodos de configuração e devolução.

As subclasses têm o nome formado a partir da abreviação "XF" mais o nome da função sem espaços. Por exemplo, para representar a função XQuery *WrappElement*, implementamos a classe `WrappElement`. As principais funções XQuery utilizadas pelo arcabouço são: *WrappElement*, que recebe uma lista de nós XML e os insere como filhos de uma *tag*, para que possam ser utilizadas por outras funções de análise. Ela é utilizada, por exemplo, para "empacotar" o resultado de funções de acesso ao código como *getMethod*, que retornam uma lista de elementos; *CompareImports*, que compara se entre as declarações de importação de pacotes do arquivo está o pacote de uma classe específica, e é utilizada em funções de análise como *MethodThatAccessAttribute*; *ExistInList* recebe uma lista de nós XML e um valor string e retorna verdadeiro caso essa string esteja em um desses nós, e é aplicada, por exemplo, quando se quer saber se uma classe está entre as subclasses de outra, através da função de análise *Subclasses*.

5.5.11. OUTROS

Além das classes e pacotes descritas anteriormente, RefaX possui o pacote `util` que contém classes isoladas que não estendem nenhuma superclasse ou implementam alguma interface. Uma delas é a classe `RefaXFacade`, que fornece todas as operações que um uma ferramenta externa ou uma interface necessita para se comunicar com as classes da nova ferramenta criada a partir da instanciação de RefaX.

5.6. DIRETRIZES PARA INSTANCIAÇÃO DO ARCABOUÇO

Instanciar RefaX significa implementar os seus pontos de extensão utilizando um conjunto de padrões e tecnologias XML para uma determinada linguagem de programação. Sempre que um desenvolvedor quiser instanciar o arcabouço para construir uma nova ferramenta de refatoração de código, ele precisa seguir os passos mostrados na Figura 29, descritos a seguir.

O primeiro passo é definir para qual linguagem a futura ferramenta RefaX se destinará. Tomada essa decisão, deve-se analisar e escolher a representação XML dessa linguagem que melhor represente o código fonte. O próximo passo, então, é definir o conjunto de padrões e tecnologias XML que serão utilizadas para implementar o conversor, reversor, processador de consulta XQuery, linguagem de atualização ou transformação, ferramenta de atualização e o repositório.

Feitas todas as escolhas, começa a fase de implementação dos pontos de extensão. Como mostrado na figura 29, existem três atividades que o desenvolvedor irá realizar na

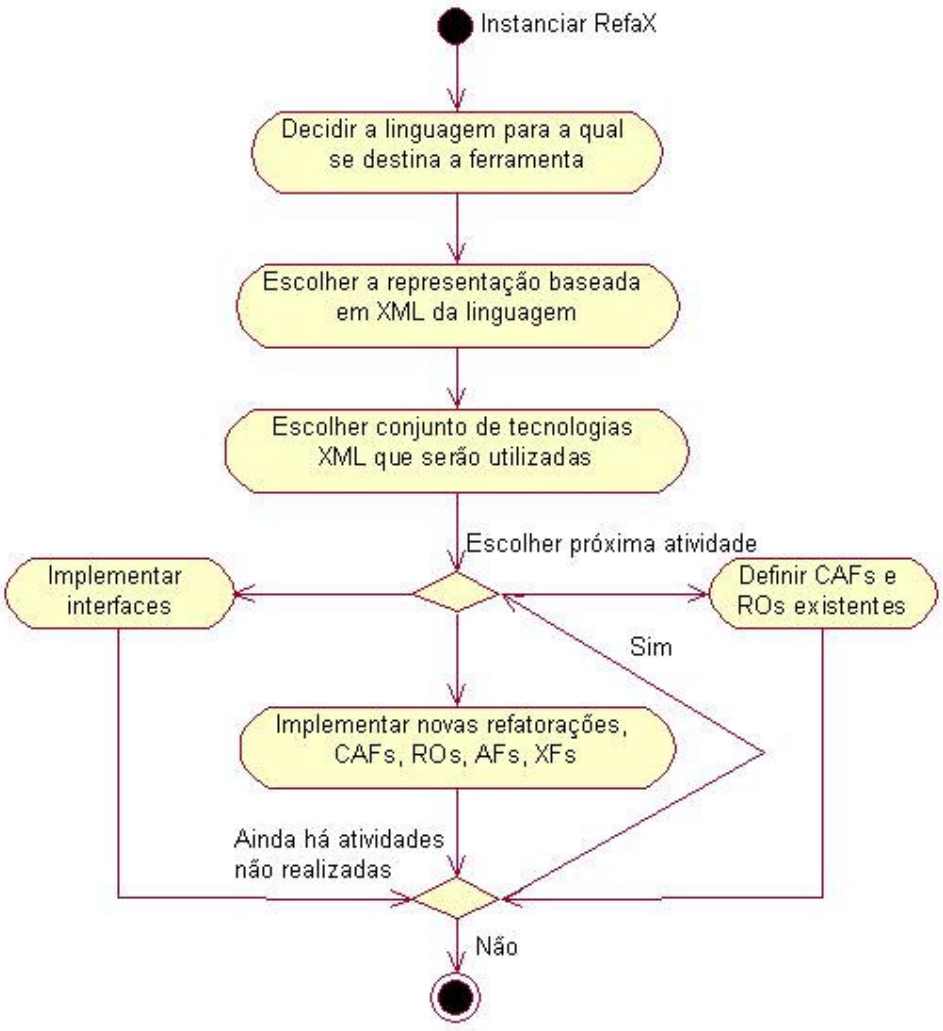


Figura 29: Diagrama de atividades para instanciação de RefaX.

ordem que ele considerar mais conveniente.

Implementar interfaces refere-se às interfaces `ConversorInterface`, `ReversorInterface`, `XqueryEngineInterface`, `UpdateEngineInterface` e `RepositoryInterface`, que devem ser implementadas com os métodos necessários para que o conversor, reversor, processador de consulta XQuery, ferramenta de atualização, e repositório, respectivamente, funcionem corretamente.

Definir CAFs e ROs existentes refere-se à instanciação das funções de acesso ao código e à implementação das operações de refatoração disponíveis no arcabouço para que as refatorações possam ser aplicadas. Para instanciar as CAFs deve-se passar como parâmetro no construtor de cada uma, a expressão de caminho relativa à representação XML da linguagem escolhida que acessa determinado elemento do código; já na implementação das ROs, deve-se especificar no atributo `roDescription` para cada uma, as operações necessárias para transformar o código em XML especificadas na linguagem de atualização escolhida.

A terceira atividade consiste em implementar novas refatorações, funções de análise, funções de acesso ao código, operações de refatoração e funções XQuery, respectivamente, que não existam no arcabouço. Para isso, o desenvolvedor precisa criar uma classe que importe o pacote correspondente ao que representa (por exemplo, se for uma função de análise, importar o pacote `refax.ufc.br.analysisFunction`), estender a classe abstrata desse pacote e então implementar seus métodos abstratos. No caso de uma nova refatoração, é aconselhável implementar esses métodos de acordo como foram implementados nas classes disponíveis em RefaX, cuja estrutura foi descrita ainda neste capítulo.

5.7. CONCLUSÃO

RefaX é um arcabouço que propicia o desenvolvimento de ferramentas de refatoração mais facilmente customizáveis, extensíveis e reutilizáveis, na medida em que baseia-se em padrões e tecnologias XML. A grande variedade de mecanismos de manipulação de dados XML, seja para consulta ou atualização, bem como a abundância de ferramentas, oferecem ao desenvolvedor uma vasta possibilidade de combinações entre elas, de modo que o desenvolvedor possa optar pelo conjunto dessas tecnologias que melhor se adeque à sua necessidade.

A principal dificuldade que enfrentamos durante o desenvolvimento do arcabouço foi decidir como representar de forma genérica cada refatoração utilizando as funções de

análise, funções XQuery auxiliares e operações de refatoração, e também descobrir quais seriam as funções de análise e funções de acesso ao código que deveriam ser implementadas. Estes itens só são descobertos à medida que tentamos implementar uma nova refatoração, e devem ser bem projetados para que sejam especificados de forma genérica, independentemente da linguagem e do esquema XML escolhidos.

Ferramentas de refatoração construídas a partir de RefaX têm sua facilidade de instanciação intimamente relacionada às tecnologias e padrões XML escolhidos pelo desenvolvedor. Quanto mais complexas as representações da linguagem em XML, maior a dificuldade de especificar as funções de acesso ao código e, principalmente, as operações de refatoração. Essas diferenças serão abordadas com mais detalhes no próximo capítulo, no qual descreveremos as ferramentas de refatoração para duas linguagens orientadas a objeto construídas a partir de RefaX.

CAPÍTULO 6

ESTUDOS DE CASO

Para demonstrar a viabilidade do arcabouço, nós conduzimos dois estudos de caso no qual usamos RefaX para implementar um protótipo de refatoração para código escrito em Java, chamado *RefaX4Java*, e outro para código escrito em C++, denominado *RefaX4C++*. Esses dois protótipos foram úteis para investigar o nível de reusabilidade oferecido pelo arcabouço e para mostrar que suas ferramentas satisfazem os requisitos discutidos na seção 5.1. As seções 6.1 e 6.2 descrevem o conjunto de rerepsentações de código fonte e tecnologias utilizadas para os protótipos RefaX4Java e RefaX4C++, respectivamente, juntamente com uma discussão sobre algumas questões envolvidas na implementação de cada protótipo. A seção 6.3 descreve um *plugin* para a plataforma de desenvolvimento Eclipse, também desenvolvido por nós, que fornece uma interface mais amigável para a utilização de RefaX4Java.

6.1. O PROTÓTIPO REFAX4JAVA

Para ilustrar como RefaX atende aos requisitos independência de esquema XML e independência de tecnologia, decidimos implementar duas versões de RefaX4Java: uma aplicável à representação JavaML proposta por Badros [4], à qual chamaremos JavaML de Badros, e outra aplicável à representação JavaML proposta por Mamas e Kontogiannis [36], à qual chamaremos JavaML de M&K.

Escolhemos essas duas representações porque ambas disponibilizam gratuitamente conversores e reversores. Além disso, suas representações XML permitem especificar mais facilmente funções de acesso ao código e operações de refatoração.

Para construir essas ferramentas, o primeiro passo é implementar os pontos de extensão do arcabouço para um determinado conjunto de tecnologias e padrões XML. A seguir mostramos que tecnologias escolhemos e o motivo:

- **Funções de acesso ao código**: foram especificadas de acordo com a representação JavaML a que se destinavam;

- **Conversor**: para JavaML de Badros usamos Jikes [28], uma adaptação do compilador Java da IBM fornecido pelo próprio Badros, enquanto para JavaML de M&K utilizamos RET4J [51], um conjunto de ferramentas para análise e transformação de programas Java;
- **Processador de Consulta**: o processador de consulta XQuery IPSI-XQ foi usado para ambas as versões, pois suporta a última versão da especificação de XQuery;
- **Linguagem de especificação de refatoração**: XUpdate [69] foi usado para ambas as versões, pois ela é uma linguagem realmente de atualização XML, não de transformação, o que torna mais fácil a especificação de consulta de atualização;
- **Repositório de dados XML**: XIndice [68], um banco de dados XML nativo disponível gratuitamente, foi utilizado em ambas as versões, pois além de sua comprovada robustez, também implementa a última versão de XUpdate;
- **Ferramenta de atualização**: utilizamos a ferramenta de atualização embutida em XIndice para ambas as versões;
- **Reversor**: para JavaML de Badros usamos transformações XSLT [65], também fornecidas pelo próprio Badros, enquanto para JavaML de M&K usamos novamente RET4J.

As funções de acesso ao código não exigiram muito esforço do desenvolvedor, pois basta que ele conheça o esquema XML que está utilizando e informe pequenas expressões de caminho. Quanto mais verbosa for uma representação XML, mais cuidado deve-se ter com a especificação das funções de acesso ao código, pois muitas vezes pode indicar um nó XML errado, o que provocaria um erro durante a aplicação da refatoração. Isso aconteceu durante a implementação da versão para JavaML de M&K. especialmente para a especificação de funções que retornam nomes, como *getClassName* e *getMethodName*, pois esses dados encontram-se geralmente nos marcadores abaixo da que representa a entidade. JavaML de Badros é uma representação menos verbosa, tanto em relação à quantidade de marcadores para representar uma entidade, quanto ao número de atributos que essas marcadores possuem. Essa característica tornou a implementação das funções de acesso ao código de RefaX para a versão JavaML de Badros um pouco mais fácil. A Tabela 3 mostra como as funções de acesso ao código foram implementadas para cada versão JavaML.

Uma tarefa um pouco mais complicada foi especificar as operações de refatoração, pois para cada uma é necessário montar uma nova consulta de atualização na linguagem escolhida pelo desenvolvedor. Mais uma vez, JavaML de Badros levou vantagem quanto à facilidade de especificar essas consultas em relação a JavaML de M&K, pelo mesmo motivo: ser uma representação menos verbosa. As Figuras 30 e 31 mostram a especificação da operação de refatoração *Add Attribute* para as versões JavaML de Badros e M&K, respectivamente. Através dela podemos ter uma idéia de como uma representação verbosa como JavaML de M&K pode aumentar o trabalho do desenvolvedor para especificar uma operação de refatoração.

Funções de acesso ao código	JavaML de Badros	JavaML de M&K
getRootNode($doc)	java-source-program	CompilationUnit
getClass($doc)	$doc//class	$doc//ClassDeclaration
getClassName($class)	$class/@name	$class/UnmodifiedClassDeclaration /@Identifier
getPackageName($doc)	$doc//package-decl	$doc//PackageDeclaration
getImport($doc)	$doc//import	$doc//ImportDeclaration
getImportName($import)	$import/@name	$import/Name/@Identifier
getSuperclass($class)	$class/superclass	$class/UnmodifiedClassDeclaration [@Extends=”true”]
getSuperclassName($superclass)	$superclass/@name	$superclass/Name/@Identifier
getConstructor($class)	$class//constructor	$class//ConstructorDeclaration
getAttribute($class)	$class//field	$class//FieldDeclaration
getAttributeName($attribute)	$attribute/@name	$attribute/VariableDeclaratorId/ @Identifier
getAttributeType($attribute)	$attribute/type/@name	$attribute/Type/Name/@Identifier
getMethod($class)	$class//method	$class//MethodDeclaration
getMethodName($method)	$method/@name	$method/MethodDeclarator /@Identifier
getMethodReturnType($method)	$method/type/@name	$method/ResultType/@Identifier
getParameter($method)	$method/formal-arguments	$method/Formalparameter
getParameterName($parameter)	$parameter/@name	$parameter/VariableDeclaratorId/ @Identifier
getParameterType($parameter)	$parameter/type/@name	$parameter/Type/Name/@Identifier
getVariable($method)	$method//local-variable	$method/LocalVariableDeclaration
getVariableName($variable)	$variable/@name	$parameter/VariableDeclaratorId/ @Identifier
getVariableType($variable)	$variable/type/@name	$variable/Type/Name/@Identifier

Tabela 3: Funções de acesso ao código para as representações JavaML de Badros e de M&K.

```
<xupdate:modifications version="1.0"
    xmlns:xupdate="http://www.xmldb.org/xupdate">
  <xupdate:insert-after select="/java-source-program/java-class-
      file/class[@name='className']/superclass">
     <xupdate:element name="field">
     <xupdate:attribute name="name">attributeName</xupdate:attribute>
     <xupdate:attribute name="visibility">public</xupdate:attribute>
     <xupdate:element name="type">
        <xupdate:attribute name="name">Object</xupdate:attribute>
     </xupdate:element>
     </xupdate:element>
  </xupdate:insert-after>
</xupdate:modifications>
```

Figura 30: Especificação da operação de refatoração *Add Attribute* em JavaML de Badros.

```
<xupdate:modifications version="1.0"
   xmlns:xupdate="http://www.xmldb.org/xupdate">
  <xupdate:insert-after select=\"/CompilationUnit/TypeDeclaration/
     ClassDeclaration/UnmodifiedClassDeclaration[@Identifier='
     className']/ClassBody\">
     <xupdate:element name="FieldDeclaration">
       <xupdate:attribute name="isFinal">false</xupdate:attribute>
       <xupdate:attribute name="isPrivate">false</xupdate:attribute>
       <xupdate:attribute name="isProtected">false
        </xupdate:attribute>
       <xupdate:attribute name="isPublic">true</xupdate:attribute>
       <xupdate:attribute name="isStatic">false</xupdate:attribute>
       <xupdate:attribute name="isTransient">false
        </xupdate:attribute>
       <xupdate:attribute name="isVolatile">false</xupdate:attribute>
       <xupdate:element name="Type">
        <xupdate:attribute name="ArraySize">0</xupdate:attribute>
        <xupdate:element name="Name">
           <xupdate:attribute name="Identifier">Object
           </xupdate:attribute>
        </xupdate:element>
        <xupdate:element name="VariableDeclarator">
         <xupdate:element name="VariableDeclaration">
           <xupdate:attribute name="ArraySize">0</xupdate:attribute>
           <xupdate:attribute name="Identifier">attributeName
           </xupdate:attribute>
         </xupdate:element>
        </xupdate:element>
       </xupdate:element>
     </xupdate:element>
  </xupdate:insert-after>
</xupdate:modifications>
```

Figura 31: Especificação da operação de refatoração *Add Attribute* em JavaML de M&K.

Contudo, nossa principal dificuldade na implementação de RefaX4Java foi criar as classes que representam as tecnologias de conversão, reversão, repositório, processador de consulta e ferramenta de atualização, na medida que utilizamos produtos de fabricantes variados e que, em alguns casos, não documentam de forma adequada suas respectivas APIs. Com isso, muito tempo foi gasto tentando entender quais as funções necessárias para que cada um pudesse funcionar e testando as classes depois de prontas para certificarmos que estavam corretas. Em algumas vezes, foi preciso entrar em contato com o fabricante de determinadas ferramentas para que ele tentasse resolver certos problemas, o que fez com que descobríssemos até defeitos e limitações com algumas tecnologias, até então nem conhecidas pelos próprios fabricantes.

Cada versão de RefaX4Java gerou um arquivo *jar* contendo as classes já compiladas e que podem então ser acoplados a uma interface mais amigável ao usuário para que este possa utilizar as refatorações implementadas. Nossa primeira idéia foi construir uma ferramenta na qual o usuário poderia criar ou importar um projeto, editar os arquivos e, claro, aplicar refatorações. Entretanto, notamos que essa não seria uma boa idéia, na medida em que estaríamos desenvolvendo um novo ambiente de programação para Java, o que não era nosso interesse. Surgiu então a idéia de desenvolver um *plugin* para uma IDE existente, o que facilitaria a disseminação e utilização dos protótipos. Com isso, decidimos desenvolver um *plugin* para o ambiente de desenvolvimento Eclipse [16], por ser um dos mais utilizados no momento e ser de fácil extensão. Na seção 6.3 mostraremos detalhes de implementação do *plugin*.

6.2. O PROTÓTIPO REFAX4C++

O protótipo RefaX4C++ foi desenvolvido com a intenção de mostrar que o arcabouço RefaX satisfaz o requisito independência de linguagem. Contudo, ao contrário do protótipo para Java, este protótipo foi mais difícil de implementar devido a várias razões. Primeiro, nós encontramos apenas duas representações baseadas em XML para código C++, ambas denominadas CppML. Uma foi proposta por Mamas e Kontogiannis [36], enquanto a outra foi fornecida como parte da ferramenta de engenharia reversa Columbus CAN [15]. Dessas duas representações, apenas a última possui um conversor disponível, mas mesmo assim sem um reversor. Como não está no escopo do nosso trabalho desenvolver tais ferramentas, não implementamos nenhum suporte para a reversão dessa representação.

Outra dificuldade deve-se ao fato de que a gramática de C++ é consideravelmente mais complexa do que a gramática de Java, o que requer a especificação de um maior número de funções de análise, funções de acesso ao código e operações de refatoração para cobrir as particularidades da linguagem. Em especial, a CppML de Columbus não representa cada arquivo *.cpp* e *.h* em XML. Ele representa o resultado do pré-processamento desses dois arquivos em XML, o que faz com que muitas informações sejam perdidas. Com isso, não é possível aplicar uma refatoração por completo, já que ela teria que se refletir nesses dois arquivos e isso não é possível com a representação de Columbus. Assim, o único ponto de extensão que pôde ser implementado por completo foram as funções de acesso ao código. A tabela 4 mostra como elas foram implementadas para a representação de Columbus e de Mamas e Kontogiannis.

Funções de acesso ao código	CppML de Columbus	CppML de M&K
getRootNode($doc)	Project	Program
getClass($doc)	$doc//Class	$doc//Class
getClassName($class)	$class/@name	$class/@name
getPackageName($doc)	-	$doc//IncludedSource
getImport($doc)	-	$doc//Include
getImportName($import)	-	$import/@name
getSuperclass($class)	-	$class/Extend
getSuperclassName($superclass)	$superclass/@name	$superclass/@name
getConstructor($class)	$class//Function [@kind="fnkConstructor"]	$class//Function [@kind = "Constructor"]
getAttribute($class)	$class//Object [@kind="fnkNormal"]	$class//Variable [@kind="RegularMember"]
getAttributeName($attribute)	$attribute/@name	$attribute/@name
getAttributeType($attribute)	$attribute/hasTypeRep/ @ref	$attribute/TypeDescriptor/ @name
getMethod($class)	$class//Function [@kind="fnkNormal"]	$class//Function [@kind = ["Regular Member"]
getMethodName($method)	$method/@name	$method/@name
getMethodReturnType($method)	$method/hasTypeRep/ @ref	$method/TypeDescriptor/ @name
getParameter($method)	$method//Parameter	$method//FunctionParameter
getParameterName($parameter)	$parameter/@name	$parameter/@identifier
getParameterType($parameter)	$parameter/hasTypeRep/ @ref	$parameter/TypeDescriptor/ @name
getVariable($method)	$method//Object [@kind="fnkLocal"]	$method//Variable [@kind='Local Member']
getVariableName($variable)	$variable/@name	$variable/@name
getVariableType($variable)	$variable/hasTypeRep/ @ref	$variable/TypeDescriptor/ @name

Tabela 4: Funções de acesso ao código para as representações CppML de Columbus e de M&K.

Se tivéssemos pelo menos um conversor e um reversor disponível para a mesma representação, poderíamos também ter desenvolvido um *plugin* que interagisse com RefaX4C++. Por esse motivo, e pelas limitações apresentadas no parágrafo anterior, RefaX4C++ tornou-se apenas um protótipo semi-pronto à espera de novas tecnologias de conversão, reversão e, até mesmo, uma nova representação XML da linguagem.

6.3. UM PLUGIN DO ECLIPSE PARA REFAX4JAVA

O plugin, denomiando RefaX4JavaPlugin, foi desenvolvido como uma interface gráfica mais amigável para a ferramenta RefaX4Java. Em comparação à arquitetura do arcabouço, RefaX4JavaPlugin representa uma camada acima da superior (Ferramentas RefaX), mas que não está contida na Figura 25, já que não faz parte do arcabouço. Essa camada poderia se chamar *Ambientes de Refatoração RefaX*, e representaria qualquer interface ou outra ferramenta que utilize uma ferramenta RefaX, como é o caso do RefaX4JavaPlugin.

Implementamos no *plugin* apenas as refatorações de renomeação (*Rename Class* e *Rename Attribute*), pois são as que fazem mais sentido de serem utilizadas na sua forma primitiva pelos usuários; as outras refatorações de RefaX4Java fazem mais sentido serem utilizadas para a composição de refatorações compostas. Um diagrama de classes simplificado do *plugin* é mostrado na Figura 32.

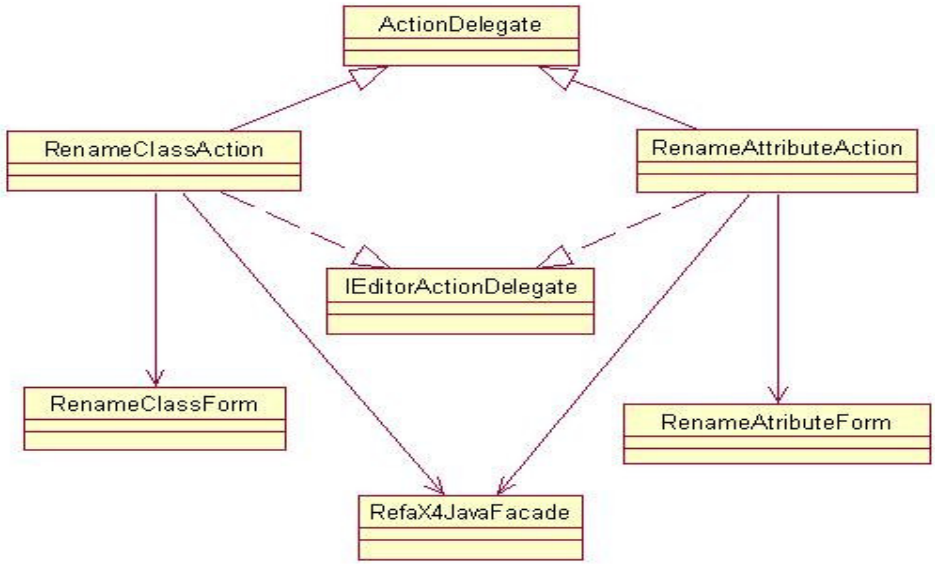


Figura 32: Diagrama de classes simplificado de RefaX4JavaPlugin.

Cada refatoração possui duas classes principais: a classe `Action`, que estende a classe do Eclipse `ActionDelegate` e implementa a interface `IEditorActionDelegate`, e a classe `Form`, que é responsável pela interação com o usuário. A classe `Action` tem como função manipular os elementos do código e os eventos ocorridos, como por exemplo, capturar as palavras corretas em uma seleção. Além disso, essa classe utiliza a classe `Facade` de Refa4Java para aplicar as refatorações. Já a classe `Form` apresenta uma tela para entrada de dados, na qual mostra o nome da classe ou do atributo selecionado e um campo em branco para que o usuário preencha com o novo nome. As Figuras 33 e 34 mostram, respectivamente, a classe Java `Funcionario`, apresentada no Capítulo 3, na qual está sendo aplicada a refatoração *Rename Class*, e o formulário de entrada de dados para essa refatoração. Por fim,

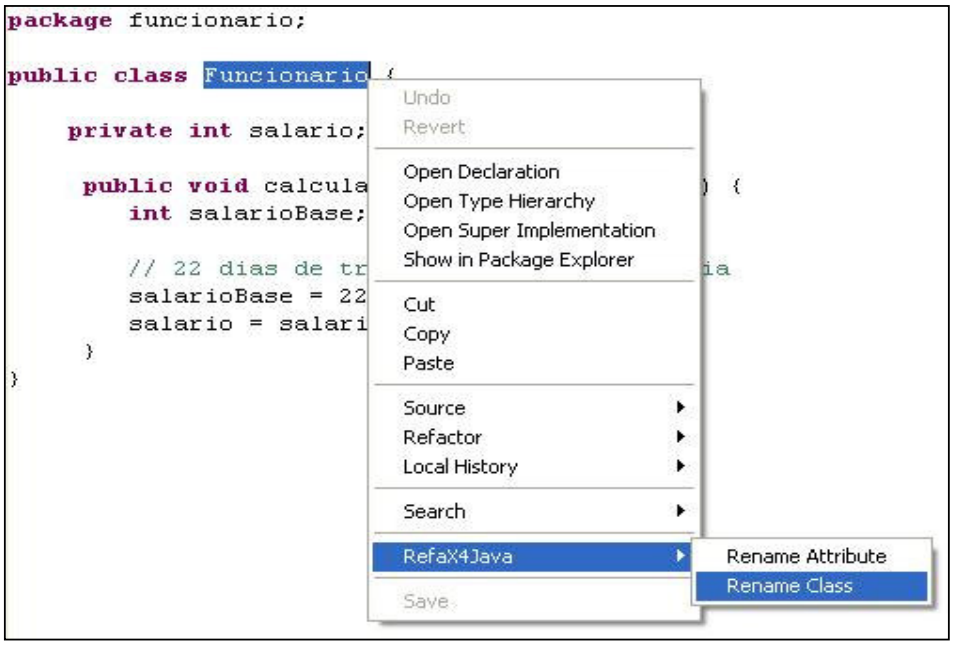


Figura 33: Exemplo de código java no qual está sendo aplicada a refatoração *Rename Class* a partir do *plugin* RefaX4JavaPlugin.

Figura 34: Tela de entrada de dados da refatoração *Rename Class*.

a Figura 35 apresenta um diagrama de atividades mostrando a interação usuário, RefaX4JavaPlugin e RefaX4Java.

Para que o *plugin* funcione na plataforma Eclipse, é necessário configurar um arquivo XML chamado *plugin.xml*, que descreve toda a integração entre as classes do *plugin* e mostra quais bibliotecas externas serão utilizadas. Outro ponto importante do arquivo XML que vale ser destacado é que é nele onde descrevemos quais pontos de extensão da plataforma estamos estendendo e que contribuições estamos utilizando. No caso de RefaX4JavaPlugin, nós estendemos o ponto de extensão `org.eclipse.ui.popupMenus`, que é o responsável por modificar o *popup* original da plataforma, e utilizamos a contribuição de visão (`viewerContribution`) para capturar os elementos selecionados no editor principal.

A maior dificuldade que tivemos para implementar o *plugin* foi, dada nossa inexperiência em estender a plataforma Eclipse, entender as classes que interagem diretamente com o código do arquivo visualizado no editor principal, já que foi encontrada pouca literatura para esse tipo de operação.

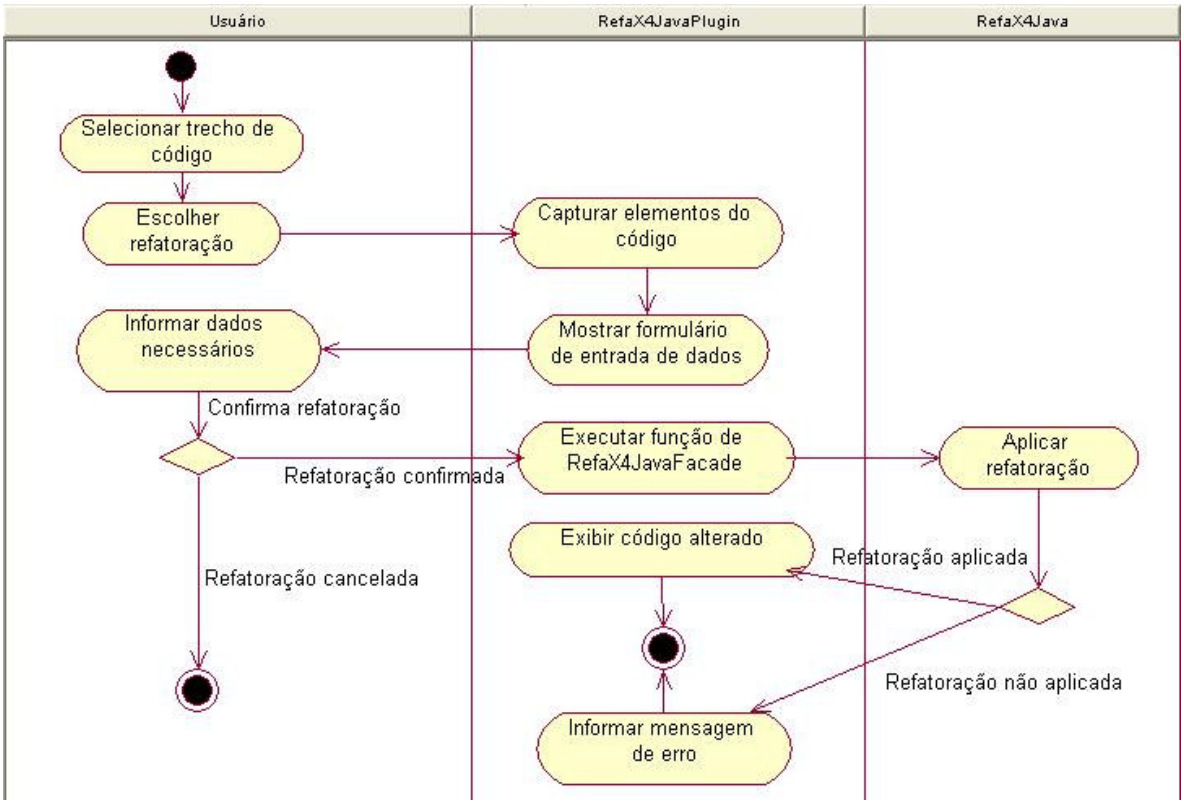


Figura 35: Diagrama de atividades mostrando interação entre usuário, RefaX4JavaPlugin e RefaX4Java.

6.4. CONCLUSÃO

Desenvolver ferramentas de refatoração a partir de RefaX é um fato possível, como foi mostrado nesse capítulo através de dois protótipos para as linguagens Java e C++. Além disso, integrar essas ferramentas a ambientes de desenvolvimento, como fizemos com RefaX4Java, ajuda a tornar o arcabouço mais conhecido e fácil de ser utilizado pelos desenvolvedores e usuários finais.

O próximo capítulo conclui a dissertação, descrevendo os principais resultados obtidos neste trabalho, e indicando sugestões de trabalhos futuros que venham a melhorar não só RefaX, mas também as ferramentas construídas a partir dele.

CAPÍTULO 7
CONCLUSÃO

Este trabalho apresentou RefaX, um arcabouço para refatoração de código baseado em um processo centrado em XML, que objetiva facilitar o desenvolvimento de ferramentas de refatoração mais flexíveis. RefaX fornece aos desenvolvedores de ferramentas de refatoração vários serviços, entre eles, conversão de artefatos de código para representações em XML; armazenamento dos artefatos convertidos em um repositório XML; verificação e execução das operações de refatorações sobre os dados do repositório; conversão da representação atualizada de volta para seu formato textual original.

Na seção 7.1, sumarizamos os principais resultados obtidos nesse trabalho e em seguida, na seção 7.2, apresentamos algumas sugestões para trabalhos futuros.

7.1. RESULTADOS

Desenvolver ferramentas de refatoração através de RefaX não é apenas viável, como mostramos com os dois protótipos para Java e C++, mas também aumenta nossa confiança de que usar XML como uma forma padronizada para desacoplar o processo de refatoração das tecnologias utilizadas pode oferecer uma efetiva contribuição para o desenvolvimento de ferramentas de manutenção de software mais flexíveis e customizáveis. Em particular, o uso de padrões de processamento e representação de código abertos e baseados em XML, de uma forma independente de tecnologia, oferece ao desenvolvedor um alto grau de flexibilidade para escolher o mais apropriado conjunto de ferramentas para cada etapa do processo de refatoração.

Os requisitos independência de linguagem de programação e de esquema, fornecidos pelo arcabouço, trazem uma contribuição adicional para promover o reuso de refatorações através de diferentes representações de código fonte, diferentes linguagens de programação e, até mesmo, diferentes ambientes de desenvolvimento e manutenção de software. Nesse sentido, a integração de ferramentas desenvolvidas usando RefaX, como RefaX4Java, a ambientes de desenvolvimento bem conhecidos, como Eclipse, pode ajudar a

impulsionar a utilização do arcabouço por parte de desenvolvedores e pesquisadores de ferramentas de refatoração.

Outra contribuição advém da disponibilização das ferramentas desenvolvidas com RefaX na forma de arquivos *jar*, que podem ser utilizadas por outras ferramentas, interfaces, ou mesmo IDEs. Esta característica indica que uma nova forma de independência pode ser obtida: a independência de ambiente.

O conjunto de funções de acesso ao código e funções de análise, implementadas no contexto de RefaX, também pode ser utilizado no desenvolvimento de outros tipos de ferramentas de manutenção, como ferramentas de análise de código, métricas e engenharia reversa. Um trabalho nessa direção já foi mostrado em [18].

Por fim, espera-se que os resultados deste trabalho também contribuam para promover a melhoria das tecnologias consideradas, especialmente com relação à criação de novas representações de código em XML, ou à revisão das representações existentes, com o objetivo de tornarem-nas mais amenas à especificação de consultas e operações de atualização.

7.2. TRABALHOS FUTUROS

Atualmente estamos trabalhando para aumentar o número de refatorações disponíveis no arcabouço. Nosso próximo passo será aplicar ferramentas baseadas em RefaX, como RefaX4Java e RefaX4C++, para sistemas de tamanhos e domínios de aplicação variados, bem como investigar sua perfomance e escalabilidade, especialmente comparando-as a outras ferramentas de refatoração para a mesma linguagem de programação a que se destinam.

Outra linha natural de pesquisa é criar novas versões de RefaX4Java para outros conjuntos de tecnologias XML e fazer uma comparação entre elas, para que possamos apontar um conjunto que maximize a performance da ferramenta. Pretendemos, também, construir mais protótipos de refatoração para outras linguagens de programação que possuam uma representação de código baseado em XML bem definida.

Também é de nosso interesse aplicar RefaX como mecanismo de transformação de linguagens, como por exemplo, de Java para AspectJ. Outro trabalho interessante seria aplicar RefaX para refatorações em nível de projeto, especialmente para diagramas UML.

Integrar ferramentas desenvolvidas com RefaX a outros ambientes de desenvolvimento, como NetBeans e JBuilder, bem como melhorar a sua reusabilidade, é um ponto que também merece atenção especial, pois facilitará a disseminação do arcabouço entre os desenvolvedores e pesquisadores da área de refatoração.

REFERÊNCIAS BIBLIOGRÁFICAS

[1] Aguiar, A., David , G., Badros, G., "JavaML 2.0: Enriching the Markup Language for Java Source Code", in Proc. of. XML: Aplicações e Tecnologias Associadas (XATA 2004), 12-13 de fevereiro, Porto, Portugal, 2004.

[2] Apache Project Home Page. Available at: www.apache.org. Last accessed on 01/04/2004.

[3] Avoisto. Project Analizer Home page. Available at: http://www.avoisto.com/project/project.html. Last accessed on 01/05/2004.

[4] Badros, G. J., "JavaML: A Markup Language for Java Source Code", in Proc. of the 9th International World Wide Web Conference (WWW9), Amsterdam, Netherlands, May 2000.

[5] BeautyJ Home Page. Available at http://beautyj.berlios.de/beautyJ.html. Last accessed on 04/03/2004.

[6] Beck, K. "Extreme Programming Explained: Embrace Change". Addison-Wesley, 1999.

[7] Bicycle Repair Man Home page. Available at: http://bicyclerepair.sourceforge.net/. Last accessed on 01/05/2004.

[8] Boger, M., Sturm, T., Fragemann, P., "Refactoring Browser for UML", in Proc. of International Conference on Extreme Programming and Flexible Process in Software Engineering, pp. 77-81, 2002.

[9] C# Refactoring Home Page. Available at: http://www.xtreme-simplicity.net/CSharpRefactory.html. Last accessed on 01/05/2004.

[10] C# Refactoring Tool home page. Available at: http://dotnetrefactoring.com/. Last accessed on 01/05/2004.

[11] Carneiro, G. F., Neto, M. G. M., "Relacionando *Refactorings* a Métricas de Código Fonte – Um Primeiro Passo para Detecção Automática de Oportunidades de Refactoring", in Proc. of 17º. Simpósio Brasileiro de Engenharia de Software (SBES 2003), Manaus, Brasil, 2003.

[12] Cinnéide, M. O., Automated Apllications of Design Patterns: A Refactoring. Tese (Doutorado em Ciências da Computação), Universidade de Dublin, 2000.

[13] Collard, M. L, Kagdi, H. H., Maletic, J. I. “An XML-Based Lightweight C++ Fact Extractor”. In Proc. of the 11th IEEE International Workshop on Program Comprehension (IWPC ‘03), Portland, USA, May 2003.

[14] Collard, M. L. “An Infrastructure to Support Meta-Differencing and Refactoring of Source Code”. In Proc. of the 18th IEEE International Conference on Automated Software Engineering, Montreal, Quebec, Canada, October 2003.

[15] Columbus Home Page. Available at http://www.frontendart.com/. Last accessed on 28/04/2004.

[16] Eclipse Home Page. Available at http://www.eclipse.org. Last accessed on 19/07/2004.

[17] Exist Home Page. Available at http://exist.sourceforge.net. Last accessed on 04/03/2004.

[18] Fonseca, L. A., Mendonça, N. C., Maia, P. H. M., “Towards Reusable Code Analysis Tools Using Standard XML Technologies”, In Anais do I Workshop de Ciências da Computação e Sistemas da Informação da Região Sul (WORKCOM-SUL), Palhoça, SC, Maio, 2004.

[19] Fowler, M. “Refactoring: Improving the Design of Existing Programs”. Addison-Wesley. 1999.

[20] Gamma, E., Helm, R., Johnson, R., Vlissides, J. “Design Patterns: Elements of Reusable Object-Oriented Software”. Addison-Wesley, 1994.

[21] Gorp, P. V., Stenten, H., Mens, T., Demeyer, S., “Toward Automating Source Consistent UML Refactorings”, in Proc. of Unified Modeling Language Conference, 2003.

[22] 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.

[23] IBM AlphaWorks. XML TreeDiff. Available at http://www.alphaworks.ibm.com/aw.nsf/techmain/xmltreediff. Last accessed on 01/05/2003.

[24] IntelliJ. IDEA Home page. Available at http://www.intellij.com/idea/. Last accessed on 01/05/2004.

[25] IPSI-XQ Home Page. Available at http://www.ipsi.fraunhofer.de/oasys/projects/ipsixq/index_e.html. Last accessed on 04/03/2004.

[26] JavaRefactor Home page. Available at http://plugins.jedit.org/plugins/?JavaRefactor. Last accessed on 01/05/2004.

[27] JFactor Home page. Available at http://www.instantiations.com/jfactor/. Last accessed on 01/05/2004.

[28] Jikes Home page. Available at http://www-124.ibm.com/developerworks/oss/jikes/. Last accessed on 04/03/2004.

[29] JRefactory Home page. Available at http://jrefactory.sourceforge.net/. Last accessed on 01/05/2004.

[30] Kataoka, Y., Ernst, M. D, Griswold, W, G., Notkin, D., "Automated Support for Program Refactoring Using Invariants", in Proc. of 17th. International Conference on Software Maintenance, pp. 736 -743, 2001.

[31] Korman, W.F. "Elbereth: Tool Support for Refactoring Java Programs". 1998. 55p. Dissertação (Mestrado em Ciências da Computação). Universidade da Califórnia, San Diego, Estados Unidos.

[32] Lakhotia, A., Deprez, J. C., "Restructuring Programs by Tucking Statements into Functions", Information and Software Tecnology, special issue on Program Slicing, vol. 40, pp. 670-689, 1998.

[33] Lehti, P. "Design and Implementation of a Data Manipulation Processor for an XML Query Language". Ph.D. Thesis, Technical University of Darmstadt, Germany, 2001.

[34] Leite, J.C.S.P., "Draco-Puc: A Technology Assembly for Domain Oriented Software Development", in: Proceedings of the Third International Conference on Software Reuse (ICSR), pp. 94-100, Rio de Janeiro, Brazil, 1994.

[35] 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.

[36] Mammas, 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.

[37] Mayrhauser, A. and Marie Vans, A., "Program Comprehension During Software Maintenance and Evolution". IEEE Computer, Vol. 28, No. 8, pp. 44–55, August 1995.

[38] Mendonça, N. C., Maia, P. H. M., Fonseca, L. A., Andrade, R. M. C., "Building Flexible Refactoring Tools with XML", in Proc. of 18o. Simpósio Brasileiro de Engenharia de Software (SBES 2004), 20 - 22 de outubro, Brasília, DF, Brasil, 2004.

[39] Mendonça, N. C., Maia, P. H. M., Fonseca, L. A., Andrade, R. M. C., "RefaX: A Refactoring Framework Based on XML", in Proc. of 20th International Conference on Software Maintenance (ICSM 2004), 11 – 14 September, Chicago, EUA, 2004.

[40] Mens, T. and Tourwé, T. "A Survey of Software Refactoring". IEEE Transactions on Software Engineering, Vol. 30, No. 2, February 2004.

[41] Mens, T., Demeyer, S., Janssens, D., "Formalising Behavior Preserving Program Transformations", Graph Transformation, 2002.

[42] Opdyke, W. F. "Refactoring: A Program Restructuring Aid in Designing Object-Oriented Aplications Framework". Ph.D. Thesis, Univ. of Illinois at Urbana-Champaign, 1992.

[43] OptimalJ Home page. Available at http://javacentral.compuware.com/pasta/. Last accessed on 01/05/2003.

[44] Philipps, J., Rumpe, B., "Refinement of Information Flow Architectures", in Proc. of International Conference on Formal Languages and Methods, 1997.

[45] Pipka, J. U., "Refactoring in a 'Test First' - World", in Proc. of 3th International Conference on Extreme Programming and Flexible Processes in Software Engineering, pp. 71-76, 2002.

[46] Pressman, Roger S. "Software Engineering – A Practitioner's Approach". McGraw-Hill, 5th Edition, 2000.

[47] Proietti, M., Pettorossi, A., "Semantics Preserving Transformation Ruler for Prolog", in Proc. of Symposium Partial Evaluation and Semantics-Based Program Evaluation, vol. 26, no. 9, pp. 274-284, May 1991.

[48] R. Fanta and V. Rajlich, "Reengineering Object-Oriented Code", in Proc. of International Conference on Software Maintenance, pp. 238-246, 1998.

[49] R. Fanta and V. Rajlich, "Restructuring Legacy C Code into C++, in Proc. of International Conference on Software Maintenance, pp. 77-85, 1999.

[50] RefactorIt Home page. Available at http://www.refactorit.com. Last accessed em 01/05/2004.

[51] Ret4J Home Page. Available at http://www.alphaworks.ibm.com/tech/ret4j. Last

accessed on 04/03/2004.

[52] Roberts, D. B. “Practical Analysis for Refactoring”. Ph.D. Thesis, Univ. of Illinois at Urbana-Champaign, 1999.

[53] Russo,A., Nuseibeh, B., Kramer, J., “Restructuring Requirements Specifications for Managing Inconsistency and Changes: A Case Study”, in Proc. of International Conference on Requirements Engineering, pp. 51-65, 1998..

[54] SAX Home page. Available at http://www.saxproject.org. Last accessed on 04/03/2004.

[55] Tamino Home Page. Available at http://www.softwareag.com/tamino. last accessed on 04/03/2004.

[56] Tatarinovi, I., Ives, Z. G., Halevy, A. Y., Weld, D.S. "Updating XML". In Proc. of ACM Special Interest Group on Management of Data (SIGMOD 2001), California, USA, May 2001.

[57] Tichelaar, S. *et al*. “A Meta-model for Language-Independent Refactoring”, in Proc. of the International Symposium on Principles of Software Evolution (ISPSE 2000). Kanazawa, Japan, November 2000.

[58] Tokuda, L., Batory, D., “Evolving object-oriented designs with refactorings”, in Preceedings of 14th IEEE International Conference on Automated Software Engineering, Cocoa Beach, FL , Estados Unidos, p. 174-181, 12 – 15 October, 1999.

[59] Tourwé, T., Brichau, J., Mens, T., “Using Declarative Metaprogramming to Detect Possible Refactorings”, in Proc. of 17th Automated Software Engineering (ASE 2002), 23–27 September, Edinburgh, SCOTLAND, 2002.

[60] Transmogrify home page. Available at http://transmogrify.sourceforge.net/. Last accessed on 01/05/2004.

[61] W3C. “Document Object Model (DOM)”. Available at http://www.w3.org/DOM. Last accessed on 04/03/2004.

[62] W3C. “Extensible Markup Language (XML)”. Available at http://www.w3.org/ TR/xml. Last accessed on 04/03/2004.

[63] W3C. “XML Path Language (XPath) Version 1.0”. W3C Recommendation November 1999. Available at http://www.w3.org/TR/xpath. Last accessed on 04/03/2004.

[64] W3C. “XQuery 1.0 : An XML Query Language”. W3C Working Draft 12 November 2003. Available at http://www.w3.org/TR/xquery/. Last accessed on 04/03/2004.

[65] W3C. “XSL Transformations (XSLT) Version 1.0”. W3C Recommendation November 1999. Available at http://www.w3.org/TR/xslt. Accessed on 04/03/2004.

[66] Warmer, J. B., Kleppe, A. G., “The Object Constraint Language: Precise Modeling With UML”. Addison-Wesley, 1st Edition, 1998.

[67] World Wide Web Consortium (W3C) Home Page. Available at http://www.w3c.org. Last accessed on 01/04/2004.

[68] XIndice Home Page. Available at http://xml.apache.org/xindice/index.html. Last accessed on 27/07/2004.

[69] XML:DB. “XUpdate – XML Update Language Working Draft”. Available at http://www.xmldb.org/xupdate/. Last accessed on 04/03/2004.

[70] XRefactoring Home page. Available at http://www.xref-tech.com/. Last accessed on 01/05/2004.