Suporte a Refatorações em um Sistema de Transformação de Propósito Geral

Gustavo Santos[1], **Paulo Borba**[1]

[1]Centro de Informática – Universidade Federal de Pernambuco (UFPE)
Av. Professor Luís Freire, s/n, Cidade Universitária, CEP 50740-540 – Recife – PE – Brasil

{gas,phmb}@cin.ufpe.br

Abstract. *Refactorings are behavior-preserving transformations for improving the programs' code quality. The benefits of this technics estimulated the appearance of an increasing number of refactoring support tools. Indeed, new refactorings appear systematically and the existing ones frequently need some customization. We present in this paper a program transformation system which provides a domain specific language for specifying refactorings. The system was developed as an extension of JaTS, a Java transformation system. We validated the language by describing several refactorings, including Extract Method.*

Resumo. *Refatorações são transformações de código que melhoram a qualidade do programa preservando seu comportamento. Os benefícios desta técnica fizeram com que surgissem várias ferramentas de apoio à aplicação de refatorações. No entanto, novas refatorações surgem sistematicamente e, mesmo as já existentes, freqüentemente precisam de customizações. Neste artigo apresentamos um sistema de transformação que oferece ao usuário uma linguagem de domínio específico capaz de expressar refatorações. O sistema foi desenvolvido como uma extensão de JaTS, um sistema de transformação para Java. A proposta foi validada através da descrição de várias refatorações, inclusive Extract Method.*

1. Introdução

A técnica de refatoração de código[1] vem se tornando cada vez mais comum em processos de desenvolvimento de software. A reestruturação contínua durante o processo de desenvolvimento contribui para um aumento da qualidade do código em termos de legibilidade, modularidade e reusabilidade. Desta forma, ganha-se em produtividade no processo como um todo. Por esta razão, refatorações são fortemente estimuladas em metodologias ágeis, como *Extreme Programming* (XP) [Beck and Andres 2004].

A popularização da técnica de refatorações fez com que a maior parte dos ambientes de desenvolvimento passassem a integrar ferramentas que permitem a automatização da aplicação de refatorações pré-definidas, como é o caso do Eclipse [EclipseProject 2005], JBuilder [Borland 2005], Intellij IDEA [JetBrains 2005] e MS Visual Studio [Microsoft 2005].

Apesar do benefício que as ferramentas de suporte a refatorações oferecem ao processo de desenvolvimento, elas compartilham uma limitação comum: oferecem um

[1]Do inglês *refactoring* - Técnica de reestruturar o código do programa sem alterar seu comportamento observável [Fowler et al. 1999].

número limitado de transformações, em geral um subconjunto das refatorações descritas em [Fowler et al. 1999]. Quando da necessidade por parte do usuário de customizar ou definir uma nova refatoração, a tarefa se mostra custosa e pouco amigável, já que é necessária a alteração direta do código fonte das ferramentas existentes.

1.1. Customização de Refatorações

A realidade tem mostrado que a necessidade de customização de refatorações é bastante comum. Mesmo as refatorações clássicas presentes nas ferramentas comerciais mostram-se incorretas em várias situações [Ettinger and Verbaere 2005]. Além disto, mesmo corretas, algumas refatorações podem gerar incorreções arquiteturais em alguns sistemas. Como exemplo, considere o código Java da Figura 1 que declara uma classe implementando a interface `java.io.Serializable`, o que indica, portanto, que a classe pode ser armazenada no sistema de arquivos ou transmitida via rede. Um bom projeto de software recomendaria encapsular os atributos `numero`, `rua` e `cidade` em uma classe `Endereco`, melhorando assim a modularidade do sistema. Neste caso a refatoração mais indicada seria *Extract Class*. No entanto, o resultado da aplicação desta refatoração geraria um código semanticamente incorreto, que não preserva o comportamento anterior do programa, como se pode constatar na Figura 2. Como a nova classe `Endereco` não implementa a interface `Serializable`, parte da informação da classe `Cliente` deixaria de ser serializável, fazendo com que as informações relativas ao endereço não fossem mais transmitidas via rede ou armazenadas em arquivo.

```
1  public class Cliente implements Serializable {
2      String nome;
3      int numero;
4      String rua;
5      String cidade; ...
6  }
```

Figura 1. Classe Serializável

Em vista deste tipo de particularidade, é natural oferecer ao usuário a possibilidade da definição e customização de suas próprias refatorações de uma forma mais amigável. Mais precisamente, é importante a existência de uma linguagem voltada para o domínio de transformações complexas, que permita a definição de refatorações de forma mais declarativa e concisa em relação às linguagens de programação de propósito geral. Desta forma, refatorações podem ser tratadas como programas e, portanto, definidas e customizadas de acordo com as necessidades do próprio usuário.

```
1  public class Cliente
2      implements Serializable {
3      String nome;
4      Endereco endereco; ...
5  }
```

```
1  public class Endereco {
2      int numero;
3      String rua;
4      String cidade; ...
5  }
```

Figura 2. Incorreção Semântica Gerada por *Extract Class*

1.2. Contribuições

O trabalho aqui apresentado tem como principal objetivo oferecer ao programador um sistema de transformação que possibilite a definição de refatorações através de uma linguagem de domínio específico. O sistema foi desenvolvido como uma extensão de JaTS [Castor et al. 2001], um sistema de transformação para a linguagem Java, descrito em mais detalhes na Seção 2.

Refatorações de código são transformações complexas que exigem manipulações de estruturas de granularidade muito fina e alto poder de análise de código por parte da ferramenta de automatização. Isto dificulta a implementação deste tipo de transformação em sistemas de propósito geral, que, na maioria dos casos, não oferecem expressividade suficiente para tal, ou mostram-se pouco amigáveis para o programador convencional. Uma análise dos principais sistemas e suas limitações é apresentada na Seção 5.

A capacidade de tratar refatorações foi introduzida em JaTS através da extensão da linguagem de *templates* original, como mostra a Figura 3. Portando, a linguagem de transformação passou a ser a composição de três sub-linguagens, sobre as quais entraremos em detalhes na Seção 3. A nova linguagem passou a incorporar as seguintes novas características:

- Suporte a micro-transformações;
- Suporte a transformações globais e tratamento de código em corpo de métodos;
- Linguagem para manipulação de meta-estruturas e aplicação de transformações;
- Linguagem para análise de código.

Figura 3. Linguagem de Transformação

Estas características são apresentadas em detalhes na Seção 3. Na Seção 4. é discutida uma avaliação da linguagem a apresentados alguns exemplos de refatorações implementadas. Na Seção 6. são discutidas algumas conclusões e trabalhos futuros.

2. JaTS Original

Apresentamos nesta seção um breve resumo das características originais de JaTS. O objetivo aqui é fornecer uma visão geral do sistema ao leitor não familiarizado com o mesmo, no entanto, maiores detalhes podem ser encontrados na literatura [Castor et al. 2001, Castor and Borba 2001].

O JaTS [Castor et al. 2001] é um sistema de transformação de propósito geral para a linguagem Java. Em sua forma original, JaTS é composto por uma linguagem de

domínio específico para definição de *templates* (JaTS-TL) e um engenho responsável por automatizar a aplicação de transformações descritas em JaTS-TL. Tais transformações são compostas por um conjunto de *templates* fonte, utilizados para extração de informações do código original através do casamento destes padrões com as classes fonte e um conjunto de padrões destino, representando a estrutura sintática do código a ser gerado. Pré-condições são opcionalmente associadas a padrões destino, determinando se a transformação deve ou não ocorrer. A Figura 4 mostra uma visão geral do processo de transformação em JaTS.

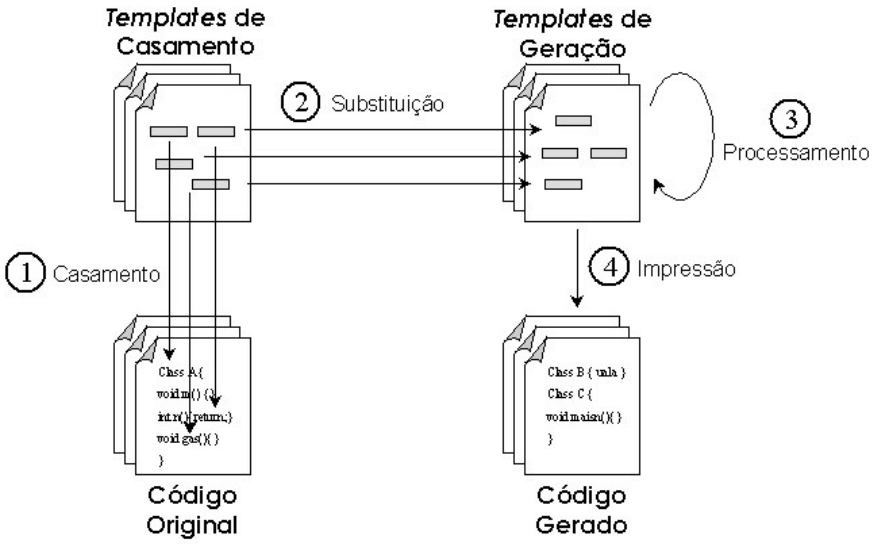

Figura 4. Ciclo de uma Transformação no JaTS Original

JaTS-TL é uma extensão sintática da linguagem Java. Através dela o usuário pode definir padrões de casamento e geração de código de maneira rápida e intuitiva, com um custo de aprendizado baixo, devido a sua similaridade com a sintaxe de Java. A estrutura básica de JaTS-TL, que possibilita a definição de padrões de casamento e o intercâmbio de informações entre os padrões fonte e destino, são as variáveis JaTS. Elas são utilizadas como armazenadoras de informações do código e possuem tipos que correspondem a estruturas sintáticas do programa. Os tipos variam desde estruturas simples, como identificador, nome qualificado, parâmetros, até estruturas mais complexas, como conjunto de métodos e conjunto de atributos. O código a seguir mostra um exemplo simples de *template* escrito em JaTS-TL. A variável `#C` representa o nome da classe e as variáveis `#fds` e `#mds` os conjuntos de atributos e métodos respectivamente.

```
public class #C #[ implements #interface ]# {
    FieldDeclarationSet:#fds;
    MethodDeclarationSet:#mds;
}
```

Além de variáveis JaTS, a linguagem de padrões disponibiliza também algumas construções mais elaboradas, que dão mais flexibilidade ao processo de casamento e mais expressividade na definição dos padrões de geração. Um exemplo deste tipo de construção é a cláusula opcional, que possibilita a definição de trechos de casamento opcional em padrões, como é o caso do trecho declarado entre "`#[`" e "`]#`" no *template* anterior,

indicando que a cláusula *implements* pode ou não aparecer no código original.

Existem ainda as construções de geração, conhecidas em JaTS como declarações executáveis, que se subdividem em expressões executáveis, declarações condicionais e declarações iterativas. Estes recursos permitem a realização de processamento em *templates* de geração. Não entraremos em maiores detalhes sobre estas construções aqui pois as mesmas já foram vastamente exploradas na literatura citada e maiores esclarecimentos fogem ao escopo deste trabalho. É importante apenas que o leitor tenha ciência da existência de construções que permitem processamento em *templates* de geração e não apenas a mera representação estrutural através de variáveis.

O JaTS original é um sistema cujo principal diferencial em relação a outros sistemas de transformação é a familiaridade oferecida ao usuário para definição de transformações. Ele mostrou-se bastante adequado ao uso em aplicações que envolviam forte geração de código, como, por exemplo, geração de padrões arquiteturais. No entanto, o JaTS original apresenta limitações quando em se tratando de transformações que envolvem análise de código e manipulações de fina granularidade. Além do mais, uma linguagem puramente declarativa não é suficiente para se expressar transformações elaboradas, que exijam, por exemplo, composição de outras transformações.

3. Suporte a Refatorações

Ferramentas com suporte a refatorações precisam dispor de alguns recursos essenciais que viabilizem este tipo de transformação, tais como:

- Manipulação de sub-estruturas de árvores sintáticas com fina granularidade;
- Consultas sobre o código;
- Análise de tipos e declarações;
- Transformações globais com efeito em múltiplas classes.

O objetivo deste trabalho é embutir tais características em uma linguagem de domínio específico que permita ao usuário definir suas próprias refatorações sem a necessidade de escrever código Java. Portanto, tal linguagem deve prover recursos que atendam cada um dos requisitos citados anteriormente, além dos requisitos básicos necessários a qualquer ferramenta de geração de código. Para isto, optamos por não apenas estender JaTS-TL, como descrito na Seção 3.1., mas também compor uma nova linguagem para manipulação de meta-estruturas, descrita na Seção 3.2. e uma linguagem de análise, descrita da Seção 3.3..

3.1. Extensões de JaTS-TL

A manipulação de sub-estruturas internas a declarações de métodos são freqüentes em refatorações. A grande maioria das transformações descritas em [Fowler et al. 1999] exigem algum tipo de processamento em corpo de métodos. Isto exigiu que alguns novos tipos de variáveis JaTS fossem definidos para que fosse possível representar estruturas como expressões e *statements* Java. Também foi necessário estender o escopo de ocorrência de declarações executáveis, que originalmente só eram possíveis em contextos externos a métodos.

Além das extensões já citadas, que intuitivamente eram naturais, algumas construções mais elaboradas foram produzidas e serão descritas ao longo desta seção.

StatementList. A representação de listas de *statements* é um problema que merece particular atenção em sistemas baseados em linguagens declarativas para casamento e geração. A principal dificuldade surge na etapa de casamento, onde a ocorrência de variáveis pode ser intercalada com valores explícitos. Considere, por exemplo, o código da Figura 5, onde a cada chamada do método `operacao()` um contador é incrementado e um registro de log é impresso.

```
1 public class Exemplo {
2     int contador; ...
3     void operacao() { ...
4         contador++;
5         System.out.println("Chamada nr.:"+contador);
6         ...
7     }
8 }
```

Figura 5. Exemplo de *logging*

Suponha que o usuário pretenda extrair a lógica de impressão para um método isolado, ou seja, substituir o código da linha 5 por uma chamada de método. Neste caso, bastaria definir um *template* de casamento contendo variáveis do tipo *StatementList*, como o seguinte:

```
1 ...
2 void operacao() {
3     StatementList:#codigoAnterior;
4     System.out.println("Chamada nr.:"+contador);
5     StatementList:#codigoPosterior;
6 }
```

E um *template* de geração que subtitui o código por uma chamada de método:

```
1 ...
2 void operacao() {
3     StatementList:#codigoAnterior;
4     log();
5     StatementList:#codigoPosterior;
6 }
7 void log() {
8     System.out.println("Chamada nr.:"+contador);
9 }
```

Num caso como este, pode haver várias situações onde um casamento determinístico com o código fonte poderia não ser possível. Por exemplo, se houver a ocorrência de múltiplas linhas de código idênticas à da linha 5 da Figura 5.

Nossa solução buscou mitigar a interferência do usuário em todos os casos onde a inferência da lógica de casamento era possível. Para tal, definimos a semântica de casamento seqüencial, que realiza o casamento de acordo com a ordem de ocorrência dos *statements*. O algoritmo garante que se houver pelo menos uma configuração de casamento possível, o sistema o realizará com sucesso. Esta técnica agrega grande poder

de expressividade à linguagem e não temos conhecimento de nenhum outro sistema de transformação que adote abordagem semelhante.

ContextDeclaration. A extração de trechos de código e a manipulação de contextos é uma técnica comum em refatorações como *Extract Method* e *Replace Temp With Query*. Nestes casos, o código a ser extraído pode estar emaranhado em qualquer parte do sistema e nem sempre variáveis do tipo *StatementList* são suficientemente expressivas, pois elas não podem representar o contexto que envolve trechos de código com aninhamento. Suponha, por exemplo, que o método da linha 3 da Figura 5 tivesse a seguinte forma:

```
1 void operacao() { ...
2     if (logging) {
3         contador++;
4         System.out.println("Chamada nr.:"+contador);
5     } ...
6 }
```

Neste caso, o código anterior à linha 4 não poderia ser representado por uma lista de *statements*, já que o *if* estaria sintaticamente incompleto. Para este tipo de situação foi definido um tipo de variável específico chamada `ContextDeclaration`, que nos permitire definir o *template* de casamento segundo o código da Figura 6:

```
1 void operacao() {
2     ContextDeclaration:#contexto{#codigo};
3 }
```

Figura 6. Declaração de Contexto

Através da linguagem de meta-programação que será apresentada na Seção 3.2. é possível a parametrização da transformação para que o casamento ocorra com sucesso. Neste caso, a variável `#codigo` receberia o valor `System.out.println("Chamada nr.:"+contador);`.

Maiores detalhes sobre manipulação de contextos e a construção `ContextDeclaration` podem ser encontradas em [Santos and Borba 2006].

Micro-transformações. Algumas refatorações exigem a transformação de subestruturas de um grau muito fino e, muitas vezes, espalhadas por toda a AST. É o que ocorre, por exemplo, na refatoração *Encapsulate Field*. Todas as expressões de acesso direto ao atributo que se deseja encapsular devem ser substituídas por chamadas a métodos de acesso. Ou seja, precisa-se realizar pequenas transformações através de uma ou várias classes. Para este tipo de situação, desenvolvemos o conceito de micro-transformações em JaTS, onde os *templates* fonte e destino de uma transformação podem representar qualquer tipo de nó da AST representáveis por variáveis JaTS. Adicionalmente, foi desenvolvida uma construção que define a estratégia de aplicação de micro-transformações. A sintaxe de declaração tem a seguinte forma:

```
#explore #<variavel> {
    #where (<tipo_de_no>) {
          <padrao_fonte> :=> <padrao_destino>
          <+ ...
    } ...
}
```

Declarações `#explore` possibilitam a composição de micro-transformações de forma a se obter uma transformação única que afete múltiplas sub-estruturas da AST isoladamente. Cada cláusula `#where` define uma micro-transformação que afeta um tipo de nó específico, determinado pelo parâmetro da cláusula.

Internamente às cláusulas `#where`, micro-transformações podem ser compostas através do operador composicional `<+`, também encontrado, com leve variação semântica, no sistema de transformação Stratego [Visser 2004]. A composição `t1 <+ t2` indica que a transformação `t1` deve ser aplicada e, apenas se falhar, aplica-se `t2`. Através do operador composicional, é possível a definição de micro-transformações mais flexíveis, com múltiplos padrões de casamento.

A Seção 4. apresenta um exemplo de uso prático desta construção na refatoração *Encapsulate Field*.

3.2. Linguagem de Meta-programação

Alguns sistemas de transformação usam estratégias implícitas para aplicação de transformações, como é o caso do JaTS original, que pressupõe a sequência descrita na Figura 4. No entanto, algumas refatorações exigem um controle mais refinado da operação, sendo vantajoso que este tipo de recurso esteja disponível para o usuário. Em JaTS, optamos por definir uma linguagem de domínio específico para manipulação de elementos de transformação, ou seja, uma linguagem para escrita de meta-programas (JaTS-ML). A linguagem foi projetada como extensão de um subconjunto da sintaxe de Java. Mais especificamente, a linguagem se constitui de *statements* Java estendidos com construções de meta-programação. Esta técnica permitiu embutir naturalmente na linguagem alguns recursos importantes para a meta-programação, como exceções, I/O, *multi-threading* e uso de APIs externas.

As construções específicas para meta-programação permitem ao usuário a definição e aplicação de transformações de forma concisa e intuitiva. Por concisão, não apresentaremos aqui a BNF da linguagem. No entanto daremos uma visão geral de suas principais construções ao longo desta seção.

Declaração de Templates. *Templates* são tipos válidos da linguagem JaTS-ML e podem ser declarados e manipulados como cidadãos de primeira classe dentro do meta-programa. Por exemplo, a declaração do *template* de uma declaração de classe seria um *statement* da seguinte forma:

```
1 ClassDeclaration #cd {
2     public class #C {
3         ...
4     }
5 };
```

Onde `ClassDeclaration` é o tipo da estrutura representada pelo *template*. Em JaTS-ML *templates* podem possuir o tipo de qualquer estrutura da AST representável por variáveis JaTS. Declarações de *templates* podem ainda ser realizadas em arquivos separados e carregadas no meta-programa.

Valoração de Meta-variáveis. Uma das formas de se parametrizar transformações de programas baseadas em padrões é a pré-valoração de meta-variáveis. Este recurso foi adicionado à linguagem através de uma construção específica para atribuição de valores a variáveis JaTS. Sendo que neste caso, "valores" são trechos de código Java.

No exemplo de *template* citado na Figura 6 mencionamos que a variável `#codigo` deveria ser valorada para que o casamento se realizasse com sucesso. A seqüência de comandos em JaTS-ML para tal seria:

```
1 Statement stm <- { System.out.println("Chamada nr.:"+contador); };
2 resultSet <- (#codigo, stm);
```

Onde `resultSet` representa o conjunto de mapeamentos de variáveis previamente declarado. A linha 2 indica a introdução do mapeamento do valor de `stm` na variável `#codigo`.

Definição de Transformações. Transformações também são tipos em JaTS-ML e, conseqüentemente, a possibilidade de definí-las explicitamente foi introduzida na linguagem. Uma transformação em JaTS possui a forma: $p \rightarrow [pre]\ p'$. Onde p representa um *template* de casamento e p' um de geração. Opcionalmente pode ser introduzida uma pré-condição, que precisa ser satisfeita para que a transformação se realize. Construções deste tipo são valores em JaTS-ML e podem ser atribuídas a variáveis do tipo `Transformation`, como no exemplo a seguir:

```
Transformation t <- #p :=> #newp;
```

Onde `#p` e `#newp` são *templates* JaTS que representam alguma estrutura sintática.

A manipulação explícita de transformações dá ao usuário o poder de lidar simultaneamente com múltiplas entidades deste tipo, o que favorece sobremaneira a definição de estratégias de aplicação.

Aplicação de Transformações. Podemos dizer que transformações em JaTS são funções de alta ordem que se aplicam a estruturas sintáticas, segundo o seguinte modelo: $\tau(\rho) \rightarrow \rho'$. Onde τ é uma transformação definida pelo usuário e ρ, ρ' são estruturas sintáticas. Em JaTS-ML utilizamos o operador `<|` para representar a aplicação de uma transformação sobre uma estrutura, da seguinte forma: `codJava <| t/resultSet`. O símbolo `/` indica que a transformação `t` deve levar em consideração o conjunto de mapeamentos `resultSet`. Em outras palavras, este recurso permite a parametrização da transformação.

A aplicação de uma transformação, se bem sucedida, produz uma estrutura sintática com tipo correspondente ao estabelecido pelo *template* de geração que compõe a transformação. Portanto, aplicações de transformações podem ser atribuídas a variáveis de tipo correspondente ao da transformação. Estruturas sintáticas Java são representadas genericamente em JaTS-ML pelo tipo `Code`. O exemplo a seguir ilustra um comando de

aplicação de transformação:

```
Code resultado <- codJava <| t/resultSet;
```

Onde `codJava` é uma variável do tipo `Code` previamente declarada e valorada. Neste caso, `resultado` receberá, após a execução do comando, o resultado da transformação.

A composição seqüencial de transformações é trivialmente definida em JaTS-ML através de comandos consecutivos, como o exposto anteriormente. Além disto, como já foi dito previamente, JaTS-ML suporta a utilização de comandos Java imersos no meta-programa. Desta forma, pode-se definir estratégias de aplicação iterativas ou mesmo capturar entradas do usuário durante a transformação.

3.3. Linguagem de Análise

Refatorações são transformações que exigem um alto grau de análise sobre o código do programa. Pré-condições e extração de informações exigem consultas elaboradas que tornam-se extremamente complexas se expressadas imperativamente em uma linguagem convencional. Por esta razão, desenvolvemos uma linguagem de expressões (JaTS-AL) que permite realizar consultas e validações sobre o código do programa de forma concisa. Esta linguagem foi inspirada na linguagem JunGL [Verbaere et al. 2006] e adota um estilo de programação similar, porém se diferencia em alguns aspectos, entre eles a sintaxe de operadores.

Basicamente, expressões de consulta em JaTS são predicados lógicos na forma de expressões regulares que descrevem um caminho na árvore do programa. No exemplo a seguir, dado que `#var` foi previamente associada a um nó da árvore que representa um `Name`, esta expressão verifica se tal variável ocorre após uma cláusula `return`.

```
#var <<* Statement:#st <: Statement:#rs >>* ReturnStatement:#p
```

O operador `<<*` indica a possibilidade de se atingir `#var` a partir de `#st` em um percurso *top-down*. O símbolo `*` tem a mesma semântica que em expressões regulares, ou seja, representa zero ou mais elementos. Portanto, a sub-expressão `#var <<* Statement:#st` indica que `#var` é filho, direto ou indireto, do *statement* `#st`. Existem operadores de ascendência (`>>`), descendência (`<<`), sucessão (`<:`) e precedência (`:>`). Sendo que cada um pode ser combinado com os modificadores + e `*`. Além dos operadores se travessia, pode-se combinar operadores lógicos `&&` (e) e `||` (ou).

Extração de informações em JaTS-AL pode ser realizada através de compreensão de conjuntos. Por exemplo, o comando para extração da lista de argumentos de um método criado a partir de um trecho de código mapeado à variável `#codigo` em um *template* como o seguinte:

```
1 void #metodo(ParameterList:#params) {
2     ContextDeclaration:#contexto{#codigo};
3 }
```

Seria:

```
NameList args <- #{ Name:#var | scope[#codigo]
      && dec(#var) <<* #params
      || (dec(#var) <<* #contexto && !(dec(#var) <<* #codigo)) }#;
```

Onde `scope[...]` indica o escopo de atuação do predicado e `dec(...)` é uma função pré-definida da linguagem que retorna o nó de declaração do nome mapeado à variável recebida como parâmetro.

4. Avaliação

Para constatarmos o poder de expressividade da linguagem de transformação, definimos, total ou parcialmente, uma série de refatorações descritas em [Fowler et al. 1999] e algumas customizações das mesmas. Nesta seção apresentamos uma visão geral de dois exemplos práticos de uso do sistema. A implementação das refatorações *Encapsulate Field* e *Extract Method*. Em particular, estas duas refatorações exploram a maior parte dos recursos necessários para que se tenha suporte a qualquer outra refatoração, como efeito em múltiplas classes, análise de código, manipulação de micro-estruturas e composição de transformações.

4.1. *Encapsulate Field*

Esta refatoração é um caso típico de transformação que afeta múltiplas partes do código em um nível de granularidade muito fino. A implementação desta refatoração resultou da composição de três transformações básicas: i) geração de métodos de acesso e substituição de modificadores do atributo; ii) atualização dos acessos ao atributo na classe local; iii) atualização dos acessos ao atributo nas outras classes do sistema. A aplicação seqüencial destas três transformações é trivialmente realizada em JaTS-ML e não entraremos em detalhes aqui. A transformação (i) já era suportada no JaTS original e foi explorada na literatura [Castor and Borba 2001]. Portanto, focaremos aqui nas transformações (ii) e (iii).

Ambas as transformações adotam a construção `#explore` para compor e aplicar micro-transformações através do código, diferenciado-se apenas pelo contexto de atuação e pela pré-condição que determina a aplicabilidade das micro-transformações. O trecho de *template* a seguir mostra parte do código utilizado na transformação (ii):

```
#explore #classDec {
    #where (UnaryExpression) {
        #var++ :=>
        ?[ dec(#var) <<* #fds ]?
        set::#var(get::#var() + 1)
    }
    #where (BinaryExpression) {
        #var + #val :=>
        ?[ dec(#var) <<* #fds ]?
        get::#var() + #val
    } ...
}
```

Neste caso, a variável `#classDec` havia sido casada previamente com a declaração da classe que contém o atributo e a variável `#var` com o nome do atributo a ser encapsulado.

A transformação (iii) adota uma técnica similar, porém a variável sobre a qual atua o `#explore` é do tipo `ClassDeclaratioSet` e representa o conjunto de classes restantes do sistema.

4.2. *Extract Method*

Do ponto de vista estrutural, a refatoração *Extract Method* pode ser dividida em dois tipos: extração de *statements* e extração de expressões. Sem dúvida, o primeiro tipo é o mais utilizado e, por esta razão, foi o abordado neste trabalho.

A principal dificuldade em se implementar esta refatoração em uma ferramenta de propósito geral está na elaborada análise de código exigida. Esta análise inclui: chegagem da validade do trecho a ser extraído, determinação dos parâmetros do novo método e determinação do tipo de retorno.

Um trecho de código só pode ser extraído para um método se nele houver apenas uma entrada e uma saída no fluxo de execução (não considerando-se interrupções por exceções). Portanto, esta validação em Java pode ser realizada através da verificação da não existência de comandos de interrupção de fluxo "órfãos" no trecho em questão. A expressão a seguir verifica que para todo comando `break` existe também um laço iterativo correspondente no trecho de código `#codigo`:

```
#forall BreakStm:#b in #codigo ->
        (#b <<* (WhileSatement:#x | ForStatement:#x) <<* #codigo)
```

A determinação dos parâmetros do método é realizada através de compreensão de conjuntos, como exemplificado na Seção 3.3..

O retorno do método é determinado por uma série de verificações também expressadas em JaTS-AL e omitidas aqui por concisão.

5. Trabalhos Relacionados

Atualmente existem diversos esforços da comunidade de transformação de programas no sentido de agregar poder a ferramentas baseadas em linguagens de transformação. Merece destaque a linguagem JunGL [Verbaere et al. 2006], desenvolvida com foco em suporte a refatorações. JunGL é uma linguagem funcional que agrega grande poder de análise, graças ao conceito de arestas dinâmicas em grafos e predicados de consulta sobre caminhos. No entanto, esta expressividade não se reflete na descrição da transformação, que acaba se mostrando menos declarativa do que em sistemas de reescrita. Este é o caso de Stratego [Visser 2004], um sistema baseado em regras de reescrita que oferece ao usuário a possibilidade de definir explicitamente a estratégia de aplicação de transformações. As regras de reescrita são recursos poderosos para definição de reestruturações de código. No entanto, a tarefa de análise não é beneficiada. Stratego busca contornar esta limitação através da introdução de regras dinâmicas durante a transformação. Porém, este recurso se mostra menos declarativo e expressivo do que linguagens específicas para análise.

A idéia da utilização de *templates* em transformações de código é também adotada por MetaJ [Oliveira et al. 2004], um ambiente extensível de meta-programação para Java. A importância de uma linguagem de consulta para a definição de refatorações foi identificada pelos autores, devido à extensão do código que se fazia necessário para

análise e extração de informações. Desta percepção foi desenvolvida SCQL, uma linguagem de consulta sobre código. Em geral, MetaJ se mostra uma ferramenta poderosa para expressão de transformações complexas, no entanto a ausência de uma linguagem de domínio específico para descrição de meta-programas torna a tarefa extensiva em refatorações elaboradas.

Também voltado para refatorações, foi desenvolvido Refax [Mendonca et al. 2004], um arcabouço para desenvolvimento de ferramentas de refatoração baseado em XML. O foco deste trabalho é oferecer ao usuário um ambiente de fácil extensão para múltiplas linguagens, beneficiando-se dos recursos já existes para manipulação de estruturas XML. A adoção destes recursos mostrou-se eficiente em situações que envolviam baixa complexidade nas refatorações, como é o caso de *Rename Class*. No entanto, apesar da definição de refatorações complexas não ter sido analisada pelos autores da ferramenta, pôde-se constatar que seria um trabalho extremamente verboso.

6. Conclusões e Trabalhos Futuros

Neste artigo apresentamos um sistema de transformação de propósito geral com suporte a refatorações de código definidas pelo usuário. A solução aqui apresentada buscou unir o poder de transformação das ferramentas específicas para refatorações e a generalidade e facilidade de extensão das ferramentas de transformação de propósito geral. O sistema foi desenvolvido como uma extensão de JaTS, um sistema de transformação para a linguagem Java baseado em padrões de reescrita. A linguagem de transformação foi definida como uma composição de três linguagens para os domínios: definição de *templates*, definição de estratégias e análise de código. A mesma mostrou-se bastante expressiva, o que permite ao usuário a definição de refatorações customizadas em menos tempo em relação a outras ferramentas e com menos possibilidade de erros. O poder da linguagem foi constatado através da definição de uma série se refatorações, entre elas *Encalsulate Field* e *Extract Method*. Esta última considerada a mais complexa das refatorações [Fowler 2001]. Apesar de ainda não termos definido todas as refatorações clássicas apresentadas em [Fowler et al. 1999], realizamos uma análise detalhada sobre o catálogo que nos permitiu constatar com bom nível de confiança que a linguagem cobre satisfatoriamente todos os casos.

Como trabalhos futuros, planejamos descrever um número maior de refatorações a fim de se comprovar a expressividade da linguagem. Além disto, é importante a conclusão da implementação, que atualmente restringe o nível de cobertura de algumas construções.

Planejamos ainda adotar a técnica de geração semi-automática de sistemas de transformação baseados em JaTS através de *bootstrapping* [Souza and Borba 2005] e estender a solução apresentada neste artigo para outras linguagens além de Java. Acreditamos que esta abordagem é possível devido a semelhança arquitetural entre as versões de JaTS e à generalidade das linguagens de meta-programação e análise definidas nesta nova versão.

7. Agradecimentos

Nossos agradecimentos aos membros do *Software Productivity Group* pelas valorosas contribuições ao trabalho, aos revisores anônimos pelas ricas sugestões que em muito

contribuíram para a melhoria deste artigo e ao CNPq pelo suporte financeiro parcial aos autores.

Referências

Beck, K. and Andres, C. (2004). *Extreme Programming Explained*. Addison-Wesley, 2th edition.

Borland (2005). JBuilder Home Page - http://www.borland.com/jbuilder/.

Castor, F. and Borba, P. (2001). A language for specifying Java transformations. In *SBLP '01: Proceedings of the 5th Brazilian Symposium on Programming Languages*, pages 236–251, Curitiba, PR, BRAZIL.

Castor, F., Oliveira, K., Souza, A., Santos, G., and Borba, P. (2001). JaTS: A Java transformation system. In *XV Brazilian Symposium on Software Engineering*, pages 374–379.

EclipseProject (2005). Eclipse Home Page - http://www.eclipse.org/.

Ettinger, R. and Verbaere, M. (2005). Refactoring bugs in Eclipse, Intellij IDEA and Visual Studio. Technical report, Computing Laboratory. Oxford University, http://progtools.comlab.ox.ac.uk/projects/refactoring/bugreports.

Fowler, M. (2001). Crossing Refactoring's Rubicon - http://www.martinfowler.com/articles/refactoringrubicon.html.

Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

JetBrains (2005). IntelliJ IDEA Home Page - http://www.jetbrains.com/idea/.

Mendonca, N. C., Maia, P. H. M., Fonseca, L. A., and Andrade, R. M. C. (2004). RefaX: A Refactoring Framework Based on XML. In *20th IEEE International Conference on Software Maintenance (ICSM'04)*, pages 147–156. IEEE Computer Society.

Microsoft (2005). MS Visual Studio Home Page - http://msdn.microsoft.com/vstudio/.

Oliveira, A. A., Braga, T. H., Maia, M. A., and Bigonha, R. S. (2004). Metaj: An extensible environment for metaprogramming in java. *J. UCS*, 10(7):872–891.

Santos, G. and Borba, P. (2006). Contextos de Primeira Classe em Transformações de Programas. In *X Brazilian Symposium on Software Engineering*, pages xx–xx. submitted - Available at http://www.cin.ufpe.br/~ gas.

Souza, A. and Borba, P. (2005). Geração de Sistemas de Transformação de Programas. In *IX Brazilian Symposium on Software Engineering*, pages 50–62.

Verbaere, M., Ettinger, R., and de Moor, O. (2006). Jungl: a scripting language for refactoring. In *International Conference on Software Engineering (ICSE'06)*, pages x–x. IEEE Computer Society. to appear.

Visser, E. (2004). Program transformation with Stratego/XT: Rules, strategies, tools, and systems in StrategoXT-0.9. In Lengauer, C. et al., editors, *Domain-Specific Program Generation*, volume 3016 of *Lecture Notes in Computer Science*, pages 216–238. Spinger-Verlag.