Contextos de Primeira Classe em Transformações de Programas

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. *General purpose transformation systems frequently provide a domain specific language for defining transformations. However, when the transformation deals with code contexts inside method bodies, transformation languages are not expressive enough and require an significative effort from the user for defining such transformations. In this paper we present a proposal for dealing with contexts by treating these structures as first class citzens. The solution was implemented through an extension of JaTS, a general purpose transformation system for the Java language.*

Resumo. *Em geral, sistemas de transformação de propósito geral oferecem ao usuário uma linguagem de domínio específico para definição de transformações. No entanto, quando estas envolvem manipulações de contextos em corpo de métodos, a falta de expressividade das linguagens de transformação neste domínio exige muito esforço do usuário para definição da transformação. Neste artigo apresentamos uma proposta para o manipulação de contextos, tratando estas estruturas como cidadãs de primeira classe em transformações. A solução foi implementada como uma extensão de JaTS, um sistema de transformação de propósito geral para a linguagem Java.*

1. Introdução

A técnica de reestruturar programas com o auxílio de ferramentas vem sendo largamente adotada na indústria de desenvolvimento de software como uma alternativa para aumentar a produtividade. Por reestruração, nos referimos a transformações que utilizam código já existente para extração de informações e conseqüente remodelagem ou geração de código novo, agregando ou não comportamento ao sistema. Entre as reestruturações mais comuns atualmente estão as refatorações, técnica que permite remodelar a estrutura do código sem alterar o comportamento observável do programa [Fowler et al. 1999].

No processo de reestruturação é freqüente a necessidade de extração de trechos de código, como ocorre, por exemplo, na refatoração *Extract Method*, e a conseqüente manipulação do contexto em que tais trechos encontram-se inseridos. Em ferramentas especializadas, tipicamente o usuário seleciona o trecho de código que deseja extrair e fornece algumas informações adicionais para que a transformação se complete. Todo o processo de manipulação do contexto fica, portanto, recôndito à percepção do usuário. Entretanto, em sistemas de transformação de programas de propósito geral, que, em sua maior parte, dispõem de uma linguagem de domínio específico para definição de transformações, a especificação de reestruturações deste tipo mostra-se deveras complexa.

Isto ocorre pelo fato de inexistir um recurso específico nas linguagens de transformação para tratar a manipulações de trechos de código imersos em contextos. Isto obriga o usuário a lidar diretamente com a AST (Árvore Sintática Abstrata) do programa, manipulando tais estruturas imperativamente.

Uma solução plausível para abrandar tal dificuldade seria fazer uso de uma característica oferecida pela maior parte das linguagens de transformação existentes hoje: a possibilidade de definição de padrões de código para extração de informações, através de casamento estrutural, e geração de código, através de modelos parametrizados (*templates*). Desta forma, transformações podem ser declaradas como um conjunto de regras do tipo $p \rightarrow p$', onde p representa um padrão de casamento e p' um padrão de geração. Considerando este tipo de regra, a principal dificuldade em se definir uma transformação envolvendo contextos é a representação de tal estrutura em forma de um padrão que permita o casamento entre o modelo descrito e código original. É natural imaginar uma representação de tal padrão da seguinte forma:

```
void metodo() {
    contexto'
    codigo
    contexto"
}
```

No código acima, `codigo` representa o trecho de código imerso no contexto e os blocos *contexto'* e *contexto"* as porções de código anterior e posterior ao trecho interno, que compostas, formam o contexto completo.

Este modelo, apesar de intuitivamente claro, não permite o tratamento do contexto como um trecho de código válido, já que a estrutura sintática representada por cada bloco estaria eventualmente inválida, como, por exemplo, no seguinte trecho de código Java:

```
void metodo() {
    ──────────────────────────────────────
    int x = 0;
    if (x == 0) {                 contexto'
    ──────────────────────────────────────
        x = 1;                    codigo
    ──────────────────────────────────────
    }
    return x;                     contexto"
    ──────────────────────────────────────
}
```

Individualmente, blocos de contexto seriam sintaticamente inválidos, o que impediria sua representação por meta-variáveis e sua manipulação explícita durante a transformação. Este é um problema que se repete em todos os casos onde o trecho interno ao contexto está aninhado em pelo menos um nível mais interno em relação à raiz do contexto.

Com o objetivo de oferecer ao usuário a possibilidade de tratar contextos de forma mais declarativa, o trabalho aqui apresentado propõe a definição de uma construção específica para manipulação deste tipo de estrutura em linguagens de transformação de

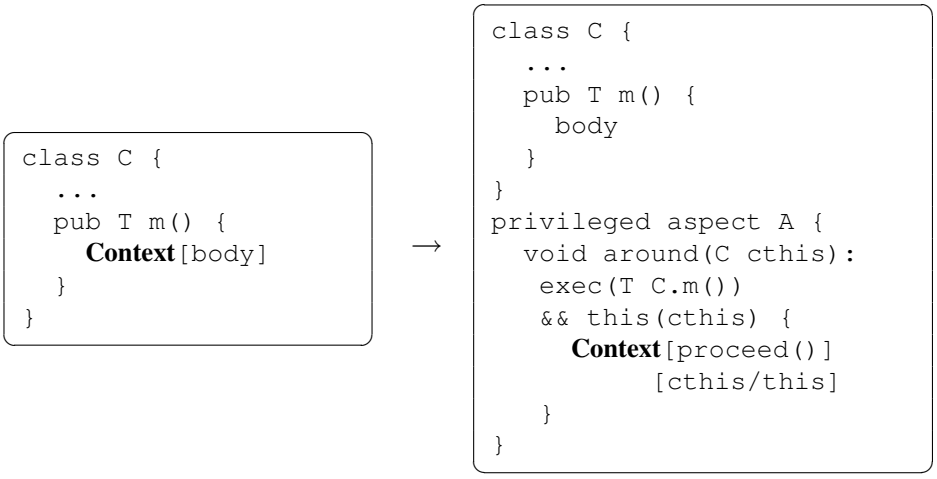

Figura 1. Extração de Contexto para Aspecto

programas, de forma a permitir não apenas a possibilidade da definição de padrões de casamento mais adequados, como também a manipulação do contexto como "cidadão de primeira classe", isto é, que possam ser manipulados como um tipo qualquer durante a transformação. Este recurso foi implementado como uma extensão de JaTS [Castor et al. 2001, Castor and Borba 2001], um sistema de transformação para a linguagem Java.

A importância de uma construção específica para tratamento de contextos em linguagens de transformação é apresentada na Seção 2. Em seguida apresentamos nossa proposta na Seção 3., mostrando detalhes conceituais e práticos da solução. A aplicabilidade da proposta é mostrada através de dois exemplos práticos na Seção 4. e algumas considerações sobre a implementação são expostas na Seção 5. Importantes trabalhos relacionados, direta ou indiretamente a esta pesquisa são avaliados na Seção 6. e, por fim, algumas conclusões e propostas de trabalhos futuros podem ser vistas na Seção 7.

2. Motivação

Transformações que manipulam código em corpo de métodos, onde as estruturas são de fina granularidade e possuem interconexões hierárquicas e seqüenciais, exigem grande poder de expressidade das ferramentas de transformação, que, em geral, não oferecem meios específicos para o tratamento deste tipo de estrutura.

Em particular, manipulações de trechos de código e respectivos contextos associados são muito freqüentes quando estamos lidando com código em nível de comandos. Além de aplicações em refatorações clássicas, como *Extract Method* e *Replace Temp with Query* [Fowler et al. 1999], a manipulação de contextos mostra-se útil em transformações relacionadas a adequação de código a novas tecnologias, como AOP [Kiczales 1996]. Já existem, por exemplo, propostas de refatorações para AspectJ [Monteiro and Fernandes 2005], como a refatoração *Extract Context* [Cole 2005], que possibilita a extração de um contexto para um aspecto, como mostra o modelo da Figura 1. Esta refatoração determina a extração de um contexto para um aspecto que afeta o método de onde o mesmo foi removido através de um *arround*, que preserva a semântica anterior.

Esta realidade atual demonstra que o suporte a tratamento de contextos já é uma carência real na área de transformação de programas. No entanto, as atuais ferramentas não atendem este domínio satisfatoriamente. Em geral, sistemas de transformação adotam a estratégia de realizar manipulação direta da AST do programa para tratar estruturas não representáveis pelas linguagens de transformação. Este é um processo extremamente verboso e que obriga o usuário a ter conhecimento da representação interna da AST do programa, o que é um contraponto em relação ao objetivo dos sistemas de transformação, desenvolvidos para eximir o usuário deste tipo de problema.

Em vista do exposto, é fundamental que haja suporte a manipulação explícita de contextos em linguagens de transformação de programas, de tal forma que se possa tratar conceitualmente este tipo de estrutura em um nível mais alto de abstração e de forma declarativa.

3. Tratando Contextos em Transformações

Seguindo a tendência dos sistemas de transformação atuais, como citamos na Seção 1., procuramos integrar o tratamento de contextos ao paradigma de transformações via padrões de reescrita. Neste caso, o sistema ideal para suportar nossa solução deveria dispor desse tipo de característica.

Enquadra-se neste perfil o sistema de transformação JaTS [Santos and Borba 2006], uma ferramenta composta por uma linguagem de transformação e um macanismo que possibilita a execução de tais transformações. A linguagem de transformação em JaTS é composta por três sub-linguagens: (i) JaTS-TL (*JaTS Template Language*), para definição de *templates* de código usados como padrões de casamento e geração, (ii) JaTS-SL (*JaTS Strategy Language*), voltada para definição de estratégias de aplicação de transformações e manipulação de meta-estruturas, (iii) JaTS-AL (*JaTS Analysis Language*), uma linguagem para consulta e análise sobre código. Identificamos que esta configuração mostrava-se bastante adequada para suportar o tratamento de contextos, por oferecer casamento de padrões (JaTS-TL) e manipulações de meta-estruturas (JaTS-SL), características essenciais ao nosso objetivo, como veremos nas sub-seções seguintes.

3.1. A construção ContextDeclaration

Intuitivamente, podemos descrever a representação de um contexto em torno de um trecho de código como $C[X]$, onde C representa o contexto e X o trecho de código envolvido. A Figura 2 ilustra como esta situação na apresenta na árvore do programa, onde $B = C[X]$.

Em nossa abordagem, X pode ser qualquer seqüência de comandos Java válida presente na AST do programa. Já o contexto C é qualquer trecho de código imerso em um corpo de método tal que $X \in C$. Portanto, o nível de aninhamento de X em relação a C pode ser zero.

O suporte a manipulação de contextos foi introduzido em JaTS através de uma nova construção exclusiva para tal fim. Seguindo o modelo $C[X]$, buscamos definir um tipo de meta-variável que representasse tal estrutura. Portanto, a variável deveria ser constituída pela composição de duas variáveis independentes, o que permitiria a manipulação do trecho de código e do contexto isoladamente. A sintaxe de declaração da construção foi definida da seguinte forma:

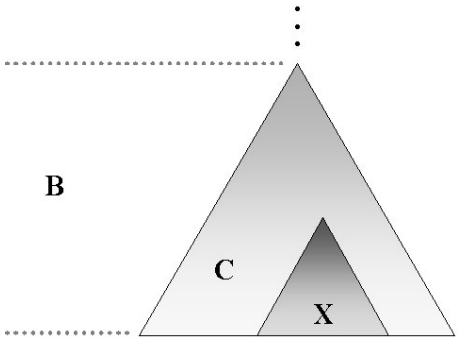

Figura 2. Esboço de um contexto na árvore

```
ContextDeclaration:<variavel_contexto>{ <variavel_codigo> }
```

Neste código, `<variavel_contexto>` e `<variavel_codigo>` representam variáveis JaTS que, isoladamente, possuem o tipo `StatementList`, um tipo JaTS que representa uma seqüência de *statements*.

Declarada em um *template* de casamento, uma construção deste tipo exige algumas informações adicionais, que possibilitem a realização do casamento de forma determinística. Tipicamente, em operações que tratam de extração de trechos de código em ferramentas de desenvolvimento, o usuário seleciona o código de interesse no próprio ambiente de edição. Esta informação é então capturada pelo engenho de transformação, que se encarrega de detectar o trecho selecionado na AST e realizar as operações. Entretanto, um sistema de transformação que utiliza uma linguagem de domínio específico para interface com o usuário (e não um ambiente visual) precisa de outro mecanismo que substitua tal papel. Conceitualmente, dois tipos de informação são fundamentais para que uma operação de seleção se realize:

1. A representação sintática do código de interesse;
2. A posição do código no contexto.

Chamamos tal seleção de *abstração* do trecho de código. Portando, dizemos que a tupla $<x, p>$ é uma abstração do código concreto X, se x é um clone de X e p a sua posição no contexto no qual se encontra.

Em JaTS, a linguagem JaTS-SL foi fundamental para a viabilidade da implementação. O item 1 já era suportado naturalmente pela linguagem, através do artifício da valoração de variáveis. Este recurso permite que variáveis JaTS recebam valores explicitamente e fora do *template* onde ocorrem. O item 2 foi implementado através da introdução de um novo tipo de dado em JaTS-SL, o tipo `Position`. Variáveis deste tipo podem representar uma posição específica na AST do programa e, quando associadas a variáveis que representam trechos de código, indicam exatamente onde tal trecho se inicia em relação ao contexto. A sintaxe de declaração de variáveis do tipo `Position` em JaTS-SL é:

```
Position <id> <- "<lista_de_indices>"
```

Sendo que `<id>` é um identificador Java válido e `<lista_de_indices>` representa uma lista de índices inteiros positivos separados por "`;`". Cada índice indica a

posição de uma sub-estrutura de um nó da AST. Por exemplo, considerando o seguinte trecho de código:

```
1 int x = 0;
2 if (condicao == true) {
3     x = 1;
4 }
```

Para indicar o bloco de código que se inicia na linha 3 seria utilizada a seguinte lista de índices:

```
[1];[1];[0]
```

Onde o primeiro índice representa o *statement* de índice 1 da lista, No caso, o *if* da linha 2. O próximo índice representa a sub-estrutura de índice 1 no nó *if*. Neste caso, o *if* é composto por três sub-estruturas: a expressão de condição, o bloco *then* e o bloco *else*. Portanto, o segundo índice representa o bloco *then*. Por fim, o terceiro índice indica o primeiro *statement* da lista que compõe o bloco *then*.

Este mecanismo de índices permite especificar qualquer posição de um *statement* no código, de forma declarativa e independente de formatação. Esta abordagem tem a vantagem de ser extremamente generalista quanto a expressividade, porém pode mostrase pouco atrativa para o usuário se dele for exigido a construção manual de tais índices. Porém, o objetivo deste trabalho é fundamentar o mecanismo básico da transformação, que poderá ser integrado, por exemplo, em um editor visual que permita construção automática de índices.

3.2. Processo de Execução

O processamento mais elaborado em relação à construção `ContextDeclaration` ocorre durante a fase de casamento, onde a representação abstrata do código é concretizada e o contexto é identificado. O processo ocorre de acordo com as seguinte fases:

1. *Identificação do contexto*: Nesta etapa o engenho identifica a porção do código original que corresponde ao contexto de interesse. Este algoritmo faz parte do processo de casamento de variáveis de tipo `StatementList`, que trata o `ContextDeclaration` como uma lista de *statements* comum;
2. *Busca da posição do trecho de interesse*: Após a identificação do bloco de código completo, o sistema busca a posição onde se encontra o trecho selecionado pelo usuário. Esta informação é extraída da variável de tipo `Position` associada previamente ao contexto;
3. *Casamento do trecho selecionado*: Uma vez identificada a posição da seleção, o sistema executa o casamento estrutural do código associado à variável com o código existente a partir daquele ponto identificado. Caso sejam compatíveis, o processo prossegue;
4. *Extração do trecho selecionado*: Verificada a compatibilidade entre o código abstrato e o concreto, ocorre a extração do trecho de interesse do contexto onde se encontra. O código extraído é então associado à variável que representa a seleção;
5. *Posição do "espaço vazio" no contexto*: Após a extração do trecho de código selecionado, o contexto passa a ser uma lista de *statements* normal. Porém a

informação de onde o trecho de código foi removido é importante para que se realizem novas inserções. Por esta razão, logo após a extração o valor `Position` é transferido da variável que representa o código selecionado para a variável que representa o contexto;

6. *Casamento do contexto*: Por fim, o código correspondente ao contexto, já sem o trecho selecionado pelo usuário, é associado à variável correspondente ao contexto. Esta variável pode ser tratada como um tipo `StatementList` em qualquer parte do sistema, o que torna o contexto um cidadão de primeira classe.

4. Exemplos

A construção aqui apresentada foi avaliada através de uma série de situações envolvendo vários tipos de manipulação de trechos de código e contextos. Apresentamos aqui dois exemplos que demonstram o poder da proposta: a refatoração *Extract Method* [Fowler et al. 1999] e um caso de transformação envolvendo extração de contexto.

4.1. Extract Method

A refatoração *Extract Method* consiste na extração de um trecho de código pré existente para um novo método a fim de se melhorar algum aspecto de qualidade do código do sistema. Este é um caso típico que exige forte manipulação de código relacionado a contexto, já que, além da simples remoção do trecho de interesse, é necessária a inserção de código no contexto (chamada ao novo método) e uma elaborada análise de código sobre o contexto para identificação de parâmetros e validade do código extraído.

Como foi mostrado na Seção 1., a inexistência de uma construção específica para representação de contextos impossibilitava a declaração de um *template* que representasse este tipo de transformação. Entretanto, a nova construção `ContextDeclaration` permitiu a definição de um *template* bastante genérico e expressivo, como o apresentado na Figura 3.

```
1 public class #C {
2     FieldDeclarationSet:#fds;
3     MethodDeclarationSet:#mds;
4
5     ModifierList:#ml Type:#t #metodo(ParameterList:#p) {
6         ContextDeclaration:#contexto{ #codigo };
7     }
8 }
```

Figura 3. *Template* de Casamento para *Extract Method*

A parametrização da transformação é realizada de acordo com as características do código a que se aplica. Ou seja, o usuário fornece apenas os valores das variáveis que sejam fundamentais para a realização do casamento. Por exemplo, se uma classe sendo transformada não possui repetição de nomes de métodos (não há *overloading*) a valoração da variável `#metodo` é suficiente para que o sistema identifique e realize o casamento de toda a estrutura externa ao método.

Para que o casamento da construção `ContextDeclaration` se realize com sucesso, é necessário o fornecimento de algumas informações acerca do trecho de código de interesse, como já foi dito anteriormente. Mais precisamente, é necessário que o usuário informe a abstração <*x*, *p*>, o que pode ser realizado utilizando-se a linguagem JaTS-SL. Para deixar mais clara nossa exposição, apresentaremos o uso desta construção através de um exemplo simples. Considere o seguinte trecho de código Java:

```
1 void m() {
2     if (LOGING) {
3         System.out.println("metodo chamado");
4     }
5     ...
6 }
```

Digamos que o usuário pretenda extrair o código da linha 3 para um novo método chamado `log()` e substituir a linha em questão por uma chamada a este método. Supondo que a classe contenha apenas um único método `m()`, o seguinte trecho de código JaTS-SL seria suficiente para que o casamento do código com o *template* da Figura 3 se realizasse com sucesso (o casamento em si não é mostrado no código a seguir):

```
1 ResultSet res;
2 Statement selecao <- { System.out.printn("método chamado"); };
3 Position p = ([0];[1];[0]);
4
5 res <- (#codigo, [selecao, p]);
```

Onde na linha 1 é declarada uma variável que representa um conjunto de mapeamentos de variáveis JaTS em valores que será utilizado durante a transformação. As linhas 2 e 3 apresentam atribuições de valores a variáveis que representam um trecho de código e uma posição no código respectivamente. Estes dois elementos irão compor a abstração <*x*, *p*>, repesentada pelo parâmetro `[selecao, p]` da linha 5. Esta mesma linha contém a introdução do mapeamento `#codigo` → `[selecao, p]` no conjunto de mapeamentos.

A próxima fase da transformação está relacionada à geração do código. Para isto, foi definido o *template* da Figura 4.

```
1  public class #C {
2      FieldDeclarationSet:#fds;
3      MethodDeclarationSet:#mds;
4
5      ModifierList:#ml Type:#t #method(ParameterList:#p) {
6          ContextDeclaration:#contexto{ #chamada };
7      }
8
9      void log() {
10         StatementList:#codigo;
11     }
12 }
```

Figura 4. *Template* de Geração para *Extract Method*

É importante notar que a variável `#codigo`, antes imersa no contexto, agora aparece de forma independente na linha 10. Já o contexto passa a conter uma outra variável `#chamada` que representará a chamada do método `log()`. Esta variável é valorada através do seguinte comando JaTS-SL:

```
1 Statement stm <- { log(); };
2 res <- (#chamada, stm);
```

Para a realização da refatoração *Extract Method* de forma genérica, ainda seriam necessárias uma série de análises sobre o código através da linguagem JaTS-AL, cuja sintaxe e semântica fogem ao escopo deste artigo, mas podem ser encontradas em maiores detalhes em [Santos 2006, Santos and Borba 2006]. Aqui, é importante apenas esclarecer ao leitor que a linguagem JaTS-AL é baseada em um paradigma lógico e permite expressar consultas e restrições sobre o código a partir da construção de expressões booleanas que descrevem caminhos sobre a árvore do programa. Tais expressões utilizam meta-variáveis para restrição de escopo e guias de consulta. O fato de existir uma variável que represente o contexto no qual um determinado trecho de código encontra-se inserido permite expressar consultas de forma bem mais concisa e objetiva.

4.2. Extração de Contexto

Um dos principais benefícios da construção `ContextDeclaration` é a possibilidade de tratamento do contexto como "cidadão de primeira classe" durante a transformação. Nesta seção apresentamos um exemplo onde ocorre a extração do contexto para um novo método enquanto o código selecionado é mantido no método original.

Considere a classe da Figura 5, onde é realizada uma operação de cálculo sobre um valor fornecido pelo usuário.

```
 1 public class Calculo {
 2   String valorDeEntrada;
 3   int valorConsolidado;
 4   ...
 5   void resultado() {
 6     if (valorDeEntrada != null) {
 7       valorConsolidado = Integer.parseInt(valorDeEntrada);
 8       int resultado = (valorConsolidado/2)%4;
 9       System.out.print(resultado);
10     }
11   }
12 }
```

Figura 5. Exemplo de Cálculo Simples

Pode-se perceber que o método que calcula e apresenta o resultado da operação contém código não relacionado à logica de negócio. Basicamente, código relativo a verificação de pré-condições, como validade do valor de entrada e conversão de tipo. Uma maneira de tornar o código mais modular seria isolar esta lógica de pré-condições em um método e fazer uso do mecanismo de *assertions* em Java. Mais precisamente, isolar o contexto em que se encontram inseridas as linha 8 e 9 do código da Figura 5. Este

tipo de transformação pode ser descrito de forma bastante simplificada através do uso da declaração de contexto.

Considere como *template* de casamento o mesmo código apresentado na Figura 3. Para adaptá-lo ao problema atual, basta definirmos a seguinte seqüência de comandos JaTS-SL:

```
1 ResultSet res;
2 Statement selecao <- { int resultado = (valorConsolidado/2)%4;
3                        System.out.print(resultado); };
4 Position p = ([0];[1];[1]);
5 res <- (#codigo, [selecao, p]);
6 Name met <- { resultado };
7 res <- (#metodo, met);
```

De fato, o processo de casamento realiza-se de forma similar ao exemplo anterior. O diferencial quando se extraindo contextos fica por conta da etapa de geração. Neste exemplo, vamos criar um método `validacao()` de tipo booleano que garantirá a validade dos dados. Portando nosso *template* de geração terá o aspecto apresentado na Figura 6.

```
1  public class #C {
2      FieldDeclarationSet:#fds;
3      MethodDeclarationSet:#mds;
4
5      ModifierList:#ml Type:#t #method(ParameterList:#p) {
6          assert validacao();
7          StatementList:#codigo;
8      }
9
10     boolean validacao() {
11         ContextDeclaration:#contexto{ return true; };
12         return false;
13     }
14 }
```

Figura 6. *Template* de Geração para Extração de Contexto

A aplicação desta transformação gera o código da Figura 7.

Por concisão, buscamos simplificar a solução aqui apresentada em detrimento da generalidade. No entanto, o *template* de geração poderia ter sido construído de forma mais genérica e parametrizado através de comandos JaTS-SL. Vale destacar também que o tipo explícito declarado na linha 11 do *template* de geração indica que a variável deve ser tratada como contexto. No entanto, esta variável é polimórfica e poderia ser utilizada como uma seqüência de *statements* qualquer, bastanto que o tipo declarado no *template* de geração fosse `StatemetList`.

4.3. Outras Aplicações

O suporte específico ao tratamento de contextos mostrou-se bastante útil também em outras transformações que envolvem extração de trechos de código[1]. Esta é uma técnica

[1]Do inglês: *Slice Extraction*.

```
1  public class Calculo {
2    ...
3    void resultado() {
4      assert validacao();
5      int resultado = (valorConsolidado/2)%4;
6      System.out.print(resultado);
7    }
8    boolean validacao() {
9      if (valorDeEntrada != null) {
10       valorConsolidado = Integer.parseInt(valorDeEntrada);
11       return true;
12     }
13     return false;
14   }
15 }
```

Figura 7. Classe Transformada

bastante utilizada para, por exemplo, pré-processamento de código. Em alguns casos, determinadas características relacionadas a testes e *loging* são removidas do código antes de sua liberação para o cliente.

Outra área de aplicação beneficiada é a construção de linhas de produtos de software. Abordagens extrativas podem ser podem ser adotas com mais eficiência utilizando sistemas de transformação de propósito geral com suporte a extração de trechos de código e manipulação de contextos.

5. Implementação

O suporte ao tratamento de contextos foi implementado como uma extensão do sistema JaTS. Por isso, apresentamos brevemente uma visão geral da arquitetura do sistema antes de expor detalhes de implementação específicos da solução.

JaTS é composto por dois módulos principais: o módulo responsável por atividades de I/O (*parsing* e *pretty-printing*) e o módulo responsável pela execução de transformações. É neste último que foi concentrado nosso trabalho.

O módulo de transformação é composto por três componentes:

1. **Matcher**: responsável por realizar a etapa de casamento entre *templates* e código concreto;
2. **Replacer**: efetua a substituição das variáveis presentes em *templates* de geração por seus respectivos valores mapeados;
3. **Processor**: realiza o processamento de construções que exigem algum tipo de manipulação além da simples substituição de variáveis por valores.

Todos estes módulos delegam para os nós sintáticos o processamento efetivo das informações obedecendo os padrões de projeto [Gamma et al. 1994] *Visitor*, para substituição de variáveis e *Interpreter* para casamento e processamento. Por esta razão, todos os nós em JaTS implementam uma interface comum que define os métodos *accept*, *match* e *process*.

A similaridade da construção ContextDeclaration com o tipo StatementList já

existente, nos permitiu implementá-la como uma derivação deste tipo, mantendo grande parte do comportamento da classe original, como mostra o diagrama da Figura 8.

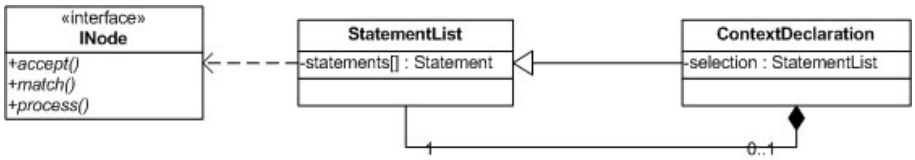

Figura 8. Relacionamentos de ContextDeclaration

Esta arquitetura fez com que toda a implementação já existente passasse a tratar a declaração de contexto como um tipo StatementList, sem necessidade de alterações. Isto foi fundamental para o processo de casamento, pois o contexto de validade de variáveis do tipo ContextDeclaration passou a ser o mesmo de variáveis StatementList.

O processo descrito na Seção 3.2. foi implementado no método *match* da classe ContextDeclaration, que sobrepõe (*overrides*) o método equivalente de StatementList. A extração do código selecionado é delegada a um módulo específico que interpreta elementos do tipo Position percorrendo a AST, realiza a verificação estrutural da representação abstrata do código com a estrutura concreta e extrai o referido código.

Antes da etapa de impressão, estruturas que representam contextos precisam ser recompostas em forma de AST puramente Java, ou seja, o código interno ao contexto precisa ser efetivamente inserido no contexto. Isto é realizado na etapa de processamento pelo método *process*.

6. Trabalhos Relacionados

Não temos conhecimento da existência de uma abordagem similar à apresentada neste artigo para tratamento de contextos. No entanto, existem sistemas de transformação que oferecem mecanismos que podem dar suporte à implementação de transformações envolvendo contextos.

O sistema Stratego [Visser 2004] oferece uma linguagem para definição de regras de reescrita e estratégias de aplicação destas regras. Apesar de não haver uma característica para tratamento direto de contextos de forma declarativa, Stratego oferece a possibilidade de definição de regras dinâmicas, que podem atuar em contextos específicos. Isto permite transformar trechos restritos da AST de acordo com regras particulares, porém seria um processo extremamente complexo em transformações elaboradas como *Extract Method*.

A possibilidade de definir *templates* utilizando a própria sintaxe concreta da linguagem objeto também é suportada por MetaJ [Oliveira et al. 2004], um ambiente extensível para meta-programação baseado em Java. *Templates* em MetaJ possuem alguns recursos interessantes ainda não oferecidos em JaTS, como a possibilidade de casamento todo-parte, que promove a decomposição estrutural de código e associa, ao mesmo tempo, a estrutura completa a uma meta-variável. No entanto, não á uma construção que suporte a representação de estruturas aninhadas no interior de métodos. A manipulação de contextos em MetaJ exigiria o uso intenso de mecanismos de caminhamento em árvores, chamados de Iteradores no ambiente. Na verdade, o usuário teria que decompor e recompor a árvore através de programação imperativa.

Existem também trabalhos relacionados na área de casamento de padrões de código. Um deles é a linguagem PDL (*Pattern Definition Language*) [Albuquerque et al. 2005], utilizada para definição de padrões em uma ferramenta de inspeção de código. Esta linguagem usa uma abordagem diferente da adotada por JaTS e MetaJ, oferecendo ao usuário a possibilidade de definir padrões utilizando código XML, o que torna o processo mais próximo da sintaxe abstrata da linguagem objeto. Esta linguagem mostrou-se uma alternativa poderosa para descrição de padrões em código aninhado e algumas de suas propriedades podem vir a contribuir significativamente em linguagens de transformação.

7. Conclusões e Trabalhos Futuros

Apresentamos neste artigo uma proposta para o tratamento de contextos em linguagens de transformações de programas. Ressaltamos a importância da existência deste tipo de recurso em sistemas de transformação de propósito geral, mostrando sua aplicabilidade em áreas como refatorações e AOP. Pudemos observar que a adoção de tal mecanismo pode trazer um ganho significativo de produtividade na tarefa de especificar transformações envolvendo contextos. Um indício disto é que pudemos, por exemplo, definir a refatoração *Extract Method*[2] em cerca de 140 linhas de código. Em uma linguagem de propósito geral, como Java, a magnitude da implementação de uma refatoração como esta mede-se na dimensão de milhares de linhas de código, como pudemos constatar a partir da análise da refatoração *Extract Method* presente na ferramenta Eclipse [EclipseProject 2005].

A proposta foi implementada como uma extensão do sistema JaTS [Santos and Borba 2006], cuja linguagem passou a agregar uma construção específica para tratamento de contextos. A solução permite a manipulação de contextos como cidadãos de primeira classe através da transformação, o que traz um ganho de expressividade significativo em relação a outras linguagens de transformação atuais, devido a natureza declarativa da abordagem. Avaliamos a proposta através da definição de uma série de transformações, sendo que a refatoração *Extract Method* foi apresentada em maiores detalhes neste artigo.

A implementação atual ainda apresenta algumas limitações que estão sendo dirimidas, entre elas a possibilidade de seleção de código em nível de expressões, já que atualmente apenas *statements* podem compor um contexto e o trecho nele selecionado. Esta restrição não é reflexo da linguagem definida, mas sim, particular à implementação atual. Além disto, em alguns casos, a especificação da posição do trecho interno ao contexto poderia ser opcional quando a estrutura sintática não se repetir internamente ao mesmo. Nestes casos o sistema deveria ser capaz de identificar a posição automaticamente, bastando para tal a integração de um mecanismo de busca por sub-árvore.

Também estamos desenvolvendo pesquisa relacionada a um mecanismo para geração automática de seqüências de índices para posições, o que exigirá a integração com um ambiente visual, onde o usuário possa realizar a seleção explicitamente a partir de um editor.

Entre os trabalhos de pesquisa futuros, pretendemos abordar um requisito que ocorre em algumas aplicações onde o foco da transformação é o contexto. Nestes casos,

[2]Por restrições da implementação atual, a refatoração *Extract Method* não abrange o nível de expressões, o que exigiria uma análise de código mais elaborada e um programa levemente mais extenso.

seria importante que houvesse a possibilidade de que, ao invés de se informar o trecho interno ao contexto, fosse informado o próprio valor do contexto, tornando o trecho interno flexível. Um exemplo de aplicação desta técnica seria a adaptação de código existente para que o mesmo passasse a utilizar o aspecto da Figura 1. Neste caso, o contexto seria previamente conhecido, já que o aspecto já estaria definido, e a variação seria seu trecho interno.

8. Agradecimentos

Gostaríamos de agradecer ao *Software Productivity Group* pelas sugestões ao longo do trabalho, ao CNPq pelo suporte financeiro parcial aos autores e ao revisores anônimos pelas excelentes sugestões que contribuíram bastante para a melhoria deste artigo.

Referências

Albuquerque, E., Garcia, A., and Couto, W. (2005). Uma ferramenta de inspeção de código para a plataforma eclipse. In *IX Brazilian Symposium on Software Engineering*, pages 243–256.

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.

Cole, L. (2005). Deriving refactorings for aspectj. Master's thesis, Federal Universisty of Pernambuco, Recife, PE, Brazil.

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

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

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: elements of reusable object-oriented software*. Addison-Wesley.

Kiczales, G. (1996). Aspect-oriented programming. *ACM Comput. Surv.*, 28(4es):154.

Monteiro, M. P. and Fernandes, J. M. (2005). Towards a catalog of aspect-oriented refactorings. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD 2005)*, pages 111–122, Chicago, USA.

Oliveira, A. A., Braga, T., Maia, M. A., and Bigonha, R. S. (2004). MetaJ: An Extensible Environment for Metaprogramming in Java. 10(7):872–891.

Santos, G. (2006). Suporte a refatorações em sistema de transformação de propósito geral. Master's thesis, Federal Universisty of Pernambuco, Recife, PE, Brazil.

Santos, G. and Borba, P. (2006). Suporte a Refatorações em um Sistema de Transformação de Propósito Geral. In *X Brazilian Symposium on Software Engineering*, pages xx–xx. 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.