Universidade Federal de Pernambuco

Centro de Informática

Implementação de uma Linguagem de Especificação de Design Rules Para Projetos Orientados a Aspectos

Trabalho de Graduação

Por

Roberta Lopes Arcoverde

Recife, Junho de 2008

Universidade Federal de Pernambuco

Centro de Informática

Graduação em Ciência da Computação

Roberta Lopes Arcoverde

Implementação de uma Linguagem de Especificação de Design Rules Para Projetos Orientados a Aspectos

Este trabalho foi apresentado ao Centro de Informática da Universidade Federal de Pernambuco como requisito parcial para a obtenção do Grau de Bacharel em Ciências da Computação.

Orientador: Paulo Henrique Monteiro Borba

Co-Orientador: Sérgio Castelo Branco Soares

"Isto também passará."

Farid Ud Din Attar

Agradecimentos

Durante os anos velozes da graduação, tive o prazer de estudar, trabalhar e conviver com pessoas fantásticas. Agradeço em especial à equipe da Commit, grandes amigos, grandes colegas de projetos, de cadeiras, de noites viradas. Aprendi bastante com todos, e espero que nossa amizade não termine junto com a faculdade. Sei que tenho fama de fuleira, mas prometo tentar ser mais presente nas nossas confraternizações.

Agradeço a meu melhor amigo, Marconi, que desde o primeiro período agüenta meus pitis, ri das minhas besteiras, e briga comigo quando estou errada – mas sem nunca deixar de me defender. Com muito orgulho eu olho pra trás e percebo que fizemos todos os nossos projetos juntos; meu parceiro insubstituível.

Ao final da graduação conheci ainda novos amigos, importantíssimos para minha formação. Ao pessoal do estágio, especialmente da minha equipe – Albérico, Mago, Shirley (in memoriam de estagium), João Paulo, Novaes e meu excelentíssimo gerente Macedo – meus sinceros agradecimentos por terem compartilhado tanto comigo, e por terem tido paciência quando eu me ausentava por causa do TG. Aos meus amigos já graduados, Jeane, Renata, Pedrinho, Turah, Thiaguinho, e àqueles que desertaram, né Ulli e Patrícia?

Ao SPG, com seus excelentíssimos membros, sempre dispostos a ajudar. Em especial a Sérgio, meu digníssimo co-orientador, a Adeline, sempre presente, e a meu orientador, Paulo Borba.

Um agradecimento especialíssimo a Sylvinha. Pelos anos de companheirismo, cumplicidade e por estar sempre ao meu lado, incondicionalmente. Passaram-se quatro anos – uma nova graduação! – e parece que foi ontem. Obrigada pela paciência e que venham muitas novas graduações!

Por fim, a minha lindíssima, maravilhosa, inadjetivável família. Meu pai, por ter me ensinado mais do que ele imagina, sendo o homem mais íntegro, ético e inteligente do mundo, além de alvirrubro! Minha mãe, essa mulher linda, carinhosa, doce, paciente… a pessoa mais bacana que conheço! E claro, a meus irmãos queridos, Renata e Guilherme, que mesmo depois de tantos anos, ainda não se acostumaram com minhas piadas infames e morrem de rir.

Resumo

O uso de programação orientada a aspectos vem sendo proclamado como solução para o problema da modularização de interesses transversais. Entretanto, o uso de tal técnica por vezes prejudica a modularidade, devido às dependências semânticas e sintáticas entre aspectos e classes. Buscando obter modularização de sistemas num contexto geral, surgiu a idéia do uso de *design rules* - regras de projeto que devem ser obedecidas em todas as fases do processo de design e desenvolvimento. Utilizando uma linguagem específica, é possível especificar *design rules* que favoreçam a modularização e o desenvolvimento paralelo de sistemas. Este trabalho propõe a implementação de uma linguagem para especificar *design rules* em projetos orientados a aspectos, definida em um trabalho de doutorado do Centro de Informática.

Palavras-Chave: Design Rules, Programação Orientada a Aspectos, Modularidade

Abstract

The use of aspect-oriented programming has been recognized as a solution to the crosscutting concerns modularization problem. Although crosscutting modularity can be achieved, current aspect-oriented languages do not address class modularity properly, due to semantic dependencies between classes and aspects. In order to obtain both crosscutting and class modularity, design rules for aspect-oriented systems should be defined. Using a specific language to specify design rules enables parallel development of classes and aspects, improving modularity. In this work, we propose an implementation of such language, which is being specified in a PhD thesis at Informatics Center.

Key-Words: Design Rules, Aspect-Oriented Programming, Modularity

Sumário

Lista de Figuras

Lista de Tabelas

Lista de Listagens

1. Introdução

A Programação Orientada a Aspectos[10] (POA) tem sido discutida como uma alternativa elegante e eficaz para implementação de interesses que não são facilmente modularizados em sistemas estruturais ou orientados a objetos. Tais interesses – chamados de transversais – ficam normalmente entrelaçados e espalhados no código-fonte de tais sistemas, o que diminui as possibilidades de reuso e a qualidade do mesmo. Entretanto, estudos recentes[1] mostram que usar POA não necessariamente implica uma maior modularidade, visto que muitas vezes o desenvolvimento do aspecto depende de informações sobre as classes, e vice-versa. Esta dependência prejudica a paralelização do desenvolvimento de sistemas orientados a aspectos.

Buscando obter modularização de sistemas num contexto geral, surgiu a idéia do uso de *design rules[8]* – regras de projeto que devem ser obedecidas em todas as fases do processo de design e desenvolvimento. Uma regra pode ser desde a necessidade de implementação de uma determinada classe até a restrição de chamadas de métodos apenas em um certo escopo, possuindo um fino nível de granularidade.

Objetivando uma linguagem para especificar *design rules* que favoreça a modularização, diminua as dependências e promova o desenvolvimento paralelo de sistemas, Costa Neto[6] definiu a LSD – *Language for Specifying Design Rules*. Para atingir seus propósitos, esta linguagem deve ser poderosa e flexível o suficiente para permitir a definição de regras de acordo com as necessidades do projetista, de forma não ambígua.

Este trabalho define uma implementação da versão atual da LSD capaz de efetuar a verificação das regras em projetos orientados a aspectos.

1.1. Objetivo

O objetivo deste trabalho de graduação é apresentar uma prova de conceito da linguagem de especificação de *design rules* proposta por Costa Neto[6]. Através da mesma, é possível analisar a expressividade da linguagem, validando-a em projetos reais e identificando melhorias.

As principais contribuições deste trabalho são:

- Implementação de um compilador para a LSD
- Validação da LSD, fornecendo uma prova de conceito da linguagem

1.2. Estrutura do Trabalho

Este trabalho é composto por cinco capítulos. No capítulo 2 abordaremos a Orientação a Aspectos, seus principais conceitos e objetivos. Ainda neste capítulo introduzimos o problema da modularização em sistemas orientados a aspectos, e como resolvê-lo utilizando *design rules*.

O capítulo 3 discorre sobre a linguagem implementada neste trabalho, a LSD – *Language for Specifying Design Rules*.

No capítulo 4 descrevemos detalhadamente a implementação de tal linguagem. Citaremos as principais tecnologias utilizadas durante o desenvolvimento, e que outras abordagens poderiam ter sido utilizadas na implementação. Apresentaremos ainda algumas métricas, sugestões de melhorias e possíveis futuras evoluções.

Por fim, no capítulo 5 constam as considerações finais, avaliação dos resultados alcançados e sugestões de trabalhos futuros.

2. Programação Orientada a Aspectos e Design Rules

A Programação Orientada a Aspectos[10] (POA) tem sido discutida como uma alternativa elegante e eficaz para implementação de interesses transversais (*crosscutting concerns*) que não são facilmente modularizados em sistemas estruturais ou orientados a objetos. Estes interesses ficam normalmente entrelaçados e espalhados no código-fonte, o que diminui as possibilidades de reuso e a qualidade do mesmo.

Entretanto, estudos recentes[1] mostram que usar POA, embora promova modularidade dos interesses transversais, por vezes gera dependências que dificultam a paralelização do desenvolvimento, visto que muitas vezes o aspecto depende de informações sobre as classes, e vice-versa.

Neste capítulo, detalharemos a técnica de POA, o problema da modularização supracitado e o uso de *design rules* como possível solução.

2.1. Programação Orientada a Aspectos

A técnica de programação orientada a aspectos surgiu como solução para separar os interesses centrais dos interesses transversais, evitando o entrelaçamento de código com diferentes propósitos e centralizando a implementação de interesses que atingem diferentes módulos do programa. Um exemplo clássico deste tipo de interesse é o uso de *logging*, comumente implementado em diversos módulos e camadas de um sistema. Alterar a tecnologia de implementação de *logging*, por exemplo, implicaria alterar todos os pontos do código que contém instruções relativas a tal interesse.

Com o uso de orientação a aspectos, uma nova unidade de encapsulamento – denominada *aspecto* – é responsável por implementar toda a lógica referente a interesses que não são facilmente modularizados apenas com o uso de orientação a objetos. O código contido nos aspectos é combinado ao código orientado a objetos diretamente no *bytecode*, através de um processo chamado *weaving*.

A implementação mais difundida de POA é a linguagem AspectJ, uma extensão orientada a aspectos da linguagem Java. A especificação da linguagem é bastante extensa e complexa, podendo ser encontrada em [17]. A seguir, resumiremos suas principais construções.

Pointcuts

Os pointcuts representam um conjunto de pontos (*join points*) no fluxo de execução de um programa. Estes pontos podem ser chamadas a métodos, execução de métodos e construtores, execução de um inicializador estático, referências a um determinado atributo, entre outros, e podem ser compostos, utilizando os operadores `||` (or), `&&` (and) e `!` (not).

Advices

Advices especificam código adicional a ser executado e quando ele deve ser executado. Existem três possíveis tipos de *advice*: *before*, *after* e *around*. O código definido nestes *advices* executará antes, depois ou durante a execução do *join point* capturado, respectivamente.

Inter-type Declarations

As *inter-type declarations* representam mudanças estruturais no código do programa, diferentemente do uso de *pointcuts* e *advices*, que modificam a dinâmica de execução. Estas mudanças estruturais podem ser tanto a introdução de novos métodos e atributos a classes ou interfaces existentes, quanto modificações da hierarquia de classes.

2.2. O Problema da Modularidade

Embora o uso de POA permita modularizar a implementação de interesses transversais, as construções utilizadas pelos aspectos para prover tal modularidade em geral precisam conhecer detalhes de implementação das classes e interfaces. Desta forma, alterações na implementação destas classes podem impactar no funcionamento dos aspectos. Além disso, a implementação destas classes não pode ser integralmente compreendida sem que o desenvolvedor tenha conhecimento de todos os aspectos que as interceptam, tornando classes e aspectos semanticamente dependentes.

O exemplo da listagem 1 ilustra o problema supracitado. A chamada ao método responsável por atualizar o *display* está espalhada em dois pontos do sistema, linhas 11 e 23, além de entrelaçada a código para reposicionamento da classe `Point`, como mostra a linha 11. Para melhorar a qualidade do código, separando os interesses e tornando-o mais fácil de manter e entender, deve ser criado um aspecto responsável pela implementação do interesse que envolve o *update* do *display*. A listagem 2 exemplifica tal aspecto.

```
1  interface IShape {
2      public void moveBy(int dx, int dy);
3  }
4
5  class Point implements IShape {
6      private int x, y;
7
8      public void moveBy(int dx, int dy) {
9          setX( x + dx )
10         setY( y + dy );
11         Display.update();
12     }
13 }
14
15 class Line implements IShape {
16     private Point p1, p2;
17
18     public void moveBy(int dx, int dy) {
19         p1.setX( p1.getX()+dx );
20         p1.setY( p1.getY()+dy );
21         p2.setX( p2.getX()+dx );
22         p2.setY( p2.getY()+dy );
23         Display.update();
24     }
25 }
```

Listagem 1 – Exemplo OO do Display Update

```
1  public aspect UpdateSignaling {
2      pointcut change():
3          execution( void IShape.moveBy(int, int) );
4
5      after() returning: change(){
6          Display.update();
7      }
8  }
```

Listagem 2 – Aspecto implementando o interesse do Display Update

Com o uso do aspecto mostrado na Listagem 2, as chamadas ao método `update` podem ser retiradas das classes `Line` e `Point`, melhorando a modularidade deste interesse transversal. Entretanto, os desenvolvedores das classes `Line` e `Point` precisam estar cientes de que existe um aspecto tratando a atualização do display, para que não insiram novamente a chamada ao método `update` dentro dos métodos `moveBy`. Esta situação gera então novos problemas de modularidade; a classe não pode mais ser totalmente compreendida sem que se considere o aspecto, e o desenvolvimento paralelo dos dois módulos é comprometido.

Isto mostra que, embora exista um nível de modularidade *crosscutting*, não se consegue facilmente paralelizar a implementação dos requisitos OO e dos requisitos AO, devido às dependências semânticas supracitadas. Este problema pode ser entretanto mitigado com o uso de regras bem definidas entre classes e aspectos, que devem ser rigorosamente obedecidas. Estas regras de design, ou *design rules*, definem interfaces entre módulos de um projeto que são menos suscetíveis a alterações, diminuindo o acoplamento entre eles, assim como interfaces diminuem o acoplamento entre componentes de software.

2.3. Design Rules

Buscando obter melhores níveis de modularidade de uma forma geral, o uso de *design rules* – regras de projeto que devem ser obedecidas em todas as fases do processo de design e desenvolvimento – foi proposto por Baldwin e Clark[8]. Estas *design rules* no processo de desenvolvimento de software correspondem às interfaces entre os módulos que são menos suscetíveis a alterações. Como exemplos de *design rules,* podemos citar as regras de comunicação entre componentes do sistema, padrões de nomenclatura, a necessidade de implementação de uma determinada classe, entre outras.

Design rules estabelecem, no contexto de orientação a aspectos, os requisitos mínimos necessários para o desenvolvimento paralelo de componentes que implementam os requisitos principais e dos que implementam interesses transversais. A especificação clara destes requisitos evita que o desenvolvedor OO escreva código que já será tratado pelos aspectos.

Utilizando uma linguagem específica, a implementação de tais *design rules* favorece a modularização e o desenvolvimento paralelo de sistemas, diminuindo as dependências sintáticas e semânticas entre classes e aspectos. Para tanto, esta linguagem deve ser poderosa e flexível o suficiente para permitir a definição de regras de acordo com as necessidades do projetista e que o desenvolvimento de módulos orientados a objetos seja simultâneo ao de módulos orientados a aspectos. Buscando tais resultados, Costa Neto[1, 6] definiu a linguagem LSD – *Language for Specifying Design Rules* – para especificação de *design rules* em sistemas orientados a aspectos, cuja implementação será o foco deste trabalho.

O próximo capítulo detalhará a estrutura da LSD.

3. Language for Specifying Design Rules - LSD

O desenvolvimento da LSD foi inicialmente motivado pela possibilidade de melhoria da modularidade de sistemas orientados a aspectos. Assim, a linguagem permite especificar os requisitos mínimos para promover o desenvolvimento paralelo de classes e aspectos, reduzindo o acoplamento entre os mesmos e melhorando a modularidade[1]. É possível especificar as *design rules* de forma não ambígua e verificar no código desenvolvido se elas foram cumpridas pelos desenvolvedores. A LSD ainda serve de guia para o desenvolvimento do sistema desde as fases iniciais.

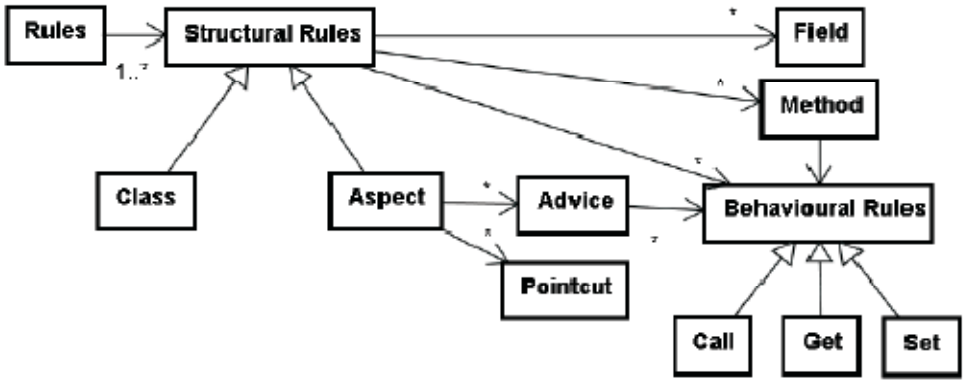

Figura 1 – Meta-modelo da LSD

A figura 1 mostra o meta-modelo da linguagem. Uma regra é composta por uma ou mais regras estruturais, podendo representar classes ou aspectos. Por sua vez, estas classes e aspectos podem definir atributos, métodos, *pointcuts* e *advices*, além de regras comportamentais. Estas regras podem estar presentes tanto em declarações de classes e aspectos quanto dentro do escopo de métodos e *advices*.

3.1. Regras Estruturais

As regras estruturais definem componentes e membros destes componentes que devem ser implementados, similarmente às interfaces em Java. Entretanto, regras estruturais aplicam-se não só a implementação de métodos, mas também a atributos que devem ser declarados, *pointcuts* e *advices*. Assim, uma *design rule* pode definir a estrutura dos componentes que devem estar presentes no projeto com um bom nível de detalhamento.

Na listagem 3, por exemplo, temos a definição da *design rule* `DRUpdateShape`, que especifica uma série de regras estruturais. Na linha 1 temos a declaração da *design rule* em si, que agrupa um conjunto de regras estruturais que devem ser seguidas por classes, interfaces e aspectos. A primeira destas regras, na linha 2, trata da implementação de uma interface, denominada `IShape`, que requer uma declaração de método com a assinatura especificada na linha 3. Deve haver ainda uma classe `Shape` que implemente a interface `IShape` (linha 6) e seu respectivo método `moveBy`, e uma classe `Display` com um único método (linhas 13 a 15). Por fim, a *design rule* `DRUpdateShape` especifica que deve haver um aspecto que declare o *pointcut change* mostrado na linha 18, e um *advice* que opere sobre tal *pointcut* (linha 20).

```
 1 dr DRUpdateShape {
 2     interface IShape {
 3         void moveBy(int dx, int dy);
 4     }
 5
 6     class Shape implements IShape {
 7         void moveBy(int dx, int dy) {
 8             xset(int Shape.dx);
 9             xset(int Shape.dy);
10         }
11     }
12
13     class Display {
14         void update();
15     }
16
17     aspect UpdateSignaling {
18         pointcut change() : execution(void IShape.moveBy(int, int));
19
20         after() returning : change() {
21             xcall(void Display.update());
22         }
23     }
24 }
```

Listagem 3 – Exemplo de design rule para o problema do Display Update

Este exemplo mostra como desenvolver uma *design rule* capaz de solucionar os problemas identificados com o exemplo do *Display Update* na seção 2.2. Como os desenvolvedores têm agora acesso a todos os requisitos que devem ser seguidos na implementação, torna-se possível e simples implantar desenvolvimento paralelo entre as classes e aspectos.

3.2. Regras Comportamentais

Estas regras especificam uma determinada ação ou comportamento que ocorre dentro de um escopo específico. Atualmente, a LSD suporta seis regras comportamentais, representando as ações de chamadas de métodos (call e xcall), atribuições a variáveis (set e xset) e recuperação de variáveis (get e xget). A linguagem dá suporte ainda à composição destas regras, através dos operadores de conjunção, disjunção e negação.

Quando uma regra comportamental é precedida por um ‘x’, caso de xcall por exemplo, o comportamento por ela definido deve se restringir ao escopo onde a regra foi declarada; isto é, não deve ocorrer em nenhum outro escopo dentre os componentes especificados na *design rule*. Por exemplo, a regra `xcall(void Display.update())` na listagem 1 indica que o método `update` da classe `Display` deve ser chamado *apenas* dentro do *advice* definido na linha 20. Caso haja mais de uma ocorrência da mesma regra nos componentes definidos pela *design rule*, a verificação deve considerar todos estes escopos. Isto é, no caso de *xcall,* por exemplo, a chamada de método deve ser feita apenas nos escopos onde a regra foi definida.

Na listagem 3 temos dois exemplos de regras comportamentais: nas linha 8 e 9, as regras xset indicam que os campos *dx* e *dy* da classe `Shape` devem ser alterados exclusivamente dentro do método `moveBy` da classe `Shape`. Já a regra xcall na linha 21, conforme anteriormente explicado, especifica que o método `update` da classe `Display` deve ser chamado exclusivamente dentro deste *advice*. Além destes exemplos, pode-se também inserir uma regra comportamental diretamente no corpo da classe, fora de métodos ou *advices*. Neste caso, o comportamento por ela definido deve se realizar em qualquer ponto da classe.

A tabela 1 resume as regras comportamentais atualmente providas pela linguagem.

Regra	Descrição
call(método)	Deve haver uma chamada ao método passado como parâmetro dentro do escopo onde a regra foi definida.
xcall(método)	Deve haver uma chamada ao método passado como parâmetro exclusivamente dentro do escopo onde a regra foi definida.
set(atributo)	Deve haver uma mudança de estado do atributo passado como parâmetro dentro do escopo onde a regra foi definida.
xset(atributo)	Deve haver uma mudança de estado do atributo passado como parâmetro exclusivamente dentro do escopo onde a regra foi definida.
get(atributo)	Deve haver uma leitura do atributo passado como parâmetro dentro do escopo onde a regra foi definida.
xget(atributo)	Deve haver uma leitura do atributo passado como parâmetro exclusivamente dentro do escopo onde a regra foi definida.

Tabela 1 – Regras comportamentais providas pela LSD

Módulo de Configuração e Parametrização

Considerando que vários componentes do sistema podem implementar uma mesma *design rule*, faz-se necessária a definição de um mapeamento que indique quais componentes implementam quais regras.

O uso de módulos de configuração é atualmente uma das opções no projeto da linguagem que permitem a realização deste mapeamento. Tais módulos possuem uma linguagem própria, onde são identificados quais componentes implementam quais regras de uma *design rule*. A listagem 4 exemplifica o uso de um módulo de configuração.

```
1
2 module ModuleUpdate implements DRUpdateShape {
3     IShape display.IShape;
4     Display display.Display;
5     UpdateSignaling display.UpdateSignaling;
6 }
```

Listagem 4 – Módulo de configuração

A linha 2 declara que a *design rule* `DRUpdateShape` será configurada no módulo `ModuleUpdate`. As linhas 3, 4 e 5 definem as classes e aspectos que implementam então as regras estruturais encontradas nesta *design rule* – no caso, `IShape`, `Display` e `UpdateSignaling`, onde o primeiro identificador refere-se ao nome da regra e o segundo ao nome qualificado do componente que a implementa.

```
1  interface IShape implements DRUpdateShape {
2      // original code
3  }
4
5  class Display implements DRUpdateShape {
6      // original code
7  }
8
9  aspect UpdateSignaling implements DRUpdateShape {
10     // original code within the pointcut description
11 }
12
13 class Line implements DRUpdateShape(Shape) {
14     // original code
15 }
16
17 class Point implements DRUpdateShape(Shape) {
18     // original code
19 }
```

Listagem 5 – Componentes e suas implementações das regras

Além da opção do uso de um módulo de configuração à parte, há ainda a possibilidade de se indicar a regra no próprio componente. A listagem 5 demonstra como este mapeamento pode ser feito. A linha 1 declara uma interface de nome `IShape` que implementa a *design rule* `DRUpdateShape`. A regra estrutural implementada é inferida pelo nome da interface, que coincide com o nome de uma regra em `DRUpdateShape`. De forma análoga, as linhas 5 e 9 declaram uma classe e um aspecto que implementam as regras `Display` e `UpdateSignaling`.

Já a linha 13 declara uma classe `Line`, cujo nome não possui nenhuma regra estrutural correspondente definida em `DRUpdateShape`. Neste caso, é necessário indicar qual regra esta classe está implementando. Isso é feito passando-se o nome da regra entre parênteses após o nome

da *design rule*, indicando neste caso que a regra `Shape` está sendo implementada. A classe definida na linha 17 demonstra como dois componentes distintos podem implementar uma mesma regra, no caso, a regra `Shape`.

Outra importante característica é a possibilidade de parametrizar as *design rules*. A parametrização é importante para prover flexibilidade na especificação de regras e promover o reuso. A listagem 6 exibe uma especificação parcial da *design rule* `DRUpdateShape`, em que o método `update`, na linha 7, é parametrizado. Assim, permite-se uma maior flexibilidade na assinatura do método que implementa a lógica de `update` do `Display`. O valor concreto do parâmetro é especificado no módulo de configuração.

```
1  dr DRUpdateShape{
2      aspect UpdateSignaling {
3          pointcut change():
4              execution( void IShape.moveBy(int, int) );
5
6          after() returning: change(){
7              xcall( <update_method> );
8          }
9      }
10     ...
11
12     class Display {
13         <update_method>;
14     }
15 }
```

Listagem 6 – Exemplo de design rule utilizando parametrização

A listagem 7 mostra o módulo de configuração da *design rule* `DRUpdateShape` parametrizada. O uso da palavra reservada *where* permite que se defina o valor do parâmetro <update_method>, conforme mostrado nas linhas 3 e 4.

```
1  module ModuleUpdate implements DRUpdateShape {
2      IShape display.IShape;
3      Display display.Display where
4          update_method: void update();
5      UpdateSignaling display.UpdateSignaling;
6  }
```

Listagem 7 – Módulo de configuração com parametrização

Buscando validar a LSD, este trabalho propõe uma implementação da linguagem, que será detalhada no próximo capítulo.

4. Implementação

Esta seção detalha a implementação da LSD realizada neste trabalho. Tal implementação resultou em um compilador da linguagem cujo código gerado verifica se as regras definidas estão corretamente implementadas por um conjunto de classes, aspectos e interfaces.

Como se trata de uma linguagem ainda em processo de especificação, sua implementação precisa ser extensível e modular, permitindo que mudanças na linguagem sejam facilmente incorporadas. Entretanto, embora o estudo e implementação de compiladores seja uma área já bastante desenvolvida, o suporte a modificações e extensões de linguagens é um problema não-trivial e difícil de ser modularizado. Idealmente, tais extensões deveriam ser auto-contidas, possibilitando a incorporação das mesmas na linguagem de forma simples. Com o propósito de tornar modulares extensões e modificações da LSD, a ferramenta JastAdd[18] foi utilizada neste trabalho.

O JastAdd provê um mecanismo orientado a aspectos estático, onde apenas são permitidos declarações inter-type e interceptação de métodos. Entretanto, outros mecanismos de promoção de modularidade são suportados, tais como gramáticas de atributos. O uso de tais técnicas ajuda a encapsular extensões nos compiladores gerados pelo JastAdd de forma simples e auto-contida[18].

A verificação das regras envolve análises no código Java/AspectJ, dependendo portanto de um compilador de tais linguagens. Nesta implementação, utilizamos o AspectBench Compiler[12] (abc) para realizar o *parsing* do código Java/AspectJ. Como o *frontend* do abc foi recentemente implementado com o JastAdd[3], quaisquer extensões necessárias para a verificação das regras poderiam ser mais facilmente incorporadas, valendo-se das vantagens supracitadas do uso do JastAdd para extensão de linguagens e compiladores.

Na próxima seção abordaremos as ferramentas utilizadas na construção do compilador LSD.

4.1. Ferramentas Utilizadas

Beaver

O Beaver[16] é um gerador de *parsers* LALR, utilizado na implementação do *frontend* do abc com o JastAdd. Embora o JastAdd se encarregue de criar as classes que representam os nós de uma linguagem, qualquer gerador de *parsers* pode ser utilizado, como o JavaCC.

A implementação final da LSD utiliza o Beaver para gerar o *parser* da linguagem, mas uma implementação experimental com o JavaCC também foi desenvolvida.

JastAdd

O JastAdd é um framework *open source* cujo principal objetivo é facilitar a extensão e implementação de compiladores e ferramentas relacionadas (como analisadores de código e ferramentas de transformação). Para tanto, o JastAdd utiliza noções de orientação a aspectos e gramática de atributos como mecanismos de modularização.

O JastAdd possui uma DSL própria para definição de ASTs, permitindo a geração automática das classes que representam os nós da árvore de uma linguagem, a partir de sua BNF. A gramática é especificada em arquivos .ast, e os nós gerados correspondem a uma hierarquia de classes Java. O quadro a seguir explica brevemente a sintaxe desta DSL e as classes geradas a partir dela.

Construção	Descrição	Classe gerada
`abstract A;`	A é uma classe abstrata e corresponde a um nó não-terminal	`abstract class A extends ASTNode { }`
`B : A;`	B é uma subclasse concreta de A e corresponde a uma produção de A	`class B extends A { }`
`C : A ::= D;`	C tem um filho do tipo D	`class C extends A { D getD(); }`
`E : A ::= [C];`	E tem um filho opcional do tipo C	`class E extends A { boolean hasC(); C getC(); }`
`F : A ::= E*;`	F tem uma lista de zero ou mais filhos do tipo E	`class F extends A { int getNumE(); E getE(int); }`

Tabela 2 – Funcionamento da DSL do JastAdd para geração de nós

Por exemplo, o código abaixo demonstra como especificar os nós que representam uma declaração de classe Java, utilizada na implementação da LSD:

```
1 abstract TypeDecl;
2 ClassDecl : TypeDecl ::= Modifiers <ID:String>
3   [SuperClassAccess:Access] Implements:Access* BodyDecl*;
```

A linha 1 mostra a declaração de um nó de nome `TypeDecl` – representando uma declaração de tipo. O modificador `abstract` intuitivamente indica que a classe que será gerada para representá-lo será abstrata. O nó `ClassDecl` representa uma declaração de classe, e herda do nó `TypeDecl` previamente definido. O operador `::=` indica os filhos deste nó. `Modifiers` representa um nó que guarda os modificadores da classe. Já a construção `<ID:String>` define um campo do tipo String no nó `ClassDecl`, cujo nome é `ID`. Este tipo de construção, entre < e >, não representa novos nós da AST, e sim *tokens* no processo de parsing. Na linha 3, temos uma declaração do tipo `Access`, de nome `SuperClassAccess`, representando a classe estendida pelo nó `ClassDecl`. Os colchetes indicam que é um nó opcional. Por fim, temos a definição das interfaces que a classe implementa, que são zero ou mais nós do tipo `Access`. O corpo da classe é definido pelo conjunto de nós do tipo `BodyDecl`.

Todos os nós da árvore da LSD foram implementados utilizando a DSL supracitada do JastAdd. Desta forma, caso seja necessário estender a linguagem, basta modificar as declarações de acordo com a sintaxe mostrada na tabela 2 e gerar o código novamente.

Gramática de Atributos

Além de prover a DSL para geração automática de nós das ASTs, o JastAdd oferece mecanismos para inserir *atributos* nestes nós. Atributos são informações que não fazem parte da estrutura da linguagem, mas estão contidas nos nós para facilitar análises e coletas de informações das ASTs.

O conceito de gramáticas de atributos foi primeiramente descrito por Knuth[5], e baseava-se no problema de como definir algoritmicamente o significado – ou semântica – de um programa. Atributos ‘decoram’ gramáticas livres de contexto, especificando esta semântica.

Os atributos gerados pelo JastAdd podem referenciar objetos, permitindo por exemplo que o acesso a uma variável tenha uma ligação direta com sua declaração, sem precisar para tanto alterar a estrutura da linguagem. Este é um exemplo clássico de atributo, que permite verificar facilmente se uma variável foi declarada antes de seu uso. Atributos podem também ser parametrizáveis, e seus valores são sempre calculados de acordo com a demanda (*demand-driven evaluation*), tornando eficiente a utilização e processamento dos mesmos.

Knuth divide atributos em dois tipos, de acordo com o nível da AST onde seus valores são calculados: atributos *synthesized,* cujos valores são calculados nos nós filhos e repassados aos nós superiores (*bottom-up*), e atributos *inherited*, cujos valores são calculados nos nós superiores e repassados aos nós filhos (*top-down*).

Como exemplo de utilização de atributos na implementação da LSD, foram utilizados atributos *inherited* para indicar se uma dada regra faz ou não parte de uma expressão. A Listagem 8 detalha esta implementação.

```
1     inh boolean RuleExpr.isComposed() ;
2
3     eq BinaryRuleExpr.getLhs().isComposed() = true;
4     eq BinaryRuleExpr.getRhs().isComposed() = true;
5     eq NegRuleExpr.getRuleExpr().isComposed() = true;
6     eq BehaviouralRuleDecl.getRuleExpr().isComposed() = false;
```

Listagem 8 – Implementação de atributo *inherited* na LSD

A linha 1 declara o atributo *isComposed* de tipo *boolean* no nó *RuleExpr*. Este nó representa uma declaração de regra comportamental no LSD. A palavra reservada *inh* indica que se trata de um atributo *inherited*, e seu valor será sempre calculado nos nós pais de *RuleExpr.* Assim, nas linhas de 3 a 6, os valores do atributo são definidos para cada nó que possui como filho um nó do tipo *RuleExpr*. Caso o nó pai seja do tipo BinaryRuleExpr ou NegRuleExpr, o atributo assume o valor *true*; em qualquer outro caso, seu valor será *false*. Assim, o valor deste atributo para regras do tipo A && B, !A ou A || B será sempre *true*.

abc

O abc é um compilador AspectJ open-source desenvolvido com o propósito de facilitar pesquisa e desenvolvimento em linguagens orientadas a aspectos.

Na implementação da LSD, o código gerado para verificar as regras precisa analisar ASTs Java/AspectJ, buscando uma correta implementação das mesmas. Para tanto, é necessário que tais ASTs sejam bem formadas, completas e fáceis de analisar e estender. O abc foi então a escolha mais apropriada para efetuar o *parsing* do programa AspectJ, inclusive por utilizar em sua implementação recente um *frontend* implementado com o JastAdd, facilitando sua extensão e integração com o compilador LSD.

4.2. Arquitetura

De forma a permitir futuras extensões e um maior nível de modularidade, a implementação do compilador LSD utilizou os seguintes componentes:

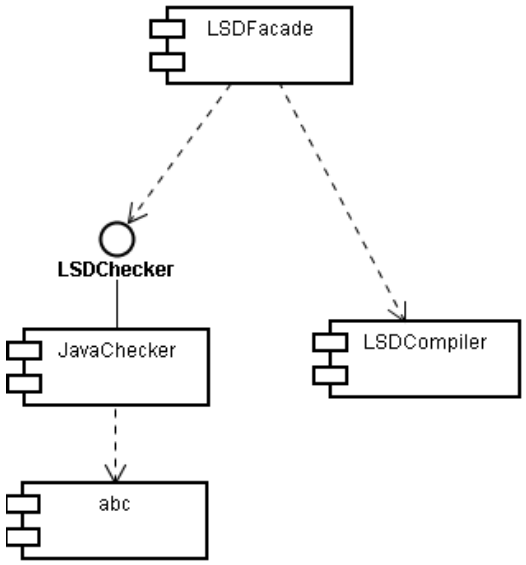

Figura 2 – Componentes da implementação da LSD

LSDFacade – representa o ponto de entrada único do verificador, provendo acesso a serviços para verificar se um conjunto de regras está sendo seguido por um conjunto de componentes.

LSDChecker – interface que define os serviços que devem ser providos por um verificador da LSD.

JavaChecker – verificador que implementa LSDChecker gerando código Java para fazer a verificação das regras. O código gerado é inserido diretamente nos nós da LSD, de forma que cada nó que representa uma regra sabe como ser verificado diante de um programa Java/AspectJ. Essa composição é feita automaticamente pelo mecanismo de orientação a aspectos e gramática de atributos do JastAdd. Desta forma, todo o código responsável por realizar as verificações das regras fica encapsulado em um módulo único.

LSDCompiler – responsável pela verificação sintática e contextual de arquivos contendo *design rules.*O LSDCompiler é responsável pela manipulação de arquivos, convertendo código-fonte em ASTs. Estas ASTs são utilizadas pelo LSDChecker para a verificação das regras.

abc – *frontend* do abc, utilizado pelo JavaChecker para efetuar o *parsing* e a análise dos componentes que implementam as regras.

4.2.1. Módulos do JastAdd

Além dos componentes supracitados, os seguintes módulos foram implementados, utilizando o mecanismo de modularização do JastAdd. Este mecanismo permite centralizar a implementação de interesses, de forma similar ao uso de *inter-type declarations* em AspectJ.

- DRChecker
    - Responsável por toda a lógica para verificação das regras estruturais.
- DRBehaviouralChecker
    - Responsável por toda a lógica para verificação das regras comportamentais. Neste módulo, todos os nós que representam uma regra comportamental possuem uma declaração de método *check*, que recebe o escopo a ser verificado e retorna se a regra está sendo seguida. Na seção 4.3.2 serão detalhadas as verificações das regras implementadas neste módulo.
- PrettyPrinter
    - Converte uma árvore LSD em código-fonte; utilizado para testar a corretude na construção das árvores criadas pelo *parser*.
- DRMatching
    - Implementa a lógica responsável por casar as estruturas definidas na LSD com as estruturas nos componentes que implementam as regras.
- XRulesLookup
    - Possui funções que auxiliam a verificação das regras precedidas por 'x'. Este módulo utiliza atributos para decorar os nós que representam escopos. Por exemplo, o atributo `verifyMethodCallScope` definido neste módulo indica se uma chamada de método passada como parâmetro ocorre dentro de um determinado nó. Este atributo é utilizado na verificação das regras call e xcall.
- ErrorCheck
    - Este módulo encapsula código para tratamento dos erros nas verificações.
- DRLanguage
    - Este módulo funciona como um módulo utilitário, reunindo funções e atributos com propósitos gerais. Alguns exemplos são métodos auxiliares para os mapeamentos entre regras e componentes e definição de atributos que armazenam o escopo de uma dada regra comportamental.

4.3. Verificação das Regras

O código gerado pelo compilador desenvolvido neste trabalho tem como propósito verificar se um conjunto de regras é corretamente implementado por um conjunto de classes e aspectos. Cada verificação envolve uma série de análises nas ASTs destas classes e aspectos, de forma que quanto mais eficientes forem estas análises, mais eficazes são as verificações.

Neste trabalho, o código gerado pelo compilador para verificar as regras foi implementado como código Java, valendo-se das características da gramática de atributos do JastAdd. Desta forma, cada nó que representa uma regra sabe como ser verificado diante de uma AST Java/AspectJ.

Para construir as ASTs das classes e aspectos, foi utilizado o *frontend* do ABC, recentemente implementado utilizando o JastAdd. Além de construir as ASTs, o frontend do ABC efetua análises semânticas e contextuais, garantindo assim que para implementar uma regra o programa deve estar bem-formado.

4.3.1 Regras Estruturais

Toda a lógica da verificação das regras estruturais foi encapsulada em um único módulo, o DRChecker. Isso foi possível graças ao mecanismo do JastAdd que utiliza orientação a aspectos para inserir esta lógica em cada nó no momento em que estes são gerados.

A verificação das regras estruturais envolve verificar a presença das estruturas representadas pelas regras no programa que está sendo analisado. Desta forma, o mais importante é definir como casar uma estrutura declarada numa *design rule* com as encontradas no programa, para identificar se a regra foi ou não cumprida.

É importante deixar claro o que é ou não considerado um *matching* entre uma regra estrutural e uma estrutura encontrada no programa sendo analisado. Explicaremos a seguir como este *matching* é realizado para cada regra.

Declaração de Classe, Interface e Aspecto

Uma design rule define quais os componentes mínimos que devem estar presentes para o desenvolvimento de um projeto. Estes componentes podem ser classes, interfaces ou aspectos, e podem ter membros e comportamentos especificados na *design rule*. Para identificar se uma declaração de classe, aspecto ou interface definida em uma *design rule* está sendo cumprida pelo projeto, as seguintes verificações são feitas:

- Os modificadores do componente não são considerados;
- Caso haja implementação de interfaces na regra, as mesmas devem também ser implementadas pelo componente;
- Caso haja uma declaração de superclasse na regra, a mesma deve também existir no componente;

O nome do componente não é considerado, pois toda verificação depende de um mapeamento entre componentes e regras que eles implementam. Assim, podemos ter uma classe `Line` que implementa a regra `Shape`, por exemplo.

Declaração de Método

Similarmente às interfaces de Java, a regra estrutural de declaração de método especifica um método que deve ser implementado. E, também similarmente às interfaces, a verificação da regra baseia-se na assinatura do método para definir se ele foi corretamente implementado ou não. Assim, as seguintes premissas são consideradas para que a verificação seja bem-sucedida:

- Os modificadores do método não são considerados;
- O tipo de retorno deve ser o mesmo definido na regra;
- A quantidade, tipo e ordem de parâmetros devem ser os mesmos definidos na regra;
- As exceções, caso especificadas, devem ser as mesmas definidas na regra;

Caso o componente que deve implementar esta regra contenha um método que a obedeça indiretamente, por meio de uma superclasse, por exemplo, considera-se que a regra está implementada normalmente.

Dentro do corpo de uma declaração de método na LSD pode haver apenas especificações de regras comportamentais, e nenhum outro tipo de *statement*. Assim, além de definir a estrutura, a regra de declaração de método pode também definir o comportamento do método. A verificação deste comportamento está explicada na seção 4.3.2.

Declaração de Atributo

Esta regra estrutural indica que um determinado atributo deve estar presente no componente que implementa a *design rule*. Assim, verificar se a regra está sendo seguida consiste em certificar a presença de tal atributo na classe ou aspecto. Para tanto, as seguintes premissas devem ser verificadas:

- Os modificadores do atributo não são considerados;
- O nome do atributo deve ser o mesmo definido na regra;
- O tipo do atributo deve ser o mesmo definido na regra;
- A verificação ignora se o atributo está ou não sendo inicializado;

Declaração de Pointcut

Para ser seguida, uma regra estrutural de declaração de pointcut deve respeitar o nome do pointcut e os *join points* que este intercepta. Para tanto, todos os designadores de *pointcuts* presentes na definição da regra devem estar também presentes no aspecto que a implementa. O uso de *wildcards* (estrela e subtipo) deve ser respeitado, visto que altera de forma significativa quais *join points* são interceptados.

- Os modificadores não são considerados;
- O nome do pointcut deve ser o mesmo definido na regra;
- Os join points interceptados devem ser os mesmos;

Declaração de Advice

A verificação desta regra possui algumas peculiaridades: como advices são construções que não possuem um identificador, precisa-se comparar o tipo do advice e os *join points* que ele intercepta. Desta forma, as seguintes premissas devem ser consideradas:

- O tipo de advice deve ser o mesmo definido na regra;
    - No caso de advices around, o tipo de retorno deve ser o mesmo
- O tipo e ordem dos parâmetros do advice devem ser os mesmos;
- O pointcut deve interceptar exatamente os mesmos join points definidos na regra, de acordo com a lógica exposta no item *Declaração de Pointcut.*

4.3.2 Regras Comportamentais

A verificação das regras comportamentais foi também encapsulada em um único módulo, DRBehaviouralChecker. No entanto, algumas extensões ao Frontend do abc foram necessárias para tornar a implementação mais eficaz e elegante. Estas extensões são especialmente úteis tanto como prova de conceito – utilizando fortemente a noção de atributos – quanto para simplificar o código que implementa a verificação das regras.

Call

Para implementar a regra call, foram necessárias duas implementações: uma para tratar da verificação da regra em si, e outra auxiliar utilizando o mecanismo de gramática de atributos para colher informações sobre chamadas de métodos em ASTs Java e AspectJ. Para melhor explicarmos o funcionamento destas implementações, dividiremos esta seção em duas partes: a implementação da regra, e a extensão do frontend do abc.

- Extensão do abc: para auxiliar a implementação da regra, o frontend do abc foi estendido, com a inclusão de um novo atributo – o methodCalls. Este é um atributo *synthesized* – ou seja, calculado nos nós filhos e passado para o nó pai. O valor deste atributo em um nó é uma lista contendo todas as chamadas feitas a métodos cujo nome é igual ao passado como parâmetro, e que são feitas no escopo representado por este nó. Por exemplo, o atributo num nó representando o corpo de um método passando como parâmetro “update” irá guardar todos os acessos ao método “update” feitos neste corpo de método. O mesmo atributo num nó representando uma classe irá guardar todas as chamadas ao método “update” dentro desta classe.

- Implementação da regra: a verificação de uma regra call é dependente do escopo onde a mesma está inserida. A extensão ao *frontend* do abc citada anteriormente adiciona o atributo methodCalls em qualquer nó que possa representar um escopo; desta forma, recuperam-se todas as chamadas a um método consultando-se apenas este atributo no escopo em que a regra está inserida. Caso haja uma ou mais chamadas ao referido método, considera-se que a regra está implementada; caso contrário, levanta-se um erro de não-implementação da regra.

Para que um método seja considerado equivalente ao método definido na regra "call", consideram-se as seguintes premissas:

- Os modificadores de acesso não são comparados;
- O tipo de retorno deve ser exatamente o mesmo definido na regra (subtipos não são considerados);
- O nome do método deve ser exatamente o mesmo;
- Os métodos devem receber os mesmos parâmetros, incluindo o tipo e a ordem em que foram definidos na regra;

Get

A implementação da regra get, similarmente à da regra call, incluiu uma extensão ao frontend do ABC, utilizando o mecanismo de gramática de atributos. Explicaremos a seguir esta extensão e como ela é usada na verificação da regra.

- Extensão do abc: com esta extensão, os nós da AST ficam decorados com um novo atributo, *fieldGetters*, que contém informações sobre todos os acessos a um determinado campo feitos nos escopos representados por estes nós. Este atributo, definido em nós que representam declarações de tipos, escopos ou *statements*, guarda todos os acessos feitos a um determinado campo cujo nome é passado como parâmetro. Desta forma, caso o nó seja um s*tatement* de qualquer outro tipo que não acesso a variável, o valor do atributo é uma lista vazia; caso o nó represente um escopo, o valor do atributo é uma lista contendo todos os acessos ao campo indicado realizados naquele escopo.
- Implementação da regra: assim como ocorre em todas as regras comportamentais, a verificação depende do escopo onde a regra está definida. Caso se trate de um corpo de método ou advice, basta recuperar o valor do atributo fieldGetters no nó da árvore que representa este corpo. Se o corpo contiver algum acesso ao campo definido na regra, o valor deste atributo será uma lista não-vazia. A verificação precisa então certificar-se de que os acessos encontrados nesta lista correspondem ao campo definido na regra get – isto é, pertencem a um objeto do tipo correto. Este pós-processamento é necessário pois o atributo fieldGetters guarda todos os acessos a

campos com o nome passado como parâmetro, mas não garante que estes pertençam ao tipo definido na regra. O exemplo a seguir ilustra este comportamento, mostrando os valores do atributo *fieldGetters* em um nó que representa uma declaração de classe.

Set

A verificação da regra set envolve verificar se um dado campo teve seu estado alterado dentro do escopo em que a regra foi definida. A utilização do mecanismo de gramática de atributos neste caso também envolveu uma extensão ao *frontend* do abc, incluindo um novo atributo.

- Extensão do abc: foi criado um novo atributo – fieldSetters – em todos os nós que representam um escopo ou um *statement*. O valor deste atributo é uma lista que contém todas as alterações de estado de um dado campo passado como parâmetro realizadas dentro do escopo daquele nó. Desta forma, caso o nó seja um s*tatement* de qualquer outro tipo que não modificação de variável, o valor do atributo é uma lista vazia; caso o nó represente um escopo, o valor do atributo é uma lista contendo todas as atribuições realizadas no campo dentro deste escopo.
- Implementação da regra: assim como no caso da regra get, a verificação da regra set depende da análise do valor do atributo *fieldSetters* no nó que representa o escopo onde a regra foi definida. Caso o valor deste atributo seja uma lista não-vazia, todos os seus elementos precisam ser verificados pra identificar se são compatíveis com o campo definido na regra.

XCall

A verificação da regra xcall assemelha-se à da regra call; no entanto, alguns comportamentos a mais devem ser também verificados. O que diferencia as duas regras é que xcall exige que a chamada ao método seja exclusiva dentro do escopo em que ela está inserida; ou seja, em nenhum outro escopo definido dentre os componentes que implementam a *design rule* poderá haver outras chamadas a este método.

Esta verificação é um tanto mais complexa, pois precisa acessar as ASTs de todas as classes e aspectos que implementam os outros módulos definidos na *design rule*, e certificar que nenhuma delas contenha uma chamada ao método indicado na regra – a não ser que seu escopo também contenha uma regra xcall idêntica.

O seguinte pseudo-código descreve o comportamento implementado por esta regra, para um dado método *m*.

```
para cada classe/aspecto

      se o método m é chamado num contexto c
```

```
            se c não corresponde a um contexto que contém um xcall(m)
                  levante erro
```

XGet e XSet

A verificação das regras xset e xget, de forma análoga à regra xcall, envolve duas análises: identificar se o atributo está sendo recuperado ou alterado dentro do escopo onde a regra foi definida e em seguida assegurar que este comportamento não ocorra em nenhum outro escopo dentre os componentes especificados na *design rule*.

Os seguintes pseudo-códigos exemplificam como ocorre a verificação nas regras xget e xset, respectivamente.

```
para cada classe/aspecto
      se o campo f é acessado num contexto c
            se c não corresponde a um contexto que contém um xget(f)
                  levante erro


para cada classe/aspecto
      se o campo f é alterado num contexto c
            se c não corresponde a um contexto que contém um xset(f)
                  levante erro
```

Composição de Regras

As regras comportamentais podem também ser combinadas com os operadores de negação, conjunção e disjunção. Assim, a verificação das mesmas deve sempre considerar se elas fazem parte ou não de uma regra composta. Por exemplo, se uma regra do tipo OU é definida, o fato de uma das regras não estar corretamente implementada é irrelevante caso a outra o esteja, e portanto nenhum erro de verificação deve ser apresentado. De forma análoga, utilizar o operador de negação em regras do tipo call, por exemplo, indicam que o método definido nesta regra não pode ser chamado no contexto onde ela está especificada.

Para facilitar estas verificações, o atributo *isComposed* é utilizado. Este atributo, localizado em nós que representam uma regra, informa se ela pertence ou não a uma expressão de composição de regras. Caso pertença, a verificação da regra deve ser feita e seu resultado é combinado com os resultados das outras regras na expressão.

A listagem 6 exemplifica como é feita a verificação para regras binárias do tipo “*OR*”.

```
1   public boolean OrRuleExpr.check(AST.ASTNode scope) {
2       return getLhs().check(scope) || getRhs().check(scope);
3   }
```

Listagem 9 – Verificação para regras comportamentais do tipo OR

A linha 1 declara o método *check* na classe que representa o nó da regra binária “OR”. Este método recebe um escopo a ser verificado e retorna um valor booleano indicando se a regra foi ou não seguida. Como se trata de uma regra binária, o resultado de sua verificação é a combinação dos resultados dos seus dois filhos; neste caso, a disjunção destes resultados.

4.4 Métricas

Buscando avaliar a qualidade da implementação, algumas métricas de complexidade foram coletadas e analisadas.

As métricas de complexidade são importantes para indicar a dificuldade de entendimento, utilização e manutenção de um software. No caso da implementação do compilador LSD, estas informações são importantes pois a linguagem encontra-se ainda em fase de especificação, devendo sofrer diversas alterações futuras, e para facilitar tal trabalho é essencial que o código seja compreensível e fácil de manter.

Algumas métricas de tamanho estão exibidas na tabela 3. É importante frisar que boa parte do código é gerada automaticamente pelas ferramentas utilizadas, como o Beaver, o JastAdd e o JFlex.

Métrica	Resultado
Número de classes	138
Número de pacotes	10
Aspectos	7

Tabela 3 - Métricas de Tamanho

Como não se trata de um projeto que utiliza diretamente orientação a aspectos, a métrica referente à quantidade de aspectos descrita na tabela 3 exibe a quantidade de módulos do JastAdd utilizados na implementação. Estes módulos utilizam *intertype declarations* para inserir código nos nós da linguagem, utilizando noções de programação orientada a aspectos.

As seções seguintes melhor detalham algumas métricas coletadas.

Source Lines of Code (SLOC)

Utilizada para coletar a quantidade de linhas de código de um programa. Na nossa abordagem, a métrica foi calculada considerando o número total de linhas, excluindo-se linhas em branco.

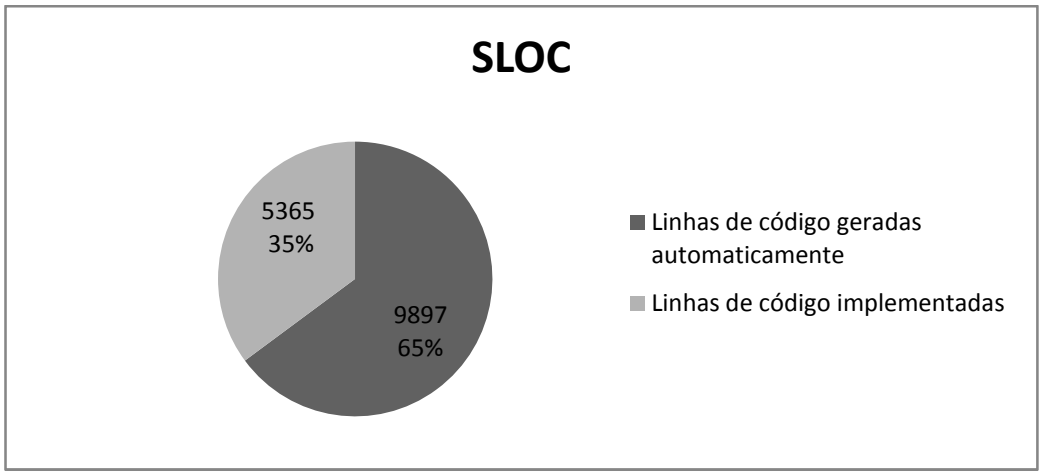

Figura 3 – Linhas de código implementadas vs. Linhas de código geradas automaticamente

A alta quantidade de linhas de código, totalizando 15262 LOC, é justificada pelo uso de ferramentas de geração automática de código (gerador de parser e dos nós da AST). Deste total, 35% apenas representam linhas de código implementadas, sendo o restante gerado automaticamente, conforme visto na figura 3.

Analisada de forma isolada, esta métrica não diz muito acerca da qualidade do código da ferramenta implementada. Entretanto, observa-se que o uso do JastAdd para geração dos nós e de seus respectivos atributos responde por aproximadamente 6 mil linhas do total, sendo o pacote que contém os nós da AST da linguagem responsável por 117 das 138 classes presentes. O JastAdd utiliza mecanismos de cache e demand-drive evaluation, que favorecem a performance e aumentam expressivamente a quantidade de código, tornando assim difícil comparar a implementação da LSD realizada com o JastAdd a outras possíveis abordagens.

É importante frisar também que o compilador da LSD reutiliza parte da implementação do frontend do abc, o que também influi na quantidade de linhas de código.

Cyclomatic Complexity Number (CCN)

Também conhecida por Complexidade de McCabe[19], esta métrica calcula a quantidade de caminhos linearmente independentes em um módulo de um programa. Quanto menor for esta quantidade, mais fácil de entender e de evoluir é uma operação.

Esta métrica teve um mau resultado, especialmente nos módulos de *parsing*, gerados automaticamente. Por priorizar performance e eficiência, o código gerado para construir o *parser* da linguagem carece de qualidade de acordo com os parâmetros da CCN. Um único método chega a ter 319 diferentes fluxos, o que dificulta o entendimento e manutenção do mesmo.

Dentre o código implementado, o pior valor encontrado foi de 10 diferentes caminhos, em funções encarregadas de verificar as regras. Este valor é ainda alto, podendo ser melhorado com o uso de *refactorings* simples, como *Extract Method*.

A importância de bons resultados nesta regra reside principalmente na necessidade de clareza e facilidade de extensão do código.

4.5 Evolução da Ferramenta

Como a LSD ainda está sendo projetada, é natural que haja mudanças tanto na estrutura sintática da linguagem, como na semântica de suas construções. Além disso, pode-se verificar no futuro a necessidade de novas regras, o que alteraria consideravelmente não só a linguagem como também o mecanismo de verificação. Para permitir que tais alterações sejam facilmente incorporadas no futuro, todo o projeto de desenvolvimento foi baseado em ferramentas extensíveis e no uso de interfaces e geração automática de código.

Um exemplo de possível extensão seria permitir o uso de *patterns* ao definir uma regra comportamental. Desta forma, o projetista poderia definir uma regra comportamental *call* que casasse não apenas com um, mas com diversos possíveis métodos. Assim, expressões como a seguir seriam possíveis:

```
xcall(public void A.*());
```

Embora o uso de *patterns* seja comum em AspectJ, a LSD atualmente implementada não dá suporte a tal tipo de construção. Veremos a seguir como poderíamos estender a linguagem de forma a aceitar o exemplo acima.

Primeiramente, a gramática da linguagem deve ser alterada para aceitar não apenas identificadores, mas também o *wildcard* estrela (*) no lugar do nome do método passado como parâmetro das regras *call* e *xcall*. A estrutura da AST atual é descrita da seguinte forma:

```
CallRuleExpr : RuleExpr ::= MethodITDSig;
XCallRuleExpr : RuleExpr ::= MethodITDSig;

MethodITDSig ::= Modifiers TypeAccess:Access TargetType:Access <ID:String>
Parameter:ParameterDeclaration* ;
```

O campo que guarda o nome do método é um *token* do tipo String e nome ID. A seguinte alteração deve ser feita para que, além de um ID, passemos a aceitar também um wildcard:

```
MethodITDSig ::= Modifiers TypeAccess:Access TargetType:Access IDPattern
Parameter:ParameterDeclaration* ;

abstract IDPattern;
NameIDPattern : IDPattern ::= <ID:String>
StarIDPattern : IDPattern;
```

A informação antes representada por um *token* agora é representada por um novo nó abstrato do tipo `IDPattern`. Por sua vez, dois filhos deste nó são definidos: um do tipo `NameIDPattern`, que contém um identificador, e um do tipo `StarIDPattern`, que representa o *wildcard* estrela. Assim, a declaração de método dentro de uma regra *call* ou *xcall* sempre poderá ter como nome tanto um identificador quanto um *wildcard*.

Uma vez alterada a estrutura da linguagem, precisamos alterar o *parser* para que tal estrutura seja corretamente identificada. Atualmente, o *parser* da LSD é gerado automaticamente através do *Beaver*. A seguinte construção implementa o *parsing* de uma estrutura do tipo `MethodITDSig,` que representa o método passado como parâmetro em regras *call* e *xcall*.

```
1     MethodITDSig method_itd_sign =
2       modifiers.a? type.b name.c DOT IDENTIFIER
3           LPAREN formal_parameter_list.e? RPAREN
```

A estrutura IDENTIFIER, na linha 2, precisa ser substituída por uma estrutura que case tanto com um identificador quanto com um *wildcard* estrela. A esta estrutura, demos o nome de `identifier_pattern`. Para implementá-la, criamos a seguinte sub-rotina:

```
1     IDPattern identifier_pattern =
2           MULT            {: return new StarIDPattern(); :}
3       |
4           IDENTIFIER.id    {: return new NameIDPattern(id); :} ;
```

Esta sub-rotina retorna um `IDPattern` de acordo com o tipo de construção encontrada. No caso do símbolo MULT (linha 2), será retornado um objeto representando um `StarIDPattern`. Caso contrário, um objeto do tipo `NameIDPattern` com o seu respectivo identificador será retornado, como mostra a linha 4.

Por fim, precisamos alterar o código que faz a verificação da regra, especificamente a parte que trata de casar o método declarado na regra com uma chamada de método feita em uma classe ou aspecto. Tal verificação considera apenas chamadas a métodos com o nome especificado; iremos estender esta verificação para ignorar o nome quando o mesmo for definido na regra como um *wildcard* estrela. Para tanto, podemos definir um atributo *isStarPattern* no nó do tipo `IDPattern`, que indica se o nome do método é ou não um *wildcard* estrela:

```
syn boolean IDPattern.isStarPattern;
eq NameIDPattern.isStarPattern = false;
eq StarIDPattern.isStarPattern = true;
```

Assim, a implementação da verificação da regra fica trivial:

```
if(!getIDPattern().isStarPattern()) {

      //não é wildcard; verifica o nome

}
```

Vale frisar que todas as alterações exemplificadas acima, desde a alteração da AST até a verificação da regra, não precisam ser intrusivas à implementação existente. Todas podem ser feitas em módulos à parte, e combinadas ao projeto no momento do *build*. Isso é possível graças ao mecanismo de composição de código do JastAdd.

5. Conclusão

Este trabalho propôs uma implementação da LSD – Language for Specifying Design Rules. Esta implementação permite a verificação de regras de design definidas nesta linguagem, de forma a assegurar que as mesmas estão sendo obedecidas.

A implementação da linguagem envolveu a construção de um compilador capaz de realizar análises léxica, sintática e semântica em *design rules* implementadas com a LSD. Um gerador de código foi também implementado, e é responsável pela verificação das regras definidas.

Dentre os objetivos atingidos, destacamos a implementação de um verificador completo para a LSD. Entretanto, algumas características não puderam ser implementadas devido à falta de tempo hábil, tais como o uso de módulos de configuração e de regras parametrizadas.

O uso de técnicas como gramáticas de atributos e orientação a aspectos foi essencial para a construção de um compilador extensível e modular. Como a LSD está ainda em processo de especificação, tais características são de grande importância.

5.1. Avaliação da Ferramenta

A ferramenta implementada, embora não contemple todas as características da LSD, é capaz de realizar a verificação de todas as regras definidas pela linguagem, tanto estruturais quanto comportamentais, de forma autônoma. Isto é, a verificação independe da existência de um compilador AspectJ instalado, pois o compilador LSD já possui uma extensão do abc embutida, como explicado no capítulo anterior.

A abordagem utilizada para implementar a verificação de regras, envolvendo o uso de gramática de atributos, facilita a extensão da linguagem e torna o código mais simples. Outra possível abordagem seria o uso do padrão de projeto Visitor[20]. O emprego deste padrão é clássico na construção de compiladores, e traz como vantagens a facilidade em definir novas operações, e a centralização de interesses em um único módulo. No entanto, adicionar novos nós à estrutura da linguagem é custoso, envolvendo a inserção de um novo método em cada Visitor; por esta razão este padrão é apenas recomendado quando a hierarquia de classes representando a AST é estável[20], o que não é o caso da LSD.

Outras ferramentas poderiam ser também utilizadas com para a implementação da verificação das regras. O SemmleCode[21], por exemplo, possui uma linguagem de domínio específico capaz de gerar *queries* para consultas e análises de código fonte. Desta forma, o código gerado para verificação poderia criar tais *queries* e executá-las, analisando assim o programa

Java/AspectJ e o cumprimento das regras. Regras estruturais como declaração de método, por exemplo, poderiam ser facilmente verificadas com a criação de uma *query* do tipo

```
from Class c where c.declaresMethod(“update”).
```

Embora conceitualmente seja viável, a versão atual do Semmle suporta apenas programas OO, não havendo suporte a *queries* para analisar programas AspectJ.

O módulo de configuração e o uso de parametrização presentes na LSD não fizeram parte do escopo de implementação deste trabalho. Desta forma, é uma evolução provável da linguagem a implementação destes recursos, que permitirão mais flexibilidade na construção e mapeamento das regras.

5.2. Trabalhos Relacionados

O uso de *design rules* vem sido discutido na literatura há algum tempo como possível solução para assegurar que determinados requisitos de projeto serão cumpridos, permitindo uma melhor modularização e paralelização no desenvolvimento. Algumas abordagens já foram sugeridas para especificar *design rules*, como utilizar o próprio modelo de *join points* de AspectJ[15] para especificar restrições e alguns tipos de regra. Utilizando as construções de *declare error/warning*, é possível definir diversas regras na forma de restrições, proibindo determinados *join points*. Entretanto, este modelo não permite a definição de restrições estruturais, como a obrigatoriedade de existência de um método. Além disso, não é possível forçar a ocorrência de uma chamada de método em um determinado escopo, por exemplo, o que limita a possibilidade de definição de regras comportamentais.

Abordaremos em seguida outros trabalhos relacionados ao uso de *design rules* em projetos orientados a aspectos.

PDL

PDL, ou Program Description Logic[4], é uma linguagem de domínio específico para verificação de *design rules*, que utiliza *pointcuts* para interceptar violações em regras. A sintaxe utilizada para definir tais *pointcuts* é bastante similar a AspectJ, entretanto o modelo de *join points* utilizado é estático; isso permite que todas as regras sejam checadas estaticamente, em tempo de compilação.

As definições de regras em PDL seguem um modelo de pares de *pointcuts-advices*, que especificam quais comportamentos devem ser identificados como violações de uma *design rule*. O exemplo abaixo demonstra como definir uma *design rule* que exige que todos os atributos sejam privados:

```
field(sourceType) && public :
      “Do not declare visible instance fields”;
```

O código acima define que qualquer atributo declarado como público deve gerar um erro de violação de regra com a mensagem definida entre aspas. A palavra reservada `field` intercepta todos os atributos encontrados no escopo passado como parâmetro; `sourceType` define que o escopo deve ser formado por todos os tipos definidos no programa. A composição com a palavra reservada `public` indica que devem ser interceptados apenas os atributos públicos. Quando as duas condições forem verdadeiras, a mensagem deve ser exibida como erro de compilação.

O diferencial de PDL é exatamente o modelo de *joinpoints* estático, que permite verificações complexas em tempo de compilação. Sua motivação foi adequar o uso de orientação a aspectos para definição de *design rules*, visto que o uso de AspectJ para tal fim, embora já amplamente estudado, apresenta problemas para expressar propriedades estáticas. Entretanto, embora eficaz para definir restrições em programas orientados a objetos, não existe no modelo da PDL construções que permitam uma clara definição das regras para promover o desenvolvimento paralelo de componentes. Embora o modelo de *join points* utilizado possa facilmente descrever violações em regras de design, não é possível definir a obrigatoriedade de determinadas regras, como forçar uma chamada de método em um certo ponto.

XPI

O uso de Crosscutting Programming Interfaces (XPIs) foi mostrado por Griswold et al. [15] como alternativa para documentar e verificar um conjunto de *design rules*, utilizando AspectJ. Entretanto, tratar-se de uma linguagem não específica leva a especificações demasiadamente complexas. A LSD, por sua especificação simples e não-ambígua, evita tal problema e facilita a definição das *design rules*.

5.3. Trabalhos Futuros

A implementação de um mecanismo de mapeamento entre as regras e os componentes que as implementam é um possível trabalho futuro, capaz de compor *design rules* complexas e de configurar suas utilizações de forma fácil. O uso de parametrização favorece ainda o reuso de tais *design rules*, diminuindo o retrabalho.

Uma análise crítica do código e da implementação do verificador LSD é também necessária, para fins de planejamento e execução de *refactorings*, melhorando a qualidade e eficiência do compilador.

Embora tenha sido testada com sucesso no exemplo de uso *Display Update*, a validação do compilador com projetos reais é essencial para analisar possíveis melhorias e identificar erros. Uma vez validado, o compilador poderá ser utilizado como meio de validação da LSD em estudos posteriores.

Por fim, a criação de uma ferramenta capaz de integrar o verificador de *design rules* a ferramentas de edição como o Eclipse, de forma integrada ao desenvolvimento de projetos, facilitaria e expandiria as possibilidades de uso da LSD, ficando também como sugestão de trabalho futuro.

Apêndice A – BNF - Language for Specifying Design Rules

```
DesignRule ::= 'dr' Id '{' Components* '}'
Components ::= InterfaceDecl | ClassDecl | AspectDecl
InterfaceDecl ::= InterfaceMod 'interface' Id ExtClause '{'
InterfaceBody '}'
ClassDecl ::= ClassMod 'class' Id ExtClause ImplClause '{'
ClassBody '}'
AspectDecl ::= AspectMod 'aspect' Id ExtClause ImplClause '{'
AspectBody '}'
ExtClause ::= [ 'extends' Id ]
ImplClause ::= [ 'implements' IdList ]
IdList ::= Id [ ',' Id ]*
InterfaceBody ::= Attribute* | MethodSig*
ClassBody ::= InterfaceBody | Method* | Rule*
AspectBody ::= ClassBody | Pointcut* | Advice* | AttITD* | MetITDSig* |
MetITD*
InterfaceMod ::= 'strictfp' | CommonCompMod
ClassMod ::= 'final' | InterfaceMod
AspectMod ::= 'privileged' | 'final' | CommonCompMod
CommonCompMod ::= 'abstract' | 'static' | VisMod
VisMod ::= 'public' | 'protected' | 'private' | 'pack'
Attribute ::= AttMod Type Id ';'
AttMod ::= 'transient ' | 'volatile' | 'static' | 'final' | VisMod
Method ::= MethodHead MethodBody
MethodSig ::= MethodHead ' ; '
MethodHead ::= MethodMod Type Id '(' ParamList ')'
MethodBody ::= '{' Rule* '}'
MethodMod ::= 'abstract' | 'final' | 'native' | 'static' | 'synchronized' |
'strictfp' | VisMod
AttITD ::= AttITDSig ';'
AttITDSig ::= AttMod Type Id '.' Id
MetITD ::= MethodITDSig MethodBody
MethodITDSig ::= MethodMod Type Id '.' Id '(' ParamList ')'
Pointcut ::= VisMod 'pointcut' Id ':' PointcutExpr
Advice ::= 'before' ':' PointcutExp AdviceBody
| 'after' ':' PointcutExp AdviceBody
| Type 'around' ':' PointcutExpr AdviceBody
AdviceBody ::= MethodBody
```

```
Rule ::= SimpleRule
| '!' Rule
| '(' Rule ')'
| Rule '&&' Rule
| Rule '||' Rule
SimpleRule ::= ( RCall | RGet | RSet | RXCall | RXGet | RXSet )
RCall ::= 'call' '(' MethodITDSig ')'
RGet ::= 'get' '(' AttITDSig ')'
RSet ::= 'set' '(' AttITDSig ')'
RXCall ::= 'xcall' '(' MethodITDSig ')'
RXGet ::= 'xget' '(' AttITDSig ')'
RXSet ::= 'xset' '(' AttITDSig ')'
```

Bibliografia

[1] Marcos Dosea, Alberto Costa Neto, Paulo Borba e Sérgio Soares. Specifyng Design Rules in Aspect-Oriented Systems. Latin American Workshop on Aspect-Oriented Software Development - LA-WASP'2007, páginas 67-78, João Pessoa-PB, Brasil, Outubro 2007.

[2] M. Xenos, D. Stavrinoudis, K. Zikouli e D. Christodoulakis. Object-Oriented Metrics – A Survey. FESMA 2000, Federation of European Software Measurement Associations, Madri, Espanha, 2000.

[3] Pavel Avgustinov, Torbjorn Ekman e Julian Tibble. Modularity First: A Case for Mixing AOP and Attribute Grammars. 7th International Conference on Aspect-Oriented Software Development, Bruxelas, Bélgica, 2008.

[4] Clint Morgan, Kris De Volder e Eric Wohlstadter. A Static Aspect Language for Checking Design Rules. 6th international conference on Aspect-oriented software development, Vancouver, Canadá, 2007.

[5] Donald E. Knuth. Semantics of context-free languages. Theory of Computing Systems, Vol. 2, No. 2. páginas. 127-145.

[6] Alberto Costa Neto. Uma linguagem para especificação de regras de projeto para sistemas orientados a aspectos, 2008. Defesa prevista para 2008.

[7] Marcos Barbosa Dosea. Uma abordagem modular para projeto de software orientado a aspectos. Dissertação de Mestrado, 2008.

[8] C. Baldwin. e K. Clark. BB. Design Rules, Vol. 1: The Power of Modularity. The MIT Press, 2000.

[9] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, e W. Griswold. Getting Started with AspectJ. Communications of the ACM, páginas 59–65, 2001.

[10] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., e Irwin. Aspect–Oriented Programming. European Conference on Object–Oriented Programming, ECOOP'97, páginas 220–242, LNCS 1241, 1997.

[11] Sullivan, K., Griswold, W. G., Song, Y., Cai, Y., Shonle, M., Tewari, N., e Rajan, H. (2005). Information Hiding Interfaces for Aspect-Oriented Design. 10th European Software Engineering Conference / 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE), páginas 166–175, New York, NY. ACM Press, 2005.

[12] Pavel Avgustinov, Aske Simon Christensen, Laurie Hendren, Sascha Kuzins, Jennifer Lhotak, Ondrej Lhotak, Oege de Moor, Damien Sereni, Ganesh Sittampalam e Julian Tibble. abc: an extensible AspectJ compiler. 4th international conference on Aspect-oriented software development (AOSD 2005), páginas 87-98, New York, NY, ACM Press, 2005.

[13] Microsystems Sun. The java language specification, Dezembro 2007. ftp://ftp.javasoft.com/docs/javaspec.ps.zip.

[14] Alberto Costa Neto, Márcio Ribeiro, Marcos Dósea, Rodrigo Bonifácio, e Paulo Borba e Sérgio Soares. Semantic dependencies and modularity of aspect-oriented software. ICSE Workshop on Assessment of Contemporary Modularization Techniques (ACoM.07) / ICSE 2007, Minneapolis, USA, Maio 2007.

[15] Griswold, W. G., Sullivan, K., Song, Y., Shonle, M., Tewari, N., Cai, Y., e Rajan, H. Modular Software Design with Crosscutting Interfaces, páginas 51–60, IEEE Software, 2006.

[16] Beaver – a LALR parser generator. Maio 2006. http://beaver.sourceforge.net/

[17] AspectJ Team. The AspectJ Project, Abril 2008. http://www.eclipse.org/aspectj

[18] T. Ekman, G. Hedin. The JastAdd System - modular extensible compiler construction**.** Science of Computer Programming, Elsevier, Outubro 2007. In Press.

[19] T.J. McCabe. A complexity measure. IEEE Transactions on Software Engineering, SE-2, Volume 4, 1976, pp 308-320.

[20] E. Gamma, R. Helm, R. Johnson e J. Vlissides. Design Patterns – Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[21] Semmle. SemmleCode: Flexible Code Queries. Maio, 2008. http://semmle.com

Folha de Aprovação

Assinatura do Orientador