VI CONGRESSO DE INICIAÇÃO CIENTÍFICA DA UNIVERSIDADE FEDERAL DE CAMPINA GRANDE

PIBIC/CNPq/UFCG-2009

UMA LINGUAGEM PARA ESPECIFICAÇÃO DE REGRAS DE DEISN PARA PROGRAMAS ORIENTADOS A ASPECTOS

Arthur Marques[1], Rohit Gheyi[2]

RESUMO

Apesar da Programação Orientada a Aspectos (OA) modularizar as preocupações transversais de um programa ela pode quebrar a modularidade das classes em certas situações. Neste caso, as classes devem estar cientes de quais aspectos interferem em sua funcionalidade. Foi observado que o uso de regras de design ajudam a diminuir o acoplamento entre classes e aspectos. Este artigo propõe uma linguagem de especificação de regras de design para declarar as dependências entre classes e aspectos. Desenvolvemos uma ferramenta que as checa automaticamente. Além disso, avaliamos o nosso trabalho especificando as regras de design de alguns dos padrões de projeto implementados em AspectJ, como o Singleton e o Template Method.

Palavras-chave: programação orientada a aspectos, regras de design, modularidade

A DESIGN RULE LANGUAGE SPECIFICATION TO ASPECT ORIENTED PROGRAMING

ABSTRACT

Despite Aspect Oriented Programming (AO) modularize program's crosscutting concerns it can break class modularity in certain circumstances. In this case, classes must known what aspects interfere on its functionalities. It was observed that design rules can mitigate the coupling between aspects and classes. This paper proposes a language to specify design rules and declare the dependences between classes and aspects. We've developed a tool to automatically check if the design rules are been followed. Therefore, we validated our work specifying design rules to some design patterns implemented on AspectJ, as the Singleton and Template Method design patterns.

Keywords: aspect oriented programming, design rules, modularity

1. INTRODUÇÃO

Orientação a Aspectos (KICZALES *et al*. 1997) propõe modularizar às preocupações transversais de um programa em módulos chamados de aspectos, melhorando, dessa forma, o design modular do projeto. Segundo Parnas (1972), um design modular possui as características: melhor compreensão, facilidade para mudanças e desenvolvimento em paralelo. Mas, ao modularizar estas preocupações em um aspecto, a modularidade das classes pode ser quebrada. As classes devem ter informações sobre quais aspectos são aplicados a ela, que funcionalidades são implementadas pelo aspecto, e que pontos de junção são capturados pelo aspecto (SULLIVAN *et al*. 2005, CLIFTONI e LEAVENS 2002, STEIMANN 2006). O desenvolvedor das classes não pode modificá-las sem antes ter informações sobre os aspectos, contradizendo os critérios de modularidade proposto por Parnas.

Essas dependências entre orientação a objetos (OO) e orientação a aspectos (OA) prejudicam o desenvolvimento do projeto. Mudanças não antecipadas podem alterar a funcionalidade do sistema. Isto ocorre, por exemplo, quando um método é extraído através de um refatoramento. Qualquer aspecto que

[1] Aluno de Curso de Ciência da Computação, Depto. de Sistemas e Computação, UFCG, Campina Grande , PB, E-mail: arthursm@dsc.ufcg.edu.br

[2] Ciência da Computação, Professor Doutor, Depto. de Sistemas e Computação, UFCG, Campina Grande , PB, E-mail: rohit@dsc.ufcg.edu.br

dependa destes pontos de junção não irá funcionar de forma adequada, a não ser que os pontos de junção sejam atualizados para capturar o novo cenário. Este tipo de problema ocorre porque o aspecto depende de algo suscetível a mudanças. Dessa forma, não temos um design modular.

Outro grave problema que ocorre é a falta de suporte ao desenvolvimento em paralelo. Muitos aspectos dependem de detalhes de como as classes estão codificadas e não podem ser criados sem algum tipo de suporte ou regra que estabeleça como pontos de junção são capturados. Aspectos podem sem aplicados depois que o programa estiver completo, extraindo as preocupações transversais e gerando uma nova versão OA do projeto, mas como softwares estão em constante manutenção e evolução este não é o cenário ideal.

Diante destas fragilidades Sullivan *et al*. (2005) discutiram como especificar regras de design usando uma linguagem natural para diminuir estas dependências. Classes e aspectos passariam a depender destas regras, elas seriam a linha guia entre os desenvolvedores, promovendo assim um melhor design modular.

Vários trabalhos (SULLIVAN *et al*. 2005, LOPES e BAJRACHARYA 2006) mostraram que projetos desenvolvidos com o apoio de regras de design possuem uma melhor modularidade em comparação a versões desenvolvidas sem o auxílio destas regras.

Griswold *et al*. (2006) utilizaram a linguagem de AspectJ (KICZALES *et al*. 2001) para especificar e checar regras de design usando pontos de junção e declarações de error e warning. Porém, usar a própria linguagem de AspectJ para estas especificações apresenta regras limitadas e complexas. Apesar de termos uma checagem automática destas regras é necessário ter um grande domínio de AspectJ para declarar estas regras.

Dósea *et al*. (2007) propuseram uma linguagem para especificar regras de design sem as limitações da linguagem de AspectJ, com regras menos complexas e mais fáceis de entender. Porém, a falta de uma semântica bem definida para a sua linguagem pode gerar regras dúbias, que poderiam comprometer a funcionalidade do programa se interpretadas de forma incorreta. Além disso, sua linguagem não é suficientemente expressiva.

Neste trabalho, propomos uma linguagem de especificação de regras de design que diminuam as dependências entre classes e aspectos, sendo uma interface entre desenvolvedores OO e OA. Esta linguagem possui uma semântica clara, já que ela é próxima da lógica de primeira ordem. O usuário pode especificar regras com operadores lógicos como negação, conjunções ou disjunções. Além disso, esta linguagem possui quantificadores universais e existenciais, permitindo declarar de forma simples e de fácil entendimento um conjunto de componentes que devem seguir uma determinada regra.

Com o objetivo de automatizar o processo de análise dessas regras criamos um suporte ferramental que guia os desenvolvedores do projeto a seguirem as regras de nossa linguagem de forma simples e prática.

Avaliamos nossa abordagem especificando regras de design para o estudo de caso sobre as figuras geométricas (KICZALES *et al*. 2001) e também para alguns dos padrões de projeto implementados em AspectJ: Singleton e Tempalte Method (HANNEMANN e KICZALES 2002).

1.1 Exemplo Motivante

Nesta seção explicamos como orientação a aspectos modulariza preocupações transversais, assim como apresentaremos os eventuais problemas que este tipo de solução pode causar. Para isso, inicialmente vamos considerar um sistema que trabalha com diversas figuras geométricas e exibe estas figuras em uma tela. Podemos, por exemplo, desenhar figuras, editar figuras existentes, movê-las, entre outras funcionalidades. À medida que realizamos estas operações temos que atualizar uma tela de visualização destas figuras. A Figura 1 apresenta um diagrama UML descrevendo parte deste modelo, onde temos uma interface *FiguraGeometrica* e as classes *Linha* e *Ponto* que implementam esta interface.

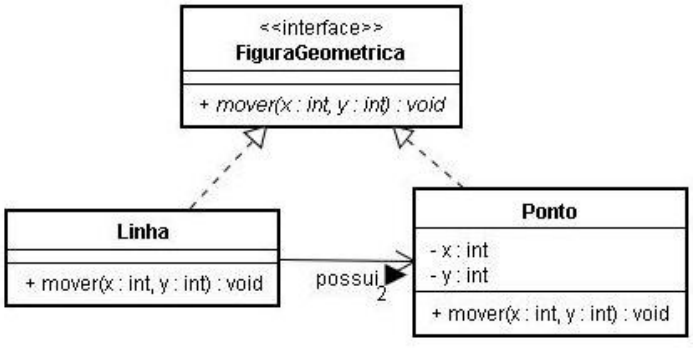

Figura 1. Diagrama de Classes das Figuras Geométricas

Com base no diagrama de classes apresentado na Figura 1, a Listagem 1 apresenta o código do método *mover* para as classes *Ponto* e *Linha*, respectivamente. Note que, após alterar as coordenadas de cada figura geométrica, chamamos o método estático *atualizar* para atualizar a tela de visualização das figuras. Neste exemplo, existem duas principais preocupações: a de editar as figuras geométricas e a de atualizar as figuras. Dizemos que a preocupação de atualizar a tela esta **espalhada**, pois ela se encontra em diversos trechos de código e também **entrelaçada**, porque ela se mistura com códigos relacionados a outras preocupações. Esse tipo de preocupação também é chamado de **preocupação transversal.**

Listagem 1. Método mover

```
public void mover(int dx, int dy) {
  setX (this.x +dx);
  setY (this.y +dy);
  Tela.atualizar( );
}
```

```
public void mover(int dx, int dy) {
  p1.setX (p1.getX( ) +dx);
  p1.setY (p1.getY( ) +dy);
  p2.setX (p2.getX () +dx);
  p2.setY (p2.getY( ) +dy);
  Tela.atualizar( );
}
```

Preocupações transversais podem prejudicar o desenvolvimento do software. Por exemplo, se mudarmos o modo como o método *atualizar* é chamado, temos de fazer essa mudança em cada trecho de código que atualiza a tela. Ou ainda, ao adicionar novas funcionalidades como cores as figuras geométricas também é necessário adicionar novas chamadas ao método *atualizar*. Quanto mais componentes envolvem o nosso sistema, mais complexo e evidente é este problema.

Desta forma, orientação a aspectos propõe modularizar às preocupações transversais em módulos chamados de **aspectos**. Através de **pontos de junção** um aspecto captura trechos do código OO e adiciona informações neste por meio de **adendos**. A Listagem 2 apresenta uma solução em AspectJ para modularizar a atualização da tela. Capturamos todas as execuções do método *mover* e após qualquer chamada a este método chamamos o método para *atualizar* a tela através do adendo deste aspecto.

Listagem 2. Aspecto para Atualizar a Tela

```
public aspect AtualizarTela {
   pointcut mudanca(): execution(void FiguraGeometrica.mover(int, int));
   after(): mudanca() {
      Tela.Atualizar();
   }
}
```

Entretanto, apesar do aspecto ter modularizado esta preocupação agora temos outro tipo de dependência: entre o aspecto e as classes, os desenvolvedores OO precisam saber que não devem mais implementar o conceito da atualização da tela e que existe um aspecto responsável por esta tarefa.

Suponha ainda que houvesse a necessidade de refatorar o código OO através de um refatoramento de nome, mudando o nome do método de *mover* para *transladar*. A funcionalidade do programa estaria comprometida, pois o ponto de junção mudança capturaria um método inexistente no programa, o adendo não seria mais aplicado a este ponto de junção. Ou ainda, caso adicionássemos novas funcionalidades que exigissem a atualização da tela estas não seriam capturadas pelo ponto de junção declarado no aspecto *AtualizarTela*. Se o desenvolvedor OO não estiver ciente do aspecto ele pode adicionar uma chamada ao método *atualizar* no método *mover*, sem saber que esta função é implementada pelo aspecto, teríamos trechos de código duplicados e desnecessários.

2. RESULTADOS E DISCUSSÃO

2.1 Linguagem de Especificação de Regras de Design

Diante destes problemas definimos uma linguagem de regras que auxiliem o desenvolvimento do nosso sistema. Expressamos essas regras através de uma linguagem chamada de interface para programas entre - cortantes (**xpi**). Podemos especificar como regras de design:

- Deve existir uma interface com o nome *FiguraGeometrica*;
- Devem existir as classes públicas *Ponto* e *Linha* que implementam esta interface;
- Deve existir o método público *mover,* com dois parâmetros inteiros, na interface *FiguraGeometrica*;

- Todos os métodos de uma classe que implemente a interface *FiguraGeometrica* não podem chamar o método *atualizar*.

Estas regras podem ser expressas por uma xpi. Cada xpi é composta por duas partes. A primeira consiste de um conjunto de variáveis que contém as declarações de nosso interesse. Podemos definir diferentes tipos de variáveis e elas podem ser utilizadas em diversas regras. O uso de variáveis torna as regras mais sucintas e mais fáceis de compreender. A segunda parte consiste das regras de design.

Para descrever a nossa linguagem inicialmente vamos apresentar exemplos das construções de nossa linguagem para então apresentar um exemplo de uma xpi, por fim, apresentamos a gramática de nossa linguagem.

2.2 Linguagem

Toda xpi possui uma estrutura básica, descrita na Listagem 3. Ela é composta de um nome, um conjunto de variáveis e um conjunto de regras.

Listagem 3. Estrutura Padrão de uma xpi

```
xpi Nome {
   ... // declaracao de variaveis
   rules
   ... // declaracao de regras
}
```

Com base neste conjunto de declarações temos uma coleção de variáveis e regras de design que compõem nossa xpi. A seguir, descrevemos como declarar cada componente presente em uma xpi.

2.3 Declaração de Variáveis

As variáveis declaram os qualificadores da maior parte dos componentes OO presentes em um projeto. Para declararmos uma variável, inicialmente definimos seu tipo e atribuímos um nome a variável. Declaramos os qualificadores presentes nesta variável de acordo com o seu tipo, de maneira semelhante à declaração deste componente em Java. Os seguintes componentes podem ser descritos como variáveis: classes, interfaces, construtores, métodos e atributos. A seguir descrevemos cada uma dessas variáveis e suas construções.

2.3.1 Declaração de Classes e Interfaces

Classes e interfaces são descritas seguindo a sintaxe Java, podemos declarar sua visibilidade, se são abstratas, seu nome, conjunto de interfaces implementadas e sua herança. Para isso, inicialmente declaramos o tipo da variável e um nome associado a ela e então descrevemos os componentes presentes nesta variável.

Por exemplo, a Listagem 4 descreve uma classe pública de nome *Linha*, note que antes do nome temos um wild card *, isso significa que esta classe pode pertencer a qualquer pacote, do contrário especificaríamos o nome do pacote.

Listagem 4. Variável de Classe

```
class c = public class *.Linha;
```

Suponha que esta classe implementasse a interface *FiguraGeometrica*, adicionamos esta restrição da mesma maneira que em Java, como apresentado na Listagem 5, que descreve as variáveis *FiguraGeometrica* e a nova variável Linha, respectivamente.

Listagem 5. Variável de Classe e Interface

```
interface i = public interface *.FiguraGeometrica;
class c = public class *.Linha implements FiguraGeometrica;
```

Em um segundo cenário suponha que *Linha* seja uma classe abstrata e que teríamos várias classes que herdassem dela, a Listagem 6 descreve este cenário, onde c1 é nossa classe abstrata e c2 e c3 são classes finais que herdam de linha.

Listagem 6. Variável de Classe Abstrata e Herança

```
class c1 = abstract class *.Linha;
class c2 = public final class *.LinhaVertical extends Linha;
class c3 = public final class *.LinhaHorizontal extends Linha;
```

Para facilitar a descrição das variáveis, poderíamos ainda substituir as declarações **implements FiguraGeometrica** por **implements i**, referenciando a variável da interface. Da mesma forma, a declaração **extends Linha** poderia ser substituída por **extends c1**, referenciando a variável *Linha*. É importante ressaltar que para qualquer declaração de variável estamos declarando o conjunto de restrições desta variável, se não especificamos nada sobre uma variável esta pode assumir qualquer valor. Por exemplo, a variável **c1** da Listagem 6 não possui restrições quanto a visibilidade, logo possíveis valores para a sua visibilidade seriam: **public**, **private**, **protected** ou **package**.

2.3.2 Declaração de Construtores e Métodos

Para declarar construtores seguimos a mesma abordagem, declarando sua visibilidade, parâmetros e exceções lançadas. É importante ressaltar que a declaração dos parâmetros segue uma semântica diferente, onde não especificar nenhum parâmetro significa que o construtor não possui parâmetros. Quando especificamos os parâmetros o construtor ou método deve ter exatamente aquela quantidade e tipos de parâmetros. A mesma semântica é usada para o conjunto de exceções lançadas por um método ou construtor. Caso não estejamos interessados no tipo de parâmetro podemos usar o wild card "*" para definir que ele pode ser de qualquer tipo, da mesma forma caso a quantidade de parâmetros não nos interessar podemos usar o wild card ".." para esta declaração, semelhante a AspectJ. A listagem 7 descreve possíveis cenários para a declaração de um construtor da classe *Ponto*, exemplificando cada caso descrito acima.

Listagem 7. Variáveis de Construtores

```
constructor crt1 = public Ponto ( );
constructor crt2 = public Ponto ( int, int);
constructor crt3 = public Ponto ( *, *);
constructor crt4 = public Ponto ( .. );
```

Métodos podem ser descritos seguindo uma abordagem semelhante, a diferença é que temos mais qualificadores. Devemos declarar o nome do método, parâmetros e retorno. Além disso, podemos declarar sua visibilidade e se um método é abstrato, final ou estático. A Listagem 8 apresenta um conjunto de possíveis métodos e suas particularidades para a classe *Ponto*.

Listagem 8. Variáveis de Métodos

```
method m1 = public boolean origem ( );
method m2 = public void setX (int);
method m3 = public void mover (int, int);
method m4 = public void setCordenadas (..) throws CoordenadasInvalidas;
```

2.3.3 Declaração de Atributos

Por fim, temos o conjunto de declarações para um atributo, devemos declarar seu nome e tipo. Podemos também declarar sua visibilidade, se é estático ou final. A listagem 9 apresenta possíveis atributos para as classes *Ponto* e *Linha,* onde temos a coordenada x de um ponto, e dois atributos do tipo *Ponto*.

Listagem 9. Variáveis de Atributos

```
attribute atr1 = public int x;
attribute atr2 = public Ponto p1;
attribute atr3 = public final Ponto ORIGEM;
```

2.4 Regras de Design

A declaração das regras depende do tipo de variáveis que utilizamos para descrevê-las. Podemos definir a existência das variáveis declaradas anteriormente no projeto. Para todas as regras que podem ser

expressas em nossa linguagem criamos regras semelhantes, mas com o operador de negação (”!"). Desta forma, expressamos o operador lógico de negação para a nossa xpi. Por exemplo, podemos declarar que uma classe não pode ser filha de outra ou que um determinado método não deve estar presente em uma classe. Também podemos expressar outros operadores como "e" ou "ou".

Existem dois principais tipos de regras. Regras básicas, que definem a existência de classes e interfaces, as quais métodos e atributos pertencem e, por fim, regras que definem a existência de chamadas a métodos por construtores ou métodos. Como segundo tipo de regra, temos regras com quantificadores universais, elas abrangem um conjunto de componentes e especificam regras para este conjunto. Apresentamos a seguir a particularidade de cada um deste tipo de regras.

2.4.1 Regras Básicas

Para definir a existência de classes e interfaces usamos a declaração **some**, isso significa que deve existir pelo menos um componente no projeto com as características descritas. Por exemplo, dada uma variável de classe **c** para definimos sua existência a partir da declaração **some c**.

Para definir componentes presentes nestas classes e interfaces usamos a função **in** onde definimos uma variável e através desta função dizemos a qual componente esta variável pertence. Dada uma variável de classe **c** definimos que um atributo, método ou construtor **x** pertence a esta classe através da declaração x.**in**(c).

Por fim, a função **call** declara quais métodos um construtor ou método chama em seu corpo. Da mesma forma que a função in, temos uma variável **y** que representa um de método ou interface e dada uma variável de método **m** dizemos que este método é chamado por um método ou construtor através da declaração y.**call**(m).

2.4.2 Regras com Quatificadores

Podemos também definir regras usando quantificadores universais. Estas regras abrangem todos os componentes no escopo do projeto com uma determinada característica. Para declará-las primeiro definimos que ela será uma regra global, atribuímos então uma variável a esta regra e definimos o tipo da variável que queremos checar. Depois de definir seu tipo, declaramos as restrições desta variável. Por fim, definimos qual regra deve ser seguida para os componentes que possuem tais especificações. Por exemplo, podemos definir que todos os métodos que contenham o nome **set** não chamem o método *atualizar* com a seguinte regra:

Listagem 10. Regras com quantificação

```
m1 = public static void atualizar ( );
rules
  all m: method | m.name(set*) => !m.call(m1);
```

Na listagem 10 apresentamos um exemplo para regras com quantificadores universais. Para o conjunto de restrições e de regras usamos um conjunto de funções auxiliares que declaram diferentes características acerca da variável, por exemplo, usamos a função **name** e a função **call**. Para o conjunto de restrições usamos os lógicos de conjunção ("||") e disjunção ("&&"). Da mesma forma que nas regras existenciais, podemos usar os operados de negação para criar estas regras.

Temos quatorze diferentes funções auxiliares para especificar restrições e regras, elas são limitadas aos tipos de suas variáveis, a função **isStatic**, por exemplo, só pode ser aplicada a métodos e atributos. A tabela 1 apresenta as funções de nossa linguagem, assim como uma breve descrição destas.

Tabela 1. Funções Auxiliares

Função	Semântica	Aplicável as variáveis
a.vis(arg)	Declara que a variável **a** deve ter visibilidade **arg**	**a**: { Classes, interfaces, métodos, construtores e atributos }
a.isFinal( )	Declara que a variável **a** deve ser final	**a**: { Classes, métodos e atributos }
a.isAbstract( )	Declara que a variável **a** deve ser abstrata	**a**: { Classes e métodos }
a.isStatic( )	Declara que a variável **a** deve ser estática	**a**: { Métodos e atributos }
a.implements(arg)	Declara que a variável **a** deve implementar **arg**	**a**: { Classes}

a.extends(arg)	Declara que a variável **a** deve herdar de **arg**	**a**: { Classes e interfaces }
a.name (arg)	Declara que a variável **a** deve ter nome **arg**	**a**: { Classes, interfaces, métodos, construtores e atributos }
a.package(arg)	Declara que a variável **a** deve estar no pacote **arg**	**a**: { Classes e interfaces }
a.in(arg)	Declara que a variável **a** deve pertencer a **arg**	**a**: { Classes e interfaces }
a.type(arg)	Declara que a variável **a** deve ter tipo **arg**	**a**: { Atributos }
a.throws(arg)	Declara que a variável **a** deve lançar **arg**	**a**: { Métodos e construtores }
a.call(arg)	Declara que a variável **a** deve chamar **arg**	**a**: { Métodos e construtores }
a.return(arg)	Declara que a variável **a** deve retornar **arg**	**a**: { Métodos }
a.class(arg)	Declara que a variável **a** pertence a **arg**	**a**: { Métodos, construtores e atributos }

As funções facilitam a declaração de regras e podem tanto usar novas construções ou variáveis em sua declaração, isto facilita seu uso e as torna melhor aproveitáveis. Por exemplo, dada uma variável de interface **i** que representa a interface *FiguraGeometrica*, poderíamos criar uma regra declarando que todas as classes que implementam a interface *FiguraGeometrica* devem ter visibilidade pública de duas formas distintas, como apresentado na Listagem 11.

Listagem 11. Funções e argumentos

```
i = public interface FiguraGeometrica;
rules
  all c: class | c.implements(FiguraGeometrica) => c.vis(public);
  all c: class | c.implements(i) => c.vis(public);
```

Como visto, ambas as funções declaram que as classes que implementam *FiguraGeometrica* devem ter visibilidade pública, a diferença entre ambas é o uso ou não da variável **i**. Este tipo de construção facilita a declaração de algumas regras onde não necessitamos de uma variável para descrever um determinado contexto.

Acreditamos que, com este conjunto de regras é possível especificar todas as restrições que classes e aspectos devem seguir de modo a obter um melhor desenvolvimento modular. Até então, apresentamos a linguagem e suas construções, a seguir apresentamos a gramática livre de contexto de nossa linguagem.

2.5 Exemplo

A Listagem 12 apresenta uma xpi que contém regras de design para o exemplo das figuras geométricas. Primeiro declaramos variáveis que correspondem às classes, interfaces e métodos de nosso interesse e, após estas declarações, são definidas as regras de design acerca do nosso projeto. Para o conjunto de regras, inicialmente expressamos a existência da interface *FiguraGeometrica* e da classe *Tela*, respectivamente. Então, declaramos que os método *mover* pertence à interface *FiguraGeometrica* e o método *atualizar* a classe *Tela*. Por fim, definimos que nenhum método de nome *mover* ou que comece com *set* deve chamar o método *atualizar*.

Listagem 12. Regras de Design para as Figuras Geométricas

```
xpi FigurasGeometricas {

  interface i = public interface *.FiguraGeometrica;
  class c = public class Tela;
  method m1 = public void mover(..);
  method m2 = public static void atualizar();

  rules
    some i;
    some c;
    m1.in(i);
```

```
        m2.in(c);
        all m: method | m.name(mover)  || m.name(set*) => !m.call(m2);
    }
```

Neste exemplo, temos diversas construções de nossa linguagem. Inicialmente temos três tipos de variáveis distintas: interface, classe e métodos, respectivamente. Estas variáveis descrevem, de acordo com o seu tipo, os componentes que serão usados na declaração das regras de design. Temos então, um conjunto de regras estruturais, elas declaram a existência de classes e interfaces, assim como que métodos pertencem a elas. Por fim, temos uma regra com um quantificador universal que, através de funções, declara que o conjunto de métodos de nome *mover* ou o conjunto de métodos que iniciam com o nome *set* deve seguir uma determinada regra. A seguir generalizamos nossa teoria mostrando cada uma das construções da linguagem, e também sua forma normal de Backus-Naur.

2.6 BNF da Linguagem

Apresentamos a nossa linguagem para especificação de regras de design, a declaração de variáveis e regras em uma xpi e diversos exemplos para cada uma destas construções. Para expressar formalmente a gramática livre de contexto de nossa linguagem utilizamos o Formalismo de Backus-Naur (BNF).

Segundo nossa gramática, uma xpi é composta de duas principais partes: o conjunto de variáveis e o conjunto de regras. Para o conjunto de variáveis temos cinco expressões para classes, interfaces, métodos, construtores e atributos, respectivamente. Já o conjunto de regras é dividido em regras existenciais e regras universais. Para o conjunto de regras existenciais temos as regras **some**, **in** e **call**. Já para as regras universais, temos expressões para as restrições e regras, que fazem uso das funções apresentadas na Tabela 1. A Listagem 13 apresenta a BNF de nossa linguagem.

Listagem 13. Forma Normal de Backus-Naur de uma xpi

```
<XPI> ::= 'xpi' <ID> '{' <Variables>* 'rules' <Rules>*  '}'

<Variables> ::=  <Class> | <Method>  | <Interface> |  <Constructors> | < Attribute >
<Class> ::= 'class' <ID>  '=' [<Visibility>] ['abstract' | 'final'] 'class' <ID> ['extends' <ID>] ('implements' <ID>)*
<Interface> ::= 'interface' <ID> '=' ['abstract'] [<Visibility>] 'interface' <ID> [<Extends> <ID>]
<Method> ::= 'method' <ID> '=' [<Visibility>]  ['abstract' | 'final'] ['static'] <Type> <ID> '(' <Parameters> ')' [
'throws' <Exceptions>]
<Constructor> ::= 'constructor' <ID> '=' [<Visibility>] <ID> '(' <Parameters> ')' [ 'throws' <Exceptions>]
<Attribute> ::= 'attribute' <ID> '=' [<Visibility>] ['final'] ['static'] <Type> <ID>

<Rules> ::= <ExistentialRule> | <QuantitativeRule>
<ExistentialRule> ::= ['!'] ('some' <ID> | <ID> '.call' '('<ID>')' |  <ID> '.in' '('<ID>')' )
<QuantitativeRules> ::= 'all' <ID> ':' <Predicate> '=>' <RuleFormula>

<Predicate> ::= (<ClassPred> |  <InterfacePred> | <MethodPred> | <ConstructorPred> | <AttributePred>)

<ClassPred> := 'class' '|' <Exp>
<InterfacePred> := 'interface' '|' <Exp>
<MethodPred> := 'method' '|' <Exp>
<ConstructorPred> := 'constructor' '|' <Exp>
<AttributePred> := 'attribute' '|' <Exp>

<Exp> ::= <BinExp> | <UnExp> | <NotExp>
<BinExp> ::= <Exp> <setOP> <Exp>
<NotExp> ::= '!' <Exp>
<UnExp> ::=  <AbstractFun> | <CallFun> | <ClassExp>| <FinalFun> |  <InheritanceFun> | <InterfaceFun> |
<MethodFun> | <NameFun> | <PackageFun> | <ReturnFun> | <StaticFun> | <ThrowsFun> |  <TypeFun>
|<VisibilityFun>

<RuleFormula> ::= <BinRule> | <UnRule> | <NotRule>
<BinRule> ::= <RuleFormula> <setOP> <RuleFormula>
<NotRule> ::= '!' <RuleFormula>
<setOP> ::= '&&' | '||'
<UnRule> ::= <AbstractFun> | <CallFun> | <ClassExp>| <FinalFun> |  <InheritanceFun> | <InterfaceFun> |
<MethodFun> | <NameFun> | <PackageFun> | <ReturnFun> | <StaticFun> | <ThrowsFun> |  <TypeFun>
```

```
|<VisibilityFun>

<AbstractFun> ::= <ID> '.isAbstract"(")'
<CallFun> ::= <ID> '.call"('<ID>')'
<ClassFun> ::= <ID> '.class"('<ID>')'
<FinalFun> ::= <ID> '.isFinal"(")'
<InterfaceFun> ::= <ID> '.implements"('<ID>')'
<InheritanceFun> ::= <ID> '.extends"('<ID>')'
<MethodFun> ::= <ID> '.in"('<ID>')'
<NameFun> ::= <ID> '.name"('<ID>')'
<PackageFun> ::= <ID> '.package"('<ID>')'
<ReturnFun> ::= <ID> '.return"('<Type>')'
<StaticFun> ::= <ID> '.isStatic"(")'
<ThrowsFun> ::= <ID> '.throws"('<ID>')'
<TypeFun> ::= <ID> '.type"('<Type>')'
<VisibilityFun> ::= <ID> '.vis"('<Visibility>')'

<Visibility> ::= 'private' | 'protected' | 'private' | 'package'
<Type> ::= 'void' | 'int' | 'double' | 'float' | 'boolean' | 'long' | <ID>
<Parameters> ::= <Type>*
<Exceptions> ::= <ID>*
```

Na Listagem 13 apresentamos a BNF de nossa linguagem. Nela, estão destacados em negrito literais e regras de derivação, quando estas se encontram no lado esquerdo da derivação. Para o conjunto de restrições e regras usamos várias funções. Dizemos que há uma sobrecarga de funções, onde elas se comportam de maneira diferente de acordo com o contexto no qual se encontram.

Os operadores de negação, disjunção e conjunção são representados pelos literais "!", "&&" e "||", respectivamente. Além destes literais, temos ainda, os wild cards de AspectJ: "*" e ".." que representam qualquer tipo de variável, qualquer número de parâmetros. Com base nesta gramática, criamos um analisador sintático que gera o nosso conjunto de regras de acordo com as derivações descritas em nossa gramática.

2.7 Estudos de Caso

Avaliamos a linguagem descrita neste artigo especificando regras de design para o exemplo das figuras geométricas. As regras produzidas nesta xpi são de fácil leitura e compreensão, o uso de variáveis permite que elas possam ser utilizadas na produção de várias regras, facilitando a criação de novas regras, além de tornar as regras mais sucintas.

A linguagem da Listagem 12 possui regras simples, definindo a existência das variáveis descritas anteriormente e também regras mais complexas que fazem uso de quantificadores. As regras com quantificadores possuem um escopo maior, checando se um conjunto de variáveis obedece a uma determinada regra. Trabalhos anteriores não tinham este tipo de quantificação, assim não era possível declarar que um conjunto de variáveis seguia uma regra. A nossa linguagem possui este tipo de quantificação, aumentando a expressividade da linguagem além de permitir que um grande conjunto de regras possa ser descrito de forma simples e prática.

Avaliamos a linguagem em alguns dos padrões de projeto implementados em AspectJ. A seguir descrevemos xpis para os padrões de projeto Singleton e Template Method, respectivamente, assim como as particularidades de cada exemplo usado para apresentar a versão OA do padrão. Estes padrões foram escolhidos devido à diferente forma como cada um interage com o projeto assim como a estratégia OA adotada por cada um para solucionar o problema. Temos um padrão de criação que usa adendos em sua estrutura e um padrão comportamental que faz uso de declarações inter-tipo. A seguir, descrevemos o estudo de cada um deles.

2.7.1 Singleton

O padrão de projeto Singleton garante a existência de uma única instância de cada objeto. Em sua implementação OO todos os construtores devem ter visibilidade privada ou protegida e novos objetos devem ser obtidos através do método *getInstance*. Na implementação deste padrão em OA, o papel de administrar uma única instância dos objetos é feita pelo aspecto.

A Figura 2 descreve o cenário utilizado por Hannemann e Kiczales para este padrão. O objetivo do exemplo é mostrar como mesmo com herança OA permite uma maior flexibilidade para definir classes singleton. Neste exemplo, temos uma interface arbitrária *Singleton* e duas classes que implementam esta

interface *Printer* e *PrinterSubClass*, que herda de *Printer*. Além disso, temos o aspecto *SingletonProtocol* que administra as instâncias de cada objeto através de um adendo, onde *Printer* seria um singleton e sua subclasse não.

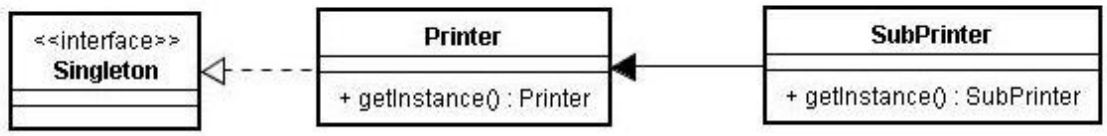

Figura 2. Diagrama de Classes do Exemplo Singleton

A Listagem 14 descreve o aspecto abstrato *SingletonProtocol*. Nele temos a estrutura de dados de uma tabela Hash para guardar cada instância de cada classe. O ponto de junção *protectionExclusions* exclui do adendo eventuais classes que implementem a interface, mas que não sejam singletons, estes pontos de junção serão definidos num aspecto que herde de *SingletonProtocol*. Por fim, o adendo captura todas as chamadas ao método *getInstance* de qualquer classe que implemente *Singleton* e não esteja no ponto de junção *protectionExclusions* recuperando a única instância dessas classes.

Listagem 14. Aspecto Singleton

```
public abstract aspect SingletonProtocol {

   private Hashtable singletons = new Hashtable();
   protected pointcut protectionExclusions();

   Object around(): call((Singleton+).new(..))  && !protectionExclusions() {
     Class singleton = thisJoinPoint.getSignature().getDeclaringType();
     if (singletons.get(singleton) == null) {
        singletons.put(singleton, proceed());
     }
     singletons.get(singleton);
   }
}
```

Com base nestas informações, temos que definir que todas as classes que implementem a interface *Singleton* devem ter o método *getInstance,* com qualquer parâmetro ou retorno, e que todos os seus construtores tem visibilidade protegida. A listagem 15 define uma xpi que especifica regras de design para este padrão de projeto.

Listagem 15. Singleton xpi

```
xpi SingletonRules {

   interface i = public interface Singleton;
   method m = public static * getInstance(..);

   rules
     some i;
     all c: class | c.implements(i) => m.in(c);
     all crt: constructor | crt.class(Singleton+) => crt.vis(protected);
}
```

Primeiro definimos a interface *Singleton*, onde de acordo com as necessidades do projeto definiremos as classes que implementam esta interface. Declaramos então que para cada classe que implemente esta interface o método *getInstance* deve pertencer a esta classe. Assim como, todos os construtores de classes que implementem a interface *Singleton* devem ter visibilidade protegida. Estes tipos de regras são importantes para avaliar a expressividade da linguagem, onde uma única regra pode checar de maneira prática todo o escopo do projeto.

2.7.2 Template Method

O padrão de projeto Template Method define o comportamento comum de um método, em OO isto é feito através de herança e métodos abstratos. Já em sua versão OA, este comportamento é definido através de um aspecto, por meio de declarações inter-tipo. No exemplo de Hanemman e Kiczales, temos classes com o objetivo de tratar uma cadeia de caracteres de diferentes formas, mas com um mesmo comportamento, que é definido no método *generate*. Este método deve receber uma String e chamar os métodos *prepare*, filter e finalize. A Figura 3 apresenta um modelo para este exemplo.

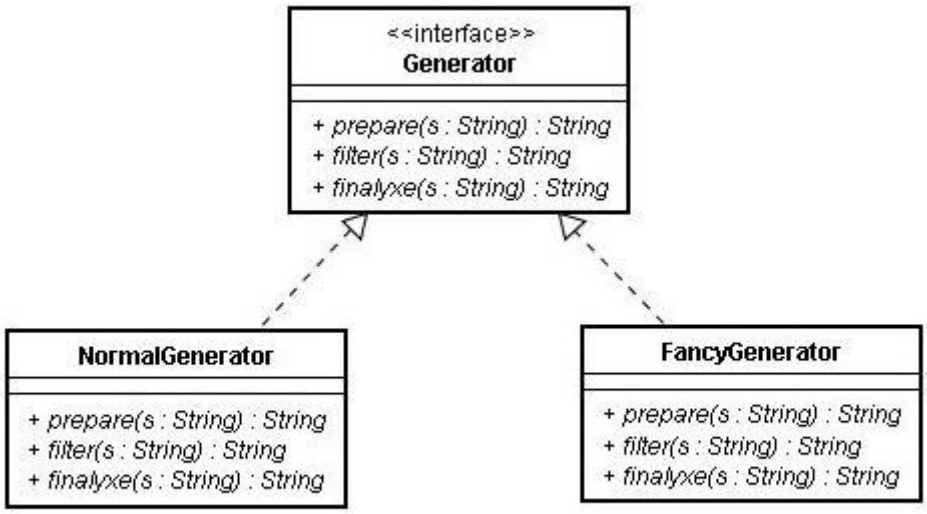

Figura 3. Diagrama de Classes do Exemplo Template Method

Visto que, estas classes implementam uma mesma interface *Generator* temos ainda um aspecto que declara que a interface possui o método *genterate*, assim como o seu comportamento, como apresentado na Listagem 16.

Listagem 16. Aspecto Template Method

```
public aspect Generating {
  public String Generator.generate(String s) {
     s = prepare(s);
     s = filter(s);
     s = finalize(s);
     return s;
  }
}
```

Dessa forma, podemos criar regras de design que declarem a estrutura desse algoritmo. Na Listagem 17 temos o conjunto de regras de design deste exemplo. Definimos as variáveis correspondentes a interface *Generator* e métodos presentes nesta interface. No conjunto de regras, definimos a existência dessa interface, assim como seus métodos. Por fim, declaramos que todas as classes que implementem esta interface não devem ter o método *generate* já que este é declarado no aspecto *Generating*.

Listagem 17. Template Method xpi

```
xpi Template {

  interface i = public interface Generator;
  method m1 = public String generate (String);
  method m2 = public String *.Generator.prepare (String);
  method m3 = public String  *.Generator.filter (String);
  method m4 = public String  *.Generator.finalize (String);
```

```
    rules
      some i;
      m2.in(i);
      m3.in(i);
      m4.in(i);
      all c: class | c.implements(i) => !m1.in(c);
  }
```

Outros estudos revelaram padrões de projeto com regras que dependem dos aspectos, como o padrão Observer, onde existem métodos e pontos de junção importantes para a funcionalidade dos observadores, que são declarados em aspectos. Porém, nossa linguagem não possui regras ou variáveis sobre aspectos, logo não pudemos verificar estes tipos de regras. Acreditamos que existe uma forma melhor de implementar este padrão.

2.8 Suporte Ferramental

Desenvolvemos uma ferramenta que verifica se o conjunto de regras definido na nossa xpi esta sendo satisfeito. Caso alguma destas regras não for cumprida uma mensagem de erro é exibida mostrando as regras que não estão sendo seguidas. Esta ferramenta foi construída usando JavaCC, um tradutor de uso em aplicações Java. Ele é baseado em uma gramática livre de contexto.

Nossa ferramenta funciona da seguinte maneira, inicialmente devemos definir qual xpi será checada e em qual escopo. Para isso, passamos para o programa como argumentos os nomes do arquivo fonte, contendo a xpi e um arquivo jar que contém o escopo do projeto a ser checado. Dados estes dois parâmetros o tradutor irá ler cada construção da xpi, verificar se ela está de acordo com a sintaxe definida para a linguagem e, caso esteja, criar e armazenar um conjunto de variáveis e regras, do contrário o analizador indica que houve um erro de sintaxe.

Dado este conjunto de regras, iteramos cada regra checando no arquivo jar se ela está de acordo com o especificado pela xpi, isto é feito através da API Design Wizard, ele extraí informações do código fonte e compara estas com as especificadas em nossa xpi. Ao fim desta iteração, temos como resultado mensagens mostrando cada regra que foi checada, se ela está em conformidade com sua descrição e, por fim, se todo o projeto está de acordo com a xpi.

A Figura 4 apresenta um modelo de como nossa ferramenta funciona. Onde fornecemos o conjunto de regras de design e o arquivo de extensão jar que será checado, estas regras são então traduzidas usando o JavaCC gerando um conjunto de regras a ser checadas e, por fim, checamos se o projeto segue cada uma destas regras.

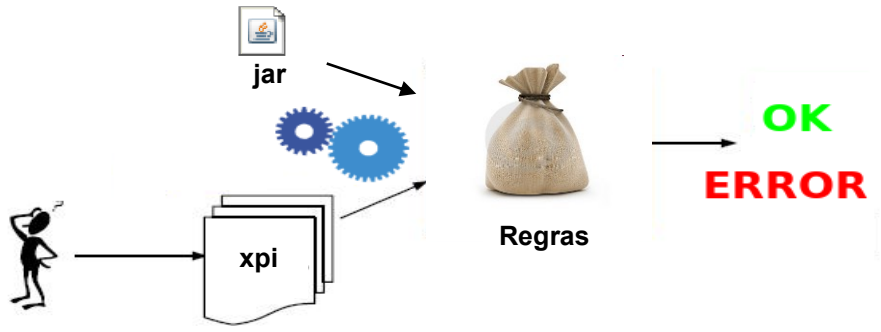

Figura 4. Ferramenta e Checagem de Regras

3. CONCLUSÕES

Neste artigo apresentamos uma linguagem de especificação de regras de design para o paradigma orientado a aspectos, declarando os componentes necessários para um desenvolvimento em paralelo entre classes e aspectos. Dentre as vantagens de nossa abordagem podemos citar o uso de operadores lógicos e quantificação universal. Esta linguagem também possui um suporte ferramental para a checagem destas regras, desta forma facilitando o desenvolvimento e validação de projetos OA e diminuindo as dependências entre aspectos e classes.

Dentre os benefícios desta linguagem ressaltamos a facilidade com a qual as regras são declaradas, onde uma mesma variável pode ser utilizada para diferentes regras. Uma semântica próxima a lógica de primeira ordem, especificando quais os qualificadores as variáveis devem possuir, o uso de negações,

conjunções e disjunções para a criação de regras. O uso de quantificadores universais também é outra vantagem da linguagem, onde podemos definir de forma simples e prática que um conjunto de variáveis deve seguir uma regra, aumentando a expressividade da linguagem. Avaliamos nossa abordagem através dos padrões de projeto Singleton e Template Method implementados em AspectJ.

3.1 Trabalhos Relacionados

Sullivan *et al*. (2005) inicialmente propuseram que as regras de design fossem criadas usando a linguagem natural, mas a partir desta proposta não teríamos garantias que estas regras fossem seguidas além disso, esta solução poderia levar a regras ambíguas e sem um suporte ferramental este trabalho estaria sucetível a muitas falhas. Nossa solução propõe uma linguagem para declarar regras de design, deste modo às regras possuem uma semântica própria e clara.

Griswold *et al*. (2006) usaram as construções de AspectJ *error* e *warning* para especificar regras de design através de uma interface para programas entre - cortantes (XPI). Mas, mesmo sendo possível checar estas regras, temos um conjunto de regras muito limitado já que este não é o propósito de AspectJ. Por exemplo, podemos declarar que um determinado método só pode ser chamado dentro do corpo de outro de outro método, mas não podemos definir a que classe este método pertence. Estas regras não eram suficientemente expressivas e era necessário um grande conhecimento de AspectJ para especificá-las. Como utilizamos uma linguagem própria para especificar estas regras, não possuímos este tipo de limitação e, apesar de não termos declarações sobre aspectos, podemos expressar um conjunto maior de regras OO em um escopo bem definido.

Dósea *et al*. (2007) propuseram uma linguagem própria (dr) para a especificação de regras de design. A sua solução declara na dr os componentes essências que deveriam estar presentes no projeto para as classes e aspectos. Apesar de sua linguagem ser de fácil compreensão, pois possui uma sintaxe semelhante a Java e AspectJ, diferente da nossa linguagem que é próxima a lógica, ela não possui uma semântica clara e não pode expressar qualquer tipo de quantificação. Por exemplo, declarações não limitavam qualificadores como visibilidade, se um método é ou não estático, não sabemos quando um método não possui parâmetros, dentre outros. No seu exemplo de dr para as figuras geométricas não é possível identificar se o método atualizar é ou não estático. A nossa linguagem provê este suporte ferramental, checando se as regras de design são seguidas. Nossa linguagem possui uma semântica mais clara já que é próxima a lógica de primeira ordem, onde declaramos exatamente quais qualificadores às variáveis devem possuir.

Neto *et al*. (2009) apresentaram uma linguagem de descrição de regras de design (dr) com uma semântica bem definida. Para isso, eles utilizaram Alloy (JACKSON 2006), uma linguagem de modelagem baseada em lógica de primeira ordem, e definiram uma semântica de tradução de suas drs para modelos em Alloy. Além disso, as regras de design podem ser checadas utilizando o JastAdd.

AGRADECIMENTOS

Gostaríamos de agradecer aos doutores a Ayla Débora a Tiago Massoni, pela revisão dos estudos de caso e a Alberto Costa Neto pela colaboração e trabalho aceito no SBLP. Agradecemos também ao CNPq por ter financiado este projeto, este trabalho foi apoiado pelo Instituto Nacional de Ciência e Tecnologia para Engenharia de Software (INES[3]), financiado pelo CNPq.

REFERÊNCIAS BIBLIOGRÁFICAS

Aldrich, J. (2005). **Open modules: Modular reasoning about advice**. In ECOOP '05: Proceedings of 19 th European Conference on Object-Oriented Programming, pages 144–168. Springer.

Avgustinov, P., Christensen, A. S., Hendren, L., Kuzins, S., Lhotak, J., Lhotak, O., de Moor, O., Sereni, D., Sittampalam, G., e Tibble, J. (2005). **abc: an extensible aspectj compiler**. In AOSD '05: Proceedings of the 4th international conference on Aspect-oriented software development, pages 87–98, New York, NY, US. ACM.

Clifton, C. e Leavens, G. T. (2002). **Observers and assistants: A proposal for modular aspect-oriented reasoning**. In Foundations of Aspect Languages, pages 33–44.

[3] www.ines.org.br

Dosea, M., Neto, A. C., Borba, P., and Soares, S. (2007). **Specifying design rules in aspect-oriented systems**. In First LA-WASP, Jo˜ao Pessoa, Brazil.

Ekman, T. and Hedin, G. (2007). **The jastadd extensible java compiler**. In OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications, pages 1–18, New York, NY, US. ACM.

Griswold, W. G., Sullivan, K., Song, Y., Shonle, M., Tewari, N., Cai, Y., and Rajan, H. (2006). Modular **software design with crosscutting interfaces**. IEEE Software, 23(1):51–60.

Jackson, D. (2006). **Software Abstractions: Logic, Language and Analysis**. MIT press. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In ECOOP: Proceedings of the European Conference on Object-Oriented Programming.

Lopes, C. and Bajracharya, S. (2006). **Assessing Aspect Modularizations Using Design Structure Matrix and Net Option Value**. LNCS Transactions on Aspect-Oriented Software Development I, pages 1–35.

Morgan, C., Volder, K. D., and Wohlstadter, E. (2007). **A static aspect language for checking design rules.** In AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development, pages 63–72, New York, NY, USA. ACM
Press.

Parnas, D. L. (1972). **On the criteria to be used in decomposing systems into modules**. Commun. ACM, 15(12):1053–1058.

Ribeiro, M., D´osea, M., Bonif´acio, R., Neto, A. C., Borba, P., , and Soares, S. (2007). **Analyzing class and crosscutting modularity with design structure matrixes**. In SBES '07: Proceedings of 21 th Brazilian Symposium on Software Engineering, pages 167– 181.

Steimann, F. (2006). **The paradoxical success of aspect-oriented programming**. SIGPLAN Not., 41(10):481–497.

Sullivan, K., Griswold, W. G., Song, Y., Cai, Y., Shonle, M., Tewari, N., and Rajan, H. (2005). **Information hiding interfaces for aspect-oriented design**. In ESEC/FSE'05, pages 166–175, New York, NY, USA. ACM.

G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. Griswold. **Getting started with AspectJ.** Communications of the ACM, 44(10):59-65, 2001.