Extensão da *UML* para Modelagem Orientada a Aspectos Baseada em AspectJ

Igor Steinmacher, José Valdeni de Lima

UFRGS – Universidade Federal do Rio Grande do Sul – Instituto de Informática
Caixa Postal 15.064 – 91.501-970 – Porto Alegre – RS – Brasil
{ifsteinmacher, valdeni}@inf.ufrgs.br

Resumo

A Programação Orientada a Aspectos (AOP) visa reduzir algumas limitações encontradas na orientação a objetos, como o espalhamento e entrelaçamento de código. Isto é feito através do encapsulamento das preocupações ortogonais (crosscutting concerns) em módulos chamados aspectos. O uso de um modelo gráfico traria as facilidades da AOP para a fase de modelagem, além de, é claro, facilitar a fase de implementação. A proposta deste trabalho é estender o diagrama de classes da UML para apoiar o desenvolvimento de sistemas orientados a aspectos, com base na linguagem AspectJ.

Palavras-chave: *Programação Orientada a Aspectos, aspectos, modelagem, UML*

Abstract

Aspect Oriented Programming (AOP) focuses on reducing some limits found on object oriented and procedural programming: code crosscutting and code tangling. This is done encapsulating the so called crosscutting concerns in modules called aspects. Using a graphical model could bring the AOP facilities to the design phase and to the implementation. The focus of this paper is to show an extension of the UML class diagram to support the development of Aspect Oriented Systems, based on AspectJ language.

Keywords: *Aspect Oriented Programming, aspects, design, UML*

1. Introdução

O sucesso do paradigma orientado a objetos se deve, principalmente, à preocupação com a modularidade, com o reuso do código e com a facilidade de manutenção de sistemas. Entretanto, este paradigma tem demonstrado algumas limitações [OSS99], como o entrelaçamento (*code tangling*) e o espalhamento de código. Estes problemas tornam a implementação mais complexa, dificultando o entendimento e a manutenção dos sistemas.

Um efeito colateral do "espalhamento" é a necessidade de atualização em vários trechos de código, em caso de manutenção de um sistema.. Isto caracteriza um problema de redundância de

código. Já o entrelaçamento se refere aos trechos de código com diferentes funcionalidades num mesmo módulo, o que pode complicar da mesma forma a manutenção de um sistema.

A Programação Orientada a Aspectos (AOP) [KIC97] reduz estes problemas aumentando a modularidade, através da separação do código que implementa funções diretamente ligadas ao sistema das preocupações ortogonais do sistema. Com esta separação, o código gerado se torna mais compreensível e de mais fácil manutenção.

AspectJ [ASP04] é uma linguagem concebida para fornecer suporte à programação orientada a aspectos, por meio de novos elementos capazes de encapsular as preocupações ortogonais ao sistema. Por ser uma linguagem com bases firmadas na AOP, é tomada como base em qualquer apresentação e exemplificação deste paradigma. Entretanto, a utilização desta ferramenta esbarra na falta de uma notação gráfica adequada. Uma modelagem gráfica não só facilitaria a implementação de sistemas, mas ajudaria no entendimento e mostraria mais claramente os conceitos da orientação a aspectos.

Este trabalho apresenta uma proposta de notação gráfica, baseada em *UML*, [UML04] para apoiar os conceitos do paradigma orientado a aspectos, usando como base a linguagem *AspectJ*. A idéia é utilizar os mecanismos de extensão oferecidos pela *UML* para se criar um *UML profile* que expresse a semântica utilizada na orientação a aspectos, evitando problemas de padronização.

2. UML

UML é uma linguagem de modelagem para especificar, visualizar, construir e documentar os artefatos de um processo de software [ZAK02]. A *UML* possui mecanismos que permitem estendê-la de forma consistente, evitando assim situações em que os seus princípios e elementos sejam desobedecidos. Esses mecanismos permitem a introdução de novos modelos com uma maior expressividade, definindo extensões específicas das linguagens de implementação ou específicas dos processos de desenvolvimento e associando informações semânticas aos elementos do modelo.

Os mecanismos de extensibilidade da *UML* especificam como os elementos do modelo *UML* podem ser personalizados e estendidos, adicionando-se nova semântica através do uso de *stereotypes*, *constraints* e *tagged values*. Um conjunto coerente dessas extensões é chamado de *UML profile*.

Stereotype permite a classificação de um elemento de modelo estendido de acordo com um elemento de modelo base já existente na *UML* e a definição de novas restrições, além de uma nova representação gráfica [CAV02]. Todos os elementos estendidos do modelo que possuírem um ou

mais *stereotypes* terão os valores adicionais, restrições definidas pelos estereótipos, além dos atributos, associações e super classes que o elemento já possui na *UML*.

Constraints são regras de restrições bem definidas. Uma *constraint* permite que uma nova especificação semântica seja atribuída ao elemento de modelo. Tal especificação é escrita numa determinada linguagem de restrições, que pode ser mais ou menos formal. Uma *constraint* pode ser adicionada tanto a um elemento de modelo diretamente quanto ao *stereotype*, neste caso sendo aplicada a todos os elementos estendidos por esse *stereotype*.

Um *tagged value* permite explicitar uma certa propriedade de um elemento através de um par: nome, valor. Essas informações podem ser adicionadas arbitrariamente a qualquer elemento do modelo, podendo ser determinado pelo utilizador ou por convenções das ferramentas de suporte.

3. AspectJ

AspectJ é uma linguagem de aspectos *Open Source* de propósito geral que estende a linguagem *Java* para suportar a programação orientada a aspectos. Esta linguagem provê uma série de construções que permitem a definição de aspectos e pontos de combinação (*join points*). Estes últimos tratam-se de pontos onde os aspectos devem ser entrelaçados ao código de componentes. *AspectJ* contém um pré-processador cuja função é combinar o código de aspectos com as classes do código de componentes. Esta combinação ocorre de acordo com os pontos de combinação especificados no código de aspectos.

A seguir são apresentadas as construções básicas apresentadas pelo *AspectJ* e que dão suporte à orientação a aspectos. Vale lembrar que em *AspectJ* os aspectos são definidos por declarações que têm forma similar às declarações de uma classe. As declarações de aspectos podem incluir *pointcuts, advices*, além de outros tipos de declarações permitidas nas declarações de classes, como métodos e atributos. Ainda é possível utilizar os conceitos de herança introduzidos pelo paradigma OO [STE03].

Pointcut se trata de um conjunto de pontos de combinação unidos através de operadores lógicos. São utilizados para indicar em que pontos do código de componentes uma determinada preocupação deve ser executada. Abaixo um trecho de código contendo um *pointcut* que identifica as chamadas aos métodos setP1 e setP2 da classe Linha:

```
pointcut Move():
call(void Linha.setP1 (Point)) ||
call(void Linha.setP2 (Point));
```

Um *advice* é uma construção que define o código adicional a ser inserido nos pontos de combinação. Cabe ao desenvolvedor definir o momento em que o *advice* deve ser disparado com relação ao ponto de combinação (antes, depois ou durante). O *advice* a seguir define que o valor *true* será atribuído a variável *flag* depois (*after*) da execução do *pointcut* Move().

```
after(): Move()   { flag = true }
```

4. Extensão da UML para Modelagem Orientada a Aspectos

Os mecanismos de extensão da *UML* são baseados principalmente nos conceitos de *stereotypes,* que provêm um meio de classificar elementos de forma que eles se comportem como se fossem instâncias do novo meta-modelo [ALD01]. No presente trabalho, serão utilizados *stereotypes* para identificar aspectos, *pointcut*s e *advices*, estendendo linguagem *UML*.

Inicialmente será definida a estrutura de uma classe cujo comportamento é alterado por um aspecto. A proposta é a criação de uma estrutura não definida nos mecanismos de extensão padrões propostos pela *UML* contendo definições de atributos, métodos e aspectos relacionados àquela classe [STE03]. Nesta estrutura aparecerão todos os aspectos que interferem em algum ponto desta classe (i.e., possuem um ponto de combinação que aponta para algum "evento" da classe).

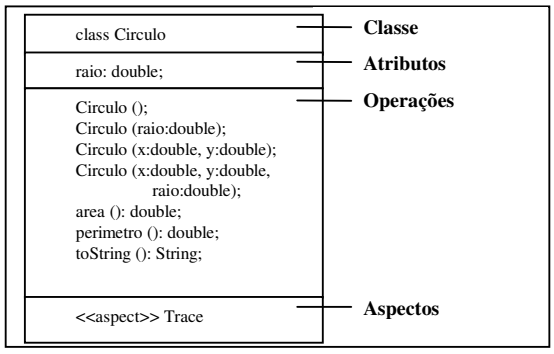

Figura 4.1 – Representando uma classe com o campo *aspect*

A Figura 4.1 mostra a estrutura proposta para a definição de uma classe. O uso da célula contendo os aspectos relacionados à classe é proposto para que não seja necessário utilizar um relacionamento entre classes e aspectos. A utilização destes relacionamentos aumenta a complexidade do modelo em sistemas de grande porte, fugindo do propósito da AOP, que busca facilitar o entendimento do sistema.

Como dito anteriormente, aspectos em *AspectJ* são semelhantes às classes *Java*. Neste sentido um aspecto será representado como uma classe, sendo apenas decorado com o *stereotype* <<aspect>>, como pode-se observar na Figura 4.2.

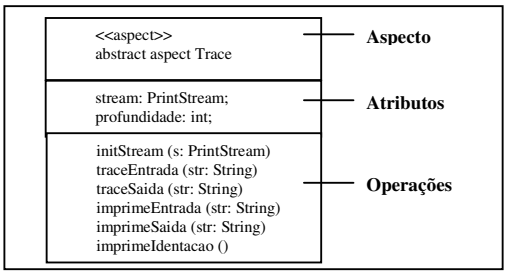

Figura 4.2 – Representação de um aspecto

Agora surge a questão: onde ficariam os *pointcuts* e os *advices*? Para modelar estas duas construções propõe-se a criação de uma estrutura que contenha ambas, e estas se relacionem com o aspecto da qual fazem parte. A estrutura citada também utiliza como base o elemento classe da *UML* para modelar um *pointcut*, sendo este decorado com o *stereotype* <<*pointcut*>>. Ao invés de modelar atributos deste *pointcut* utilizar-se-á o *stereotype* <<*joinpoint*>> para mostrar o conjunto de pontos de combinação pertencentes a cada *pointcut,* reduzindo a possibilidade de existirem diferentes interpretações dos *pointcuts*. E, por último, os *advices* referentes a cada *pointcut* são definidos dentro da mesma estrutura, sendo estes identificados pelo *stereotype* <<*advice*>>. *Pointcuts,* pontos de combinação e *pointcuts* são mantidos em uma mesma estrutura trazendo para a fase de modelagem a modularidade proposta pela AOP.

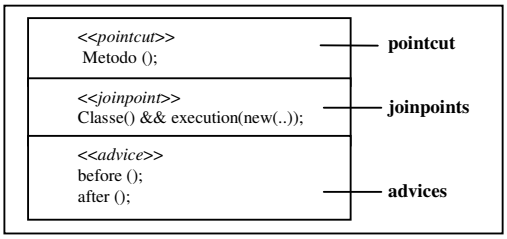

Figura 4.3 – Representação de um *pointcut* com seus pontos de combinação e *advices*

A Figura 4.3 mostra a proposta de representação de um *pointcut* estendendo-se o elemento classe da *UML*. O entendimento se torna fácil, alterações, quando necessárias, são realizadas de maneira a não comprometer outras estruturas do modelo. Para relacionar os *pointcuts* aos respectivos aspectos se fez uso de relacionamentos de generalização. Estes relacionamentos terão

como função demonstrar quais *pointcuts* estão relacionados a um determinado aspecto. Pode-se visualizar a utilização dos relacionamentos na Figura 4.4. Este tipo de utilização não compromete a funcionalidade dos mesmos e mantém a padronização UML.

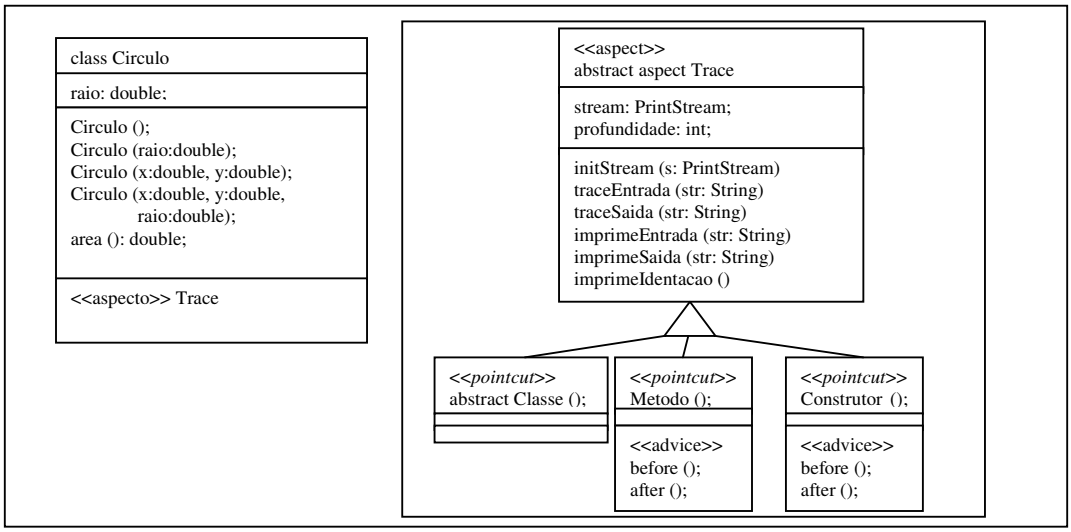

Figura 4.4 – Relacionando Classes, Aspectos e *Pointcuts*

5. Trabalhos Relacionados

A busca por um padrão de modelagem para a programação orientada a aspectos tem sido foco de muitas pesquisas. A seguir são destacadas as propostas de modelos de Siobhán Clarke [CLA02a] [CLA02b], Zakaria, Hosny e Zeid [ZAK02] e Stein, Hanenberg e Unland [STE02].

5.1. Abordagem de Clarke

É a abordagem mais bem fundamentada e mais aceita por parte da comunidade que utiliza orientação a aspectos. Clarke estende a *UML* através de um novo conceito de modelagem, os chamados *composition patterns*. Estes *patterns* são na verdade *templates* de pacotes que utilizam classes e operações através de relacionamentos de composição especiais. *Composition patterns* são baseados numa notação especial para a *subject oriented programming*, chamada *subject oriented design model.*

Como pode-se perceber a abordagem de Clarke foi desenvolvida especificamente para modelar sistemas baseados na *subject oriented programming*[OSS99], mas a autora busca demonstrar como os *composition patterns* podem ser utilizados para modelar programas orientados a aspectos. Clarke [CLA02a] mostra como implementar um sistema em *AspectJ*, utilizando uma

modelagem que utiliza *composition patterns*. É possível entender a implementação a partir da abordagem, mas a maneira proposta não é compatível com a semântica do *AspectJ*. Um *advice* em *AspectJ* é executado dentro do escopo do *aspect* correspondente, enquanto utilizando a abstração proposta por Clarke é executado dentro do escopo da classe correspondente. Esta abordagem se diferencia da proposta deste trabalho pelo fato de utilizar *templates* UML em sua modelagem.

5.2. Abordagem de Zakaria, Hosny e Zeid

Esta abordagem utiliza como base as estruturas da linguagem *AspectJ*. Os autores a utilizam a proposta de Aldawud, Elrad e Bader [ALD01], em que os mecanismos de extensão da *UML* são baseados nos *stereotypes*. É utilizada uma nova notação gráfica para representar aspectos e *pointcuts* como extensões do meta-modelo de classes *UML*. Os relacionamentos entre classes e aspectos são feitos através de relacionamentos de associação. Para tal é criado um conjunto de *tags* que identificam o tipo do relacionamento.

Esta proposta é muito bem fundamentada, e possui algumas facilidades para implementação em *AspectJ*, uma vez que traz todas as estruturas suportadas pela linguagem modeladas, mas peca em alguns sentidos. O número de *tags* introduzidas por esta proposta é excessivo, somente nas associações aspecto-classe, existem oito novas *tags*. Os relacionamentos entre classes e aspectos podem aumentar a complexidade em sistemas de maior porte. No caso desta proposta, tem-se um problema ainda maior, pois é criado um relacionamento aspecto-classe, um relacionamento aspecto-*pointcut* e um relacionamento classe-*pointcut* para cada ponto de combinação.

5.3. Abordagem de Stein, Hanenberg e Unland

Esta abordagem também é baseada nas construções introduzidas pelo *AspectJ*, sendo a que mais se aproxima de sua semântica. Nela é criada uma notação única para esta linguagem, tomando como base todos os elementos descritos na especificação da linguagem. Todos os novos elementos criados têm base nos *stereotypes*, estendendo os elementos classe da *UML*.

Para representar os aspectos é utilizado o *stereotype* <<*aspect*>> seguido do nome do aspecto. Dentro desta estrutura pode-se encontrar as estruturas básicas de uma classe (atributos e operações). Os atributos funcionariam normalmente, visto a possibilidade de um aspecto em *AspectJ* suportar a existência de atributos. Dentro do espaço reservado para as operações encontram-se os *pointcuts* e os *advices*. Os *pointcuts* são decorados com o *stereotype* <<*pointcut*>> e são declarados como um método normal, exceto pela existência de restrições. A principal dessas restrições diz respeito aos pontos de combinação. Os *advices* também são decorados, mas com o *stereotype* <<*advice*>> e são declarados logo após os *pointcuts*.

Esta abordagem foi utilizada como base para a proposta deste trabalho, pois consegue se aproximar muito da semântica do *AspectJ*, trazendo para a fase de modelagem as facilidades de modularidade da programação orientada a aspectos. Apesar disso, podem ser citados alguns pontos que não foram aceitos na criação da abordagem mostrada neste trabalho. A utilização de *pointcuts* e *advices* dentro da estrutura do aspecto pode causar uma certa dificuldade ao desenvolvedor. Esta dificuldade pode ser encontrada quando for necessária a alteração de algum *pointcut* ou *advice*, o que dificultaria a busca e alteraria a estrutura de todo o aspecto.

6. Estudo de Caso

O modelo proposto neste trabalho foi aplicado a um estudo de caso, afim de validá-lo. O aspecto escolhido para tal estudo foi o *tracing*. Um sistema simples de cálculo de medidas de figuras geométricas foi utilizado para a realização deste *tracing.*

Inicialmente utilizou-se UML para modelar classes, atributos e métodos do sistema escolhido. A seguir realizou-se a implementação utilizando-a linguagem *Java.* Em seguida foram localizados os pontos a serem mapeados pelo *tracing*. Feito isso foram modeladas e implementadas as rotinas de *tracing*, sem utilização de aspectos.

Os mesmos pontos mapeados na implementação orientada a objetos foram utilizados na implementação do aspecto de *tracing*. Utilizou-se o diagrama de classes (antes da inclusão das rotinas de *tracing*) do sistema orientado a objetos para inclusão do aspecto. O aspecto, assim como os *joinpoints* e *advices*, foram modelados e, em seguida, mapeados nas classes relacionadas a eles. Realizou-se, então, a implementação do aspecto modelado utilizando *AspectJ.* A seguir utilizou-se o sistema com a inclusão do aspecto. Tal modelagem e a implementação dos aspectos e de uma das classes são apresentados, respectivamente, nas Figuras 6.1 e 6.2. É possível verificar a proximidade existente entre a modelagem e o código implementado, sem a necessidade de excesso de conexões entre classes e aspectos. O modelo fica muito mais "limpo", refletindo o que acontece no código.

Como conclusão, pode-se dizer que o estudo de caso foi concluído com êxito, e pôde demonstrar a facilidade de entendimento do modelo quando da implementação do sistema. A alteração realizada no diagrama de classes não trouxe modificações às classes do mesmo. A forma com que foi modelado o aspecto, mostrou claramente o desacoplamento deste com relação às classes nas quais ele interfere. Através da implementação sem aspectos e, em seguida, com aspectos, foi possível perceber as facilidades e a limpeza no código introduzida pela orientação a aspectos. Apesar destes pontos ainda se faz necessária a aplicação deste modelo em sistemas de maior porte que possam apontar possíveis falhas.

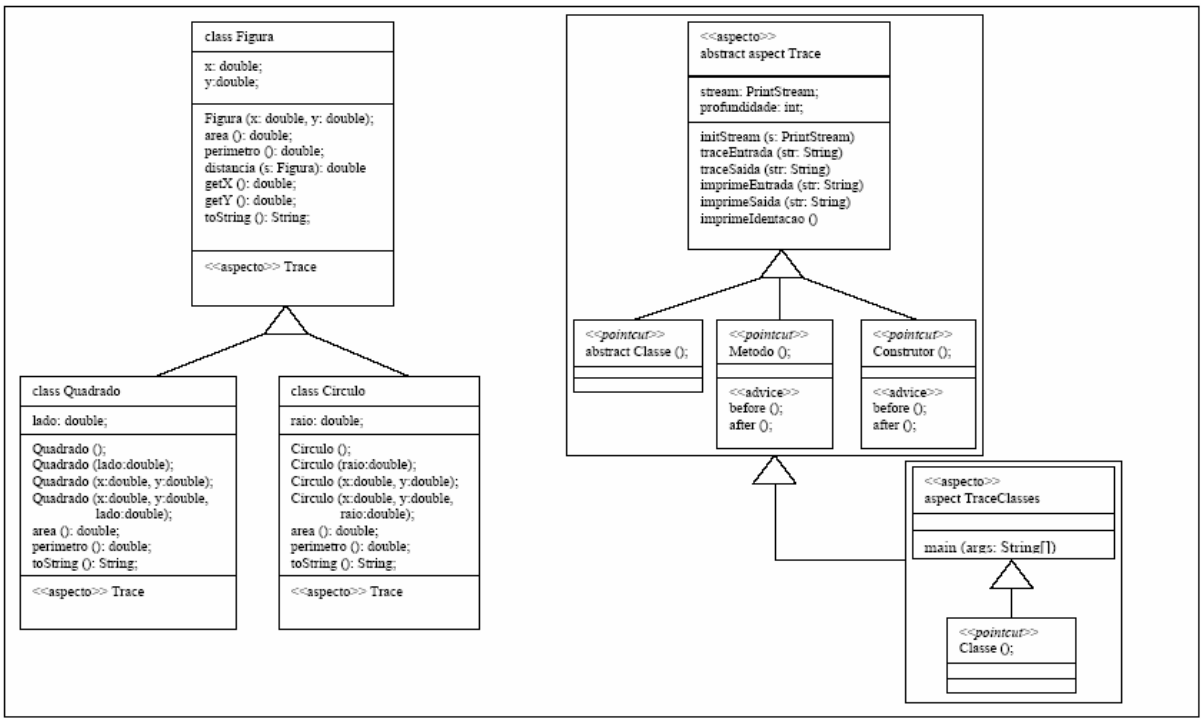

Figura 6.1 – Modelagem do estudo de caso: um sistema com *Tracing*.

```
abstract aspect Trace {
    protected static PrintStream stream;
    protected static int profundidade;
    public static void initStream(PrintStream s) {
        stream = s;
    }
/* PointCuts */
    abstract pointcut Classe();
    pointcut Construtor(): Classe() && execution(new(..));
    pointcut Metodo(): Classe() && execution(* *(..));
/*advices*/
    before(): Construtor() {
        traceEntrada("" + thisJoinPointStaticPart.getSignature());
    }
    after():  Construtor() {
        traceSaida("" + thisJoinPointStaticPart.getSignature());
    }
    before(): Metodo() {
        traceEntrada("" + thisJoinPointStaticPart.getSignature());
    }
    after(): Metodo() {
        traceSaida("" + thisJoinPointStaticPart.getSignature());
    }
}
```

(a)

```
public aspect TraceClasses extends Trace {

    pointcut Classe():  within(Figura)  ||
                        within(Circulo) ||
                        within(Quadrado);
    public static void main(String[] args) {
        Trace.initStream(System.out);
        ExampleMain.main(args);
    }
}
```

(b)

(c)

```
public class Circulo extends Figura {
    protected double raio;
    public Circulo(double x, double y, double raio) {
        super(x, y);
        this.raio  = raio;
    }
    public Circulo(double x, double y) {
        this(x, y, 1.0);
    }
    public Circulo(double raio) {
        this(0.0, 0.0, raio);
    }
    public Circulo() {
        this(0.0, 0.0, 1.0);
    }
    public double perimetro() {
        return 2 * Math.PI * raio;
    }
    public double area() {
        return Math.PI * raio*raio;
    }
    public String toString() {
        return ("Raio do Circulo = " + String.valueOf(raio) + super.toString());
    }
}
```

Figura 6.2 – Implementação dos aspectos *Trace* (a) e *TraceClass* (b) modelados na Figura 6.1 e da classe Circulo (c).

7. Conclusões

Este trabalho apresenta uma proposta de modelagem que suporte a AOP, baseando-se na semântica do *AspectJ*. Com ela a tarefa de se modelar um aspecto ficou muito mais próxima da implementação dos mesmos, facilitando o trabalho do programador. A proposta mostra claramente o desacoplamento existente entre classes e aspectos, sendo desnecessária a utilização de relacionamentos entre estes. Outro ponto a ser considerado é a facilidade de se lidar com os *pointcuts*, uma vez que estes são modelados através de uma estrutura, que se associa ao aspecto através de relacionamentos (herança). O grande diferencial introduzido pela proposta é a forma clara de se modelar aspectos graficamente, excluindo-se a necessidade de relacionamentos entre classes e aspectos, reduzindo a complexidade dos modelos.

O modelo foi aplicado a um estudo de caso simples, no qual se "encapsulou" o aspecto de *tracing*. Verificou-se a diminuição da mão-de-obra na implementação de um sistema contendo este aspecto utilizando-se *AspectJ*. O estudo mostrou ainda a clareza com que o modelo apresenta o desacoplamento real existente entre aspectos e classes, a modularidade e a facilidade de lidar com aspectos.

Não foram encontrados problemas a primeira vista, mas é necessária a aplicação de outros estudos de casos de maior porte, e estudos de caso que envolvam pessoas alheias ao desenvolvimento da extensão apresentada.

7. Referências Bibliográficas

[ALD01] ALDAWUD, Omar, ELRAD, Tzilla and BADER, Atef, "A *UML* Profile for Aspect Oriented Modeling", *OOPSLA 2001 - Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, Tampa Bay, Florida - Outubro, 2001

[ASP04] *ASPECTJ*, *AspectJ* Home Page, disponível na Internet, http://www.aspectj.org, 24/07/2004

[CAV02**]** CAVALCANTI, Marcus R., MACHADO, Maria L., "Uma comparação na extensibilidade de Meta-Modelos", *Mestrado IM/NCE – UFRJ*, Rio de Janeiro,2002

[CHA02] CHAVEZ, Cristina, LUCENA, Carlos, "A Metamodel for Aspect-Oriented Modeling", *First International Conference on Aspect-Oriented Software Development*, Enschede, Holanda, Abril, 2002.

[CLA02a] CLARKE, Siobhán, WALKER, Robert J., "Towards a Standard Design Language for AOSD", *First International Conference on Aspect-Oriented Software Development*, Enschede, Holanda - Abril , 2002

[CLA02b] CLARKE, Siobhán, "Extending standard *UML* with model composition semantics", *Science of Computer Programming*, Volume 44, pag. 71-100 – Julho, 2002

[ELR01a] ELRAD, Tzilla, FILMAN, Robert E., BADER, Atef, "Aspect-Oriented Programming", *Communications of the ACM*, Vol. 44 – Outubro, 2001

[KIC97] KICKZALES, Gregor, IRWIN, John, LAMPING, John, MENDHEKAR, Anurag, MAEDA, Chris, LOPES, Cristina Videira, LOINGTIER, Jean-Marc. "Aspect-Oriented Programming", *ECOOP'97*, Finlândia – 1997

[KIC01a] KICKZALES, Gregor, HILSDALE, Erik, HUGUNIN, Jim, KERSTEN, Mik, PALM, Jeffrey, GRISWOLD, William,, "An Overview of AspectJ", *ECOOP'01* – 2001

[KIC01b] KICKZALES, Gregor, HILSDALE, Erik, HUGUNIN, Jim, KERSTEN, Mik, PALM, Jeffrey, GRISWOLD, William,, "Getting Started with AspectJ", *Communications of the ACM*, Vol. 44 – Outubro 2001

[MUR01] MURPHY, Gail C., BANIASSAD, Elisa L. A., ROBILLARD, Martin P., LAI, Albert, KERSTEN, Mik A., "Does Aspect-Oriented Programming Work?", *Communications of the ACM*, Vol. 44 – Outubro 2001

[OSS99] OSSHER, Harold and TARR, Peri, "Using subject–oriented programming to overcome common problems in object–oriented software development/evolution", *International Conference on Software Engineerin'99*, pages 698–688. ACM – 1999

[SOA02] SOARES, Sérgio, BORBA, Paulo, "*AspectJ* - Programação orientada a aspectos em *Java*", *SBLP2002 – Simpósio Brasileiro de Linguagem de Programação*, Rio de Janeiro – 2002

[STE02] STEIN, Dominik, HANENBERG, Stefan, UNLAND, Rainer, "A *UML*-based Aspect-Oriented Design Notation For *AspectJ*", *First International Conference on Aspect-Oriented Software Development* - Enschede, Holanda – April, 2002.

[STE03] STEINMACHER, Igor Fábio, GIMENES, Itana Maria de Souza, "Estudo de Princípios para Modelagem Orientada a Aspectos", *Trabalho de Graduação TG17-02 – Universidade Estadual de Maringá,* Brasil – Abril de 2003

[UML04] UML HomePage, disponível na Internet, www.uml.org, 24/07/2004

[ZAK02] ZAKARIA, Aida Atef, HOSNY, Hoda , ZEID, Amir, "A *UML* Extension for Modeling Aspect-Oriented Systems", *Aspect-Oriented Modeling with AOP as part of the AOSD*, Enschede, Holanda – Abril de 2002