AspectJ — Programação orientada a aspectos em Java

Sérgio Soares[1]*, **Paulo Borba**[1]†

[1]Centro de Informática
Universidade Federal de Pernambuco
Caixa Postal 7851 – CEP 50740-540 Recife, PE

scbs,phmb@cin.ufpe.br

Abstract. *This document presents a tutorial about AspectJ, an aspect-oriented extension to Java. Aspect-oriented programming (AOP) tries to solve some inefficiency of object-orientation, such as tangled code and code spread over several units, making harder software development and maintainability. AOP increases system modularity by separating code that implements specific functions, affecting different parts of the system, called crosscutting concerns. We present the main constructions of AspectJ, as well as examples of aspects to assist the assimilation of the concepts. We also discuss using design patterns to implement some features of AspectJ, and its benefits and liabilities.*

Resumo. *Este documento apresenta um tutorial sobre AspectJ, uma extensão orientada a aspectos de Java. Programação orientada a aspectos (AOP) procura solucionar algumas ineficiências da orientação a objetos, como o entrelaçamento e espalhamento de código com diferentes propósitos. Este entrelaçamento e espalhamento tornam o desenvolvimento e a manutenção destes sistemas extremamente difícil. AOP aumenta a modularidade separando código que implementa funções específicas, afetando diferentes partes do sistema, chamadas preocupações ortogonais (*crosscutting concern*). Nós apresentamos as principais construções de AspectJ, bem como exemplos de aspectos para auxiliar a assimilação dos conceitos. Também discutimos o uso de padrões de projetos para implementar algumas características de AspectJ, e discutimos suas vantagens e desvantagens.*

1. Introdução

A necessidade de desenvolver software de qualidade aumentou o uso da orientação a objetos [Meyer, 1997, Booch, 1994] em busca de maiores níveis de reuso e manutenibilidade, aumentando a produtividade do desenvolvimento e o suporte a mudanças de requisitos. Entretanto, o paradigma orientado a objetos tem algumas limitações [Ossher et al., 1996, Ossher and Tarr, 1999], como o entrelaçamento e o espalhamento de código com diferentes propósitos, por exemplo, o entrelaçamento de código de negócio com código de apresentação, e o espalhamento de código de acesso a dados em vários módulos do sistema. Parte destas limitações podem ser compensadas com o uso de padrões de projetos [Buschmann et al., 1996, Gamma et al., 1994].

*Apoiado pela CAPES
†Apoiado em parte pelo CNPq, processo 521994/96–9.

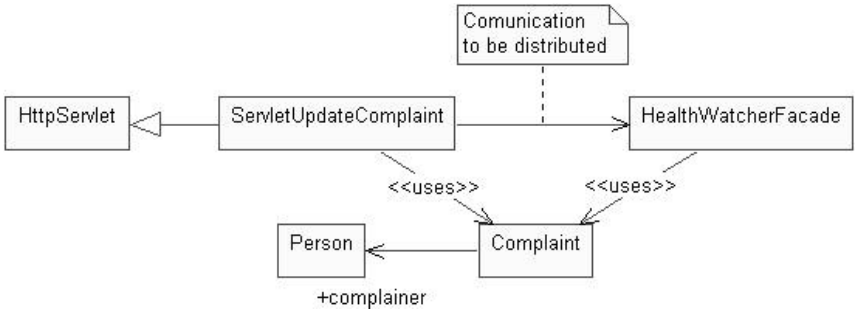

Figura 1: Diagrama de classe do sistema *Health Watcher*.

Por outro lado, extensões do paradigma orientado a objetos, como *aspect–oriented programming* (programação orientada a aspectos) [Elrad et al., 2001], *subject–oriented programming* [Ossher and Tarr, 1999], e *adaptive programming* [J. et al., 1994], tentam solucionar as limitações do paradigma orientado a objetos. Estas técnicas visam obter uma maior modularidade de software em situações práticas onde a orientação a objetos e padrões de projetos não fornecem suporte adequado.

Uma destas técnicas, programação orientada a aspectos (AOP), parece bastante promissora [Murphy et al., 2001, Elrad et al., 2001, Laddad, 2002]. AOP procura solucionar a ineficiência em capturar algumas das importantes decisões de projeto que um sistema deve implementar. Esta dificuldade faz com que a implementação destas decisões de projeto sejam distribuídas pelo código, resultando num entrelaçamento e espalhamento de código com diferentes propósitos. Este entrelaçamento e espalhamento tornam o desenvolvimento e a manutenção destes sistemas extremamente difícil. Desta forma, AOP aumenta a modularidade separando código que implementa funções específicas, afetando diferentes partes do sistema, chamadas preocupações ortogonais (*crosscutting concern*). Exemplos de *crosscutting concerns* são persistência, distribuição, controle de concorrência, tratamento de exceções, e depuração. O aumento da modularidade implica em sistemas legíveis, os quais são mais facilmente projetados e mantidos.

AOP permite que a implementação de um sistema seja separada em requisitos funcionais e não–funcionais. Dos requisitos funcionais resultam um conjunto de componentes expressos em uma linguagem de programação atual, por exemplo, a linguagem Java [Gosling et al., 2000]. Já dos requisitos não–funcionais resultam um conjunto de aspectos (*crosscutting concerns*) relacionados às propriedades que afetam o comportamento do sistema. Com o uso da abordagem, os requisitos não–funcionais podem ser facilmente manipulados sem causar impacto no código de negócio (requisitos funcionais), uma vez que estes códigos não estão entrelaçados e espalhados em várias unidades do sistema. Desta forma, AOP possibilita o desenvolvimento de programas utilizando tais aspectos, o que inclui isolamento, composição e reuso do código de implementação dos aspectos.

2. Um exemplo de *crosscutting concern* — distribuição

Vamos tomar como exemplo um sistema que registra queixas para o sistema público de saúde. O nosso objetivo é distribuir este sistema utilizando RMI [Microsystems, 2001].

A Figura 1 é um diagrama de classes UML [Booch et al., 1999] que mostra parte das classes do sistema *Health Watcher*. No diagrama estão representadas a

classe fachada [Gamma et al., 1994], o ponto único de acesso ao sistema, um Servlet Java [Hunter and Crawford, 1998], que implementa a interface com o usuário, e neste caso é responsável por atualizar informações sobre uma queixa no sistema. Além disso, estão presentes classes que modelam as queixas (`Complaint`) do sistema e o responsável pela queixa (`Person`). A comunicação a ser distribuída é a entre a classe fachada e a interface com o usuário (servlets).

Distribuição sem AOP

Para implementar a distribuição no sistema sem utilizar AOP teremos de alterar uma série de classes. A Figura 2 mostra como o código de distribuição afeta o código fonte das classes do sistema. O código que está selecionado (por retângulos) nestas classes é responsável pela implementação da distribuição com RMI.

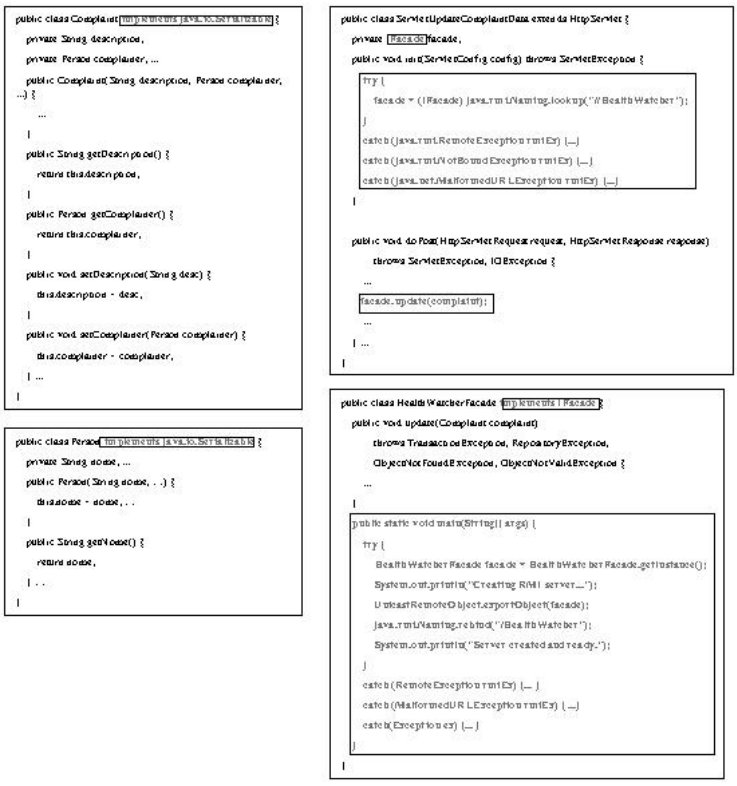

Figura 2: Código fonte do sistema distribuído sem AOP.

Note que o código de distribuição (selecionado por caixas) está entrelaçado com código funcional das classes e está espalhado por várias classes do sistema. Esta falta de modularidade torna o sistema difícil de manter e evoluir. Imagine que queiramos alterar o protocolo de comunicação, por exemplo, para CORBA [Orfali and Harkey, 1998], ou outro protocolo qualquer. Teríamos que alterar diretamente várias classes do sistema.

Distribuição com AOP

Para distribuir o mesmo sistema utilizando AOP, teríamos de definir um aspecto de distribuição que faz as alterações necessárias no sistema, para torná–lo distribuído. Lembre-se que AspectJ pode alterar a estrutura estática e dinâmica das classes de um sistema. A Figura 3 mostra o resultado do uso de AOP.

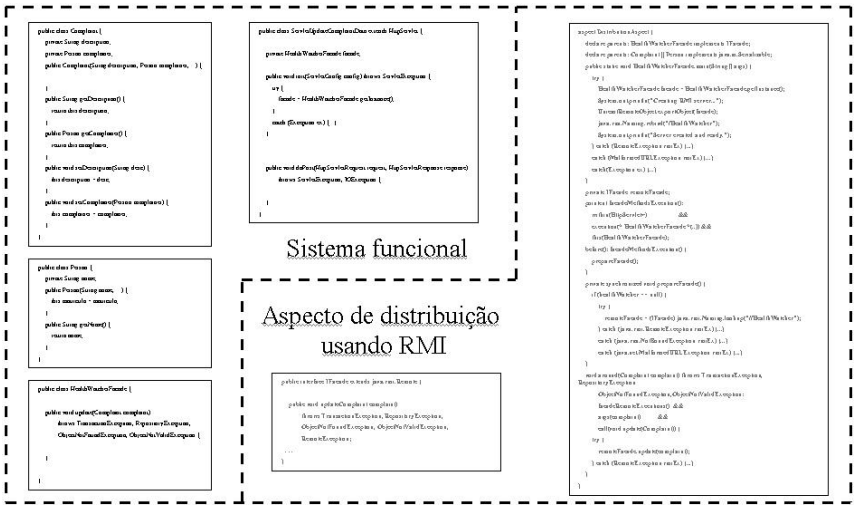

Figura 3: Código fonte do sistema distribuido com AOP.

Note que as classes do sistema foram preservadas. Além de não haver código de distribuição espalhado pelo sistema e entrelaçamento com código funcional, a alteração do sistema não é invasiva, ou seja, não é feita com alteração direta do código fonte. A mesma é feita automaticamente através da composição do sistema com o aspecto definido. Este processo é chamado de *weaving*, ou recomposição aspectual. A Figura 4 mostra o processo de recomposição.

O *weaver* é um compilador que dado o código fonte de um sistema e um conjunto de aspectos gera uma versão do sistema com estes aspectos, como, por exemplo, o aspecto de distribuição. Para tornar o sistema distribuído utilizando outro protocolo, seria necessário implementar outro aspecto que utiliza o protocolo requerido e em seguida utilizar o *weaver* de AspectJ para gerar a versão do sistema distribuída com tal protocolo.

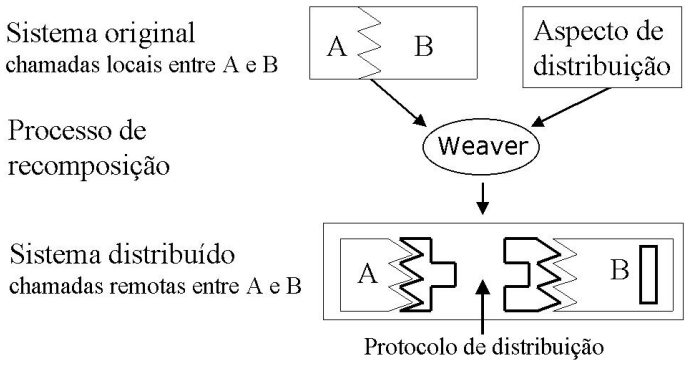

Figura 4: Recomposição aspectual, ou *weaving*.

Desta forma teremos a definição de um aspecto de distribuição que afeta as classes do sistema para implementar a sua distribuição com RMI, como mostrado no diagrama de classes na Figura 5.

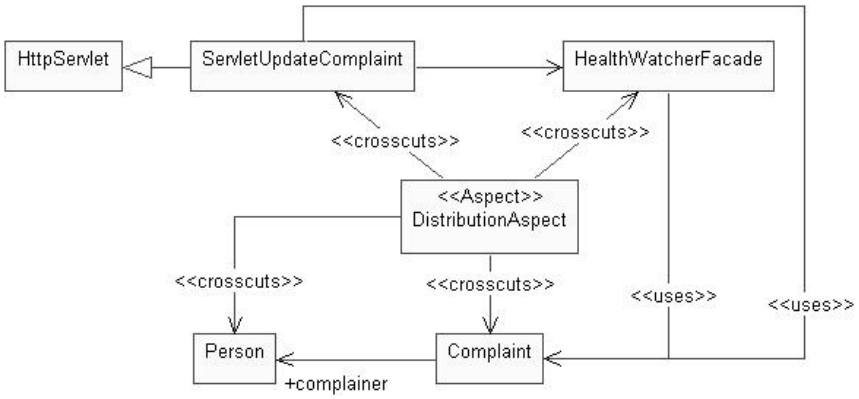

Figura 5: Diagrama de classe do sistema *Health Watcher* levando em conta o aspecto de distribuição.

Observe que o aspecto de distribuição afeta a interface com o usuário, a fachada e os dados que são transmitidos entre eles.

3. AspectJ

Nesta seção apresentamos a linguagem AspectJ [Kiczales et al., 2001], uma extensão orientada a aspectos, de propósito geral, da linguagem Java [Gosling et al., 2000].

3.1. Anatomia de um aspecto

A principal construção em AspectJ é um aspecto. Cada aspecto define uma função específica que pode afetar várias partes de um sistema, como, por exemplo, distribuição. Um aspecto, como uma classe Java, pode definir membros (atributos e métodos) e uma hierarquia de aspectos, através da definição de aspectos especializados.

Aspectos podem alterar a estrutura estática de um sistema adicionando membros (atributos, métodos e construtores) a uma classe, alterando a hierarquia do sistema, e convertendo uma exceção checada por uma não checada (exceção de *runtime*). Esta característica de alterar a estrutura estática de um programa é chamada *static crosscutting*.

Além de afetar a estrutura estática, um aspecto também pode afetar a estrutura dinâmica de um programa. Isto é possível através da interceptação de pontos no fluxo de execução, chamados *join points*, e da adição de comportamento antes ou depois dos mesmos, ou ainda através da obtenção de total controle sobre o ponto de execução. Exemplos de *join points* são: invocação e execução de métodos, inicialização de objetos, execução de construtores, tratamento de exceções, acesso e atribuição a atributos, entre outros. Ainda é possível definir um *join point* como resultado da composição de vários *join points*.

Normalmete um aspecto define *pointcuts*, os quais selecionam *join points* e valores nestes *join points* e *advices* que definem o comportamento a ser tomado ao alcançar os *join points* definidos pelo *pointcut*.

Nas seções seguintes, à medida que apresentamos as construções de AspectJ, exemplificamos as mesmas mostrando a implementação do aspecto de distribuição para o sistema *Health Watcher*.

3.2. Modelo de *join point*

Um *join point* é um ponto bem definido no fluxo de execução de um programa. A Figura 6 mostra um exemplo de fluxo execução entre dois objetos e identifica alguns *join points* no mesmo.

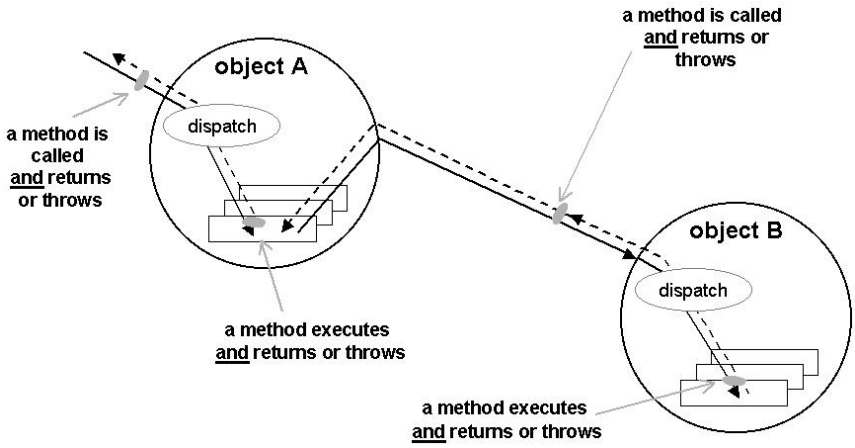

Figura 6: *Join points* de um fluxo de execução [Hilsdale and Kiczales, 2001].

O primeiro *join point* é a invocação de um método do objeto A, o qual pode retornar com sucesso, ou levantar uma exceção. O próximo *join point* é a execução deste método, que por sua vez também pode retornar com sucesso ou levantar uma exceção. Durante a execução do método do objeto A é invocado um método do objeto B. A invocação e execução deste método são *join points*, e da mesma forma que os do objeto A podem retornar com sucesso ou levantar uma exceção.

3.3. *Pointcut*

Pointcuts são formados pela composição de *join points*, através dos operadores && (e), || (ou), e ! (não). Utilizando *pointcuts* podemos obter valores de argumentos de métodos, objetos em execição, atributos e exceções dos *join points*.

Considere a definição do aspecto de distribuição para o sistema *Helth Watcher*. O trecho de código a seguir é a definição de um *pointcut* que identifica as invocações de todos os métodos da classe fachada, com quaisquer tipo de retorno, nomes, ou parâmetros, devido ao uso dos *wildcards* $*$ e $..$[1].

```
pointcut facadeMethodsCall():
    within(HttpServlet+)  &&
    call(* IFacade.*(..));
```

[1]Este *wildcard* é utilizado para representar qualquer seqüência de parâmetros

Além disso o *pointcut* restringe os join points àqueles os quais o código em execução pertencer aos subtipos (devido ao uso do *wildcard* $+$) da classe `HttpServlet`. Ou seja, este *pointcut*, chamado `facadeMethodsCall` identifica as invocações de métodos feitas à classe fachada pela interface com o usuário. Estas são as chamadas que deverão ser executadas remotamente.

Para definir *pointcuts*, identificando os *join points* a serem afetados, utilizamos construtores de AspectJ chamados designadores de *pointcut* (*pointcut designators*), como os apresentados a seguir.

`call`(*Assinatura*)	Invocação de método/construtor identificado por *Assinatura*
`execution`(*Assinatura*)	Execução de método/construtor identificado por *Assinatura*
`get`(*Assinatura*)	Acesso a atributo identificado por *Assinatura*
`set`(*Assinatura*)	Atribuição de atributo identificado por *Assinatura*
`this`(*PadrãoTipo*)	O objeto em execução é instância de *PadrãoTipo*
`target`(*PadrãoTipo*)	O objeto de destino é instância de *PadrãoTipo*
`args`(*PadrãoTipo*, ...)	Os argumentos são instâncias de *PadrãoTipo*
`within`(*PadrãoTipo*)	O código em execução está definido em *PadrãoTipo*

Onde *PadrãoTipo* é uma construção que pode definir um conjunto de tipos utilizando *wildcards*, como $*$ e $+$. O primeiro é um *wildcard* conhecido, pode ser usado sozinho para representar o conjunto de todos os tipos do sistema, ou com depois de caracteres, representando qualquer seqüência de caracteres. O último deve ser utilizado junto ao nome de um tipo para assim representar o conjunto de todos os seus subtipos.

A lista completa de *wildcards* e *pointcut designators* pode ser encontrada no guia de programação de AspectJ [Team, 2002].

3.4. *Advices*

Advices são construções que definem código adicional que deverá executar nos *join points*.

Considerando o aspecto de distribuição vamos definir um *advice* que utiliza o *pointcut* definido na seção anterior. O *advice* deverá se assegurar de que a instância remota está disponível antes de executar qualquer método na interface com usuário. Desta forma devemos utilizar um *advice* `before` da seguinte forma.

```
before(): facadeMethodsCall() {
  this.getRemoteInstance();
}
```

Onde o método `getRemoteInstance` obtém uma referência para a instância remota da classe fachada, que é armazenada por um atributo do aspecto

```
private IFacade remoteFacade;
private synchronized void getRemoteInstance() {
  if (healthWatcher == null) {
    try {
      remoteFacade = (IFacade) java.rmi.Naming.lookup("...");
    }
    catch (java.rmi.RemoteException rmiEx) { ... }
    catch (java.rmi.NotBoundException rmiEx) { ... }
    catch (java.net.MalformedURLException rmiEx) { ... }
  }
}
```

Ainda é necessário redirecionar chamadas locais para a instância remota da fachada. Para isso devemos definir um *pointcut* que recupere a instância local da fachada

```
pointcut redirectFacadeCall(IFacade facade):
  facadeMethodsCall() &&
  target(facade);
```

o qual compõe o *pointcut* `facadeMethodsCall` com o *designator* `target`, de modo a obter o objeto alvo das chamadas locais, ou seja a instância local da fachada. Note que o *pointcut* `redirectFacadeCall` tem a fachada como parâmetro. Este *pointcut* é utilizado no seguinte *advice* `around`, que também deve declarar um parâmetro do mesmo tipo do *pointcut*

```
Object around(IFacade facade) throws ... :
    redirectFacadeCall(facade) {
  try {
    return = proceed(this.remoteFacade);
  }
  catch (RemoteException rmiEx) { ... }
}
```

e que substitui a instância local pela remota, ao mandar a computação do *join point* prosseguir utilizando a instância remota, através da invocação do método `proceed`, que continua a executar o *join point*. Os parâmetro a serem passados no método `proceed` são os mesmos do *advice* `around`.

Entretanto, devido a um aparente bug, em análise pelo time de desenvolvimento de AspectJ, este *advice* gera um código que leva a um erro de execução. Desta forma a solução é a definição de um *advice* para cada método da classe fachada. A seguir mostramos um exemplo de definição deste *advice* para o método que atualiza queixas no sistema.

```
void around(Complaint complaint) throws ... :
    facadeMethodsCall()  &&
    args(complaint)           &&
    call(void update(Complaint)) {
  try {
    remoteFacade.update(complaint);
  }
  catch (RemoteException rmiEx) { ... }
}
```

Onde são identificadas invocações do método `update` que recebe uma queixa como parâmetro, e através do *designator* `args` obtém–se acesso ao argumento utilizado no momento da chamada do método, o qual também é um parâmetro do *advice*. Todavia, esta abordagem nos obriga a adicionar um novo *advice* sempre que um método for adicionado na classe fachada. Impacto similar ocorre no caso de alteração de algum método da fachada, onde seríamos obrigados a alterar o *advice* correspondente, o que não seria necessário com a abordagem anterior.

Os *advices* de AspectJ são apresentados a seguir.

`before`	Executa quando o *join point* é alcançado, mas imediatamente antes da sua computação
`after returning`	Executa após a computação **com** sucesso do *join point*
`after throwing`	Executa após a computação **sem** sucesso do *join point*
`after`	Executa após a computação do *join point*, em qualquer situação
`around`	Executa quando o *join point* é alcançado e tem total controle sobre a sua computação

3.5. *Static crosscutting*

Como mencionamos anteriormente, a linguagem AspectJ permite alterar a estrutura estática de um programa através da adição de membros de classe, da alteração a hierarquia de classes ou da substituição de exceções checadas por não checadas. Nas seções seguintes apresentamos e exemplificamos estas situações.

Introduction

O mecanismo que adiciona membros a uma classe é chamado *introduction*. Em AspectJ nós podemos introduzir métodos concretos ou abstratos, construtores e atributos em uma classe.

Até o momento, definimos *pointcut* e *advices* que afetam apenas as classes de interface com o usuário. O seguinte trecho de código introduz na classe fachada um método `main`, responsável por exportar e nomear, utilizando a API de RMI, uma instância da classe fachada tornando-a assim disponível para responder a invocações remotas.

```
public static void HealthWatcherFacade.main(String[] args) {
  try {
    HealthWatcherFacade facade = HealthWatcherFacade.getInstance();
    UnicastRemoteObject.exportObject(facade);
    java.rmi.Naming.rebind("/HealthWatcher");
  }
  catch (RemoteException rmiEx) { ... }
  catch (MalformedURLException rmiEx) { ... }
  catch(Exception ex) { ... }
}
```

A seguir apresentamos as construções do tipo *introduction* de AspectJ.

Modificadores Tipo PadrãoTipo.Id(*Formais*) { *Corpo* }	Define um método nos tipos em *PadrãoTipo*
`abstract` *Modificadores Tipo PadrãoTipo.Id*(*Formais*);	Define um método abstrato nos tipos em *PadrãoTipo*
Modificadores PadrãoTipo.new(*Formais*) { *Corpo* }	Define um construtor nos tipos em *PadrãoTipo*

Outras construções

Devido a uma exigência da API de RMI devemos definir uma interface para a classe fachada adicionando a exceção específica da API (`RemoteException`) nas clausulas `throws` dos métodos. Esta interface é um tipo auxiliar do aspecto de distribuição. Além da definição da interface remota

```
public interface IFacade extends java.rmi.Remote {

  public void update(Complaint complaint)
    throws TransactionException, RepositoryException,
    ObjectNotFoundException, ObjectNotValidException,
    RemoteException;
  ...
}
```

temos de alterar a hierarquia da classe fachada para que a mesma passe a implementar a interface remota através da seguinte construção

```
declare parents: HealthWatcherFacade implements IFacade;
```

Outra alteração na hierarquia do programa é necessária. Os tipos dos objetos que são recebidos e retornados pela classe fachada devem implementar a interface `java.io.Serializable` de modo a permitir a sua serialização no canal de comunicação entre a interface com o usuário e a fachada.

```
declare parents: Complaint || Person implements Serializable;
```

A seguir apresentamos as outras construções em AspectJ que alteram a estrutura estática de um programa.

`declare parents` : *PadrãoTipo* `extends` *ListaTipos*;	Declara que os tipos em *PadrãoTipo* herdam dos tipos em *ListaTipos*
`declare parents` : *PadrãoTipo* `implements` *ListaTipos*;	Declara que os tipos em *PadrãoTipo* implementam os tipos em *ListaTipos*
`declare soft` : *PadrãoTipo*: *Pointcut*;	Declara que qualquer exceção de um tipo em *PadrãoTipo* que for lançada em qualquer *join point* identificado por *Pointcut* será encapsulada em uma exceção não checada

Maiores informações sobre *crosscutting concerns* podem ser encontradas no guia de programação de AspectJ [Team, 2002].

3.6. Aspectos reusáveis

AspectJ permite a definição de aspectos abstratos, os quais devem ser estendidos provendo a implementação do componente abstrato. Os componentes abstratos podem ser métodos, como em uma classe Java, e *pointcuts*, os quais devem ser definidos em um aspecto concerto, permitindo o reuso do comportamento dos aspectos abstratos.

Vamos considerar um aspecto de tratamento de exceções. Podemos definir um aspecto geral que define um *pointcut* abstrato, responsável por definir os pontos em que as exceções são lançadas e um método abstrato responsável por aplicar o tratamento necessário. Dessa forma definimos o aspecto `ExceptionHandlingAspect` que utiliza o *pointcut* abstrato e o método abstrato em um *advice* que captura a exceção lançada após a execução dos *join points* e chama o método abstrato que deve prover o tratamento adequado para a exceção.

```
public abstract aspect ExceptionHandlingAspect {
  public abstract pointcut exceptionJoinPoints();
  after() throwing (Throwable ex): exceptionJoinPoints() {
    this.exceptionHandling(ex);
  }
  protected abstract void exceptionHandling(Throwable ex);
}
```

A partir deste aspecto podem ser criados vários aspectos especializados para prover tratamentos de exceção para diferentes interfaces com o usuário, como o aspecto a seguir que provê tratamentos de exceção para servlets Java. Note que apenas a implementação do método `exceptionHandling` é fornecida, logo o aspecto ainda é abstrato.

```
public abstract aspect ServletsExceptionHandlingAspect
    extends ExceptionHandlingAspect {
  protected void exceptionHandling(Throwable ex) {
    // handling exceptions in servlets Java
  }
}
```

Em seguida são definidos aspectos mais especializados para serem utilizados por outros aspectos, como o de distribuição.

```
public aspect DistributionExceptionHandlingAspect
    extends ServletsExceptionHandlingAspect {
  public pointcut exceptionJoinPoints():
    DistributionAspect.facadeMethodsCall();
}
```

o qual identifica como *join point* passíveis de tratamento de exceção os mesmos alterados pelo aspecto de distribuição. Note o reuso do *pointcut* `facadeMethodsCall` do aspecto de distribuição (`DistributionAspect`). Observe este exemplo de *crosscutting concerns*, onde distribuição e o tratamento de suas exceções são definidos como preocupações em separado.

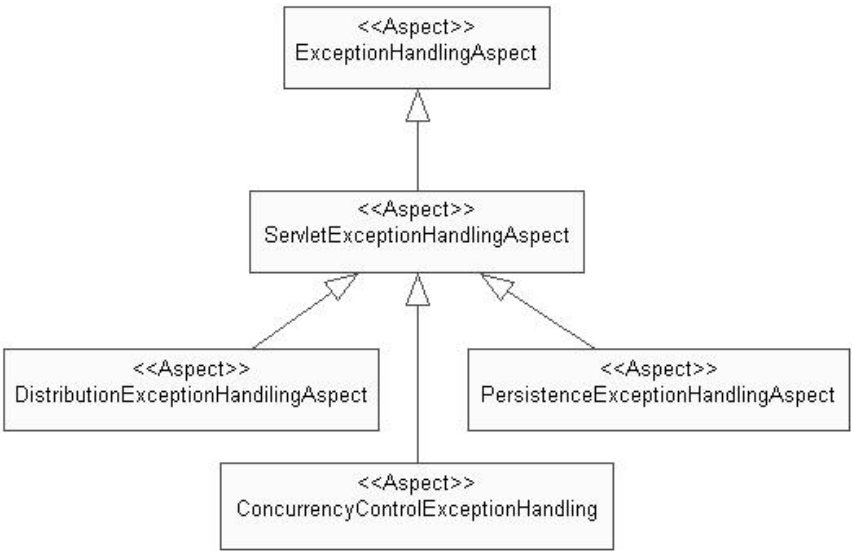

Figura 7: Diagrama de classes com aspectos de tratamento de exceções.

Com esta abordagem teríamos a hierarquia de aspectos mostrada no diagrama da

Figura 7, onde seriam definidos vários aspectos de tratamento de exceção para tratar as exceções adicionadas por outros aspectos.

Maiores informações sobre aspectos de distribuição e tratamento de exceções podem ser encontradas em outro trabalho [Soares et al., 2002].

3.7. Um exemplo simples

O seguinte código em AspectJ define um aspecto responsável por imprimir na saída padrão todos os comandos sql [Elmasri and Navathe, 1994] antes dos mesmos serem executados no banco de dados. A idéia é permitir ao programador visualizar a string resultante, o que normalmente resulta de uma série de concatenações de strings, muitas vezes difíceis de corrigir.

```
aspect DatabaseDebugging {
  private interface TypesDebugged { }

  declare parents : DataCollection1 ||
                    DataCollection2 ||
                    ...
                    DataCollectionN implements TypesDebugged;

  pointcut queryExecution(String sql):
    call(* Statement.*(String)) &&
    this(TypesDebugged) &&
    args(sql);

  before(String sql): queryExecution(sql) {
    System.out.println(sql);
  }
}
```

onde declaramos uma interface que é utilizada para marcar as classes que serão depuradas, e fazemos estas classes implementarem a interface. Em seguida é definimos um *pointcut* que identifica as invocações de todos os métodos de objetos do tipo `Statement` (da API de JDBC [White and Hapner, 1999]), responsável pela execução dos comandos sql, que recebam como parâmetro um string e retornem qualquer tipo. O *pointcut* ainda obtém a string utilizada como argumento nessas invocações de métodos. Em seguida definimos um *advice* que imprime a string (sql) que será executada no banco de dados. Isto deve ser feito antes da sua execução de modo que a mesma seja visualizada antes de ser submetida ao banco de dados.

4. AOP e padrões de projeto

Parte da funcionalidade da programação orientada a aspectos pode ser implementada por padrões de projetos [Gamma et al., 1994], como a separação de código com funções específicas. Por exemplo, podemos utilizar o padrão *Adapter* [Gamma et al., 1994] para adicionar um comportamento a um método de uma classe.

Considere o diagrama de classes da Figura 8 onde a classe `Source` usa o serviço `m` de uma interface `ITarget`.

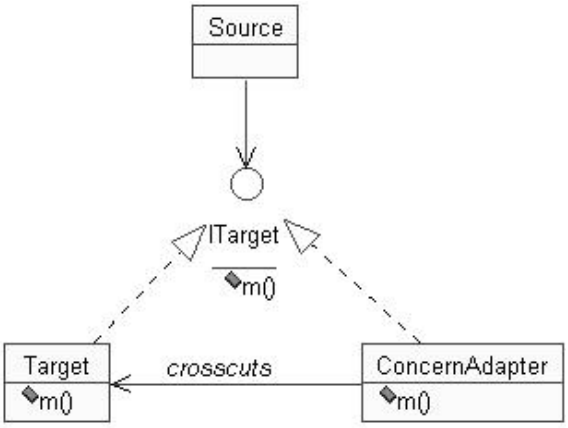

Figura 8: Adaptadores implementando separação de preocupações.

A classe `Target` contém o código funcional do sistema, ou seja, o código que implementa os requisitos funcionais do sistema. Considere a necessidade de implementar de um aspecto para afetar as execuções do método `m` da classe `Target`. Para isso um adaptador, que possui uma referência para uma instância de `Target`, deve implementar a interface `ITarget` adicionando o comportamento desejado e delegando, se for o caso, à sua instância de `Target` a execução da parte funcional.

Observe que esta abordagem é análoga a do *advice* `around`, onde o mesmo detém controle total sobre o fluxo de execução do *join point*, assim como o método `m` do adaptador detém controle total sobre o fluxo de execução do método `m` da classe `Target`.

Porém, esta abordagem de padrões de projetos levaria a uma duplicação de código, uma vez que para cada classe interceptada seria necessário um adaptador, mesmo que para a implementação de um mesmo aspecto, como depuração, por exemplo. Em AspectJ basta definir no *pointcut* quais pontos devem ser afetados pelo aspecto. Outra desvantagem é que caso o método `m` de `Target` invoque outro método da classe `Target` que seria afetado pelo aspecto essa invocação não seria capturada pelo adaptador. Já em AspectJ essa invocação seria capturada. E por fim, no caso de uma mudança da implementação do aspecto, a alteração de a que componente a classe `Source` esta ligada seria invasiva, ou seja, feita diretamente no código fonte. Com AspectJ esta mudança seria automática, bastando recompor (*weave*) o sistema utilizando o novo aspecto.

5. Conclusões

A Programação orientada a aspectos trás vários benefícios devido à possibilidade de tornar programas mais modulares. O objetivo da técnica é separar preocupações ortogonais (*crosscutting concerns*) de modo a evitar o entrelaçamento de código com diferentes propósitos e o espalhamento de código com propósito específico em várias partes do sistema. Desta forma obtemos ganhos com manutenção e evolução do sistema além de favorecer o reuso de suas partes.

A linguagem AspectJ é uma extensão orientada a aspectos da linguagem Java, permitindo assim a programação orientada a aspectos em Java. Entre as desvantagens da linguagem estão a necessidade de se familiarizar com as novas construções de AspectJ e a pouca maturidade do ambiente de desenvolvimento, resultando em um ambiente ainda não muito estável. Entretanto, o ambiente teve uma evolução considerável, estando hoje bem mais estável que nas versões anteriores. Além disso, o time de suporte está pronto a tirar eventuais dúvidas e até mesmo sugerir soluções para problemas, através de uma lista de discussão. Pontos a melhorar no ambiente de desenvolvimento são o tempo de compilação (*weaving*) e o tamanho do *bytecode* gerado.

Outra fraqueza de AspectJ é a sua política de tratamento de exceções. O *advice* `around` é o único tipo de *advice* que suporta a declaração de uma cláusula `throws`. Nos demais não é possível lançar uma exceção checada que já não seja tratada pelos métodos afetados. A solução dada é utilizar exceções *soft*, ou seja, encapsular exceções checadas em uma exceção não checada (exceções do tipo *runtime*). Como este tipo de exceção não obriga o programador a prover o tratamento para a mesma, o programa pode gerar erros inesperados, caso se esqueça de prover o tratamento adequado.

AspectJ provê construtores muito poderosos, que devem ser utilizados com precaução, uma vez que o uso de *wildcards* pode afetadar várias partes de um programa, inclusive partes indesejadas. Para aumentar ainda mais a expressividade da linguagem, sugerimos [Soares et al., 2002] a criação de uma funcionalidade que permita introduzir uma exceção na cláusula `throws` de um método. Esta funcionalidade seria mais um tipo de *static crosscutting*. Desta forma poderíamos introduzir exceções checadas, as quais devem, obrigatoriamente, ser tratadas, não deixando a cargo do programador assumir o compromisso de tratar exceções do tipo *runtime*.

A definição de um *pointcut* em AspectJ requer a identificação de pontos específicos de um programa. Como estes pontos são identificados através de nomes e tipos de métodos, parâmetros e etc., os aspectos ficam dependentes do sistema, ou da nomenclatura utilizada por ele, dificultando as chances de reuso. Por exemplo, para identificar todas as chamadas de métodos que inserem dados no sistema poderíamos identificar as invocações a métodos com nome `insert`, o que obriga a definição de métodos que inserem dados sempre com este nome, e não `register` ou `add`. Isto mostra a necessidade de uso de um padrão de nomenclatura, o que também beneficia a legibilidade do sistema.

Outro suporte ao desenvolvimento por parte de AspectJ são extensões para IDEs (ferramentas CASE de programação) bem disseminadas, como Borland JBuilder. Com isto é possível utilizar estes ambientes de programação para desenvolver sistemas com AspectJ. Estas extensões também permitem visualizar que partes do código são afetadas pelos aspectos. Atualmente estão sendo desenvolvidas extensões para outras IDEs.

A maior vantagem de AspectJ é a possibilidade de implementar funcionalidades em separado da parte funcional do sistema, e automaticamente inseri ou remover tais aspectos do mesmo. Para remover um aspecto do sistema basta gerar uma nova versão do sistema sem o aspecto que se quer remover. Além disso, para alterar um aspecto, como mudar o protocolo de distribuição de um sistema, basta implementar outro aspecto de distribuição, e passar a usá–lo no processo de recomposição. Com a separação, a legibilidade do código funcional é favorecida, uma vez que não há códigos com diferentes

propósitos entrelaçados entre si e com código funcional. Isto também permite a validação precoce dos requisitos funcionais, antes mesmo da implementação de aspectos como persistência, distribuição, e controle de concorrência, tendo assim um desenvolvimento progressivo [Soares and Borba, 2002] do sistema.

Referências

Booch, G. (1994). *Object–Oriented Analysis and Design with Applications*. Benjamin/Cummings, second edition.

Booch, G., Jacobson, I., and Rumbaugh, J. (1999). *Unified Modeling Language – User's Guide*. Addison–Wesley.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996). *A System of Patterns: Pattern–Oriented Software Architecture*. John Wiley & Sons.

Elmasri, R. and Navathe, S. (1994). *Fundamentals of Database Systems*. Addison–Wesley, second edition.

Elrad, T., Filman, R. E., and Bader, A. (2001). Aspect–oriented programming. *Communications of the ACM*, 44(10):29–32.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object–Oriented Software*. Addison–Wesley.

Gosling, J., Joy, B., Steele, G., and Bracha, G. (2000). *The Java Language Specification*. Addison–Wesley, second edition.

Hilsdale, E. and Kiczales, G. (2001). Aspect–oriented programming with AspectJ. In *OOPSLA'01, Tutorial*, Tampa FL.

Hunter, J. and Crawford, W. (1998). *Java Servlet Programming*. O'Reilly & Associates, Inc., first edition.

J., L. K., I., S.-L., and et al (1994). Adaptive Object–Oriented Programming Using Graph–Based Customization. *Communications of the ACM*, 37(5):94–101.

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

Laddad, R. (2002). I want my aop!, part 1. *JavaWorld*. Avaliable at http://www.javaworld.com/javaworld/jw-01-2002/jw-0118-aspect.html.

Meyer, B. (1997). *Object–Oriented Software Construction*. Prentice–Hall, second edition.

Microsystems, S. (2001). Java Remote Method Invocation (RMI). Disponível em http://-java.sun.com/products/jdk/1.2/docs/guide/rmi.

Murphy, G. C., Walker, R. J., Baniassad, E. L., Robillard, M. P., Lai, A., and Kersten, M. A. (2001). Does aspect–oriented programming work? *Communications of the ACM*, 44(10):75–77.

Orfali, R. and Harkey, D. (1998). *Client/Server Programming with Java and CORBA*. Wiley.

Ossher, H., Kaplan, M., Katz, A., Harrison, W., and Kruskal, V. (1996). Specifying suject–oriented composition. *TAPOS*, 2(3):179–202. Special Issue on Subjectivity in OO Systems.

Ossher, H. and Tarr, P. (1999). Using subject–oriented programming to overcome common problems in object–oriented software development/evolution. In *International Conference on Software Engineering, ICSE'99*, pages 698–688. ACM.

Soares, S. and Borba, P. (2002). Progressive implementation with aspect–oriented programming. In Verlag, S., editor, *The 12th Workshop for PhD Students in Object-Oriented Systems*, Malaga, Spain. To appear.

Soares, S., Laureano, E., and Borba, P. (2002). Implementing distribution and persistence aspects with AspectJ. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'02)*, Seattle, WA, USA. ACM Press. To appear.

Team, A. (2002). The AspectJ Programming Guide. Disponível em http://aspectj.org.

White, S. and Hapner, M. (1999). JDBC 2.1 API. Version 1.1. Sun Microsystems.