Programação Orientada a Aspectos com AspectJ

Autor: Torsten Nelson

O curso segue a estrutura do livro AspectJ in Action, de Ramnivas Laddad, que foi a bibliografia recomendada.

AULA	TÓPICO
Módulo 1 - Revisão de orientação a objetos	
1	Fundamentos da orientação a objetos
2	Java - Herança e polimorfismo
3	Java - outros tópicos
Módulo 2 - Orientação a Aspectos	
4	Interesses transversais
5	Exemplo prático de interesses transversais
6	Anatomia de linguagens orientadas a aspectos
Módulo 3 - AspectJ	
7	O compilador, "Hello, World", e o IDE Eclipse
8	Visão geral do AspectJ
9	O modelo de join points
10	Pointcuts
11	Advice
12	Introduções
Módulo 4 - AspectJ avançado	
13	Reflexão em Java
14	Reflexão em AspectJ
15	Precedência entre aspectos
16	Associações entre aspectos
Módulo 5 - Aplicações de POA	
17	Padrões de projeto e aspectos
18	Aplicações da orientação a aspectos

Aula 1 - Orientação a objetos

Um objeto é um *componente* de software - uma parte de um sistema que exibe, ou deveria exibir, certas características específicas. Nessa aula vamos ver algumas dessas características

1. **Comportamento e estado**
   - O *estado* de um objeto é o conjunto de suas propriedades;
   - O *comportamento* de um objeto são as funções que afetam suas propriedades ou as de outros objetos;
   - Cada objeto tem seu próprio estado
2. **Instâncias e classes**
   - Cada objeto é uma instância de alguma classe;
   - Uma classe define quais as propriedades que tem cada uma de suas instâncias e comportamento delas;
   - Há apenas uma "cópia" do comportamento (métodos da classe).
3. **Acoplamento e coesão**
   - Acoplamento é o nível de dependência entre classes;
   - Deve-se tentar minimizar o acoplamento para evitar a propagação de mudanças e para possibilitar a reutilização de classes;
   - Coesão é o nível de integridade interna de uma classe;
   - Classes com alta coesão têm responsabilidades bem definidas e são difíceis de dividir em duas ou mais classes;
   - Classes com baixa coesão tratam de responsabilidades diversas e em geral podem ser divididas.
   - Deve-se tentar maximizar a coesão das classes.
4. **Interfaces e implementações**
   - Cada objeto define uma *interface* pública;
   - A interface consiste dos comportamentos e estado do objeto que podem ser acessados por outros objetos;
   - A interface é um *contrato*; mudar a interface significa violar o contrato e pode ter consequências graves;
   - A implementação é o modo como o objeto realiza as obrigações impostas pelo contrato;
   - Outros objetos não deveriam ser afetados por mudanças na implementação.
   - A interface de um método também é chamada de *assinatura*, e consiste do nome do método mais seus parâmetros formais.
   - Um método é *sobrecarregado* quando uma classe tem mais de um método com o mesmo nome, mas diferentes listas de parâmetros.

Exemplo: a cozinha inteligente de Torsten

Eu gostaria de um sistema que soubesse que ingredientes eu tenho em casa (na despensa e geladeira) e que guardasse minhas receitas favoritas. Ele me permitiria fazer três consultas: descobrir se tenho em casa os ingredientes necessários para preparar uma receita, descobrir que receitas posso preparar com os ingredientes que tenho em casa, e descobrir que receitas posso preparar com um conjunto específico de ingredientes.

- Quais são as classes desse sistema?
- Qual o estado de cada classe?
- Qual a funcionalidade que cada classe deve prover?
- Como são as dependências entre as classes?

Aula 2 - Herança

Suponha que você está fazendo um sistema para gerenciar uma locadora de vídeo que aluga e vende DVDs. Naturalmente, você precisa de pelo menos duas classes para cuidar dos seus produtos:

```
class DVDdeVender {
   private float preço;
   public void vender();
   public void devolver();
   public void recibo();
}

class DVDdeAlugar {
   private float preço;
   private Date dataDevolução;
   public void alugar();
   public void devolver();
   public void recibo();
}
```

No entanto, você nota que as duas classes são muito semelhantes e gostaria de não ter de repetir esforços na sua criação e uso. O mecanismo que permite isso é a *herança*.

Herança é um relacionamento entre classes que é uma das características principais da orientação a objetos. A classe herdada é a classe pai, ou superclasse, e a classe herdadora é a classe filha, ou subclasse.

A classe filha herda todos os membros (dados e métodos) da classe pai. No entanto, a classe filha só tem acesso direto a membros declarados como *public* ou *protected*.

Interfaces e implementações:

- A interface da classe filha contém a interface da classe pai.
- A classe filha pode acrescentar membros à sua interface.
- A classe filha NÃO pode remover membros da interface herdada.
- A classe filha pode alterar a implementação de um método herdado, *sobrepondo* a implementação com uma nova; instâncias da classe filha usarão a nova implementação e não a herdada.

Há duas motivações principais para o uso da herança:
- Herança para construção
- Herança para substituição

Herança para construção

- Usa-se herança para construção quando o objetivo é reutilizar uma classe existente para criar uma semelhante.
- Um ou mais métodos são sobrepostos para se mudar a funcionalidade da classe.
- Objetivo é reaproveitamento de código.
- Em geral, o uso de herança somente para construção é fruto de de um projeto ruim; deve-se buscar soluções que usem também herança para substituição.

Exemplo:

```
class DVDdeVender {
   private float preço;
   public void adquirir();
   public void devolver();
   public void recibo();
}

class DVDdeAlugar extends DVDdeVender {
   private Date dataDevolução;
   public void adquirir();
   public void devolver();
}
```

Herança para substituição

No exemplo acima, o uso de herança reduziu a quantidade de código necessária da construção da classe. A herança também nos permite reduzir o código necessário para o *uso* da classe. Por exemplo, suponha que tenhamos a seguinte classe para armazenar nosso estoque de DVDs:

```
public class ColeçãoDeDVDs {
   private DVDdeAlugar[] dvdsDeAlugar;
   private DVDdeVender[] dvdsDeVender;
   private int numeroDVDsAlugar;
   private int numeroDVDsVender;

   public void acrescentarDVDdeVender(DVDdeVender d) {
                  dvdsDeVender[numeroDVDsVender] = d;
                  numeroDVDsVender++;
   }
   public void acrescentarDVDdeAlugar(DVDdeAlugar d) { ... }

   // Imprimir todos os DVDs
   public void relatório() {
      for (int i=0; i < numeroDVDsAlugar; i++) {
              dvdsDeAlugar[i].imprimir();
          }
      for (int i=0; i < numeroDVDsVender; i++) {
              dvdsDeVender[i].imprimir();
          }
   }
}
```

A *regra da substituição* diz que sempre que um programa espera um objeto (por exemplo, como parâmetro de um método, ou em uma atribuição) podemos substituir o objeto por outro objeto que seja instância de uma classe que é filha da classe esperada.

Deve-se sempre seguir a regra do É-UM para se saber se uma subclasse é apropriada. Por exemplo, um DVDdeAluguel não É-UM DVDdeVender. São coisas diferentes. No entanto, tanto DVDdeAluguel É-UM DVD quanto DVDdeVender É-UM DVD. Vamos então reescrever a nossa hierarquia:

```
public class DVD {
   private float preço;
   public void adquirir();
   public void devolver();
```

```
   public void recibo();
}

public class DVDdeVender extends DVD {
   public void adquirir();
   public void devolver();
}

public class DVDdeAlugar extends DVD {
   private Date dataDevolução;
   public void adquirir();
   public void devolver();
}
```

A classe DVD contém todos os dados e funções comuns às duas classes anteriores. Cada classe herda esses membros e ainda acrescenta aquilo que lhe é particular.

A distinção entre interface e implementação é importantíssima aqui: como a subclasse necessariamente contém a interface da classe pai, garante-se que a substituição é possível.

Outra regra importante de se lembrar é que pode-se sempre atribuir uma instância de uma subclasse a uma variável da superclasse, mas nunca se pode fazer o contrário. Ou seja,

```
    DVD d1 = new DVDdeAlugar();  // OK
    DVDdeAlugar d2 = new DVD();  // Não é permitido.
```

O princípio da substituição nos permite reescrever a classe ColeçãoDeDVDs:

```
public class ColeçãoDeDVDs {
   private DVD[] dvds;
   private int numeroDVDs;

   public void acrescentarDVD(DVD d) {
                  dvds[numeroDVDs] = d;
                  numeroDVDs++;
   }

   // Imprimir todos os DVDs
   public void relatório() {
      for (int i=0; i < numeroDVDs; i++) {
               dvds[i].imprimir();
         }
   }
}
```

Usando o princípio da substituição, podemos usar a classe ColeçãoDeDVDs para acrescentar ambos os tipos de DVD:

```
public static void main() {
   ColeçãoDeDVDs c;

   c.acrescentarDVD(new DVDdeAlugar());
   c.acrescentarDVD(new DVDdeVender());
   c.relatório();
}
```

Polimorfismo é um termo usado no mundo OO para se referir à substituição de classes por

subclasses.

Classes abstratas e interfaces

O procedimento de devolução de DVDs vendidos e alugados é bem diferente. O que colocar na classe DVD, então? Uma classe pode ter métodos sem implementação, apenas interface. Uma classe que tenha algum método sem implementação é chamada *classe abstrata*

```
public abstract class DVD {
   private float preço;
   public abstract void adquirir();
   public abstract void devolver();
   public void recibo();
}
```

Classes abstratas não podem ser instanciadas e servem apenas como base para substituição.

Uma *interface* (em Java) é uma classe que não tem nenhum dado e em que todos os métodos são abstratos. Herdar de interfaces tem uma sintaxe própria, sendo um tipo especial de herança, porque uma mesma classe pode herdar de várias interfaces.

No exemplo abaixo, *Cloneable* é uma interface da biblioteca padrão Java que define um único método, *clone()*, e *Comparable* é uma interface que define o método *compareTo()*.

```
public class DVDdeAlugar extends DVD implements Cloneable, Comparable {
    ...
}
```

Pense: o que se ganha em declarar que se está implementando uma interface? Por que não simplesmente implementar *clone()* e *compareTo* sem declarar que se está implementando a interface?

Aula 3 - Java

Tratamento de Exceções

Uma *exceção* é um evento que foge ao fluxo normal esperado de um programa.

Métodos devem capturar as exceções geradas. Se um método não capturar a exceção, abandona-se o método e retorna-se ao método chamador, e assim por diante, até que a exceção seja capturada ou que se saia do programa principal (main).

Exceções são objetos também. Em Java, existem várias classes de exceções, todas elas subclasses de java.lang.Exception. O usuário pode criar novas subclasses.

Quando um método gera uma exceção, diz-se que ele joga (throws) a exceção. Em Java, usa-se o comando *throw*. A assinatura do método deve declarar todas as exceções jogadas por ele.

```
void usaDisco( ) throws IOException {
    if (deuProblemaDeDisco) {
              throw (new IOException() );
    }
}
```

Exceções são capturadas em blocos *try..catch*. Para capturar uma exceção, o método deve ser chamado dentro do bloco *try*. Se a exceção for jogada, o controle passa para o bloco *catch*. Por exemplo,

```
void interpretar(String s) {
   try {
      int numero;
      numero = Integer.parseLong(s);
   } catch (NumberFormatException e) {
      System.out.println("Erro: o string nao é um numero valido");
          e.printStackTrace();
   }
}
```

Se o método não capturar a exceção, ele tem de declarar que ela é jogada:

```
void interpretar(String s) throws NumberFormatException {
   int numero;
   numero = Integer.parseLong(s);
}
```

Deve-se usar exceções unicamente para situações excepcionas; nunca se deve usar exceções para regular o fluxo normal do sistema.

Gerenciamento de memória

Em Java, variáveis de tipos primitivos são alocadas automaticamente quando declaradas e liberadas quando saem do escopo em que foram declaradas. A passagem de parâmetro dessas variáveis é feita por valor, ou seja, é feita uma cópia da variável para servir de parâmetro.

Por outro lado, objetos são tratados de maneira completamente diferente. Não são alocados

automaticamente. A declaração apenas cria uma referência a um objeto. Para se alocar um objeto, usa-se o comando *new*. Objetos não são liberados quando saem de escopo, mas sim liberados automaticamente quando não há mais nenhuma referência a eles, um processo chamado de "coleta de lixo" (*garbage collection*).

```
// O parâmetro x é alocado e criado quando o método é executado.
void teste(int x) {
  int i;       // Um inteiro é alocado aqui
  String s;    // Um String é declarado, mas não existe ainda
   s = new String()    // Agora sim um String foi alocado
}
// x e i deixam de existir aqui e são liberados. O string s
// poderia continuar existindo, mas não há mais referências a ele
// pois "s" está fora de escopo, então o string é liberado.
```

Construtores

Java tem oito tipos primitivos: boolean, byte, char, short, int, long, float e double. Tipos primitivos são alocados automaticamente quando declarados.

Os tipos restantes são classes. Classes não são alocadas automaticamente. Objetos declarados inicialmente tem o valor *null*. Para se criar um objeto, usa-se o comando *new*.

```
int i;   // tipo primitivo
Date d = new Date();   // Classe Date
```

Quando o objeto é criado, um método especial chamado construtor é executado. O construtor tipicamente inicializa o objeto com valores aceitáveis. Construtores são frequentemente sobrecarregados para aceitarem listas de valores para inicialização.

O construtor de uma subclasse deve chamar o construtor da superclasse para garantir que os membros herdados sejam executados devidamente. A chamada se faz com o comando *super()*, que deve ser o primeiro comando do construtor. Na verdade, pode-se acessar qualquer membro da superclasse com a notação *super.membro()*.

Métodos úteis

Todas as classes são subclasses da classe *Object*. Dela são herdados alguns métodos que podem ser sobrepostos e são muito usados. Por exemplo:

- `public boolean equals(Object obj);`

  Retorna verdadeiro se dois objetos são equivalentes. A comparação entre variáveis de tipos primitivos pode ser feita com o operador ==. No entanto, quando usado entre dois objetos, == retorna verdadeiro somente se os dois objetos sendo comparados são o *mesmo* objeto. Para compararmos o conteúdo, devemos sempre usar *equals*.

  ```
  String s1, s2, s3;
  s1 = "Bom dia";
  s2 = "Bom dia";
  s3 = s1;
  // Todos os 3 valem "Bom dia"
  s1 == s2       // Falso
  s1 == s3       // Verdadeiro
  ```

```
s1.equals(s2)  // Verdadeiro
```

A implementação de *equals* herdada de Object faz uma comparação simples, bit-a-bit, dos objetos. Se a classe contiver apenas dados formados por tipos primitivos, a implementação herdada é suficiente. Se ela contiver outros objetos, no entanto, é preciso sobrepor o método herdado com um mais apropriado (por quê? e como é esse método sobreposto?).

- `public String toString();`

  Retorna uma representação do valor do objeto em forma de String.
- `public Object clone();`

  O método *clone* retorna uma cópia do objeto alvo. Se o objeto contiver referencias a outros objetos, a cópia conterá as mesmas referências - chama-se isso de cópia "rasa" (*shallow copy*). Para se duplicar toda a estrutura do objeto, é necessário sobrepor o método herdado com um que faça a cópia corretamente.

Typecasting

Há ocasiões em que é necessário converter um objeto de um tipo em outro. Por exemplo, o método *clone* acima retorna uma instância de *Object*. O que aconteceria no caso a seguir?

```
String s1, s2;
s1 = "Bom dia";
s2 = s1.clone();
```

Na última linha, estamos tentando atribuir um objeto da classe *Object* a um objeto da classe *String*. Lembre-se que não podemos atribuir um objeto da superclasse a um objeto da subclasse, apenas o contrário. Assim, é necessário "forçar" o valor retornado a ter o tipo que queremos. Isso se chama "typecasting" (moldar o tipo):

```
s2 = (String) s1.clone();
```

Só se pode fazer typecasting de uma superclasse para uma subclasse. Não se pode fazê-lo entre classes "irmãs".

Pacotes

Cada classe Java pertence a um pacote, que é um conjunto de classes relacionadas. A biblioteca de classes padrão Java contém inúmeros pacotes com classes úteis, como por exemplo:

```
java.lang     // Classes básicas da linguages como Integer, String, etc.
java.util     // Classes "utilitarias" como conjuntos, datas, etc.
java.io       // Classes para entrada/saída de dados (arquivos, terminal)
```

Cada classe é identificada unicamente pelo seu nome completo, que é o nome do pacote mais o nome da classe. Por exemplo, a classe *java.util.ArrayList* é uma lista encadeada implementada com um vetor.

Pacotes devem ter nomes "universalmente únicos". Ou seja, se voce colocar suas classes em um pacote, deve escolher um nome que ninguém mais tenha usado. A regra usada para isso é usar o URL da sua empresa/instituição como prefixo, de trás pra frente. Por exemplo, poderíamos usar

o pacote *br.pucminas* ou até simplesmente *pucminas*.

Os pacotes são localizados pelo compilador através de um esquema de diretórios. As classes do pacote *br.pucminas* devem estar em um diretório *br/pucminas/* dentro do CLASSPATH.

Uma classe que faça parte do pacote *br.pucminas* deve começar com a declaração *package br.pucminas;*

Pode-se referir a classes no mesmo pacote pelo nome curto (sem o pacote). Para não precisar se referir a classes de outros pacotes pelo nome completo, pode-se importar as classes do pacote com a declaração *import*.

```
import java.util.ArrayList;  // Importa a classe ArrayList
import java.io.*;            // Importa todas as classes do pacote java.io
```

Aula 4 - Interesses entrecortantes

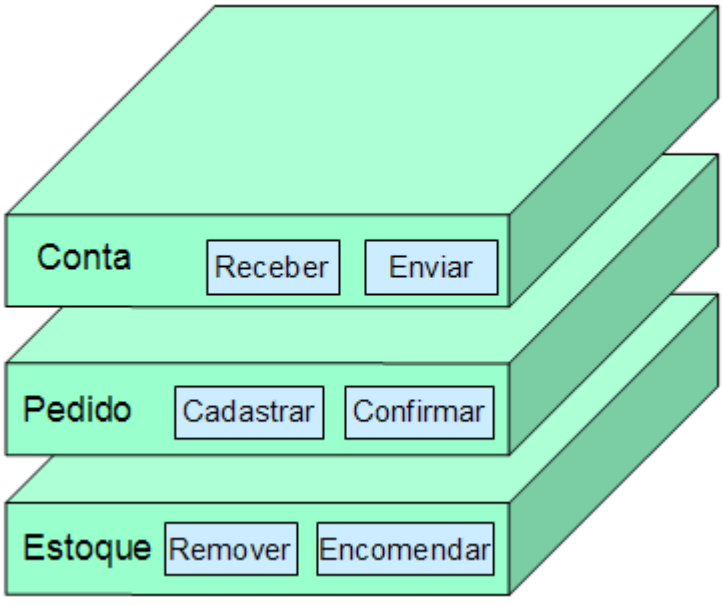
Os termos *desenvolvimento estruturado* e *orientação a objetos* dizem respeito à *modularidade* do sistema. São formas distintas de se dividir um sistema em partes.

A divisão em partes é importante para se reduzir a complexidade. É muito difícil para um ser humano compreender um sistema de grande porte se este for monolítico, sem fronteiras claras que definem suas funções.

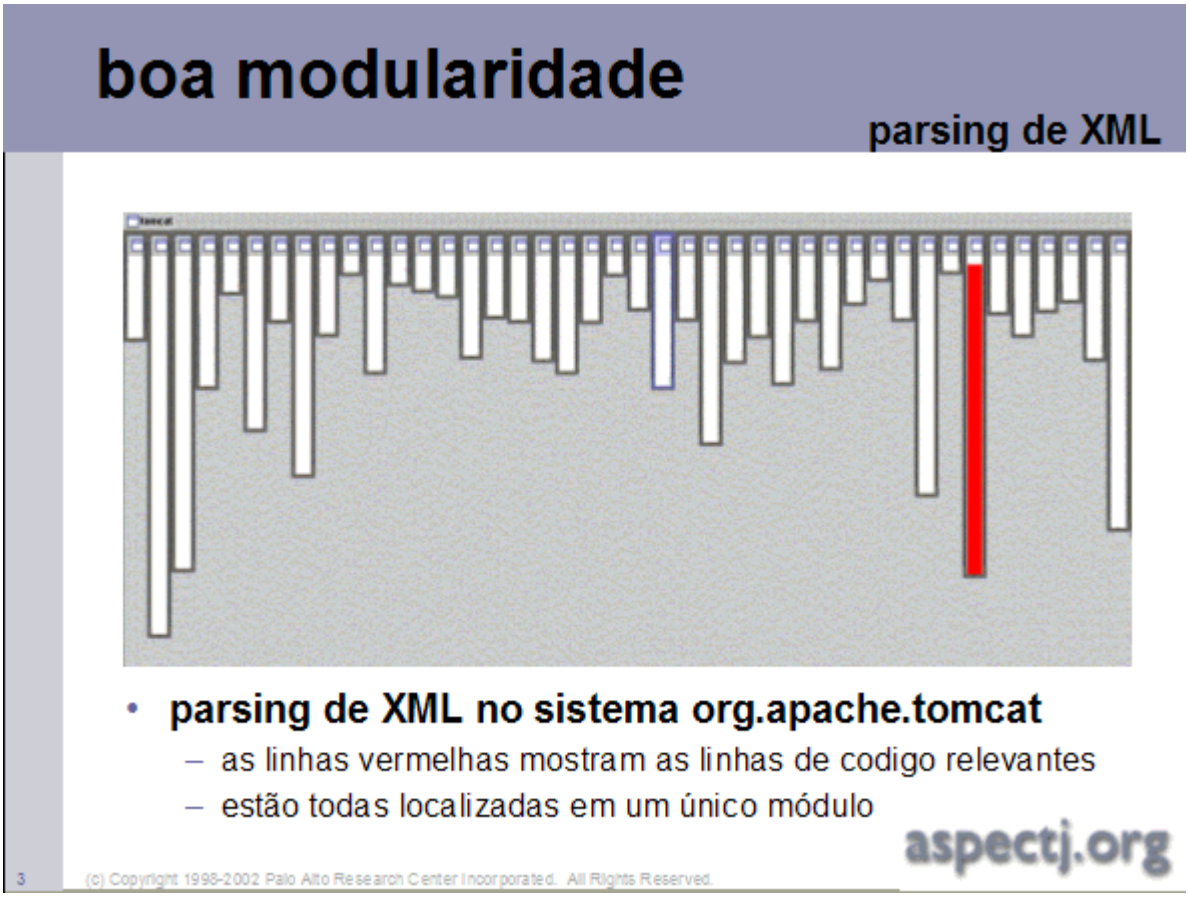
O termo *separação de interesses* foi cunhado por Edsger Dijkstra em 1974 para denotar o princípio que guia a divisão em partes: todo sistema de software lida com diferentes interesses, sejam eles dados, operações, ou outros requisitos do sistema. O ideal seria que a parte do programa dedicada a satisfazer a um determinado interesse estivesse concentrada em uma única localidade física, separada de outros interesses, para que o interesse possa ser estudado e compreendido com facilidade.

O desenvolvimento estruturado realizou a separação de interesses orientando-se através das diferentes funcionalidades oferecidas pelo software. Cada função é implementada em um único módulo, ou procedimento. Daí surgiram conceitos que ajudam a manter a separação de interesses, como o baixo acoplamento e a alta coesão.

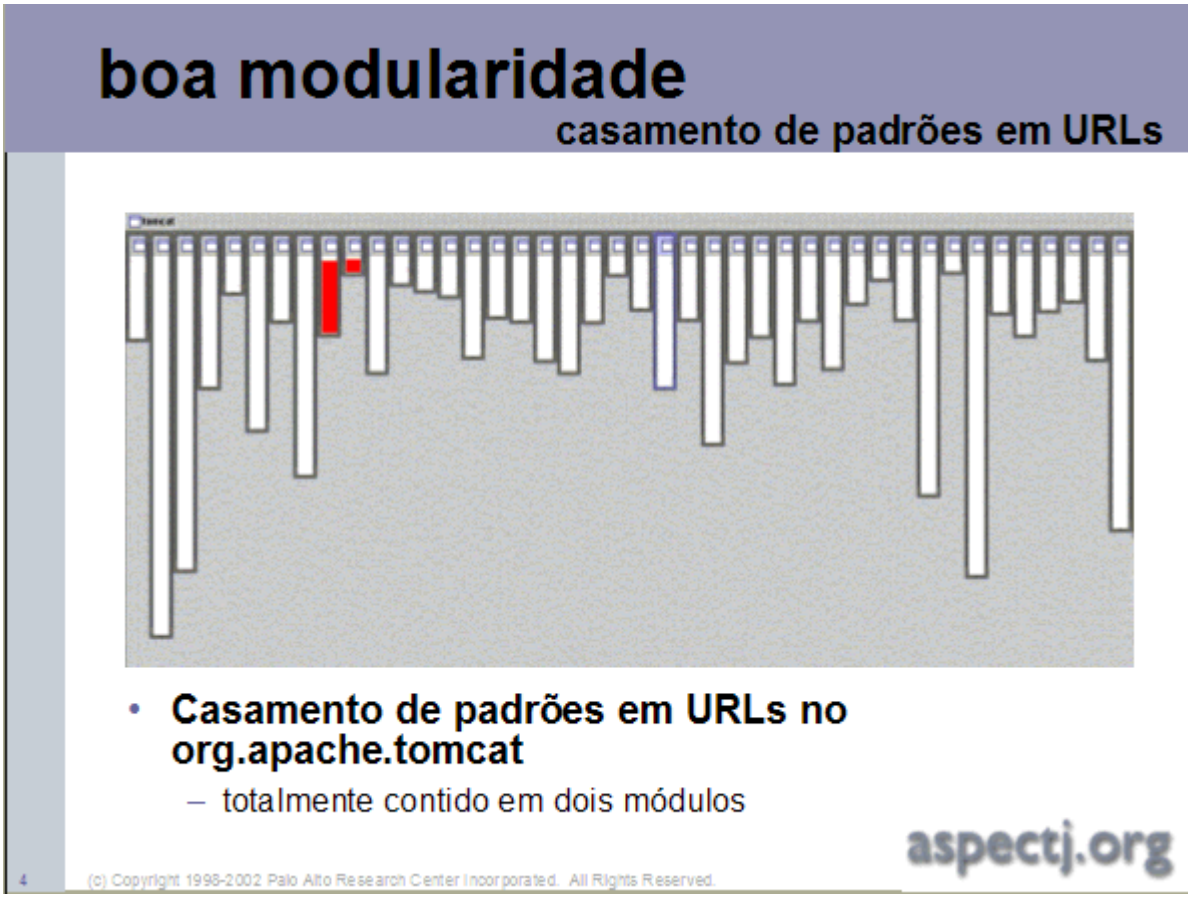
A orientação a objetos veio como forma de sanar uma das deficiências do desenvolvimento estruturado. Apesar de interesses relativos a funcionalidades ficarem separados, interesses relativos a dados ficavam distribuídos em diversos módulos. O paradigma OO definiu que a separação deveria acontecer em duas dimensões, primeiro dividido em termos de dados e depois em termos das funções que utilizam cada tipo de dados.



A orientação a objetos melhorou as possibilidades de separação de interesses. No entanto, ainda tem deficiências nessa área. Os diagramas abaixo mostram uma representação gráfica do código do sistema Tomcat, um servidor web com capacidade de executar *servlets* Java. Cada coluna representa um módulo do sistema, sendo que o tamanho de cada coluna mostra o número proporcional de linhas de código daquele módulo. Como podemos ver no diagrama abaixo, alguns interesses estão muito bem separados.

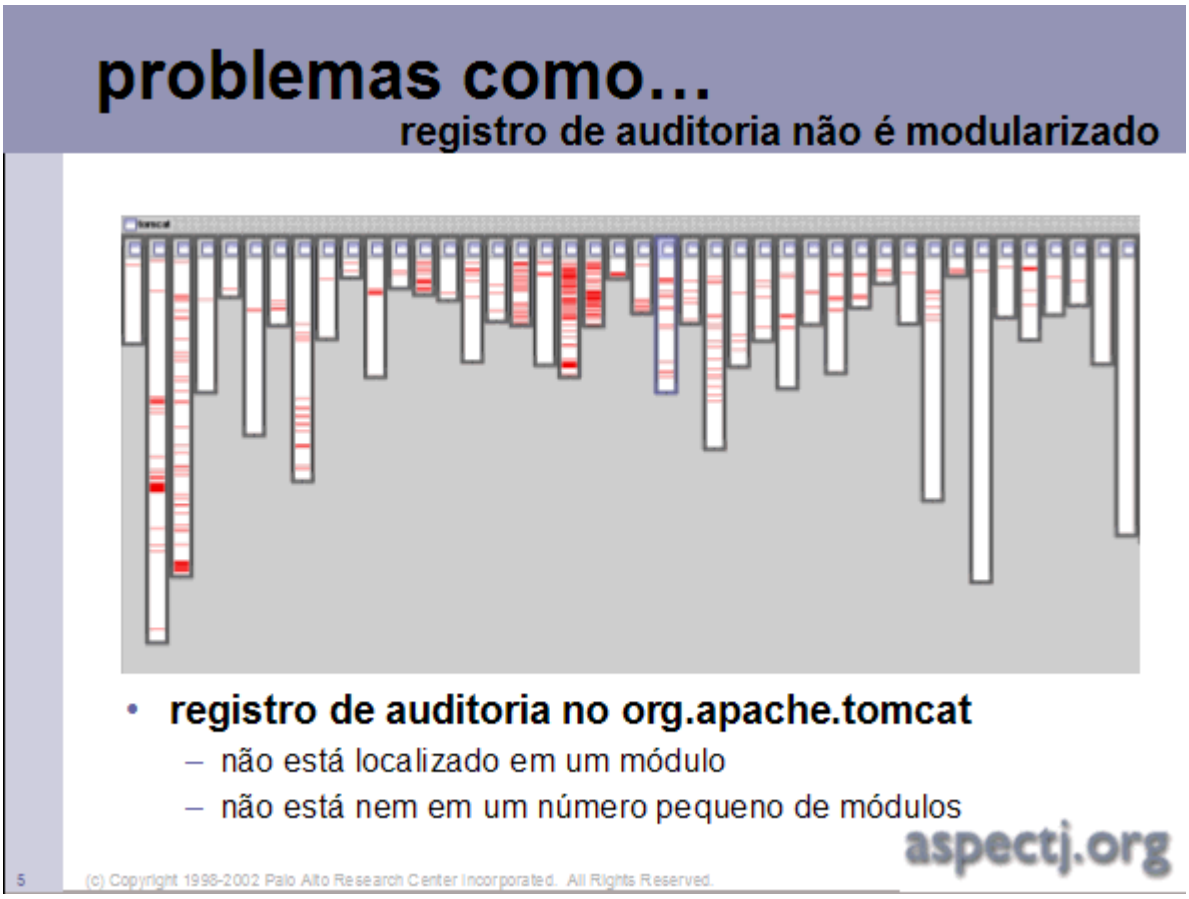


(figura extraída e traduzida do AspectJ Tutorial, de Erik Hilsdale e outros)



(figura extraída e traduzida do AspectJ Tutorial, de Erik Hilsdale e outros)

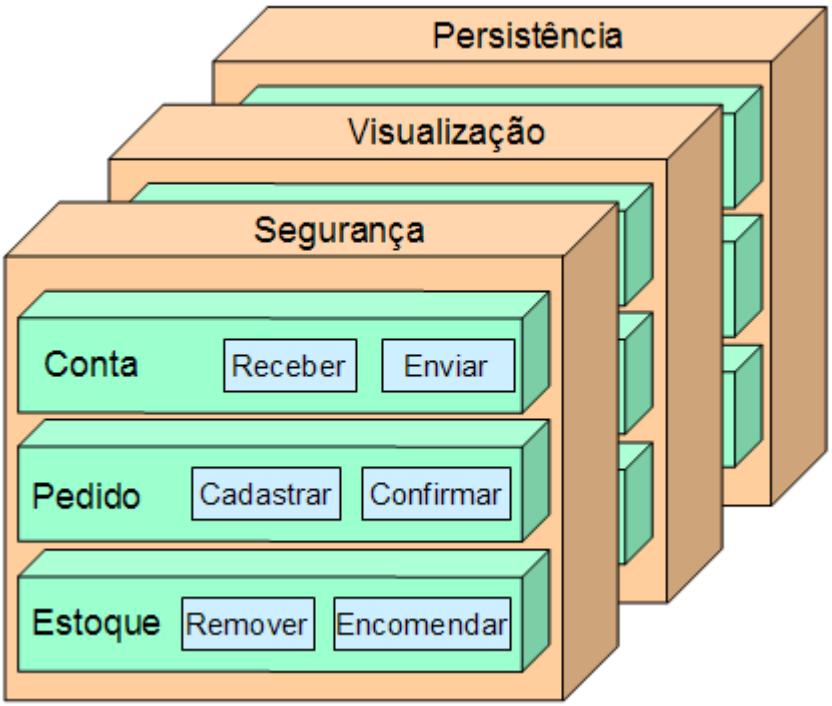
No entanto, isso nem sempre é verdade. Se considerarmos a funcionalidade de guardar registros para auditoria, isto é, registrar as ações tomadas pelo servidor para se detectar violações de segurança, erros, etc., então vemos que o código responsável por esse comportamento está espalhado por quase todos os módulos.



(figura extraída e traduzida do AspectJ Tutorial, de Erik Hilsdale e outros)

Na terminologia de orientação a aspectos, diz-se que a função de registro para auditoria é um *interesse entrecortante*, porque a sua implementação "corta" a estrutura de módulos do sistema. Praticamente todo programa orientado a objetos não-trivial contém interesses entrecortantes.

O objetivo do desenvolvimento orientado a aspectos é encapsular interesses entrecortantes em módulos fisicamente separados do restante do código. Esses módulos são denominados *aspectos*. Pensando em termos abstratos, a orientação a aspectos introduz uma terceira dimensão de decomposição. Além de decompor o sistema em objetos (dados) e métodos (funções), decompomos cada objeto e função de acordo com o interesse sendo servido e agrupamos cada interesse em um módulo distinto, ou aspecto.

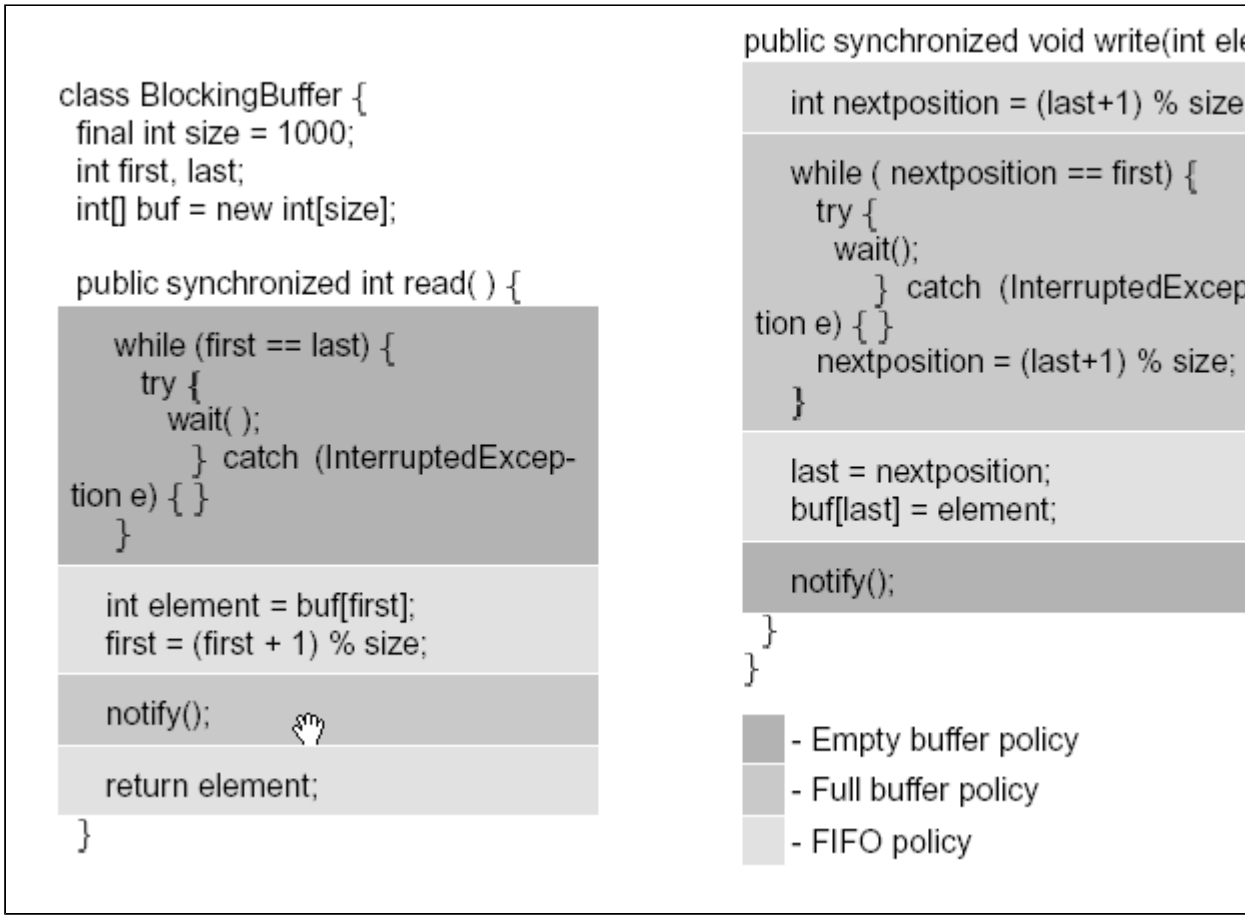

Persistência
Visualização
Segurança
Conta
Receber
Enviar
Pedido
Cadastrar
Confirmar
Estoque
Remover
Encomendar

Aula 5 - Exemplo prático

Considere um buffer de números inteiros, isto é, uma estrutura de dados compartilhada entre diversos processos, sendo que alguns escrevem dados no buffer e outros lêem e removem os dados.

Se um processo tenta ler dados do buffer vazio, ele é bloqueado até que haja algum elemento no buffer. Da mesma forma, se um processo tenta escrever em um buffer cheio, ele é bloqueado até que alguém leia algum elemento.

A figura abaixo mostra os métodos "ler" e "escrever" do buffer. Podemos dividir cada um desses métodos em três partes: a parte que lida com o que fazer quando o buffer está vazio (*política de buffer vazio*), a parte que lida com o que fazer quando o buffer está cheio (*política de buffer cheio*) e a parte que lida com o funcionamento normal do buffer, que diz que os elementos devem ser lidos na mesma ordem que são lidos (*política FIFO*).



Na terminologia de POA, dizemos que cada política está *espalhada* por diversos métodos. Analogamente, dizemos que cada método contém um *entrelaçamento* de diversas políticas. Em POA, tentamos agrupar funcionalidades espalhadas em entidades separadas, de forma a minimizar o entrelaçamento presente em cada método.

Por exemplo, abaixo temos duas dessas políticas separadas e agrupadas em um módulo fictício chamado "concern" (interesse). Cada método contém apenas a parte da operação que diz respeito à política em questão. Para obtermos a operação completa, no entanto, precisamos de

alguma forma de unir os módulos.

```
concern FIFOPolicy {
  class BlockingBuffer {
    final int size = 1000;
    int first, last;
    int[] buf = new int[size];

    public int read( ) {
      int element = buf[first];
      first = (first + 1) % size;
      return element;
    }
    public void write(int element) {
      int nextposition = (last+1) % size;
      last = nextposition;
      buf[last] = element;
} } }
```

```
concern FullPolicy {
  class BlockingBuffer {
    int size, first, last;

    public synchronized int read( ) {
      notify();
    }
    public synchronized void write(int
      int nextposition = (last+1) % size
      while ( nextposition == first) {
        try {
          wait();
        } catch (InterruptedException
        nextposition = (last+1) % size;
} } } }
```

Aula 6 - Anatomia de linguagens orientadas a aspectos

Composição de partes

O paradigma de orientação a aspectos envolve duas etapas de trabalho. A primeira é a decomposição do sistema em partes não entrelaçadas e não espalhadas --- essa é a parte fácil. A segunda envolve juntar essas partes novamente de forma significativa para se obter o sistema desejado. O processo de juntar as partes se chama *composição*. Há três questões a serem definidas em qualquer linguagem orientada a aspectos para se fazer a composição: a *correspondência*, a *semântica composicional*, e o *tempo de ligação*.

- **correspondência -** A forma de correspondência da linguagem é o modo com o qual se descreve quais entidades serão compostas entre si. A correspondência pode ser implícita (determinada por regras da linguagem) ou explícita (descrita pelo programador). Por exemplo, no exemplo acima que mostra duas políticas, podemos dizer que há uma correspondência implícita entre eles - os métodos com os mesmos nomes estão relacionados.
- **semântica composicional -** é o que deve acontecer com os elementos que correspondem. Em geral, linguagens de POA modificam a semântica das chamadas a métodos:
    - Em linguagens procedurais, chamar a função F implica em executar a função F.
    - Em linguagens orientadas a objetos, chamar o método M implica em executar algum método M em uma das subclasses que definem M.
    - Em linguagens orientadas a aspectos, chamar o método M pode ter diversas consequências:
        - M é executado, ou
        - N (algum outro método) é executado, ou
        - M+N são executados, em alguma ordem definida

  A linguagem pode definir diversas semânticas diferentes que em geral são escolhidas pelo programador. No exemplo da última aula, uma linguagem de POA permitiria que o usuário definisse o que acontece com os métodos que têm o mesmo nome (ou seja, que correspondem): se um for chamado, os dois devem ser executados? em que ordem? apenas um deve ser executado? qual?
- **tempo de ligação -** diz respeito ao momento em que a correspondência passa a surtir efeito; pode ser *estático* (em tempo de compilação) ou *dinâmico* (em tempo de execução).

A ferramenta ou compilador que faz a composição dos elementos em POA é chamado *weaver* (tecelão), pois "tece" os vários fragmentos de programa em um programa único.

A forma de composição das partes é o que realmente distingue linguagens orientadas a aspectos de outras linguagens. Em linguagens procedurais ou orientadas a objetos, a composição é feita através de chamadas de procedimentos ou métodos. Ou seja, uma parte (por exemplo, uma classe) usa a funcionalidade de outra chamando um método.

Em POA, não há chamadas explícitas de métodos entre partes. Ao invés disso, especifica-se, em uma parte separada, como uma parte deve reagir a eventos que acontecem em outra parte. Essa estratégia reduz o acomplamento entre as partes, pois as partes não se acessam diretamente.

Benefícios da programação orientada a aspectos

- *Menos responsabilidades em cada parte* - Como interesses entrecortantes são separados em seus próprios módulos, as partes do programa que lidam com a lógica de negócios não

ficam poluídas com código que lida com interesses periféricos.
- *Melhor modularização* - Como os módulos em AOP não se chamam diretamente, há uma redução no nível de acomplamento.
- *Evolução facilitada* - Novos aspectos podem ser acrescentados facilmente sem necessidade de alterar o código existente
- *Mais possibilidades de reutilização* - Como o código não mistura interesses, aumentam-se as possibilidades de se reutilizar módulos em sistemas diferentes.

Mitos e realidades da programação orientada a aspectos

- *É difícil seguir a lógica de programas orientados a aspectos* - Verdadeiro. Como um módulo não chama outro diretamente, é difícil inferir o comportamento do sistema como um todo avaliando-se módulos individuais. No entanto, esse mesmo fato torna mais fácil a compreensão de cada módulo isoladamente, pois ele não mistura outros interesses.
- *A POA não resolve nenhum problema novo* - Verdadeiro. A POA não tenta resolver problemas não solucionados. Há outras soluções para os problemas de espalhamento e entrelaçamento que não envolvem a criação de novas tecnologias. Por exemplo, há padrões de projeto (*design patterns*) que atacam problemas semelhantes, como o padrão *Strategy*.
- *A POA incentiva projetos mal feitos* - Falso. Soluções orientadas a aspectos não são a cura para programas ruins, são apenas ferramentas que fornecem novas formas de resolver problemas em áreas onde a orientação a objetos é deficiente. Na verdade, para se usar a POA com sucesso é preciso muito esforço no desenvolvimento do projeto.
- *A POA não é necessária pois com interfaces abstratas resolve-se os mesmos problemas em OO* - Falso. De fato, a melhor forma de se resolver o problema de interesses entrecortantes em sistemas OO é usando interfaces abstratas que vão implementar esses interesses. Apesar dessa abordagem ser muito usada em projetos OO, ainda é necessário que o restante do código chame os métodos fornecidos pelas interfaces abstratas. Ou seja, o problema é diminuído, mas não resolvido.
- *A POA quebra o encapsulamento* - Verdadeiro. Porém, as violações de encapsulamento são úteis se usadas de maneira controlada. Veremos mais sobre isso em uma aula futura.
- *A POA vai substituir a orientação a objetos* - Falso. A orientação a aspectos é uma tecnologia complementar à orientação a objetos e não funciona sem esta. As várias partes que compõem um programa orientado a aspectos ainda são implementadas dentro do modelo OO.

Aula 7 - O compilador, o IDE eclipse e um primeiro aspecto

O compilador

Programas orientados a aspectos precisam de um compilador específico. No caso da linguagem AspectJ, o compilador ajc transforma um programa escrito em AspectJ em um programa em bytecode Java, que pode ser executado por qualquer máquina virtual Java (JVM).

O IDE Eclipse

O Eclipse é um IDE (*Integrated Development Environment*, ou ambiente integrado de desenvolvimento) gratuito, produzido pela IBM, totalmente desenvolvido usando tecnologia Java.

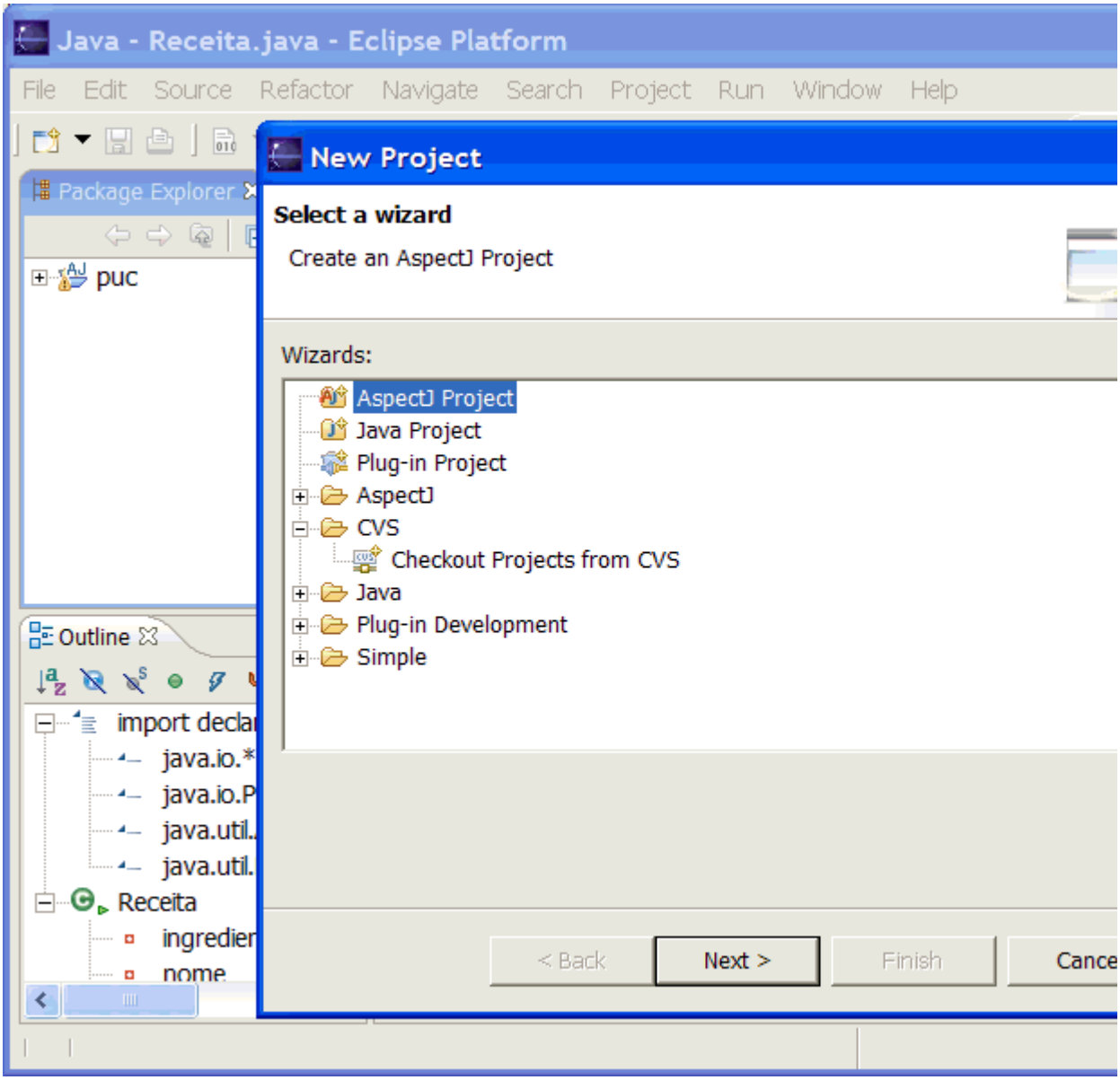
O Eclipse é extensível através de plugins. Existem inúmeros plugins disponíveis. Um deles, o AJDT, fornece suporte ao desenvolvimento com o AspectJ. As telas a seguir ilustram o uso do Eclipse com o AspectJ (as telas mostram o Eclipse versão 3.0; o laboratório da PUC tem a versão 2.1, então espere que haja pequenas diferenças).

Instalação do AJDT

- Ver as instruções originais em inglês, ou siga os passos abaixo

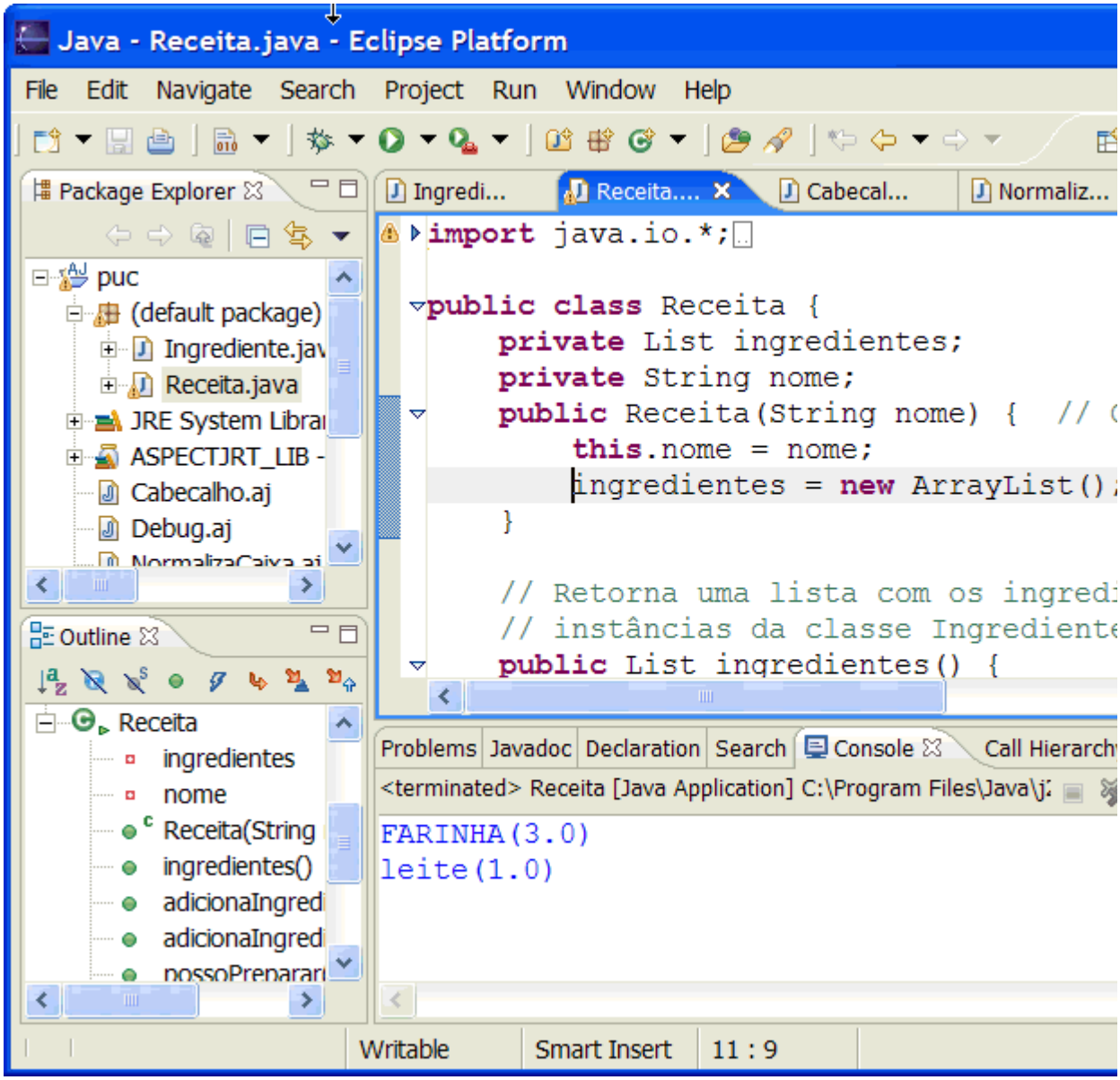
- **Eclipse 3.0**
  - Selecionar Help->Software Updates->Find and Install…
  - Entre com o nome "AJDT" e o URL "http://download.eclipse.org/technology/ajdt/30/update"
  - Clique OK
  - Abra (clicando no +) o nodo que aparece e selecione "AspectJ." Clique next. Selecione "Eclipse AspectJ Development Tools 1.1.12"
  - Clique next, clique na opção de "I accept..." e siga com a instalação até o final.
- **Eclipse 2.1**
  - Selecionar Help->Software Updates->Update Manager
  - Clique com o botão da direita no painel "Feature Updates" e selecione New->Site Bookmark, do menu que aparece.
  - Entre com o nome "AJDT" e o URL "http://download.eclipse.org/technology/ajdt/update"
  - Clique Finish
  - Abra (clicando no +) o nodo que aparece (AJDT) e selecione "AspectJ." Abra novamente (no +) e selecione "Eclipse AspectJ Development Tools 1.1.4"
  - Clique "Install now..." no painel da direita.

Para começar a desenvolver um sistema em AspectJ, é preciso primeiro criar um projeto. Para isso, selecione *File->New->Project*. Dentre as opções que aparecem, selecione "AspectJ Project".



Um projeto AspectJ tem classes Java comuns e aspectos. Vamos acrescentar os arquivos Receita.java e Ingrediente.java ao projeto. Para isso, é preciso selecionar *File->New->Class* e entrar com o nome da classe que se quer criar. O Eclipse criará uma classe em branco. Como já temos a classe feita, podemos copiar-e-colar o corpo da classe para dentro do ambiente do Eclipse.

Tendo criado as duas classes, podemos executá-las e ver seu resultado. O Eclipse 3.0 compila as classes automaticamente, mas no Eclipse 2.1 é preciso compilar as classes usando Project->Rebuild All. Para executar as classes, é só selecionar no painel "Package explorer" a classe que se quer executar (ou seja, a que define o método *main*; no nosso caso, a classe *Receita*) e selecionar *Run->Run As...->Java Application*. O resultado aparece no painel "Console".



Agora pode-se acrescentar aspectos ao projeto. Para tanto, basta incluir, por exemplo, os arquivos Cabecalho.aj, NormalizaCaixa.aj, e Debug.aj. Depois é só compilar (Eclipse 2.1) e executar novamente a classe receita. Note que a saída é modificada pelos aspectos simplesmente por terem sido incluídos, sem ter de mudar as classes Receita ou Ingrediente.

Aula 8 - Visão geral do AspectJ

Elementos do AspectJ

- **Pontos de junção** (*join points*) - um ponto de junção é qualquer ponto identificável pelo AspectJ durante a execução de um programa. O AspectJ permite diversos tipos de pontos de junção: entradas e saídas de métodos, tratamento de exceções, acessos a variáveis de objetos, construtores, entre outros.
- **Pontos de corte** (*pointcuts*) - pontos de corte são construções de programa que permitem a seleção de um ou mais pontos de junção. Pode-se usar expressões regulares para se especificar os pontos de corte, permitindo grande flexibilidade.
- **Advice** - advice, ou "conselho", é um trecho de código que deve ser executado em pontos de junção selecionados por um ponto de corte. O advice pode ser executado antes, depois, ou "em volta" (*around*) do ponto de junção. O around pode ser usado para substituir o ponto de junção pelo advice, ou para executar código antes e depois do ponto de junção.
- **Introduções** (*introduction*) - uma introdução é uma forma de um aspecto introduzir mudanças em classes, interfaces e aspectos do sistema. Por exemplo, pode-se acrescentar uma variável ou método a uma classe existente.
- **Aspectos** - o aspecto é a unidade modular principal em AspectJ, assim como a classe é a unidade principal em Java. Um aspecto agrupa pontos de corte, advice, e introduções.

Alguns Aspectos

O elemento principal do AspectJ é o *aspecto*. Cada aspecto assemelha-se a uma classe e pode ter tudo que uma classe tem: definições de variáveis, métodos e restriçoes de acesso. Além desses elementos, no entanto, aspectos podem ter outros elementos. Os dois tipos principais de elementos no AspectJ são *pointcuts* ("pontos de corte") e *advice* ("conselhos").

Pontos de corte são definições de instantes durante a execução de um programa. Os pontos de corte não tem nehum comportamento; são definições de *correspondência* (ver aula 6). Eles denotam *onde* e *quando* os aspectos terão efeito no programa como um todo.

Um advice é um comportamento. Ele especifica não só o que será feito, na forma de uma sequência de operações Java, mas também o momento em que serão feitas as operações. Todo advice é relacionado a um ou mais pontos de corte.

Vejamos um exemplo de aspecto simples. O aspecto a seguir define um ponto de corte que equivale a uma chamada ao método *print* da classe *Receita*. O asterisco no pointcut denota que o ponto de corte se refere a qualquer tipo de retorno, ou seja, ele se refere ao método Receita.print (), não importa qual o tipo do valor retornado pelo método.

```
public aspect Cabecalho {
        // Ponto de corte
        pointcut cabecalho() : call (* Receita.print());

        // Advice
        before() : cabecalho() {
                System.out.println("-------RECEITA DA COZINHA INTELIGENTE-----
        }
}
```

Além do ponto de corte, o aspecto define um advice do tipo *before* (antes). Além disso, ele se

refere ao pointcut definido anteriormente. Isso significa que o programa definido dentro do advice será executado antes do instante definido pelo pointcut. Como o ponto de corte define o instante da chamada ao método print() da classe Receita, o advice define que antes de cada chamada desse tipo será impressa uma mensagem.

O compilador AspectJ verifica se existe ou não um ponto de corte que case com o definido dentro do aspecto e gera código para aplicar o advice no local apropriado. A mudança no programa é automática. A classe Receita é executada e, sem que haja nenhuma referência ao aspecto, este é executado no instante definido pelo pointcut.

Outros aspectos podem ser acrescentados ao projeto da mesma forma. O aspecto a seguir define um ponto de corte que engloba todos os métodos *toString()* de todas as classes. O advice presente nele é do tipo *around*, que significa que o código do advice será executado ao invés do método chamado, ou seja, o método *toString* chamado não é executado, e sim o código do advice. Este, por sua vez, chama o método *proceed()* (prosseguir), que é uma palavra-chave do AspectJ que executa o método que foi substituído. Nesse exemplo, o objetivo é chamar o método *toString()* correspondente ao aspecto e guardar seu resultado na variável *s*. Depois disso, o método converte o string recebido em letras minúsculas usando *toLowerCase()* e retorna o string convertido para o método que chamou *toString()* originalmente.

```
public aspect NormalizaCaixa {
        pointcut string() : call (* *.toString());

        String around() : string() {
                 String s = proceed();
                 return s.toLowerCase();
        }
}
```

O último aspecto, chamado Debug, define um ponto de corte que inclui todos os métodos do sistema, exceto os do próprio aspecto Debug. O advice associado é do tipo *after*, que executa após terminar a execução do método correspondente no ponto de corte. O advice usa a técnica da introspecção (ou reflexão), uma tecnologia que permite que um programa descubra fatos sobre sua própria estrutura. Nesse caso, o aspecto está verificando o nome do método que foi chamado e após o qual o advice está executando. Ele então imprime uma mensagem com o nome do método.

```
public aspect Debug {
        pointcut debug() : call(* *.*(..)) && !within(Debug) && !within(Normal

        after() : debug() {
                String nomeMetodo = thisJoinPointStaticPart.getSignature().get
                System.out.println("DEBUG: Saindo do metodo " + nomeMetodo);
        }
}
```

O compilador AspectJ gera bytecode com os aspectos entremeados entre as classes. Só para ter uma idéia de como seria a saída do compilador se este transformasse os aspectos em Java puro, seria algo assim, no caso do primeiro aspecto, "Cabeçalho":

```
public class Cabecalho {
    void before$print( ) {
              System.out.println("-------RECEITA DA COZINHA INTELIGENTE-------'
    }
}
```

```
public class Receita {

    Cabecalho c;

    ...  // parte da classe removida

        public Receita() {  // construtor
             c = new Cabecalho();
    }

        public void print() {
            c.before$print();
                System.out.println(this.toString());
        }
}
```

Aula 9 - Pontos de junção

Um ponto de junção é qualquer ponto identificável pelo AspectJ durante a execução de um programa. Aspectos podem ser associados a pontos de junção e executados antes, depois, ou ao invés deles. Existem diversos pontos de junção reconhecidos pelo AspectJ:

- **Métodos**
  Existem dois tipos de pontos de junção relacionados a métodos: pontos de chamada de métodos e pontos de execução de métodos
  - **Chamada de métodos** - São os pontos indicativos de onde, no código, o método é chamado.
  - **Execução de métodos** - É o corpo dos métodos propriamente ditos.

  Por quê precisamos de dois pontos de junção diferentes? Não bastaria um desses? Eles não são equivalentes?
  Na verdade, há uma diferença: como em linguagens orientadas a objetos existe polimorfismo, é possível chamarmos um método de uma classe e executarmos um método de uma de suas subclasses. Assim, é útil fazer a distinção em se tratando de aspectos. Além disso, como a chamada e execução podem estar em arquivos fisicamente diferentes, pode-se gerar pontos de corte compostos onde a diferença entre chamada e execução torna-se mais importante.

- **Construtores**
  Assim como métodos, construtores têm dois pontos de junção: um para chamadas ao construtor e outro para a execução do construtor. Pontos de junção de chamada de construtores são encontrados em locais onde um objeto é instanciado com comandos *new*. Pontos de junção de execução de construtores são o próprio código do construtor.

- **Acesso a campos**
  Existe um ponto de junção em cada acesso aos dados de uma classe. Os pontos são divididos em pontos de leitura, onde o dado é usado mas não modificado, e pontos de escrita, onde o dado é modificado.

  É importante notar que só há pontos de junção em dados declarados no escopo da classe. Não há pontos de junção para variáveis locais a métodos.

- **Tratamento de exceções**
  Um ponto de junção de tratamento de exceções é o interior de um bloco *catch* que 'captura' algum tipo de exceção.

- **Inicialização de classes**
  São pontos de junção que representam o instante em que uma classe é carregada na memória e seus dados estáticos são inicializados.

  Há outros tipos de pontos de junção menos usados: inicialização de objetos, pré-inicialização de objetos, e execução de *advice*.

Aula 10 - Pontos de corte (*pointcuts*)

Sintaxe básica

Um ponto de corte (*pointcut*) é uma construção sintática do AspectJ para se agrupar um conjunto de pontos de junção. Sua sintaxe básica é a seguinte (ilustrada com um exemplo):

```
public pointcut nome() : call (void Receita.print())
```

- A primeira parte é a declaração de restrição de acesso - nesse caso, *public*, mas pointcuts podem ser *private* ou *protected*.
- A palavra-chave *pointcut* denota que estamos declarando um ponto de corte.
- Todo pointcut tem um nome qualquer, e pode receber parâmetros - nesse caso, o pointcut não recebe parâmetros.
- Depois dos dois pontos (:) obrigatórios vem o tipo dos pontos de junção agrupados pelo pointcut - nesse caso temos um pointcut do tipo *call*, que indica uma chamada de método.
- Finalmente vem a assinatura do pointcut, uma especificação dos pontos de junção aos quais o pointcut se refere.

Caracteres especiais

Na descrição da assinatura, pode-se usar alguns caracteres especiais para incluir mais de um ponto de junção no pointcut. Os caracteres são:

Caractere	Significado
*	Qualquer sequência de caracteres não contendo pontos
..	Qualquer sequência de caracteres, inclusive contendo pontos
+	Qualquer subclasse de uma classe.

Expressões lógicas

Um pointcut pode conter várias assinaturas ligadas através de operadores lógicos. Os operadores são:

Operador	Significado	Exemplo	Interpretação do exemplo
!	Negação	! Receita	Qualquer classe exceto Receita
||	"ou" lógico	Receita || Ingrediente	Classe receita ou classe ingrediente
&&	"e" lógico	Cloneable && Runnable	Classes que implementam ambas as interfaces Cloneable e Runnable

Pointcuts por tipo de ponto de junção

Já vimos pointcuts do tipo *call*, que se refere a chamadas de métodos. Além desse, existem outros tipos, listados a seguir com suas sintaxes:

Categoria de ponto de junção	Sintaxe do pointcut
Chamada de método ou construtor	call(AssinaturaDoMetodo)

Execução de método ou construtor	execution(AssinaturaDoMetodo)
Inicialização de classe	staticinitialization(Classe)
Leitura de dado de classe	get(AssinaturaDoCampo)
Escrita de dado de classe	set(AssinaturaDoCampo)
Tratamento de exceção	handler(Exceção)

Pointcuts de fluxo de controle

Há dois tipos de pointcuts que se referem a um conjunto de pontos de junção que ocorrem em um **fluxo de controle**. Um fluxo de controle consiste de todos os comandos que são executados entre a entrada em um método e a saída deste, inclusive considerando comandos em outros métodos chamados pelo primeiro.

Os dois pointcuts são *cflow( )* e *cflowbelow( )*. A única diferença entre eles é que *cflow( )* inclui entre os pontos de junção resultantes o próprio ponto de junção dado como parâmetro, enquanto *cflowbelow( )* inclui apenas os pontos de junção que estão "abaixo" do ponto dado. A tabela abaixo mostra um exemplo de cada tipo:

Ponto de corte	**Descrição**
cflow(call (Receita.print ())*	Todos os pontos de junção que ocorrem durante a chamada ao método *print* da classe *Receita*, incluindo a própria chamada.
cflowbelow(execution(void Receita.print())	Todos os pontos de junção que ocorrem durante a execução do método *print* da classe *Receita*, NÃO incluindo a própria execução.

Pointcuts baseados na estrutura léxica

Dois tipos de pointcuts levam em consideração trechos do código do programa, sem levar em consideração a execução. O primeiro é *within(Tipo)*. Esse pointcut inclui todos os pontos de junção dentro de uma classe ou aspecto passada como parâmetro. O segundo é *withincode (Método)*, que inclui todos os pontos de junção dentro de um método ou construtor.

Ponto de corte	**Descrição**
within(Receita+)	Todos os pontos de junção que ocorrem dentro da classe Receita ou de qualquer de suas subclasses.
withincode(Receita.set*(..)*	Todos os pontos de junção que ocorrem dentro de métodos da classe Receita cujos nomes começam com "set".

Pointcuts de objetos em execução

O pointcut *this(Classe)* compreende todos os pontos de junção de um objeto que é da classe dada ou de qualquer subclasse dela. A diferença para *within* é basicamente que *within* é calculado em tempo de compilação e *this* em tempo de execução, e *within* não inclui subclasses.

Pointcuts condicionais

O pointcut *if(condição)* compreende todos os pontos de junção em que a condição for verdadeira, em tempo de execução.

Exercício em aula

Considere a seguinte classe:

```
public class ExemploPointcut extends Exemplo {
   String titulo;
   int dificuldade;

   ExemploPointcut() {
       titulo = new String("Exemplo de pointcut");
           dificuldade = 1;
   }

   ExemploPointcut(String s, int diff) {
       titulo = new String(s);
           dificuldade = diff;
   }

   void Resolver(int diff) {
        if (dificuldade == diff) {
                    System.out.println("Resolvido!");
            }
        }
}
```

Escreva declarações de pointcuts para os seguintes pontos de junção:

1. A execução do método Resolver dessa classe.
2. Chamadas a de qualquer dos construtores dessa classe.
3. Acessos para leitura à variável "dificuldade".
4. Acessos para escrita de qualquer variável.
5. Chamadas a qualquer método dessa classe ou da classe "Receita".
6. Chamadas a qualquer método que comece com "R" no sistema.
7. Execuções de métodos que recebem um inteiro como parâmetro.
8. Tratamentos da exceção IOException em qualquer método da classe "Receita".

Aula 11 - Advice

Tipos de advice

Advice é uma estrutura que denota o que um aspecto deve fazer, ou seja, qual o comportamento do aspecto. Em termos mais formais, o advice designa a *semântica comportamental* do aspecto. Todo advice está associado a um pointcut, que define pontos de junção. Há três tipos de advice:

1. *before*: executa antes do ponto de junção
2. *after*: executa depois do ponto de junção
3. *around*: executa "em volta" do ponto de junção; esse tipo de advice serve para substituir a execução do ponto de junção pela execução do advice, ou executar parte do advice antes do ponto e junção e outra parte depois.

Todos os tipos de advice são definidos de forma semelhante:

```
before():                              // declaração do advice
   ponto_de_corte() {                  // ponto de corte associado
      System.out.println("teste");     // corpo do advice
      System.out.println("tchau");
   }
```

Advice do tipo *before*

O advice do tipo *before* é o mais simples: ele simplesmente executa antes do ponto de junção. O único detalhe a se observar é que se o advice levantar uma exceção, o ponto de junção não será executado.

Advice do tipo *after*

O advice do tipo *after* executa após o ponto de junção. Existem três variacões desse advice, que dependem de como terminou a execução do ponto de junção: se terminou normalmente, com uma exceção, ou de qualquer forma.

after() : ponto_de_corte()	executa depois do ponto de junção, independente de como ele retornou
after() returning : ponto_de_corte()	executa depois do ponto de junção se ele tiver terminado normalmente (sem exceção)
after() throwing : ponto_de_corte()	executa depois do ponto de junção somente se este tiver saído com uma exceção

Tanto a forma *returning* quanto a *throwing* podem definir um parâmetro que conterá o valor retornado ou a exceção jogada, para ser usada pelo aspecto. Por exemplo:

```
after() returning(String s) : call(* Receita.getNome()) {
     System.out.println("O metodo retornou o string " + s);
}
```

Advice do tipo *around*

O corpo de um advice around é executado substituindo o ponto de junção. No entanto, o ponto

de junção pode ser executado de dentro do corpo do advice usando-se o comando *proceed( )*.

A chamada a *proceed( )* retorna o mesmo tipo retornado pelo ponto de junção substituído. Além disso, se o ponto de junção é um método que espera receber argumentos, estes devem ser passados no *proceed( )*. Veremos como passar esses argumentos na seção seguinte.

Todo advice *around* deve declarar um tipo a ser retornado. O tipo deve ser o mesmo tipo dos pontos de junção associados a ele. Por exemplo, se o advice está associado à chamada de um método que retorna um inteiro, ele deve retornar um inteiro. Todos os pontos de junção de um advice *around* devem retornar um tipo compatível com o tipo retornado pelo advice. Por exemplo:

```
String around() : call (* *.toString()) {
    String s = proceed();
    return s.toLowerCase();
}
```

Se os pontos de junção retornam tipos diferentes, pode-se contornar isso declarando que o advice retorna *Object*. O AspectJ cuidará para que o tipo correto seja retornado ao chamador, mesmo que o tipo seja um tipo primitivo.

Da mesma forma que tipos de retorno, o advice deve declarar que joga (throws) exceções jogadas pelo método capturado.

Passagem de contexto

O aspecto pode usar informações do ponto de junção capturado. Essa informação faz parte do *contexto* do ponto de junção. É dever do ponto de corte (pointcut) expor o contexto desejado para que o advice possa usá-lo. Por exemplo, suponha que queiramos imprimir uma mensagem quando inserimos um ingrediente a uma receita, detalhando a operação efetuada:

```
before(Receita r, Ingrediente i) :
   call(void adicionaIngrediente(Ingrediente)) &&
   target(r) &&
   args(i) {
        System.out.println("Inserindo ingrediente"
                  + i + " na receita "
                  + r.getNome() );
   }
}
```

O pointcut *target* captura o objeto que é o alvo da chamada de método (o *call* especificado). O pointcut *args* captura os argumentos passados como parâmetros (no caso, um objeto da classe Ingrediente). Pointcuts do tipo *this* também servem para capturar contexto. No caso, o *this* captura o objeto onde está o ponto de junção (o call, nesse caso). No exemplo, poderíamos substituir "this" por "target" e obter o mesmo efeito? Experimente!

Pointcuts já podem ser definidos com os parâmetros desejados. Poderíamos escrever o exemplo anterior da seguinte forma:

```
pointcut insercao(Receita r, Ingrediente i) :
   call(void adicionaIngrediente(Ingrediente)) &&
   target(r) && args(i);
```

```
before(Receita r, Ingrediente i) : insercao(r, i) {
        System.out.println("Inserindo ingrediente"
                    + i + " na receita " + r.getNome() );
   }
}
```

Aula 12 - Introduções

Introduzindo novos membros em classes

Além de modificar o comportamento das classes de um sistema, um aspecto pode introduzir novos membros (dados ou métodos) a classes existentes. Esse mecanismo é chamado de *introdução*.

Por exemplo, suponha que queremos um mecanismo para nos avisar quando o nível de um ingrediente estiver baixo na despensa da casa. Suponha que a despensa tem um método chamado *usarIngrediente(Ingrediente i, int quantidade)*, que diminui a quantidade em estoque do ingrediente desejado. Podemos incluir nosso mecanismo de aviso com o seguinte aspecto:

```
public aspect QuantidadeMinimaEmEstoque {

   private int Despensa.quantidadeMinima;

   public boolean Despensa.abaixoDoMinimo(Ingrediente i) {
        return i.getQuantidade >= quantidadeMinima;
   }

   after(Despensa d) : execution(Despensa.new(..)) && this(d) {
        d.quantidadeMinima = 3;
   }

   after(Despensa d, Ingrediente ing, int quantidade) :
        execution(* Despensa.usarIngrediente(Ingrediente, int)) &&
            args(ing, quantidade) && this(d) {
             if (d.abaixoDoMinimo(i)) {
                            System.out.println("Comprar mais " + i.getNome());
                        }
   }
}
```

Modificando a hierarquia de classes

Além de introduzir membros, o aspecto pode acrescentar relacionamentos de herança a classes existentes. As duas possibilidades são (ilustradas com exemplos):

- Fazer com que uma ou mais classes implementem uma ou mais interfaces:

  ```
  declare parents : Receita* implements Cacheable, Cloneable;
  ```

  (todas as classes que começam com "Receita" vão implementar as duas interfaces enumeradas).

- Fazer com que uma ou mais classes sejam filhas de uma classe:

  ```
  declare parents : Ingrediente extends Comida;
  ```

  (a classe Ingrediente torna-se filha da classe Comida).

A nova hierarquia de classes só é útil no contexto dos aspectos, ou seja, como o restante do programa não sabe que as classes têm a hierarquia definida no aspecto, não podem fazer uso

dela. No entanto, o aspecto pode. Além disso, o aspecto não pode violar as regras de herança da linguagem Java, ou seja, uma classe não pode herdar de mais de uma classe. Portanto, a declaração de *extends* acima não pode ser usadas com classes que já herdam de alguma outra classe.

Introduzindo erros de compilação e warnings

O aspecto pode forçar o compilador a apresentar erros caso algum ponto de junção seja encontrado. Isso é útil, por exemplo, para verificar políticas estabelecidas de uso de programas terceiros. Suponha que o arquiteto do sistema tenha decidido que não se deve usar o tipo de dados *ArrayList*, por questões de desempenho. Ele pode validar o sistema com a seguinte declaração em um aspecto:

```
declare error: call(java.lang.ArrayList.new(..)) :
    "Não é permitido usar ArrayLists";
```

Ao invés de *declare error*, pode-se usar *declare warning* para gerar um warning no compilador. Note que essas declarações somente podem ser usadas com pointcuts resolvidos estaticamente, pelo compilador, ou seja, não podem ser usados com cflow ou cflowbelow, nem this, target, args, ou if.

Aula 13 - Reflexão em Java

O que é reflexão?

Reflexão ou *introspecção* é a capacidade de um programa de investigar fatos sobre si próprio. Por exemplo, um objeto pode "perguntar" a outro quais os métodos que ele possui. A linguagem de programação precisa oferecer recursos para que a reflexão seja possível. A linguagem Java oferece esses recursos e o AspectJ oferece recursos adicionais para o uso de reflexão em aspectos. O objetivo dessa aula é dar uma visão geral dos mecanismos de reflexão da linguagem Java; a próxima aula vai tratar de introspecção em AspectJ.

A classe *Class*

Toda classe Java que tem alguma instância existe na memória na forma de uma instância da classe *Class*. Essas instâncias contêm todas as informações de uma classe - seu nome, dados, métodos, etc - além de métodos para recuperar essas informações. Há três formas de obtermos instâncias da classe *Class*:

1. Usando o método getClass() em um objeto qualquer:
   Class class = r.getClass();
2. Usando o sufixo *.class* em uma classe conhecida:
   Class class = java.io.FileReader.class;
3. Usando o método estático *Class.forName(String nome)*:
   Class class = Class.forName(stringComNomeDaClasse);

De posse de um objeto do tipo *Class* podemos chamar os seguintes métodos para obter mais informações sobre a classe (a lista a seguir é apenas um pequeno subconjunto dos métodos da classe Class):

- *String getName()* - retorna o nome da classe
- *Object newInstance()* - retorna um novo objeto que é instância da classe
- *Method[] getMethods()* - retorna uma lista de objetos da classe *Method*, representando os métodos da classe
- *Field[] getFields()* - retorna uma lista de objetos da classe *Field*, representando os campos da classe
- *Method getMethod(nome, tipos dos parâmetros)* - retorna o método que tem o nome e parâmetros dados, se existir.

De posse de um objeto e um método, podemos chamar o método usando *invoke*, da classe *Method*:

Object invoke(Object instância, List argumentos)

Observe a seguinte função que recebe o nome de uma classe como parâmetro e executa o método print() dessa classe, se ele existir:

```
static public void executaMetodoPrintDaClasse(String s) {
    try {
            Class c = Class.forName(s);
            Method m = c.getMethod("print", null);
            Object o = c.newInstance();
            m.invoke(o, null);
```

```
        } catch (Exception e) {
                System.out.println("Deu errado.");
        }
    }
```

Aula 14 - Reflexão em AspectJ

Já vimos três formas de capturar o contexto de um ponto de corte em tempo de execução, usando as cláusulas *this()*, *target()* e *args()*. Além dessas formas, é possível usar reflexão para descobrir detalhes sobre o ponto de junção que está sendo tratado por um aspecto. Nessa aula veremos quais são as possibilidades oferecidas pelo AspectJ em termos de reflexão. Note que a reflexão é relativamente lenta, então só devemos utilizá-la quando não for possível obter a informação necessária usando *this()*, *target()* e *args()*.

Existem três objetos que podem ser acessados de dentro de um advice, para determinar informações sobre onde o aspecto está sendo executado:

- *thisJoinPoint* - contém informações dinâmicas sobre o ponto de junção. Os seguintes métodos podem ser chamados no objeto thisJoinPoint:
  - *Object[] getArgs()* - retorna os argumentos da função, semelhante ao que se obtém com o ponto de corte args().
  - *Object getTarget()* - retorna o objeto que foi alvo do ponto de corte, semelhante ao ponto de corte target() - por exemplo, no caso de uma chamada a método, o alvo é o objeto que recebe a chamada do método.
  - *Object getThis()* - retorna o objeto onde está o ponto de corte, semelhante ao ponto de corte this() - por exemplo, no caso de uma chamada a método, o objeto é aquele que está fazendo a chamada.
  - *getStaticPart()* - retorna o objeto *thisJoinPointStaticPart*, descrito a seguir.
- *thisJoinPointStaticPart* - contém informações estáticas (nome, assinatura, etc.) sobre o ponto de junção. Os seguintes métodos podem ser chamados no objeto thisJoinPointStaticPart:
  - *String getKind()* - retorna um string com o tipo do ponto de junção (call, execute, set, handler, etc.)
  - *Signature getSignature()* - retorna um objeto do tipo *Signature*, descrito mais abaixo, que contém a assinatura do método relativo ao ponto de junção.
  - *SourceLocation getSourceLocation()* - serve para descobrir a localização do ponto de junção no código fonte (arquivo, número da linha, etc.). Não usaremos esse método.
- *thisJoinPointEnclosingStaticPart* - contém informações estáticas (nome, assinatura, etc.) sobre o contexto que contém o ponto de junção. Por exemplo, se o ponto de junção é uma chamada de método, o contexto é a execução do método onde é feita a chamada. Esse objeto é do mesmo tipo que o *thisJoinPointStaticPart* e portanto tem os mesmos métodos.

Uma das classes utilizadas na reflexão é a classe *Signature*, que representa a assinatura de um ponto de junção. A classe *Signature* tem diversas subclasses (que não estudaremos) específicas para métodos, campos, exceções, e assim por diante. Alguns dos métodos da classe são:

- *Class getDeclaringType()* - retorna uma instância de *Class* que é o tipo retornado pelo método.
- *String getName()* - retorna o nome do ponto de junção (método ou variável).

Exemplo

Considere a seguinte adaptação do aspecto "Debug":

```
public aspect Debug {
```

```
        pointcut debug() : call(* *.*(..)) && !within(Debug);

        before() : debug() {
                String nomeMetodo = thisEnclosingJoinPointStaticPart.getSignat
                System.out.println("DEBUG: Metodo " + nomeMetodo);
        }
}
```

Considere agora a execução do seguinte método "main". Qual a saída do programa?

```
public class Receita {
        private List ingredientes;
        private String nome;

        // Acrescenta um ingrediente à receita
        public void adicionaIngrediente(Ingrediente i) {
                ingredientes.add(i);
        }

        // Programa main que demonstra alguns dos métodos implementados
        public static void main(String[] args) {
                Receita r = new Receita("bolo");
                r.adicionaIngrediente(new Ingrediente("FARINHA", 3));
        }
}
```

Resposta:

```
DEBUG: Metodo main
DEBUG: Metodo adicionaIngrediente
```

E se trocarmos o thisEnclosingJoinPointStaticPart por thisJoinPointStaticPart?

```
DEBUG: Metodo adicionaIngrediente
DEBUG: Metodo add
```

Aula 15 - Precedência entre aspectos

O que acontece quando dois "advice" se aplicam a um mesmo ponto de junção? Se estão declarados no mesmo aspecto, o AspectJ dá preferência aos que estão declarados primeiro. Se estão em aspectos diferentes, de modo geral a ordem em que os dois advice são executados é imprevisível. No entanto, o AspectJ permite que se determine a ordem de execução explicitamente.

Se um advice A do tipo "before" tem precedência sobre outro advice B do mesmo tipo, o A será executado antes do B. Se os dois fossem do tipo "after", o A seria executado *depois* do B, ou seja, o de maior precedência vem por último. Se os dois fossem do tipo "around", o de maior precedência seria executado e os outros não, a não ser que o método *proceed()* seja chamado. O mesmo vale se um advice "around" tiver maior precedência que outro do tipo "before" ou "after" - o outro só será executado se proceed() for chamado.

Ordenação de "advice" em um único aspecto

Quando temos um único aspecto com mais de um advice que se aplicam ao mesmo ponto de junção, os advice têm prioridade conforme a ordem em que estão definidos dentro do aspecto. Por exemplo, se há um advice "before" definido antes de um advice "around", ambos aplicados ao mesmo ponto de corte, o "before" será executado antes do "around". Se a ordem de definição deles for invertida, o "around" será executado primeiro e o "before" só será executado se o "around" chamar *proceed()*.

Ordenação explícita de "advice"

A ordenação é feita definindo-se quais aspectos têm *precedência* sobre outros. O AspectJ provê um comando para se definir a precedência:

```
declare precedence: aspecto1, aspecto2, ...
```

Essa declaração quer dizer que os advice do aspecto aspecto1 têm precedência sobre o aspecto2, o 2 sobre o 3, e assim por diante.

As duas regras de precedência devem ser combinadas quando necessário. No exemplo anterior, se o aspecto2 tiver mais de um advice, todos os advice do aspecto2 terão menor precedência que os do aspecto1.

Aula 16 - Associações entre aspectos e objetos

Por default, uma única instância de cada aspecto em um sistema é criada quando o sistema é executado, e essa instância continua a existir até que o sistema termine. No entanto, é possível fazer com que exista mais de uma instância de um aspecto.

As instâncias de aspectos nunca são criadas explicitamente pelo programador. São criadas sempre pelo próprio AspectJ e podem estar associadas a alguma entidade do sistema. A associação define quando a instância é criada e quando ela deixa de existir, além de determinar qual instância é usada quando um ponto de junção é atingido. Além da associação default, que é de uma instância por sistema, existem dois outros tipos de associação: por objeto e por fluxo de controle.

Associações por objetos

Quando definimos que um aspecto deve ter uma instância por objeto, a instância é criada automaticamente quando um objeto do tipo desejado é criado. O aspecto fica associado ao objeto durante toda a vida do objeto, e é usado sempre que algum dos pontos de junção do objeto for interceptado.

Há dois tipos de associação por objeto, *perthis(ponto_de_corte)* e *pertarget(ponto_de_corte)*. Ambos recebem um ponto de corte como parâmetro, que indica os pontos de junção que vão levar à criação de uma instância. A diferença entre eles é a mesma diferença que existe entre os pontos de corte *this()* e *target()*: o perthis() se refere ao objeto onde está o ponto de junção, e o pertarget() se refere ao objeto que é alvo do ponto de junção.

O seguinte exemplo mostra a sintaxe das definições de associação e será usado para ilustrar seu comportamento:

```
public aspect testaAssociacao perthis(execucoesReceita()) {
    pointcut execucoesReceita() : execution(* Receita.*(..));

        int x;

        // Construtor
        public testaAssociacao() {  x = 0;      }

        after() : execucoesReceita() {
            x = x+1;
            System.out.println("x = " + x);
        }
}
```

Se executarmos a classe Receita com o seguinte *main()*:

```
        Receita r1 = new Receita("bolo1");
        Receita r2 = new Receita("bolo2");
        r1.adicionaIngrediente(new Ingrediente("FARINHA", 3));
        r1.adicionaIngrediente(new Ingrediente("leite", 1));
        r2.adicionaIngrediente(new Ingrediente("FARINHA", 2));
        r2.adicionaIngrediente(new Ingrediente("leite", 1));
```

Obteremos a seguinte saída:

```
x = 1
x = 2
x = 1
x = 2
```

Observe que existe um contador (x) distinto para cada um dos objetos do tipo Receita (r1 e r2). Se removermos a cláusula perthis() do aspecto acima e executarmos novamente o main(), teremos a seguinte saída:

```
x = 1
x = 2
x = 3
x = 4
x = 5
```

Note que há apenas um contador para os dois objetos. Por que o contador chegou a 5 dessa vez, ao invés de 4? Porque o ponto de corte inclui o método *main( )*. Por que então o main( ) não foi considerado na primeira execução? Porque o main( ) é um método estático; métodos estáticos não pertencem a nenhuma instância; como o aspecto está associado somente a instâncias, o main ( ) não tem um aspecto associado.

Associações por fluxo de controle

Quando o aspecto é associado a um fluxo de controle, uma instância do aspecto é criada assim que começa a execução do ponto de junção especificado, e é removida quando o ponto de junção termina.

Há dois tipos de associação por fluxo de controle, *percflow(ponto_de_corte)* e *percflowbelow (ponto_de_corte)*. Sua aplicação é semelhante aos pontos de corte *cflow( )* e *cflowbelow( )*.

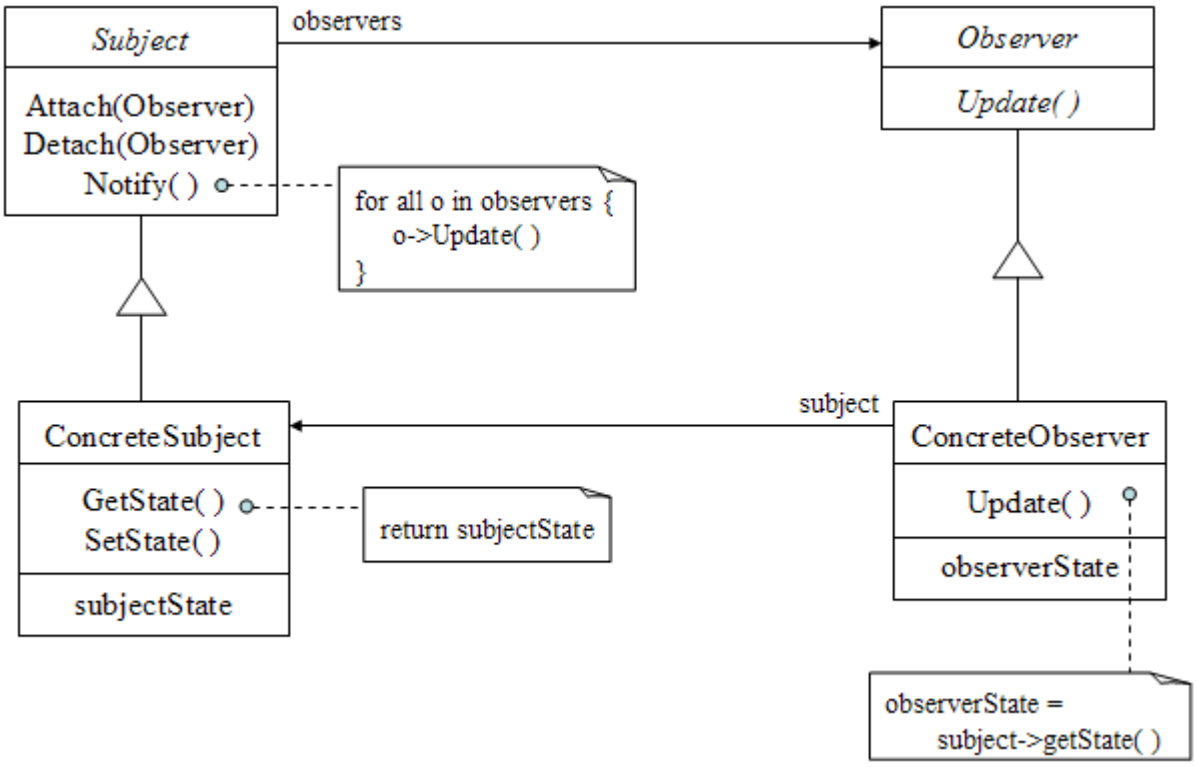
Se trocarmos o *perthis* do exemplo acima por *percflow*, obteremos a seguinte saída:

```
x = 1
x = 1
x = 1
x = 1
x = 1
```

Aula 17 - Aspectos e padrões de projeto

Padrões de projeto (*design patterns*) são modelagens de soluções para pequenos problemas que aparecem frequentemente durante a modelagem de sistemas orientados a objetos (pegue aqui uma apresentação PowerPoint sobre padrões de projeto).

Cada padrão é ilustrado através de um diagrama abstrato de classes. Por exemplo, abaixo temos o diagrama que demonstra o padrão *Observer*:



Um problema com o uso de padrões é que os padrões estão embutidos no código, ou seja, podemos ver o padrão como um interesse entrecortante. Seria interessante se pudéssemos remover o padrão do código, na forma de um aspecto, e aplicá-lo às classes que desejássemos sem termos de modificar essas classes. Nessa aula veremos um exemplo de aplicação de um padrão usando aspectos. O escolhido será o padrão Observer, mostrado acima. Nossa solução é baseada em uma solução encontrada em um artigo de Hannemann e Kiczales, "Design Pattern Implementation in Java and AspectJ".

Inicialmente, é preciso deixar claro que não é possível implementar o padrão Observer com o mesmíssimo comportamento do padrão definido acima, pois no padrão original cada observador "escolhe" qual o sujeito que vai observar. É impossível fazermos essa "escolha" automaticamente. Assim, vamos implementar uma versão do Observer em que todos os observadores vão se ligar ao mesmo sujeito.

Nossa implementação usará *pontos de corte abstratos*. Esses são pontos de corte cuja definição será feita em uma subclasse do aspecto. O aspecto com o padrão tem toda a parte genérica, mas teremos de criar um aspecto concreto para especificar os observadores e sujeito específicos da nossa aplicação.

```
import java.util.*;

public abstract aspect PadraoObserver {

        protected interface Subject { }
        protected interface Observer {
                void update(Subject s);
        }

        Subject sujeito;
        List observadores;

        abstract pointcut criarSujeito(Subject s);
        abstract pointcut criarObservador(Observer o);
        abstract pointcut modificarSujeito();

        after(Subject s) : criarSujeito(s) {
                if (sujeito == null) {
                        sujeito = s;
                }
        }

        after(Observer o) : criarObservador(o) {
                observadores.add(o);
        }

        after() : modificarSujeito() {
                Iterator i = observadores.iterator();
                while (i.hasNext()) {
                        Observer o = (Observer) i.next();
                        o.update(sujeito);
                }
        }
}
```

Agora precisamos implementar o aspecto concreto. Vamos usar como exemplo de sujeito uma classe MP3Player que representa um tocador de músicas MP3. Temos duas formas de aumentar o volume, um slider que movemos com o mouse e um valor numerico da percentagem do volume máximo, que podemos editar. Mudar um deve necessariamente mudar o outro. Por exemplo, se mexermos no slider, o valor do percentual deve mudar automaticamente.

```
public aspect ObserverConcreto extends PadraoObserver {
        pointcut criarSujeito(Subject s) : execution(MP3Player.new()) && this
        pointcut criarObservador(Observer o) :
                (execution(Slider.new()) || execution(TextLevel.new())) && th

        pointcut modificarSujeito() : execution(* MP3Player.setVolume(..));

        declare parents: MP3Player implements Subject;
        declare parents: Slider implements Observer;
        declare parents: TextLevel implements Observer;

        public void Slider.update(Subject s) {
                MP3Player p = (MP3Player) s;
                position = p.getVolume();
        }

        public void TextLevel.update(Subject s) { ... }
```

```
}
```

O aspecto concreto define os pontos de corte correspondentes ao papel do sujeito (MP3Player) e observadores (TextLevel e Slider). Além disso, define que essas classes são subclasses das classes Subject e Observer, através da implementação de interfaces. Finalmente, implementa os métodos *update* que são responsáveis por lerem do sujeito os valores desejados e atualizarem cada observador.

Aula 18 - Aplicações da orientação a aspectos

Logging, tracing e profiling

Logging consiste em guardar registros das ações executadas por um programa para fins de análise posterior. Em geral não se sabe se o log será necessário, mas ele pode vir a ser importante para encontrar defeitos no sistema. Logging é o exemplo canônico de POA: a forma mais simples de se mostrar as vantagens do paradigma é com um aspecto que intercepta todos os pontos de junção de chamadas de métodos e usa reflexão para mostrar informações sobre os métodos chamados. Assim pode-se demonstrar como aspectos simples podem acrescentar comportamentos em diversos pontos do sistema simultâneamente, sem modificá-los.

Tracing é basicamente o mesmo que logging, mas com objetivos diferentes: o resultado é usado somente para depuração de programas. Em geral o resultado é descartado após o uso e as rotinas de tracing nunca fazem parte do produto final que vai para o cliente.

Profiling é semelhante ao tracing, mas aqui o objetivo é juntar informações sobre o tempo de execução dos métodos para fins de otimização do sistema. Aspectos são úteis aqui também: pode-se interceptar as chamadas dos métodos que se deseja medir, zerar um contador de tempo antes da sua execução, e imprimir o contador após seu retorno.

Garantir o cumprimento de regras arquiteturais

Toda arquitetura define restrições, ou seja, ela define não só os subsistemas que se comunicam mas também os subsistemas que *não* devem se comunicar. No entanto, é comum que programadores violem regras arquiteturais, incluindo comunicações entre subsistemas que deveriam ser independentes, muitas vezes para contornar falhas ou dificuldades. No entanto, aquilo que torna o código correto também torna a arquitetura e o projeto incorretos, pois passam a não refletir a realidade do código.

Aspectos podem ser usados para garantir que a arquitetura seja seguida. Para isso, usa-se *declare error* e *declare warning* com expressões que identificam as chamadas inválidas. Por exemplo, suponha que classes do pacote puc.si.fronteira só possam ser utilizadas por classes do pacote puc.si.controlador. Podemos garantir isso com a seguinte declaração:

```
declare error: call(* puc.si.fronteira...(..)) && ! within(puc.si.controlador.
```

Otimização: pooling e caching

Muitas vezes a criação de um recurso é uma operação cara em termos de tempo de execução. Por exemplo, isso acontece com conexões ao banco de dados - leva tempo estabelecer uma conexão autenticada com um banco remoto. Se isso for feito a cada consulta, o programa será muito lento. A solução mais comum é manter um "pool" de conexões - um conjunto de conexões que permanecem abertas e que partes do programa requisitam quando precisam delas, depois as liberam quando não são mais necessárias.

Esse "pool" pode ser totalmente implementado através de aspectos. O aspecto pode criar o pool, interceptar as chamadas que solicitam os recursos (por exemplo, os pedidos de abertura de conexão) e retornar recursos do pool para quem os solicitou.

Outra forma de otimização é o uso de caches. Um cache rápido em memória pode ser utilizado

para guardar resultados de operações caras. Esses resultados podem ser usados se a operação tiver de ser repetida pelo sistema. Da mesma forma que com pooling, um aspecto de caching implementa o cache e intercepta todas as operações caras. Se o resultado desejado estiver no cache, ele é retornado imediatamente. Caso contrário, o aspecto executa a operação original e guarda o resultado no cache.

Acesso a recursos compartilhados em sistemas concorrentes

Em sistemas com várias *threads* ou processos paralelos que acessam memória compartilhada, é essencial que se implemente uma política para impedir condições de corrida e deadlocks no acesso à memória. Se o sistema já não tiver sido projetado com essas condições em mente, aspectos podem ser utilizados para garantir a segurança no acesso a recursos compartilhados.

Um aspecto pode interceptar todas as chamadas que utilizam o recurso compartilhado e manter uma "fila de espera" para que os interessados acessem o recurso um de cada vez. Pode-se inclusive implementar políticas diferentes para leitura e escrita, permitindo vários leitores simultâneos mas apenas um escritor.

Autenticação e autorização

É comum que sistemas tenham mais de um ponto de entrada possível. Por exemplo, em sistemas web, o usuário pode criar um "bookmark" que aponta diretamente para uma página que lhe é útil, de forma que ele não precise navegar sempre através da página principal. No entanto, muitos desses sistemas exigem que o usuário se autentique com sucesso para ter acesso aos dados. Isso apresenta mais uma oportunidade excelente para o uso de aspectos.

As rotinas de autenticação podem ser implementadas através de aspectos, e os aspectos podem ser associados a todas as possíveis "portas de entrada". Todas as chamadas são interceptadas e redirecionadas ao aspecto de autenticação. Se o usuário já estiver autenticado, o controle retorna à chamada original. Caso contrário, o usuário deverá se autenticar antes.

Acesso transacional

Uma *transação* é uma operação que envolve várias operações menores mas onde se garante que ou todas as operações serão executadas com sucesso, ou nenhuma delas será. Por exemplo, em um sistema de comércio eletrônico, ao se efetuar uma compra, é feito um pedido de remessa ao cliente e um pedido de cobrança. Se houver algum problema em apenas uma das operações, o usuário poderá receber algo pelo qual não pagou, ou ser cobrado por algo que não receberá.

Usando-se aspectos, podemos interceptar as chamadas às operações que juntas formarão a transação. O aspecto pode realizar cada operação verificando se ela foi bem-sucedida e armazenando dados relativos à operação. Se alguma operação não for bem-sucedida, o aspecto deve tomar medidas para "desfazer" as operações que já tiverem sido realizadas.