UNIP – Universidade Paulista – Campus Tatuapé - SP
Ciência da Computação / Sistemas de Informação
Tópicos Especiais de Computação

Introdução a Programação Orientada a Aspectos

Parte 1 - Orientação a objetos

Um objeto é um componente de software - uma parte de um sistema que exibe certas características específicas. A seguir são 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.

Prof. Marcelo Nogueira

Parte 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*.

2.1 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.

2.1.1 Herança para construção

- Usa-se herança para construção quando o objetivo é reutilizar uma classe existente para criar semelhante;
- Um ou mais métodos são sobrepostos para se mudar à funcionalidade da classe;
- Objetivo é reaproveitamento de código;

- Em geral, o uso de herança somente para construção é fruto de um projeto ruim;
- Devem-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();
}
```

2.1.2 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 se pode 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.

2.2 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 {
...
}
```

Parte 3 - 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.

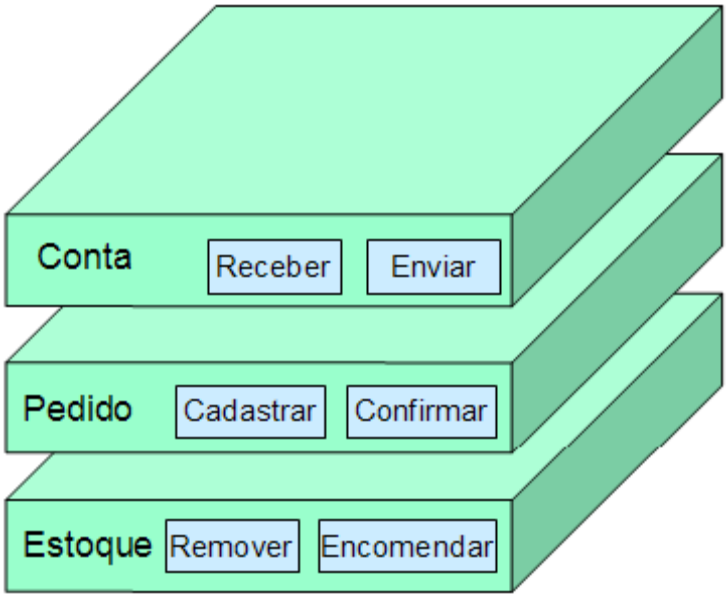
A divisão em partes é importante para se reduzir à complexidade. É muito difícil para um ser humano compreender um sistema de grande porte se este for monolítico, sem fronteiras claras que definem suas funções.

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.

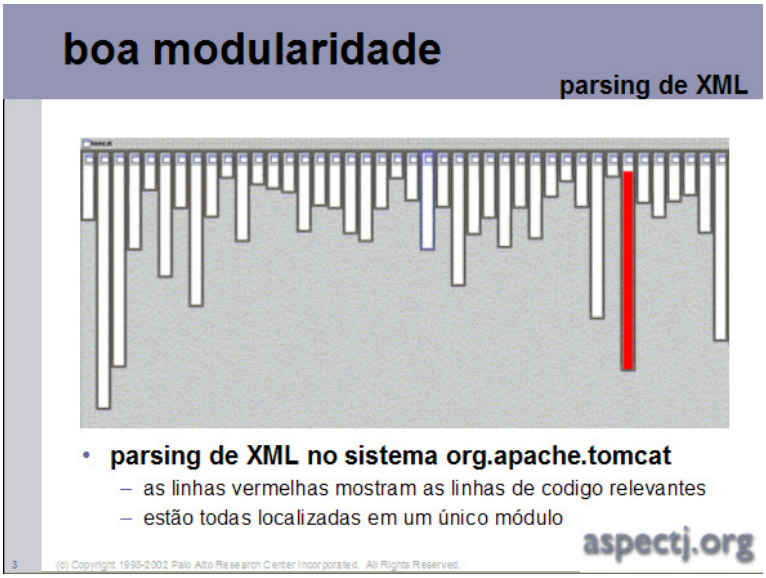
A orientação a objetos veio como forma de sanar uma das deficiências do desenvolvimento estruturado. Apesar de interesses relativos a funcionalidades ficarem separados, interesses relativos a dados ficavam distribuídos em diversos módulos.

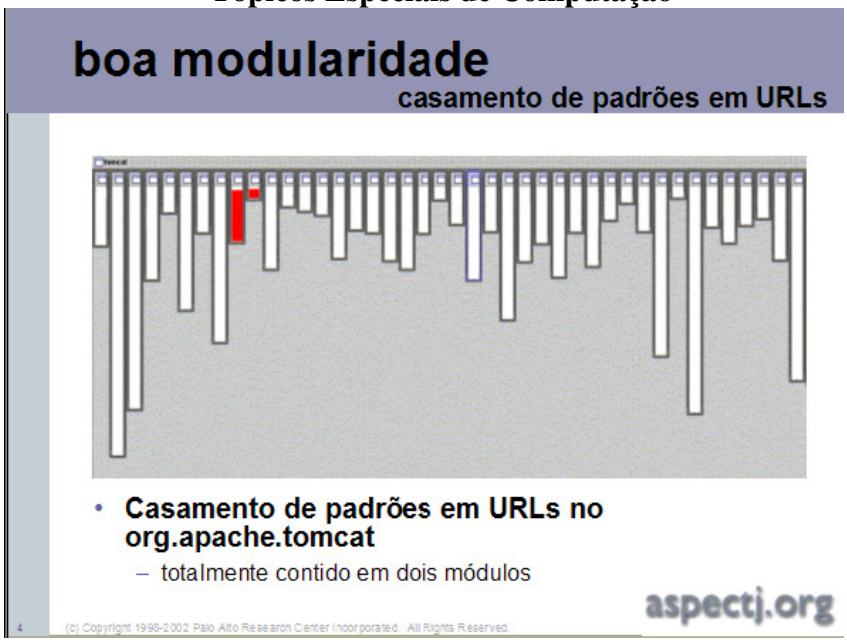
O paradigma OO definiu que a separação deveria acontecer em duas dimensões, primeiro dividido em termos de dados e depois em termos das funções que utilizam cada tipo de dados.

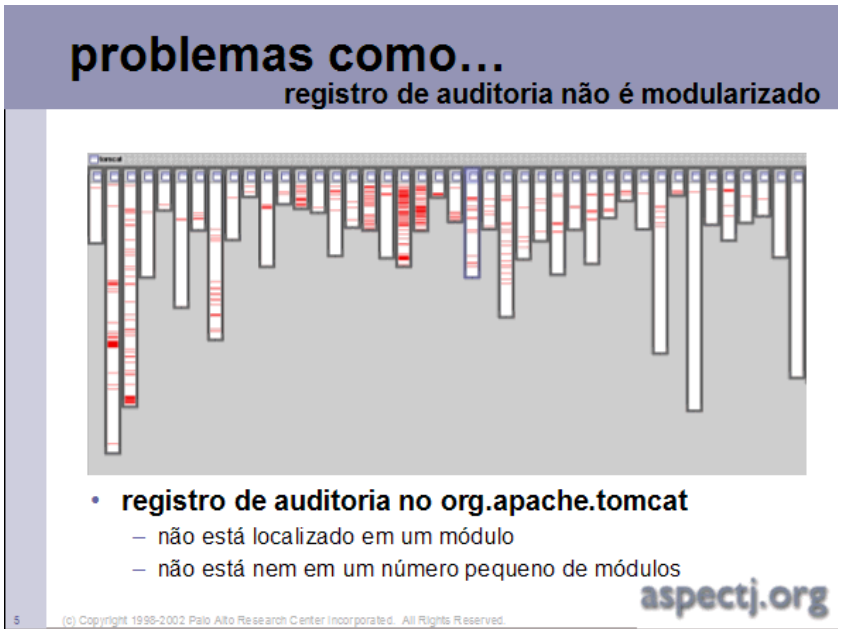


A orientação a objetos melhorou as possibilidades de separação de interesses. No entanto, ainda tem deficiências nessa área. Os diagramas abaixo mostram uma representação gráfica do código do sistema Tomcat, um servidor web com capacidade de executar servlets Java. Cada coluna representa um módulo do 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.

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 veremos que o código responsável por esse comportamento está espalhado por quase todos os módulos.





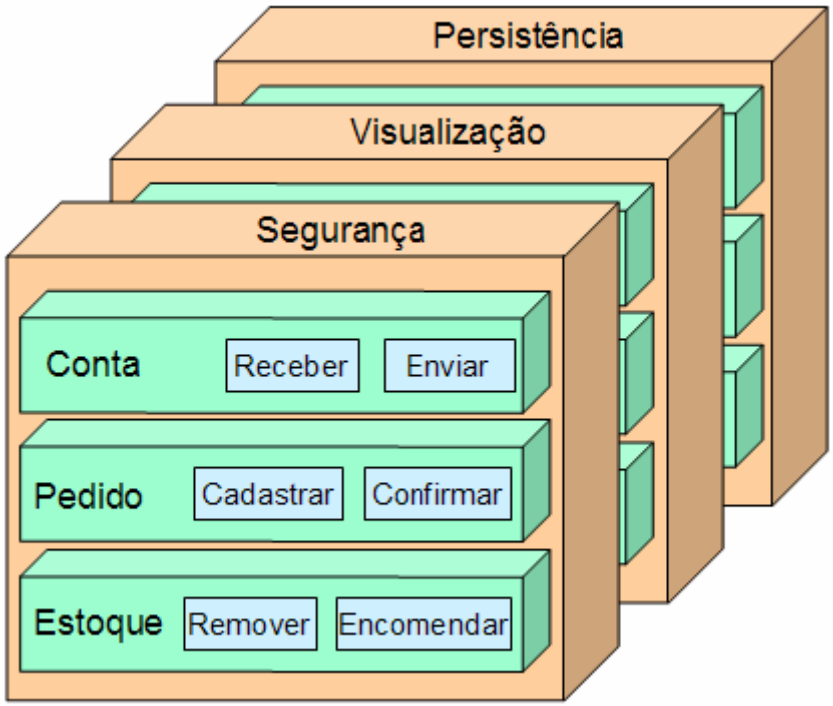


Na terminologia de orientação a aspectos, diz-se que a função de registro para auditoria é um interesse entrecortante, porque a sua implementação "corta" a estrutura de módulos do sistema.

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.



Parte 4 - Anatomia de linguagens orientadas a aspectos

4.1 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 conseqüências:
  - 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.
- 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 acoplamento entre as partes, pois as partes não se acessam diretamente.

4.2 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 Programação 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 acoplamento;
- 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.

4.3 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 de esta 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.

Referências Bibliográficas

Torsten Nelson, Programação Orientada a Aspectos com AspectJ,