Módulo III
Padrões GOF

Professores
Eduardo Bezerra – edubezerra@gmail.com
Ismael H F Santos – ismael@tecgraf.puc-rio.br

Ementa

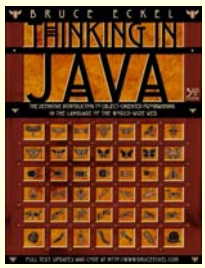
- Introdução aos padrões de projeto (GoF)
- Conceitos preliminares
  - Mecanismos de Herança
  - Princípio de Substituição de Liskov
  - Delegação
  - Delegação x Herança

Bibliografia

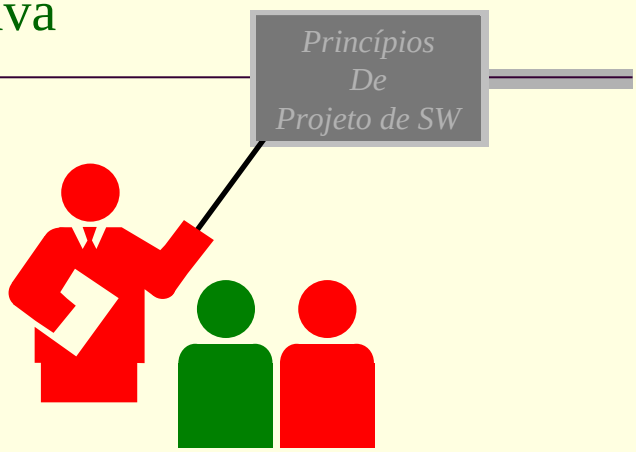
- *Craig Larman, Utilizando UML e Padrões, Ed Bookman*
- *Eric Gamma, et ali, Padrões de Projeto, Ed Bookman*
- *Martin Fowler, Analysis Patterns - Reusable Object Models, Addison-Wesley,1997*
- *Martin Fowler, Refatoração - Aperfeiçoando o projeto de código existente, Ed Bookman*

Livros

- **Core Java 2**, Cay S. Horstmann, Gary Cornell
  - Volume 1 (Fundamentos)
  - Volume 2 (Características Avançadas)
- **Java: Como Programar**, Deitel & Deitel
- **Thinking in Patterns with JAVA**, Bruce Eckel
  - **Gratuito**. http://www.mindview.net/Books/TIJ/



POO-Java



Padrões GoF

- Erich Gamma et al descreveram 23 padrões que podem ser aplicados ao desenvolvimento de sistemas de software orientados a objetos.
  - Gamma e seus colaboradores são conhecidos como a *Gangue dos Quatro* (Gand of Four, GoF).
- Os padrões catalogados pela equipe GoF possuem diversos nomes alternativos:
  - Padrões de projeto
  - Padrões GoF
  - Design patterns

Categorias de padrões GoF

- Padrões de projeto estão relacionados a
  - questões de comportamento de objetos
  - ciclo de vida de objetos
  - a interface dos objetos
  - relacionamentos estruturais entre objetos.
- Em função disso, os padrões GoF foram divididos em três categorias:
  - ***Criacionais***: têm a ver com inicialização e configuração de objetos.
  - ***Estruturais***: têm a ver com o desacoplamento entre a interface e a implementação de objetos.
  - ***Comportamentais***: têm a ver com interações (colaborações) entre sociedades de objetos.

Categorias de padrões GoF

Criacionais	*Estruturais*	*Comportamentais*
Abstract Factory Builder Factory Method Prototype Singleton	Adapter Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Interpreter Iterator Mediator Memento Observer State Strategy Template Method Visitor

POO-Java

Conceitos Preliminares

Mecanismos de herança

- **Herança estrita (extensão):** subclasse estende a superclasse, acrescentando novos membros (atributos e/ou métodos).
  - A superclasse permanece inalterada.
- **Herança de interface (especificação):** a superclasse especifica o que uma subclasse deve oferecer, mas não implementa nenhuma funcionalidade. Apenas a *interface* da superclasse é herdada pela subclasse.
- **Herança polimórfica:** a subclasse herda a interface e uma implementação de (pelo menos alguns) métodos da superclasse. A subclasse pode então redefinir métodos para especializar o comportamento em relação ao que é oferecido pela superclasse, ou ter que oferecer alguma implementação para métodos que a superclasse tenha declarado mas não implementado.

Mecanismos de herança em Java

- Uma interface define assinaturas de métodos e constantes.
  - Uma interface pode *estender* (i.e., ou seja, herdar de) zero, uma, ou muitas interfaces.
  - Uma interface não define quaisquer comportamento ou atributos.
- Uma classe define atributos e métodos.
  - Uma classe pode *estender* (i.e., herdar comportamento e atributos de) zero ou mais classes.
  - Uma classe pode *implementar* (i.e., estar de acordo com) zero ou mais interfaces, além de poder estender sua super classe.

Mecanismos de herança em Java

- **Extensão simples**
  - palavra-chave extends; para herança estrita, marcar todos os métodos da superclasse como final.
- **Especificação**
  - uma especificação é implementada como uma interface; subclasses da especificação usam a palavra-chave implements para indicar este tipo de herança.
  - separa interface e implementação
  - implementações podem ser transparentemente substituídas
  - Diminui o acoplamento

Mecanismos de herança em Java

- **Herança polimórfica**
  - como a extensão simples, usa a palavra-chave extends. Usando na superclasse as palavras-chave final e abstract, é possível indicar que partes da superclasse não podem ser modificadas ou devem ser modificadas.

Princípio de Substituição de Liskov

- Liskov Substitution Principle (LSP), por Barbara Liskov, em 1993.
- Princípio: *Todas as classes derivadas de uma classe devem ser trocáveis quando usadas como a classe base*
- Exemplo:
  - Seja A uma classe e B uma de suas subclasses. Seja ainda o método **m(A a) { … }**
  - Se **m** se comporta corretamente quando o parâmetro é uma instância de **A**, ele deve se comportar corretamente quando o parâmetro é uma instância de **B**
  - Isso sem que **m** precise saber que existe a classe **B**

LSP – exemplo clássico

- Seja a classe abaixo.

```
class Rectangle {
    protected double h,w;
    protected Point top_left;
    public double setHeight (double x) { h=x; }
    public double setWidth (double x) { w=x; }
    public double getHeight () { return h; }
    public double getWidth () { return w; }
    public double area() { return h*w; }
     …
}
```

- Suponha que tenhamos várias aplicações clientes da classe Rectangle…

LSP – exemplo clássico (cont.)

- Adicionando um quadrado…
- Um quadrado é um tipo de retângulo, certo?
- Então: **class Square extends Rectangle { … }**
- A princípio, não precisamos modificar o código cliente pré-existente.
  - e.g., **void m(Rectangle x) { … }** não precisa de modificações quando da adição dessa nova classe Square.
- Mas, há problemas…

LSP – exemplo clássico (cont.)

- Problema: com a solução anterior, podemos “degenerar” um quadrado!
- Uma segunda solução: redefinir os métodos setHeight e setWidth na classe Square:

```
class Square extends Rectangle {
    public void setHeight(double x) {
      h=x; w=x;
    }
    public void setWidth(double x) {
      h=x; w=x;
    }
}
```


LSP – exemplo clássico (cont.)

- Considere agora o trecho de código (cliente) a seguir:

```
void m(Rectangle r) {
    r.setHeight(5);
    r.setWidth(4);
    assert (r.area() == 20);
}
```

- Quando temos apenas objetos retângulo, o código acima é válido; no entanto, este código não é válido quando, além de retângulos, temos também quadrados.
- Não há nada de errado com **m**
- O que está errado em **Square?**

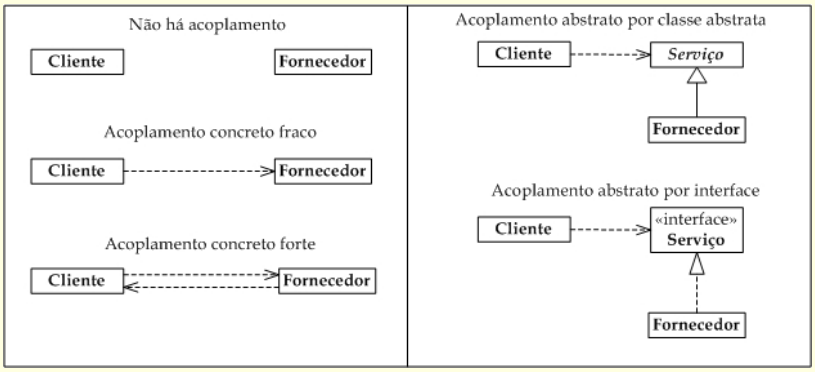
O papel do LSP

- LSP é um princípio bastante restritivo. Em geral, os desenvolvedores apóiam LSP e o têm como uma meta.
- Deve ser usado como um sinalizador
  - É possível e aceitável que se viole esse princípio, mas a violação deve ser examinada cuidadosamente.
- Depende do cliente da hierarquia de classes
  - E.g., se temos um programa no qual altura e comprimento nunca são modificados, é aceitável ter um Square como uma subclasse de Rectangle.
- "Square subclass of Rectangle" e "Eclipse subclass of Circle" têm sido fontes de guerras religiosas na comunidade OO por anos (vide http://ootips.org)

Acoplamentos concreto e abstrato

- Usualmente, um objeto A faz referência a outro B através do conhecimento da classe de B.
  - Esse tipo de dependência corresponde ao que chamamos de ***acoplamento concreto***.
- Entretanto, há outra forma de dependência que permite que um objeto remetente envie uma mensagem para um receptor sem ter conhecimento da verdadeira classe desse último.
  - Essa forma de dependência corresponde ao que chamamos de ***acoplamento abstrato***.
  - A acoplamento abstrato é preferível ao acoplamento concreto.
- Classes abstratas e interface permitem implementar o acoplamento abstrato.

Acoplamentos concreto e abstrato (cont)



Reuso através de generalização

- No reuso por generalização/especialização subclasses herdam comportamento da superclasse.
  - Exemplo: um objeto ContaCorrente não tem como atender à mensagem para executar a operação debitar só com os recursos de sua classe. Ele, então, utiliza a operação herdada da superclasse.
- Vantagem:
  - Fácil de implementar.
- Desvantagem:
  - Exposição dos detalhes da superclasse às subclasses (Violação do *princípio do encapsulamento*).
  - Possível violação *do Princípio de Liskov (regra da substituição)*.

Reuso através de delegação

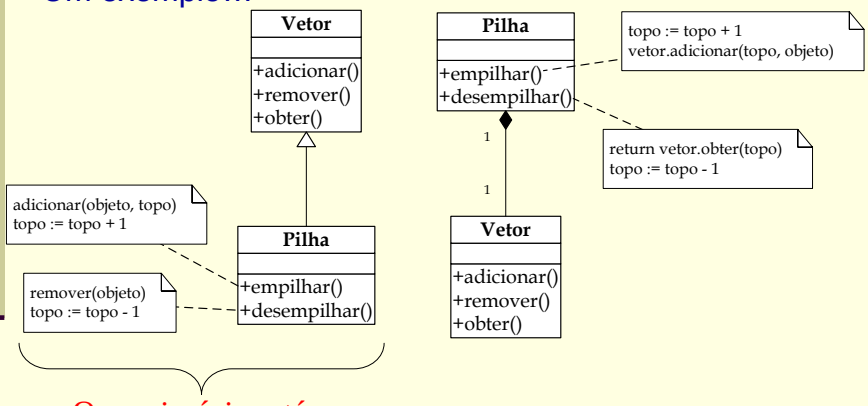
- A delegação é outra forma de realizar o reuso.
- "Sempre que um objeto não pode realizar uma operação por si próprio, ele delega uma parte dela para outro(s) objeto(s)".
  - Ou seja, o objeto reusa as operações dos objetos para os quais ele delega responsabilidades.
  - A delegação implica na composição de objetos.
- A delegação é mais genérica que a generalização.
  - um objeto pode reutilizar o comportamento de outro sem que o primeiro precise ser uma subclasse do segundo.

Reuso através de delegação

- Uma outra vantagem da delegação sobre a herança de classes é que o compartilhamento de comportamento e o reuso podem ser realizados em tempo de execução.
  - Na herança de classes, o reuso é especificado estaticamente e não pode ser modificado.
- Desvantagens:
  - desempenho (implica em cruzar a fronteira de um objeto a outro para enviar uma mensagem).
  - não pode ser utilizada quando uma classe parcialmente abstrata está envolvida.

Delegação x Herança de classes

- Um exemplo…



Que princípio está sendo violado aqui?

Generalização versus delegação

- Há vantagens e desvantagens tanto na generalização quanto na delegação.
- De forma geral, não é recomendado utilizar generalização nas seguintes situações:
  - Para representar papéis de uma superclasse.
  - Quando a subclasse herda propriedades que não se aplicam a ela.
  - Quando um objeto de uma subclasse pode se transformar em um objeto de outra subclasse.
    - Por exemplo, um objeto Cliente se transforma em um objeto Funcionário.