Módulo I
Padrões de Projeto GoF

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

Ementa

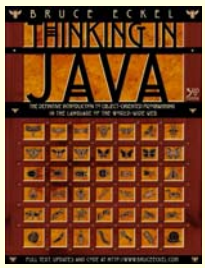
- Padrões de Projeto GoF
  - Introdução
  - Singleton
  - Iterator
  - Factory Method
  - Abstract Factory
  - Command
  - Template Method
  - Adapter
  - Composite
  - Observer
  - MVC

Bibliografia

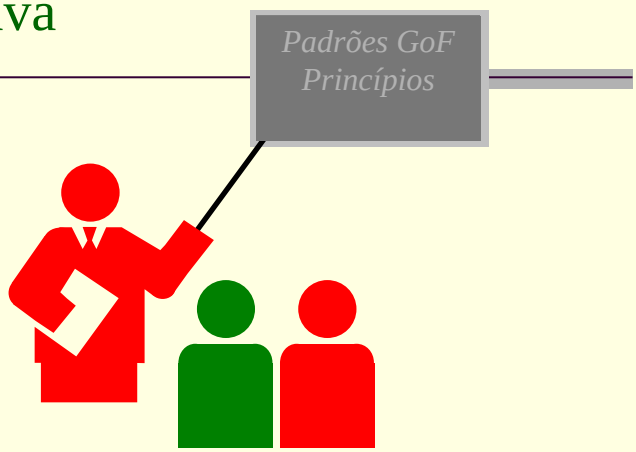
- *Craig Larman, Utilizando UML e Padrões, Ed Bookman*
- *Eric Gamma, et ali, Padrões de Projeto, Ed Bookman*
- *Martin Fowler, Analysis Patterns - Reusable Object Models, Addison-Wesley,1997*
- *Martin Fowler, Refatoração - Aperfeiçoando o projeto de código existente, Ed Bookman*

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

- Em 1995, Erich Gamma, JohnVlissides, RalphJonhsone RichardHelm 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).
- Não são invenções. São documentação de soluções obtidas através da experiência. Foram coletados de experiências de sucesso na indústria de software, principalmente de projetos em C++ e SmallTalk

Categorias de padrões GoF

- ***Padrões de projeto*** estão relacionados a questões de comportamento de objetos, ciclo de vida de objetos, a interface dos objetos e a relacionamentos estruturais entre os objetos.
- ***Padrões de Projeto*** permitem desenvolver software de melhor qualidade uma vez que utilizam eficientemente polimorfismo, herança, modularidade, composição, abstração para construir código reutilizável, eficiente, de alta coesão e baixo acoplamento.
- Ajuda na documentação e na aprendizagem. O conhecimento dos padrões de projeto torna mais fácil a compreensão de sistemas existentes.

Categorias de padrões GoF

- Os padrões catalogados pela equipe GoF possuem diversos nomes alternativos: Padrões de projeto, Padrões GoF ou Design patterns
- 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

		Propósito		
		1. Criação	2. Estrutura	3. Comportamento
Escopo	Classe	Factory Method	Class Adapter	Interpreter Template Method
	Objeto	Abstract Factory Builder Prototype Singleton	Object Adapter Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

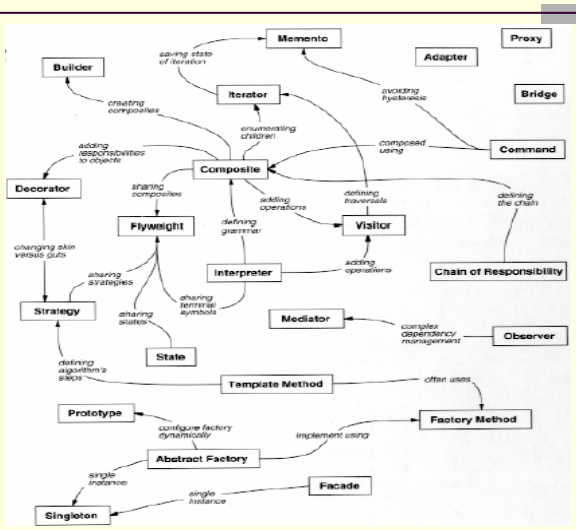
Categorias de padrões GoF - Metsker

- Um padrão "GoF" também é classificado segundo o seu escopo; de classe ou de objeto. Nos padrões com escopo de classe os relacionamentos que definem este padrão são definidos através de herança e em tempo de compilação. Nos padrões com escopo de objeto o padrão é encontrado no relacionamento entre os objetos definidos em tempo de execução.
- Metsker classifica os padões GoF em 5 grupos, por intenção (problema a ser solucionado):
  - (1) oferecer uma interface,
  - (2) atribuir uma responsabilidade,
  - (3) realizar a construçãode classes ou objetos
  - (4) controlar formas de operação
  - (5) implementar uma extensãopara a aplicaçãoPadrões

Categorias de padrões GoF - Metsker

Intenção	Padrões
1. Interfaces	Adapter, Facade, Composite, Bridge
2. Responsabilidade	Singleton, Observer, Mediator, Proxy, Chain of Responsibility, Flyweight
3. Construção	Builder, Factory Method, Abstract Factory, Prototype, Memento
4. Operações	Template Method, State, Strategy, Command, Interpreter
5. Extensões	Decorator, Iterator, Visitor

Relacionamento entre os Padrões



Outros Padrões

- Há vários catálogos de padrões em software
  - Muitos são específicos a uma determinada área (padrões J2EE, padrões de implementação em Java, em C#, padrões para concorrência, sistemas distribuídos, etc.)
  - Os padrões apresentados aqui são aplicáveis em Java e outras linguagens
- Dois outros padrões são muito populares atualmente
  - Dependency Injection: um caso particular de um dos padrões GRASP (Indirection) bastante popular no momento (também conhecido como Inversão de Controle)
  - Aspectos: uma extensão ao paradigma orientado a objetos que ajuda a lidar com limitações dos sistemas OO

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!, ie, podemos criar quadrados com lados diferentes !?@
- 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?**

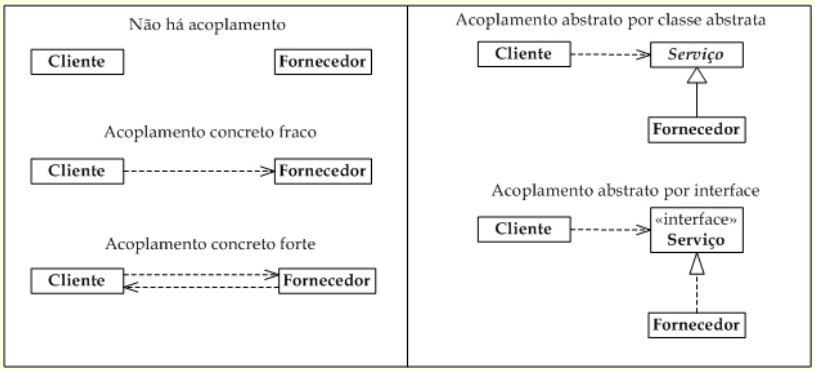
O papel do LSP

- 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

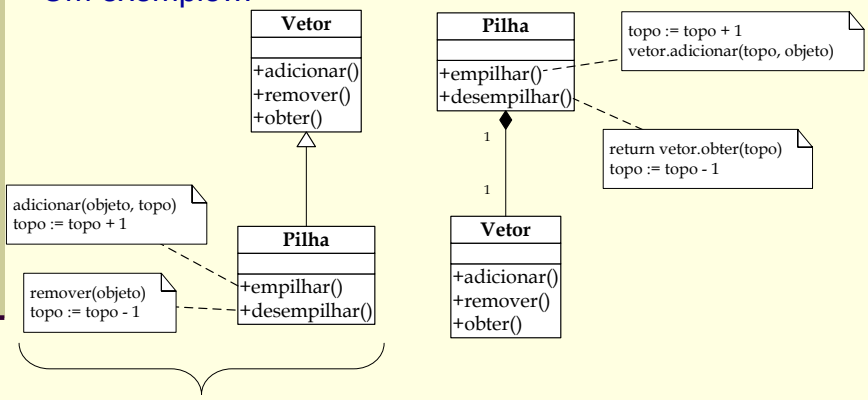
- A delegação é outra forma de realizar o reuso.
- “Sempre que um objeto não pode realizar uma operação por si próprio, ele delega uma parte dela para outro(s) objeto(s)”.
  - Ou seja, o objeto reusa as operações dos objetos para os quais ele delega responsabilidades.
  - A delegação implica na composição 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.

Padrões para Adaptação Interfaces

- Padrões de design que concentram-se principalmente na adaptação de interfaces são:
  - Adapter: para adaptar a interface de uma classe para outra que o cliente espera
  - Façade: oferecer uma interface simples para uma coleção de classes
  - Composite: definir uma interface comum para objetos individuais e composições de objetos
  - Bridge: desacoplar uma abstração de sua implementação para que ambos possam variar independentemente

POO-Java

Padrões GoF
Singleton

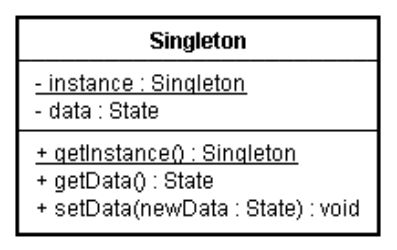
Singleton

- **Motivação**: algumas classes devem ser instanciadas uma única vez:
  - Um spooler de impressão
  - Um sistema de arquivos
  - Um Window manager
  - Um objeto que contém a configuração do programa
  - Um ponto de acesso ao banco de dados
- Obstáculo: a definição de uma variável global deixa a instância (objeto) acessível mas não inibe a instanciação múltipla.
- Como assegurar que somente uma instância de uma classe seja criada para toda a aplicação?

Singleton

- **Intenção**: garantir que uma classe tem apenas uma instância, e prover um ponto de acesso global a ela;
- **Solução**: fazer com que a própria classe seja responsável pela manutenção da instância única, de tal forma que:
  - Quando a instância for requisitada pela primeira vez, essa instância deve ser criada;
  - Em requisições subseqüentes, a instância criada na primeira vez é retornada.
  - A classe **Singleton** deve:
    - armazenar a única instância existente;
    - garantir que apenas uma instância será criada;
    - prover acesso a tal instância.

Singleton (estrutura)



Singleton (implementação)

```
public final class Singleton {
  private static Singleton instance = null;

  private Singleton () {
   ...
  }

  public static Singleton getInstance() {
   if (instance == null) {
     instance = new Singleton ();
   }
   return instance;
  }
  ...
}
```

Singleton (uso)

```
public class UsoDoSingleton {
     :
  Singleton obj;
     :
  obj = Singleton.getInstance();
     :
}
```


POO-Java

Padrões GoF
Iterator

Iterator

- Toda coleção possui uma representação interna para o armazenamento e organização de seus elementos.
  - Por outro lado, essa coleção deve permitir que seus elementos sejam acessados sem que sua estrutura interna seja exposta.
- De uma maneira geral, pode-se desejar que estes elementos sejam percorridos de várias maneira, sem no entanto ter que modificar a interface da coleção em função do tipo de varredura desejado.
  - de frente para trás, vice-versa, ou mesmo em ordem aleatória.

Iterator

- O padrão **Iterator** permite descrever uma forma de percorrer os elementos de uma coleção sem violar o encapsulamento dessa coleção.
- **Intenção**: iterar sobre (percorrer seqüencialmente) uma coleção de objetos sem expor sua representação.
  - Obedecer o princípio do encapsulamento

Iterator

- **Solução:** um objeto intermediário (**iterator**) é usado entre o cliente e a coleção de objetos.
  - Este objeto conhece a estrutura interna da coleção a ser percorrida, e apresenta uma interface para percorrer tal estrutura.
  - Esta interface é independente dessa estrutura interna.
  - Os clientes que desejam percorrer a coleção utilizam a interface do objeto intermediário, em vez de se comunicarem diretamente com a coleção de objetos.

Iterator

- Requisitos de um iterador
  - Um modo de localizar um elemento específico da coleção, tal como o primeiro elemento.
  - Um modo de obter acesso ao elemento atual.
  - Um modo de obter o próximo elemento.
  - Um modo de indicar que não há mais elementos a percorrer.
- Exemplo em Java
  - As classes List, Set e Sorted são subclasses de Collection, e herdam um método iterator() que retorna um objeto iterador.
  - O objeto Iterator possui métodos hasNext() e next().

Iterator (exemplo)

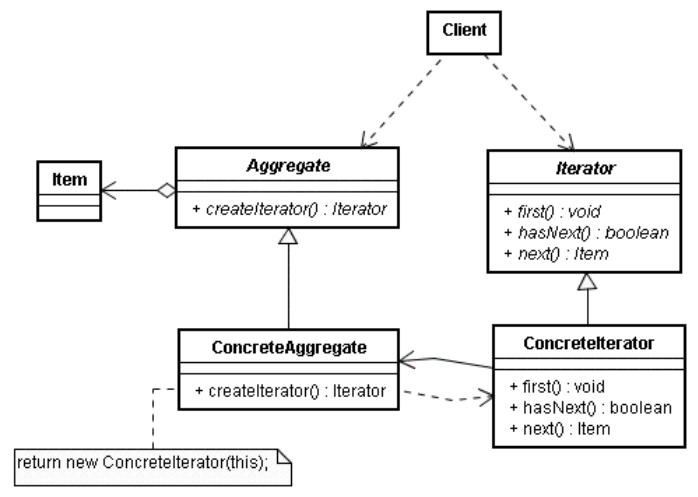
```
// ICollection.java
// interface para obtenção de Iterator para coleções

public interface ICollection
{
  // obtenção de um Iterator
  public IIterator getIterator();
  // determina existência de um elemento
  public boolean has(Object object);
  // adição de um elemento
  public boolean add(Object object);
  // remoção de um elemento
  public boolean remove(Object object);
  // remoção de todos os elementos
  public void removeAll();
}
```


Iterator (exemplo)

```
// IIterator.java
public interface IIterator {
  // verifica a existência de um próximo elemento
  public boolean hasNext();
  // retorna o próximo elemento
  public Object next();
}
```

Iterator (estrutura)



Iterator (participantes)

- *Iterator*
  - Define um interface para o acesso e varredura;
- *ConcreteIterator*
  - Implementa a interface do *Iterator*;
  - Mantém referência (cursor) ao objeto que está sendo percorrido, podendo calcular qual o elemento seguinte.
- *Aggregate*
  - Define um interface para a criação do objeto Iterator;
- *ConcreteAggregate*
  - Implementa o método da interface que retorna uma instância do ConcreteIterator.

Iterator (aplicabilidade)

- O uso do padrão Iterator se aplica quando se quer:
  - acessar o conteúdo de objeto agregados sem expor sua representação interna;
  - dar suporte a mais de uma maneira de percorrer a lista;
  - prover interface única para percorrer estruturas agregadas diferentes.

Iterator (conseqüências)

- Mantém separadas a representação interna e a responsabilidade de *navegação* pelas partes.
  - O iterador conhece a estrutura interna das partes, mas os clientes do iterador não conhecem.
- Move da coleção de objetos para o objeto iterator a responsabilidade de acesso e varredura da coleção.
- A coleção ainda é responsável por criar seus próprios iteradores e o faz através do padrão "*Factory Method*".
- Há a possibilidade de utilizar mais de um iterador simultaneamente.
  - Dá suporte a múltiplas maneiras de percorrer a coleção e, se necessário, essas varreduras podem ocorrer ao mesmo tempo.

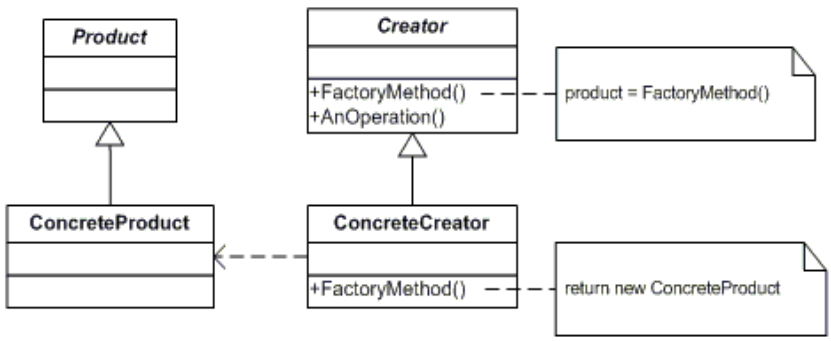
POO-Java

Padrões GoF
FactoryMethod

Factory Method

- **Intenção**: definir uma interface para criação de um objeto, permitindo que as suas subclasses decidam qual classe instanciar. O Factory Method deixa a responsabilidade de instanciação para as subclasses.

Factory Method (estrutura)



Factory Method (consequências)

- Provê ganchos para subclasses;
- Conecta hierarquia de classes paralelas quando há delegação.

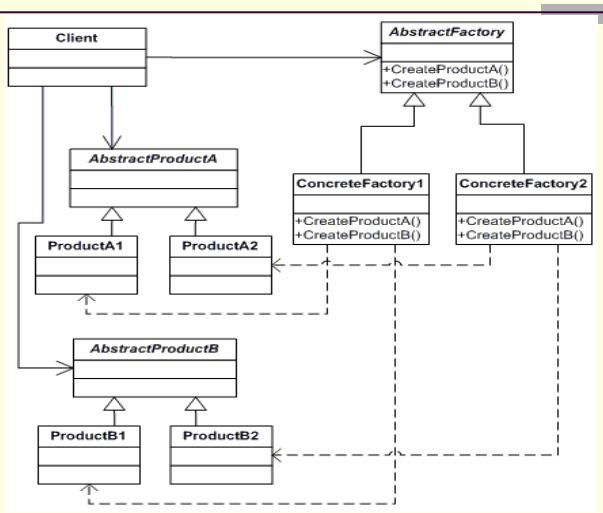
POO-Java

Padrões GoF Abstract Factory

Abstract Factory

- **Intenção**: fornecer uma interface comum para a criação de famílias de objetos relacionados ou dependentes sem especificar suas classes concretas.
- **Solução**: crie uma interface para representar uma fábrica para cada família de objetos. As subclasses concretas instanciam cada família específica.

Abstract Factory (estrutura)



Abstract Factory (participantes)

- **Fábrica Abstrata**: declara uma interface para operações criam objetos-produto abstratos;
- **Fábrica Concreta**: implementa as operações para criar objetos-produto concretos;
- **Produto Abstrato**: declara uma interface para um tipo de objeto produto.
- **Produto Concreto**: implementa a interface abstrata de Produto Abstrato e define um objeto-produto a ser criado pela Fábrica Concreta correspondente.
- **Cliente**: utiliza apenas as interfaces declaradas por Fábrica Abstrata e Produto Abstrato.

Abstract Factory (conseqüências)

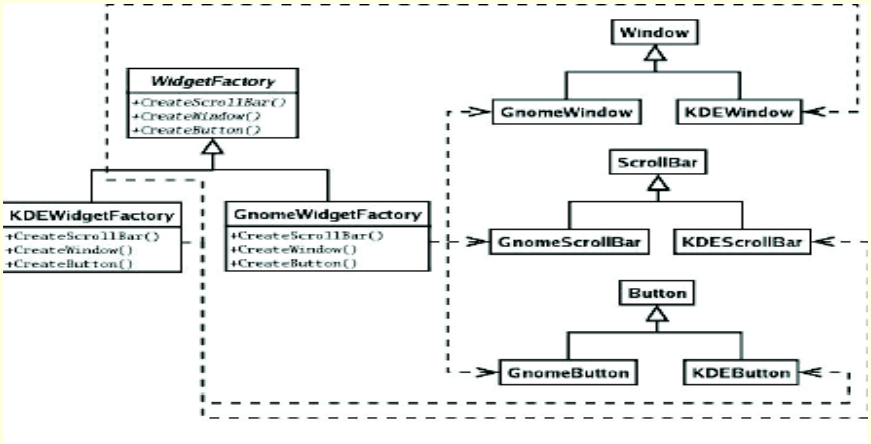
- Isola classes concretas: uma vez que uma fábrica encapsula a responsabilidade e o processo de criação de objetos-produto, ela isola clientes das classes de implementação.
- Fica mais fácil a troca de uma família de produtos, bastando trocar a fábrica concreta usada pela aplicação.
- Promove consistência entre produtos.
- O suporte a novos tipos de produtos é dificultado, já que a interface definida em AbstractFactory fixa o conjunto de produtos que podem ser criados.

Abstract Factory (aplicabilidade)

- Quando o sistema deve ser independente de como seus produtos são criados, compostos e representados.
- Quando o sistema deve ser configurado com uma dentre várias famílias de produtos.
  - É necessário fornecer uma biblioteca de classes, mas não é desejável revelar que produto particular está sendo usado.
- Quando uma família de produtos relacionados foi projetada para ser usada em conjunto, e esta restrição tem de ser garantida.

Abstract Factory (exemplo)

- Exemplo: portabilidade entre bibliotecas GUI (Gnome,KDE)



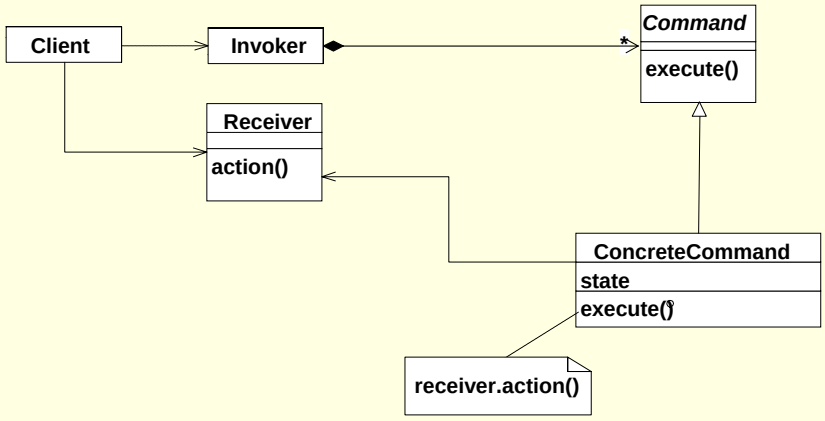
POO-Java

Padrões GoF
Command

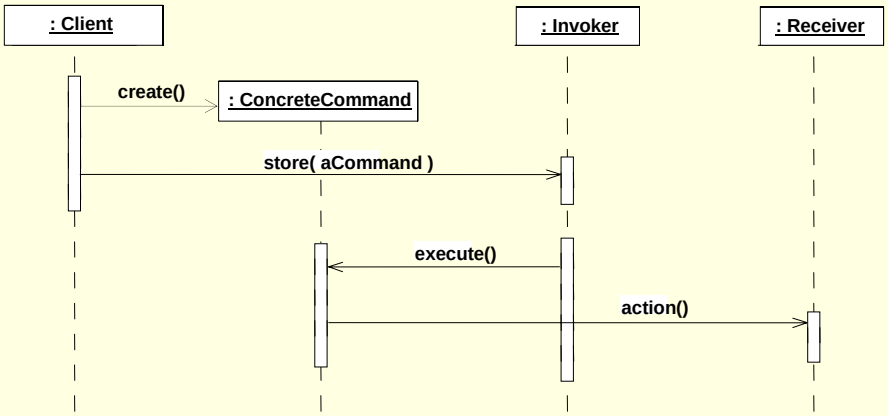
Command

- **Intenção**: associar uma ação a diferentes objetos através de uma interface conhecida. Permitir que objetos deste tipos tenham tais ações executadas sem que conheçamos o tipo de tais objetos ou a natureza das ações.
- **Solução**: encapsular uma requisição como um objeto, permitindo a parametrização de clientes com diferentes requisições.

Command (estrutura)



Command (exemplo de interação)



Command (conseqüências)

- Isola requisitante do executor;
- Permite registro (log) e/ou retrocesso (undo) de ações;
- Permite execução em instante posterior à requisição
  - i.e., permite enfileirar ações para processamento em outro momento.

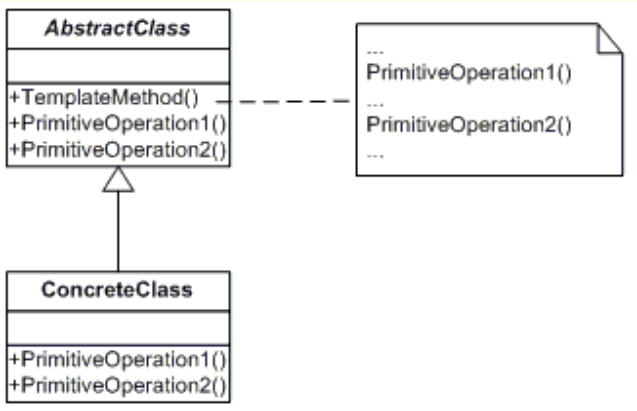
POO-Java

Padrões GoF
Template Method

Template Method

- **Intenção**: definir o esqueleto de um algoritmo em uma operação, postergando (delegando) a definição de alguns passos desse algoritmo para subclasses.
- **Solução**:
  - Em uma classe X, definir a parte invariável do algoritmo em uma operação. Essa operação é denominada método template (template method).
  - Nesta mesma operação, fazer chamadas a operações que representam a parte variável do algoritmo. Essas operações são denominadas operações gancho (hook operations).
  - Essas operações gancho devem então ser implementadas pelas subclasses de X.

Template Method (estrutura)



primitiveOperation1 e **primitiveOperation2** são "hook operations"

Template Method (aplicabilidade)

- Quando queremos implementar partes invariáveis de um algoritmo e deixar que as subclasses implementem os comportamentos variáveis;
- Quando comportamentos comuns entre subclasses devem ser fatorados e localizados em uma superclasse comum.
  - evitando assim duplicação de código;
- Quando queremos controlar a extensão das subclasses.
  - Pode-se definir um *template method* que chama *hook operations* em pontos específicos, permitindo desse modo extensões apenas nesses pontos.

Template Method (conseqüências)

- Permite que as subclasses redefinam certos passos de um algoritmo sem mudar a estrutura desse algoritmo.
  - i.e., Template Method usa herança para variar partes de um algoritmo.
- Um operação gancho não precisa necessariamente ser abstrata.
- Padrão importante no desenvolvimento de frameworks, pois fornece uma maneira de separar o comportamento variável do comportamento invariável em uma aplicação.
  - Permitem a implementação do Princípio de Hollywood (ou inversão de controle): “não nos ligue; nós ligaremos pra você”.
  - Inversão: uma superclasse chama operações de sua subclasse.

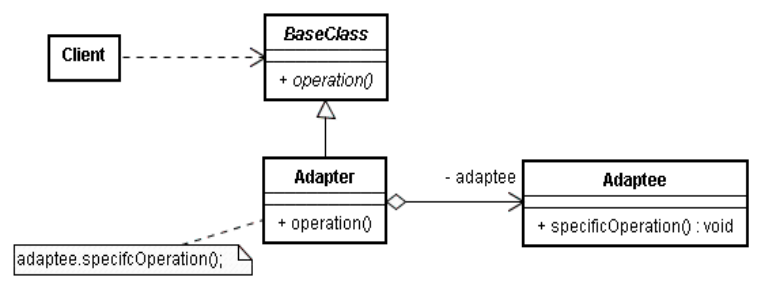
POO-Java

Padrões GoF
Adapter

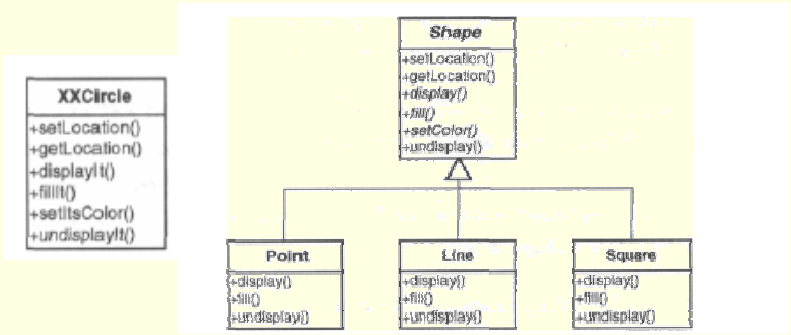
Adapter

- **Intenção**: adaptar um objeto preexistente para uma interface específica com a qual um outro objeto espera se comunicar.
- **Solução**: Definir uma classe que serve como um adaptador e que age como um intermediário entre o objeto e seus clientes (utilizar herança ou composição). O adaptador traduz comandos do cliente para o fornecedor e os resultados do fornecedor para o cliente.

Adapter (estrutura)



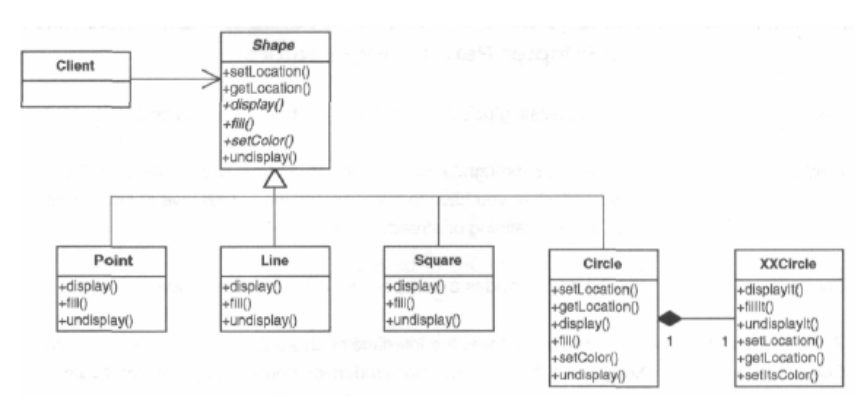
Adapter (exemplo)



Adapter (exemplo)

- Não posso usar XXCircle diretamente porque quero preservar o comportamento polimórfico em Shape.
  - Diferentes nomes e listas de parâmetros
  - XXCircle não deriva de Shape
- Solução: definir uma classe Circle que sirva como um adaptador para XXCircle.
  - Circle deriva de Shape
  - Circle contém XXCircle
  - Circle repassa mensagens enviadas para ele diretamente para XXCircle.

Adapter (exemplo)



Adapter (aplicabilidade)

- Quando se quer usar uma classe já existente e sua interface não combina com a esperada pelo cliente;
- Quando se quer criar uma classe reutilizável que coopera com classes não relacionadas ou não previstas, isto é, classes que não necessariamente tenham interfaces compatíveis;
- Quando se necessita usar várias classes existentes, mas é impraticável adaptar através da transformação de suas interfaces para transformá-las em subclasses de uma mesma classe.

Adapter (conseqüências)

- Adapta a classe Adaptee à BaseClasse pelo comprometimento com a classe concreta Adapter.
  - Como resultado, a classe Adapter não funcionará quando se quiser adaptar uma classe e todas as suas subclasses;
- Um único objeto Adapter trabalha com várias classes Adaptee
  - Quer dizer, a própria classe Adaptee e todas as suas subclasses (se houver).
  - O objeto Adapter pode adicionar funcionalidades a todas as classes Adaptee de uma só vez.

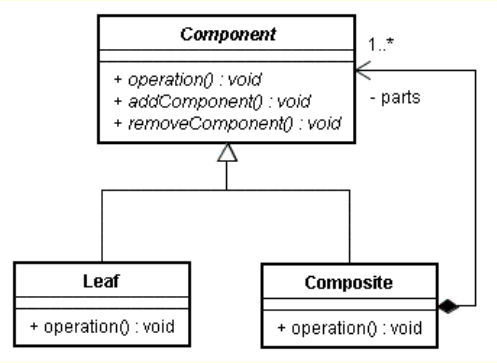
POO-Java

Padrões GoF
Composite

Composite

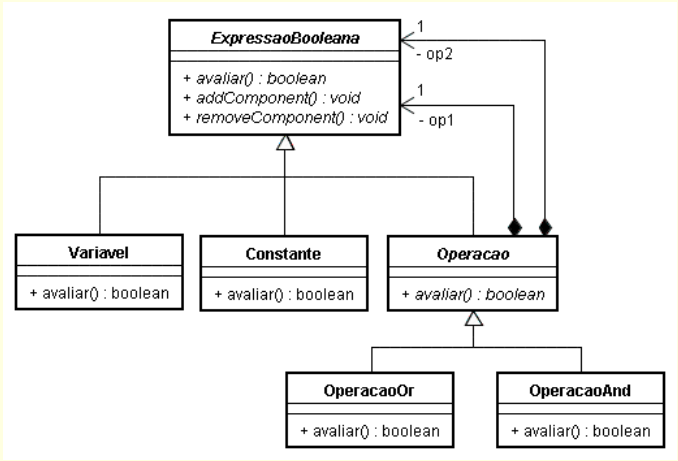
- São comuns as situações onde temos que lidar com uma coleção de elementos estruturada hierarquicamente (em vez coleções “lineares”).
- Problema: como criar objetos utilizando partes de tal forma que tanto o objeto todo quanto os objetos parte forneçam a mesma interface para os seus clientes?
- Composições podem cumprir com este requisito e ainda permitir:
  - o tratamento da composição como um todo;
  - ignorar as diferenças entre composições e elementos individuais;
  - a adição transparente de novos tipos a hierarquia;
  - a simplificação do cliente.

Composite (estrutura)



Composite (exemplo)

- Expressões lógicas (booleanas)



Composite (conseqüências)

- Objetos complexos podem ser compostos de objetos mais simples recursivamente.
  - Permite forma assim uma hierarquia de objetos
- O cliente pode tratar objetos "parte" e objetos "todo" da mesma forma.
  - Isso resulta na simplificação deste cliente.
  - Os clientes normalmente não sabem (e nem devem se preocupar) se eles estão tratando um componente individual ou composto.
- Facilita a adição de novos componentes: o cliente não tem que mudar com a adição de novos objetos
  - Sejam eles simples ou compostos

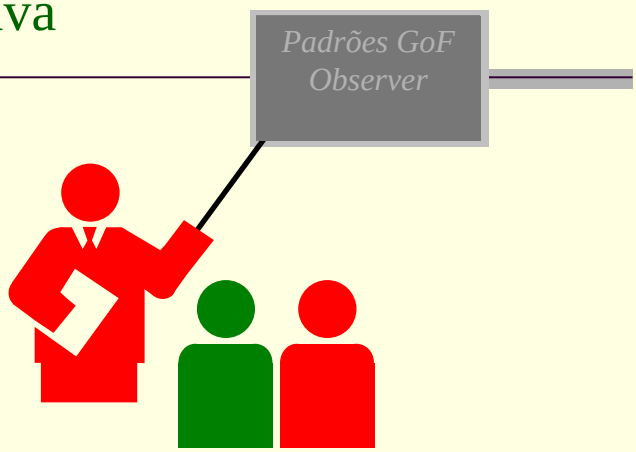
Composite (conseqüências)

- O projeto pode ficar geral demais, o que torna mais difícil restringir os possíveis componentes de um objeto composto.
  - Por exemplo, em uma hierarquia que contenha documentos e suas partes (seções, parágrafos, etc.), podemos compor seções com documentos, etc. o que não faz sentido

Composite (aplicabilidade)

- Quando é necessário representar hierarquias do tipo todo-parte.
- Quando é necessário tratar todo e respectivas partes de forma indistinta.

POO-Java

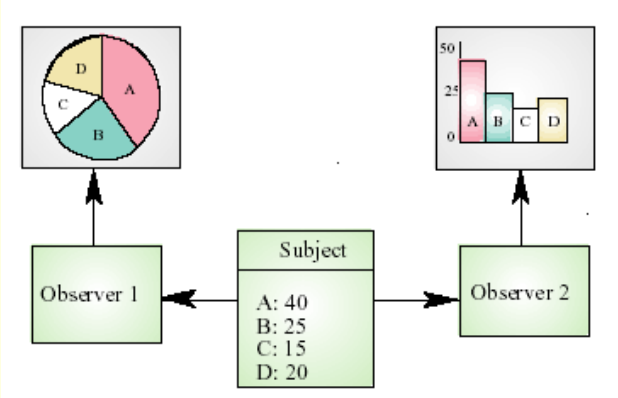


Observer

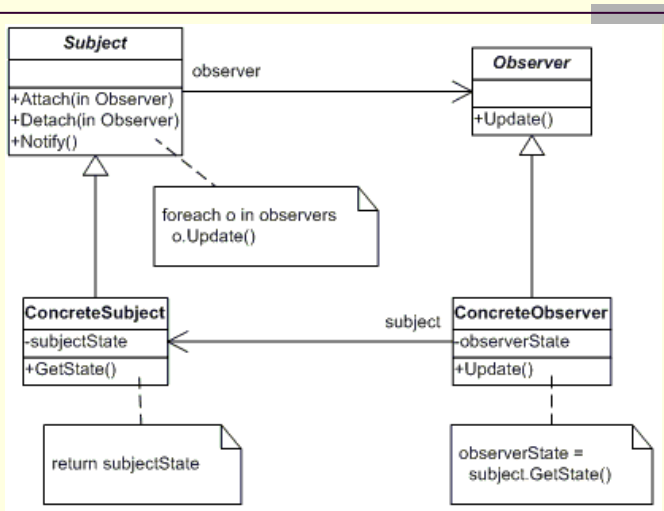
- Existem situações onde diversos objetos mudam seu estado de acordo com a mudança de estado de outro objeto.
  - e.g. as *views* e o *model* no framework MVC
- Define uma relação de dependência 1:N entre objetos, de tal forma que, quando um objeto (assunto) tem seu estado modificado, os seus objetos dependentes (observadores) são notificados.
  - Assunto → subject
  - Observadores (objetos dependentes) → observers

Observer (motivação)

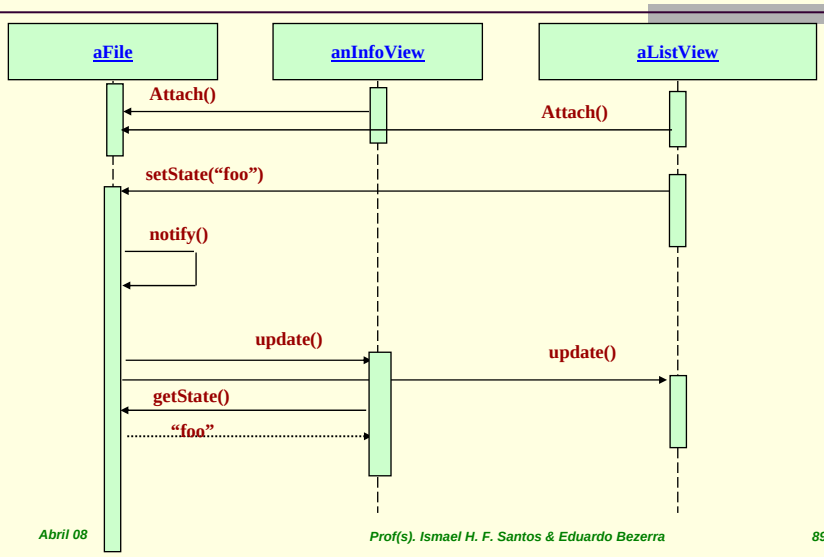
- Exemplo clássico



Observer (estrutura)



Observer (exemplo de interação)



Observer em Java

```
public class Observable extends Object {
   Collection<Observer> observers;
   public void addObserver(Observer o);
   public void deleteObserver(Observer o);
   public boolean hasChanged();
   public void notifyObservers();
   public void notifyObservers(Object arg);
   …
}

public abstract interface Observer {
   public abstract void update(Observable o, Object arg);
}

public class Subject extends Observable{
   public void setState(String filename);
  public string getState();
}
```

Observer (aplicabilidade)

- Quando uma abstração tem dois aspectos, um dependente do outro. Encapsulando-se esses aspectos em objetos separados fará com que se possa variá-los e reusá-los independentemente;
- Quando uma mudança em um objeto requer uma mudança em outros, e não se sabe como esses outros objetos efetivamente fazem suas mudanças;
- Quando um objeto deve poder notificar outros objetos sem assumir nada sobre eles. Dessa forma evita-se que os objetos envolvidos fiquem fortemente acoplados.

Observer (conseqüências)

- Possibilita baixo acoplamento entre os objetos dependentes (os observadores) e o assunto.
- Acoplamento Abstrato
- Suporte para *broadcast*
- Dificuldade em saber o que foi mudado?

POO-Java

Padrões GoF
Model View Controller

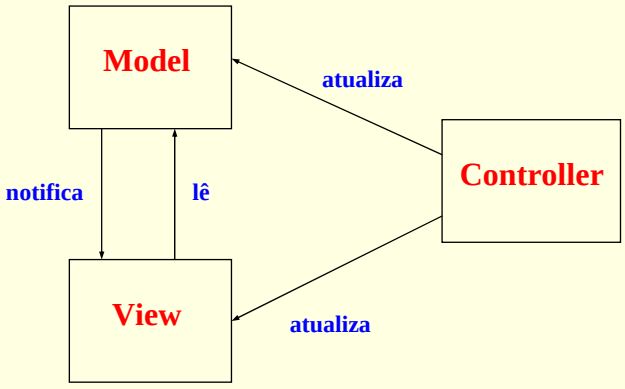
POO-Java

MVC em Swing

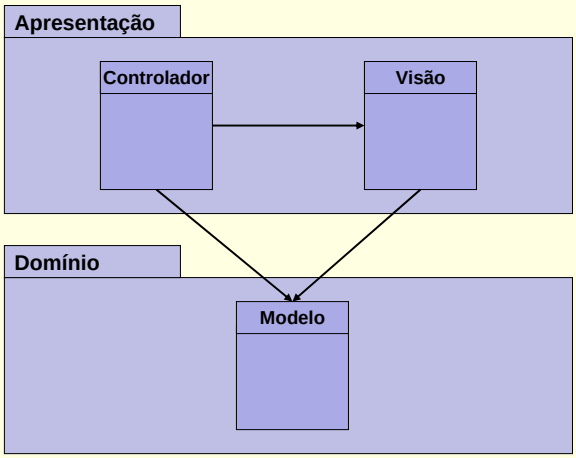
Arquitetura MVC

- O Swing adota uma arquitetura conhecida como *Model-View-Controller* (MVC)
  - Modelo = dados / conteúdo
    - estado de um botão, texto
  - Visão = aparência
    - cor, tamanho
  - Controle = comportamento
    - reação a eventos

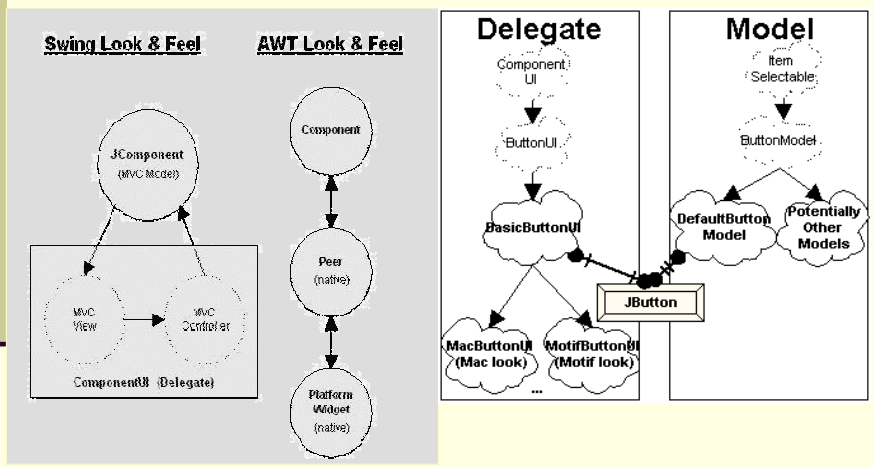
Interação entre os objetos MVC



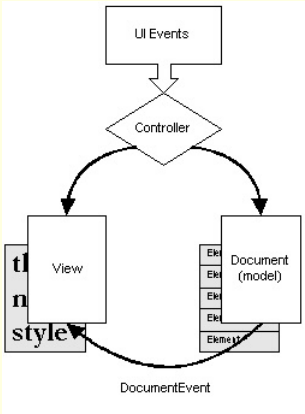
Arquitetura MVC no Swing



Modelo-Delegado



Documentos Swing



Explorando a Arquitetura MVC

- Como os dados (o modelo) não fazem parte integrante do elemento de interface que os exibe, podemos gerenciá-los em separado
- Por exemplo, é possível exibir um mesmo conjunto de dados em mais de um elemento de interface, simultaneamente
- Também é possível fazer com que o elemento de interface use os dados originais, sem copiá-los

Exemplo de Uso

- Suponha que você tem uma lista de nomes muito grande e deseja exibí-la em uma **JList**
- Usando a forma que vimos, esses nomes seriam copiados para dentro da lista
- Para evitar essa replicação, podemos utilizar um modelo próprio, que permitirá à **JList** acessar diretamente a lista de nomes

Interface **ListModel**

- Define o modelo usado pela classe **JList**
- Abrange dois aspectos:
  1. o acesso aos dados
  2. o controle da modificação dos dados
- Métodos de ListModel

```
int getSize()
Object getElementAt(int index)
void addListDataListener(ListDataListener l)
void removeListDataListener(ListDataListener l)
```

De Volta ao Exemplo

- Imagine que os nomes estão armazenados em um *array* de **String**
- Assumindo que a lista de nomes não é modificada, podemos ignorar o *listener*
- Basta, então, definir uma classe que implemente ListModel e utilize o *array* como fonte dos dados

Criando um Modelo

```
class ListaDeNomes implements ListModel {
  private String[] nomes;
  ListaDeNomes(String[] nomes) {
    this.nomes = nomes;
  }
  public int getSize() {
    return nomes.length;
  }
  public Object getElementAt(int index) {
    return nomes[index];
  }
  public void addListDataListener(ListDataListener l) {}
  public void removeListDataListener(ListDataListener l) {}
}
```

Usando o Modelo

```
JFrame f = new JFrame("Teste");
f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
String[] nomes = {"a", "b", "c", "d", "e", "f"};
JList l = new JList(new ListaDeNomes(nomes));
Container cp = f.getContentPane();
cp.add(new JScrollPane(l));
f.pack();
f.setVisible(true);
```

Exercícios – Questão 24 (again) - Exemplo com DefaultListModel !

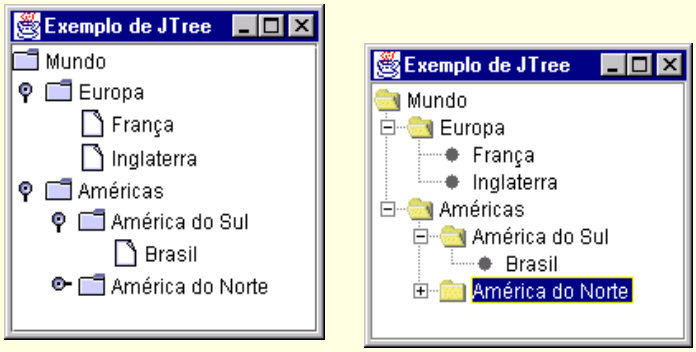
Classe **JTree**

- Componente que exibe uma estrutura de dados hierárquica (árvore)
- Segue o padrão MVC: os dados a serem exibidos são obtidos de um modelo (**TreeModel**)
  - o modelo a ser utilizado é fornecido no construtor do objeto **JTree**

Terminologia

- Uma árvore é composta de nós
  - um nó ou é uma folha ou possui nós filhos
  - todo nó, com exceção da raiz, tem exatamente um nó pai
  - toda árvore tem exatamente um nó raiz
- Tipicamente, o usuário pode ***expandir*** ou ***colapsar*** nós, tornando seus filhos, respectivamente, visíveis ou invisíveis

Exemplos



Interface **TreeModel**

- Define um modelo de dados adequado para um JTree
- Pertence ao pacote **javax.swing.tree**
- O Swing oferece uma implementação dessa interface: a classe **DefaultTreeModel**
  - modelo de árvore que utiliza TreeNodes

Métodos de **DefaultTreeModel**

```
DefaultTreeModel(TreeNode root)

Object getRoot()
int getChildCount(Object parent)
Object getChild(Object parent, int index)

void insertNodeInto(MutableTreeNode child,
                    MutableTreeNode parent,
                    int index)
void removeNodeFromParent(MutableTreeNode node)

void addTreeModelListener(TreeModelListener l)
```

Interface **MutableTreeNode**

- É uma subinterface de **TreeNode**
- Modela um nó que pode ser modificado
  - adição/remoção de filhos
  - modificação do conteúdo armazenado no nó (“user object”)
- O Swing oferece uma implementação dessa interface: a classe **DefaultMutableTreeNode**

Métodos de **DefaultMutableTreeNode**

```
DefaultMutableTreeNode(Object userObject)
DefaultMutableTreeNode(Object userObject,
                       boolean allowsChildren)
void add(MutableTreeNode child)
void remove(MutableTreeNode child)

Object getUserObject()
void setUserObject(Object userObject)
String toString()

boolean isLeaf()
Enumeration children()
```

Criando um JTree

```
DefaultMutableTreeNode mundo =
      new DefaultMutableTreeNode (“Mundo”);
DefaultMutableTreeNode europa =
      new DefaultMutableTreeNode (“Europa”);
DefaultMutableTreeNode americas =
      new DefaultMutableTreeNode (“Américas”);
mundo.add(europa);
mundo.add(americas);
…
JTree arvore = new JTree(new DefaultTreeModel(mundo));
```


Modos de Seleção

- O modo de seleção de um JTree é configurado (e gerenciado) por um “modelo de seleção” (**TreeSelectionModel**)
- Modos disponíveis:
  - SINGLE_TREE_SELECTION
  - CONTIGUOS_TREE_SELECTION
  - DISCONTIGUOUS_TREE_SELECTION

Configurando o modo de seleção

- Configurando modo de seleção

```
JTree arvore = new JTree(raiz);
int modo = TreeSelectionModel.SINGLE_TREE_SELECTION;
TreeSelectionModel tsm = arvore.getSelectionModel();
tsm.setSelectionMode(modo);
```

- Obtendo a seleção corrente

```
TreePath path = getSelectionPath()
if (path != null) {
 DefaultMutableTreeNode selNode =
(DefaultMutableTreeNode)path.getLastPathComponent();

String selValue = (String)selNode.getUserObject();
 …
}
```


Eventos de Seleção

- Eventos de seleção são gerados sempre que a seleção de uma árvore é alterada.
- Esses eventos podem ser tratados através da adição de um **TreeSelectionListener**.
- A interface **TreeSelectionListener** pertence ao pacote javax.swing.event e define apenas um método: `valueChanged`

Exercícios – Questão 27
Exercícios – Questão 28

Conclusões

- Componentes essenciais de uma aplicação Swing:
  - Contêiners são janelas ou painéis que contêm componentes.
  - Layouts especificam como arranjar componentes em um contêiner.
  - Componentes: são os controles da interface gráfica com o usuário.
  - Ouvintes (listeners) são conectados a componentes e contém o código que é executado quando o componente é usado.
    - É desse modo que uma ação do usuário sobre um componente é conectada a um método Java.

Conclusões

- Alguns IDEs têm eles próprios facilidades de construção da interface gráfica (editores de formulários)
  - e.g. NetBeans - www.netbeans.org
- Também há ferramentas específicas para a criação de GUIs em Java. Exemplos são:
  - XUI - http://xui.sourceforge.net/
  - UICompiler - http://uic.sourceforge.net/