Reuso de Software Aula 03

POO e Padrões de Projetos

Eduardo Figueiredo
http://www.dcc.ufmg.br/~figueiredo
reuso.software@gmail.com

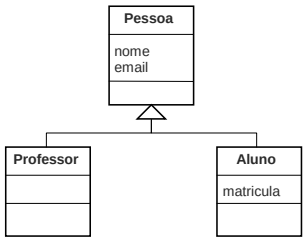
12 Março 2012

Tópicos da Aula

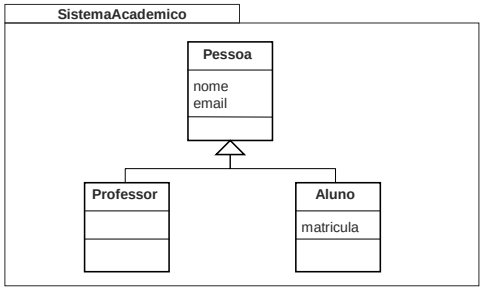
- Programação orientada a objetos
  - Reuso de classes
  - Bibliotecas
  - Frameworks
- Padrões de projeto
  - Padrões de Criação
  - Padrões estruturais
  - Comportamentais

Reuso de Classes

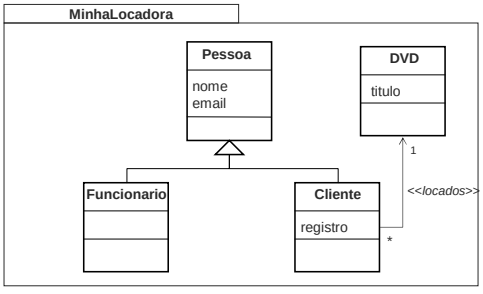
Considere três classes...



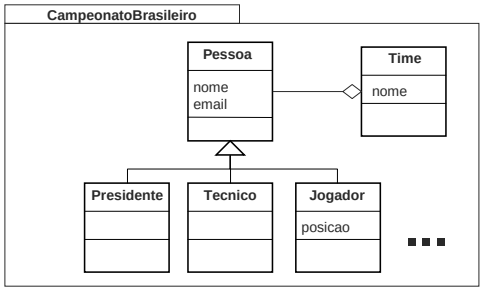
Sistema Acadêmico



Locadora de DVD



Campeonato de Futebol



Bibliotecas

Bibliotecas de Software

- Bibliotecas implementam serviços que podem ser usados por programas
  - É um forma bem comum de reuso
- Disponibiliza funcionalidades comuns a diferentes tipos de sistemas
  - Converter informação entre formatos conhecidos (e.g., string para inteiro)
  - Acesso a recursos, arquivos, BD, etc.
  - Tipos abstratos de dados: fila, pilha, lista...

Use de Biblioteca em Java

```
import java.util.Vector;

public class Customer {

  String name;
  Vector phoneNumbers = new Vector();

  void removePhoneNumber(String c){
    phoneNumbers.removeElement(c);
  }

  void addPhoneNumber(String c){
    phoneNumbers.addElement(c);
  }

  ...
}
```

Importar Classe da Biblioteca

```
import java.util.Vector;

public class Customer {

  String name;
  Vector phoneNumbers = new Vector();

  void removePhoneNumber(String c){
    phoneNumbers.removeElement(c);
  }

  void addPhoneNumber(String c){
    phoneNumbers.addElement(c);
  }

  ...
}
```

A classe Vector é incorporada ao sistema.

Instanciar um Objeto da Biblioteca

```
import java.util.Vector;

public class Customer {

  String name;
  Vector phoneNumbers = new Vector();

  void removePhoneNumber(String c){
    phoneNumbers.removeElement(c);
  }

  void addPhoneNumber(String c){
    phoneNumbers.addElement(c);
  }

  ...
}
```

Um objeto da classe Vector é criado da mesma forma que qualquer outro objeto do sistema.

Acessar Funções da Biblioteca

```
import java.util.Vector;

public class Customer {

  String name;
  Vector phoneNumbers = new Vector();

  void removePhoneNumber(String c){
    phoneNumbers.removeElement(c);
  }

  void addPhoneNumber(String c){
    phoneNumbers.addElement(c);
  }

  ...
}
```

As funcionalidades *removeElement* e *addElement* estão implementadas na biblioteca.

Produtividade e Confiabilidade

- Desenvolvedores não tem que "re-inventar a roda"
  - Alguns funcionalidade estão disponíveis
- Bibliotecas são bem testadas por muitos usuários
  - Mesmos situações pouco usuais e valores extremos já foram explorados

Principais Desvantagens

- Difícil fazer adaptações para atender detalhes específicos do sistema
  - Melhor performance
  - Mais robusto
- Tempo e custo para aprender ou encontrar a funcionalidade desejada

Frameworks

Motivação

- Objetos e funções (em bibliotecas) são geralmente pequenos e especializados demais
  - Torna-se necessário o reuso de abstrações maiores, como *frameworks*
- *Framework* é um conjunto de classes e interfaces que formam uma estrutura genérica

Framework

- *Frameworks* são aplicações incompletas
  - São formados por interfaces, classes abstratas e classes concretas
  - Permitem criar vários aplicações diferentes
- Detalhes específicos do sistema devem ser implementados
  - Pela adição de novas classes
  - Pela implementação das classes abstratas
  - Por arquivos de configuração, etc.

Características Principais

- Estensibilidade
  - Framewoks são extensíveis
- Inversão de controle
  - Diferente de bibliotecas, o fluxo de controle da aplicação é geralmente coordenada pelo framework
- Código não pode ser modificado
  - Diferente das aplicações, o código do *framework* não deve ser modificado

Extensão de *Frameworks*

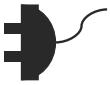
- *Frameworks* são entidades grandes que devem ser estendidas
- Exemplos de extensão
  - Adição de classes concretas que herdam operações de classes abstratas
  - Adição (sobrescrita) de métodos que respondem os eventos conhecidos do framework

Frozen Spots e *Hot Spots*

- Frameworks são consistuídos de *frozen spots* e *hot spots*
- *Frozen spots* definem a arquitetura principal do *framework*
  - Eles não são alterados em nenhuma extensão do *framework* (congelados)
- *Hot spots* definem os locais que programadores usam para extensão

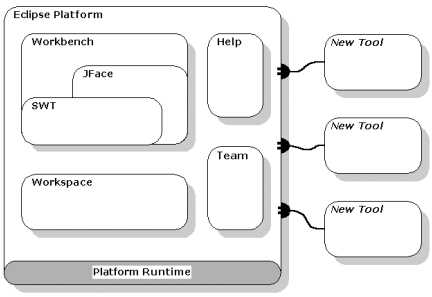
Representação (*Hot Spot*)

Ponto de extensão do framework



Extensão da aplicação

Exemplo de Extensão



Principal Problema

- *Framework* são geralmente entidades grandes e complexas
  - Leva um longo tempo para entendê-los e usá-los efetivamente
- Em grandes organizações, é comum ter profissionais especialistas em *frameworks* específicos

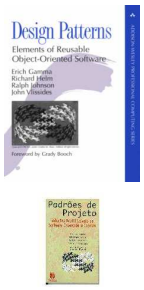
Padrões de Projeto

Padrões de Projeto

- Um padrão é uma descrição do problema e a essência da sua solução
- Documenta boas soluções para problemas recorrentes
  - Permite o reuso de conhecimento anterior documentados em boas práticas
- Deve ser suficientemente abstrato para ser reusado em aplicações diferentes

Os 23 Padrões de Projeto

- Os 23 padrões de projeto mais conhecidos foram popularizados pelo livro de E. Gamma, R. Helm, R. Johnson e J. Vlissides
  - Conhecido como Gang-of-Four (GoF)



Além dos 23 Padrões GoF

- Novos padrões de projeto surgem a todo momento
  - Padrões para áreas específicas, como padrões de interface, padrões de persistência, padrões de arquitetura, etc.
- Existem vários livros sobre padrões de projeto
  - A maioria deles se concentra nos 23 padrões GoF

Classificação dos Padrões

- Segundo o seu propósito
  - De Criação: tratam da criação de objetos
  - Estruturais: tratam da composição de classes e objetos
  - Comportamentais: tratam das interações e responsabilidades entre classes e objetos
- Segundo seu escopo
  - Classes: lidam com os relacionamentos entre classes e subclasses
  - Objetos: lidam com os relacionamentos entre objetos

Documentação de um Padrão

- Nome e classificação
  - Outros nomes
- Motivação
- Aplicações
- Estrutura
- Participantes
- Colaboração
- Consequências
- Implementação
- Código de exemplo
- Usos conhecidos
- Padrões relacionados

Elementos Principais

- Nome
  - Um identificador significativo para o padrão
- Descrição do problema
- Descrição da solução
  - Um *template* de solução que pode ser instanciado em maneiras diferentes
- Consequências
  - Os resultados e compromissos de aplicação do padrão

Padrões de Criação

Padrões de Criação

- Abstract Factory
- Builder
- **Factory Method**
- Prototype
- Singleton

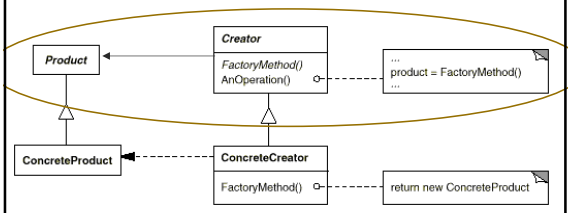
Problema (Factory Method)

- Uma classe não sabe antecipar o tipo dos objetos que a mesma precisa criar
  - É preciso adiar a instanciação de objetos para as subclasses
- Exemplo: suponha uma aplicação lida com diversos tipos de documentos
  - Ela sabe quando os documentos devem ser criados, mas não sabem que documentos criar

Padrão Factory Method

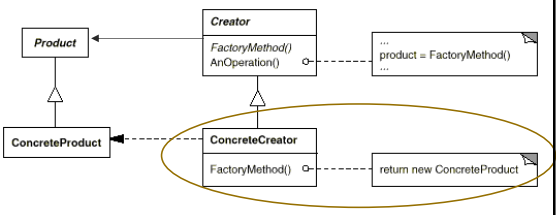
- Nome
  - Factory Method
- Descrição do problema
  - *Slide anterior*
- Descrição da solução
  - *Próximo slide*
- Consequências
  - Eliminam dependências de classes específicas (código lida com as interfaces)

Solução do Factory Method



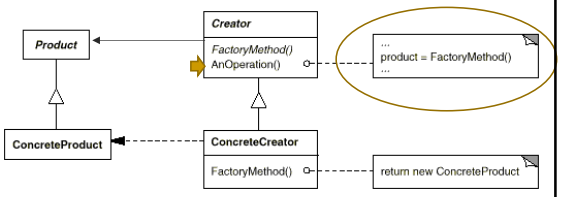
Super-classes não conhecem o produto específico.

Solução do Factory Method



O método fábrica cria e retorna o objeto no momento adequado.

Solução do Factory Method



O produto geral pode seu usado pela superclasse, mesmo sem conhecer o produto específico.

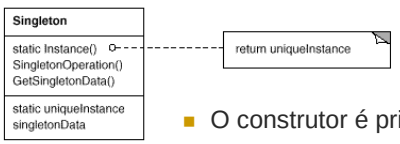
Padrões de Criação

- Abstract Factory
- Builder
- Factory Method
- Prototype
- **Singleton**

Padrão Singleton

- Nome
  - Singleton
- Descrição do problema
  - Uma classe precisa ter uma única instância
- Descrição da solução
  - *Próximo slide*
- Consequências
  - Fácil acesso e gerência de recursos compartilhados, como variáveis globais

Solução do Singleton



- O construtor é privado
- O método *instance()* é público e estático
  - Retorna a única instância que é guardada em uma variável de classe

Exemplo: ClasseSingleton

```
public class ClasseSingleton {

  private static ClasseSingleton instance;
  private int numInstances = 0;

  private ClasseSingleton() {
    numInstances++;
  }

  public static ClasseSingleton getInstance() {
    if (instance == null) instance = new ClasseSingleton();
    return instance;
  }

  public void printNumInstances() {
    System.out.println("numInstances = "+numInstances);
  }
}
```

Atributo que armazena a única instancia da classe.

Exemplo: ClasseSingleton

```
public class ClasseSingleton {

   private static ClasseSingleton instance;
   private int numInstances = 0;

   private ClasseSingleton() {
      numInstances++;
   }

   public static ClasseSingleton getInstance() {
      if (instance == null) instance = new ClasseSingleton();
      return instance;
   }

   public void printNumInstances() {
      System.out.println("numInstances = "+numInstances);
   }
}
```

Construtor da classe é privado para evitar criação acidental de outras instancias.

Exemplo: ClasseSingleton

```
public class ClasseSingleton {

   private static ClasseSingleton instance;
   private int numInstances = 0;

   private ClasseSingleton() {
      numInstances++;
   }

   public static ClasseSingleton getInstance() {
      if (instance == null) instance = new ClasseSingleton();
      return instance;
   }

   public void printNumInstances() {
      System.out.println("numInstances = "+numInstances);
   }
}
```

Única instancia só pode ser obtida pela chamada ao método *getInstance()* da classe.

Exemplo: ClasseSingletonTest


```
public class ClasseSingletonTest {

   public static void main(String[] args) {
      ClasseSingleton singleton = ClasseSingleton.getInstance();
//    ClasseSingleton singleton = new ClasseSingleton();
      singleton.printNumInstances();
   }
}
```

Tentar criar uma instância pela chamada ao construtor causa erro de compilação.

Padrões Estruturais

Padrões Estruturais

- **Adapter**
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

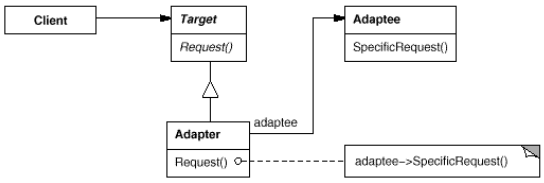
Problema (Padrão Adapter)

- Uma biblioteca é projetada para ser reusada
- Entretanto, a interface da biblioteca (assinatura dos métodos) pode não ser exatamente a esperada pela aplicação
  - Não é desejável alterar o código da aplicação
  - Não é desejável alterar a interface da biblioteca

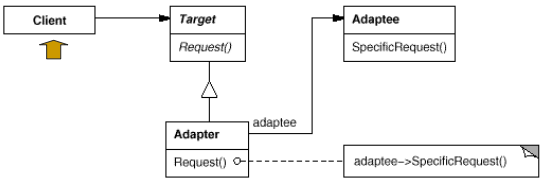
Padrão Adapter

- Nome
  - Adapter
- Descrição do problema
  - Permitir que classes com interfaces incompatíveis trabalhem juntos
- Descrição da solução (próximo slide)
- Consequências
  - Reuso de funcionalidades de uma classe sem alterar sua interface

Solução do Adapter

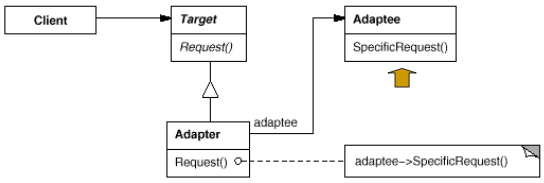


Solução do Adapter



O cliente (*Client*) precisa de uma funcionalidade.

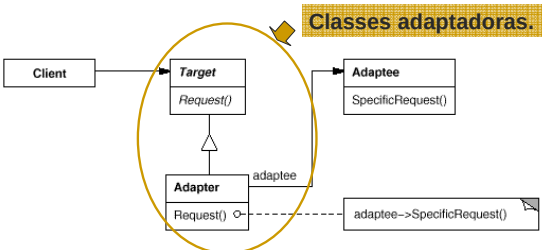
Solução do Adapter



A funcionalidade já está implementada no *Adaptee*. Entretanto, os métodos tem assinaturas diferentes.

Solução do Adapter

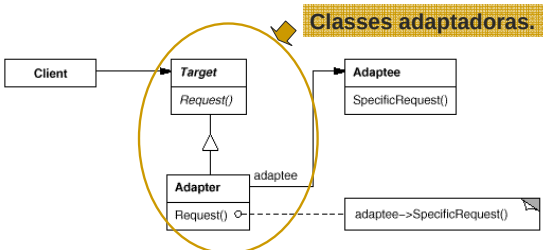
Classes adaptadoras.



A classe *Target* tem a assinatura do método que o cliente precisa.

Solução do Adapter

Classes adaptadoras.



A classe *Adapter* converte a assinatura do método em Adaptee para aquela que o cliente precisa.

Padrões Estruturais

- Adapter
- Bridge
- **Composite**
- Decorator
- Facade
- Flyweight
- Proxy

Problema (Padrão Composite)

- Algumas aplicações permitem construir entidades complexas a partir de elementos mais simples
- Exemplo: Editor de Figuras

Figuras Primitivas

Linha

Círculo

Problema (Padrão Composite)

- Algumas aplicações permitem construir entidades complexas a partir de elementos mais simples
- Exemplo: Editor de Figuras

Figuras Compostas

Quadrado

Triângulo

Problema (Padrão Composite)

- Algumas aplicações permitem construir entidades complexas a partir de elementos mais simples
- Exemplo: Editor de Figuras

Figuras Compostas

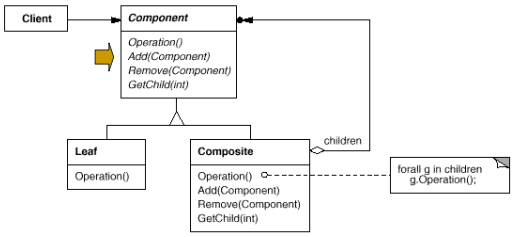
Casa

Rosto

Padrão Composite

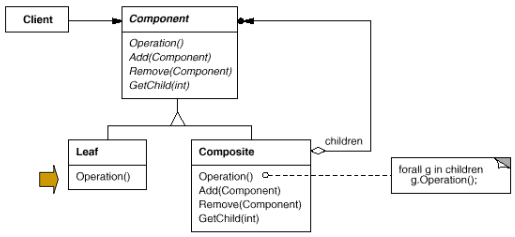
- Nome
  - Composite
- Descrição do problema
  - Compor objetos em estrutura hierárquica (todo-parte)
- Descrição da solução (próximo slide)
- Consequências
  - Objetos primitivos e compostos são tratados de maneira uniforme
  - Fácil incluir novos objetos

Solução do Composite



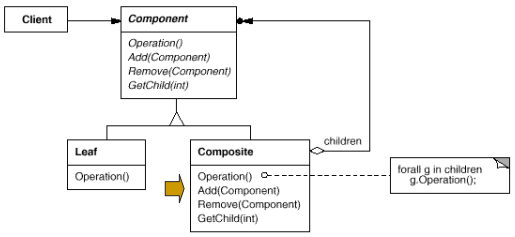
***Component* define as operações genéricas.**

Solução do Composite



***Leaf* define operações de elementos primitivos.**

Solução do Composite



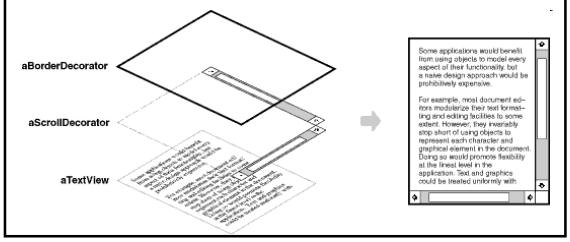
***Composite* representa elementos compostos.**

Padrões Estruturais

- Adapter
- Bridge
- Composite
- **Decorator**
- Facade
- Flyweight
- Proxy

Problema (Padrão Decorator)

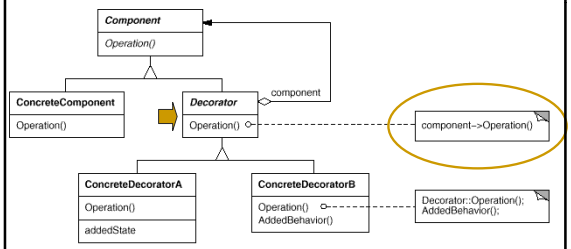
- Adicionar e remover responsabilidades dinamicamente a um objeto



Padrão Decorator

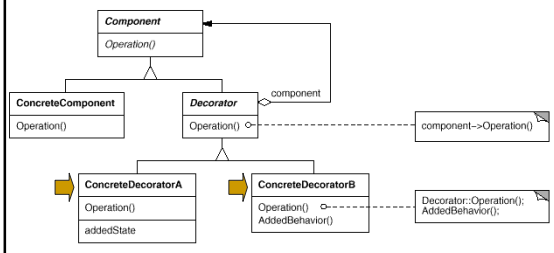
- Nome
  - Decorator
- Descrição do problema
  - Agregar dinamicamente novas responsabilidades a um objeto
- Descrição da solução (próximo slide)
- Consequências
  - Maior flexibilidade que herança
  - Evita sobrecarregar hierarquia de classes

Solução do Decorator



***Decorator* conhece o componente a ser decorado.**

Solução do Decorator



Podem haver vários decoradores concretos.

Padrões Comportamentais

Padrões Comportamentais

- Chain of Responsibility — **Vide Aula 02**
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- **Observer**
- State
- Strategy
- Template Method
- Visitor

Padrões Comportamentais

- Chain of Responsibility
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- **Strategy**
- Template Method
- Visitor

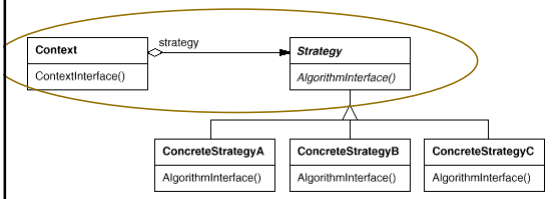
Problema (Strategy)

- Existem vários algoritmos para um mesmo problema
  - Exemplo: vários algoritmos de ordenação de array (BubbleSort, QuickSort, etc.)
- O objetivo é implementar e executar diferentes algoritmos usando uma mesma interface
  - Encapsular os diferentes algoritmos e torná-los intercambiáveis

Padrão Strategy

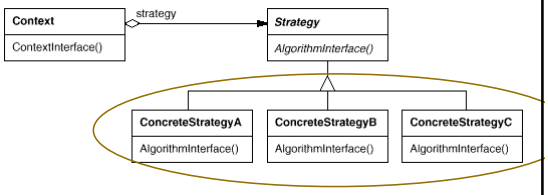
- Nome
  - Strategy
- Descrição do problema
  - Definir uma família de algoritmos
- Descrição da solução
  - próximo slide
- Consequências
  - Permite alternar entre diferentes algoritmos sem o uso de condicionais

Solução do Strategy



A aplicação usa o algoritmos sem conhecer sua real implementação.

Solução do Strategy



Diferentes implementações do algoritmo.

Referências

- Ian Sommerville. **Engenharia de Software**, 9ª Edição. Pearson Education, 2011.
  - Cap. 16 Reuso de Software
- E. Gamma, R. Helm, R. Johnson, J. Vlissides. **Padrões de Projeto**, 1a. Edição. Bookman, 2000.
  - Padrões: Factory Method, Singleton, Adapter, Composite, Decorator, Strategy