DCC / ICEx / UFMG

Padrões de Projeto

Eduardo Figueiredo

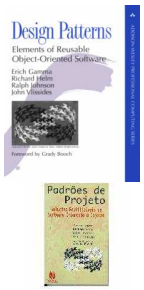
http://www.dcc.ufmg.br/~figueiredo

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

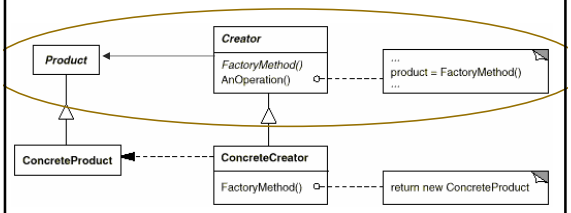
Problema (Factory Method)

- 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

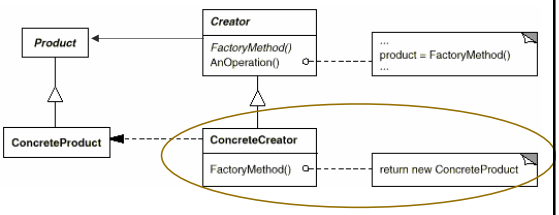
- Nome
  - Factory Method
- Descrição do problema
  - *Slide anterior*
- Descrição da solução
  - *Próximo slide*
- Consequências
  - Eliminam dependências de classes específicas (código lida com as interfaces)

Solução do Factory Method



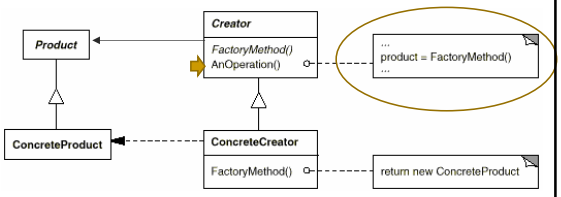
Super-classes não conhecem o produto específico.

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.

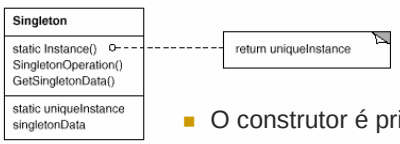
Padrões de Criação

- 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

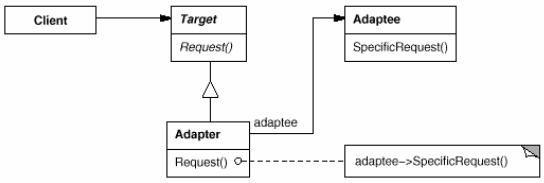
Problema (Padrão Adapter)

- 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

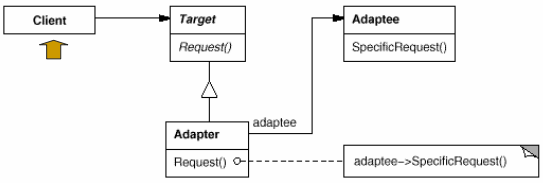
Padrão Adapter

- 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

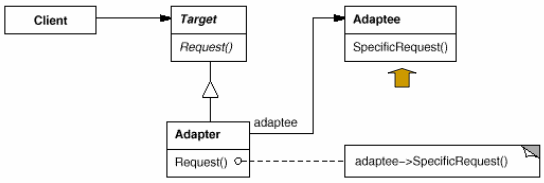


Solução do Adapter



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

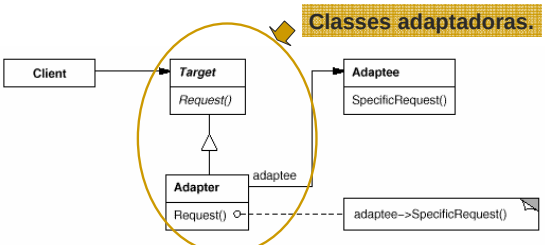
Solução do Adapter



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

Solução do Adapter

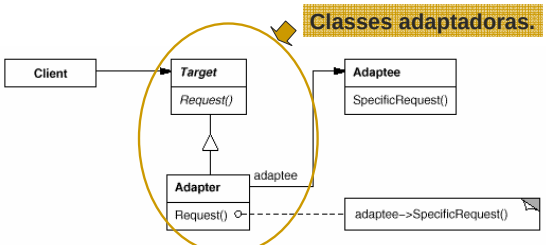
Classes adaptadoras.



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

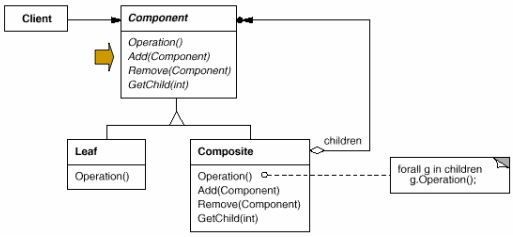
Casa

Rosto

Padrão Composite

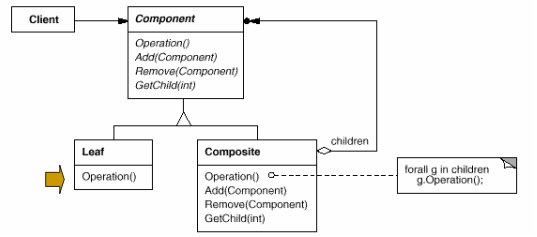
- Nome
  - Composite
- Descrição do problema
  - Compor objetos em estrutura hierárquica (todo-parte)
- Descrição da solução (próximo slide)
- Consequências
  - Objetos primitivos e compostos são tratados de maneira uniforme
  - Fácil incluir novos objetos

Solução do Composite



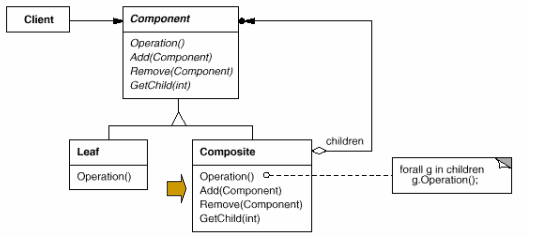
***Component* define as operações genéricas.**

Solução do Composite



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

Solução do Composite



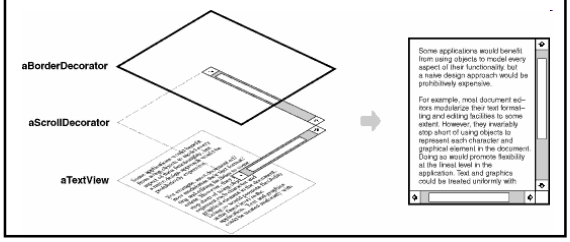
***Composite* representa elementos compostos.**

Padrões Estruturais

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

Problema (Padrão Decorator)

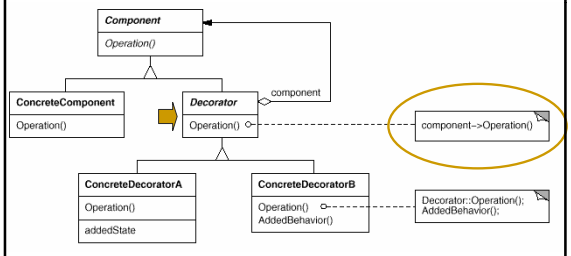
- Adicionar e remover responsabilidades dinamicamente a um objeto



Padrão Decorator

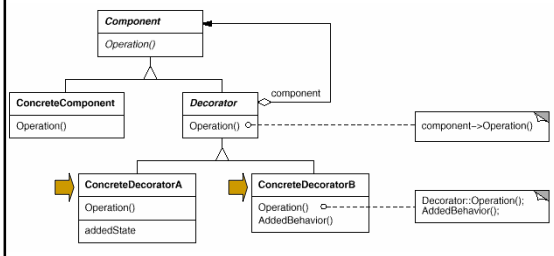
- Nome
  - Decorator
- Descrição do problema
  - Agregar dinamicamente novas responsabilidades a um objeto
- Descrição da solução (próximo slide)
- Consequências
  - Maior flexibilidade que herança
  - Evita sobrecarregar hierarquia de classes

Solução do Decorator



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

Solução do Decorator



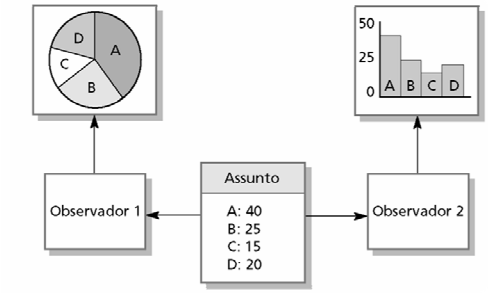
Podem haver vários decoradores concretos.

Padrões Comportamentais

Padrões Comportamentais

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

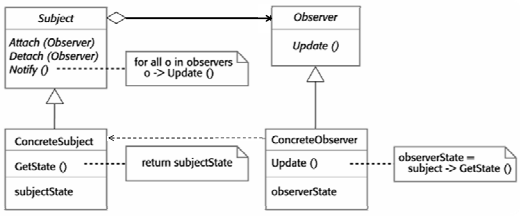
Problema (Padrão Observer)



Padrão Observer

- Nome
  - Observer
- Descrição do problema
  - Separar o objeto de sua forma de apresentação
- Descrição da solução (próximo slide)
- Consequências
  - Otimizações para melhorar a atualização da apresentação

Solução do Observer



Padrões Comportamentais

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

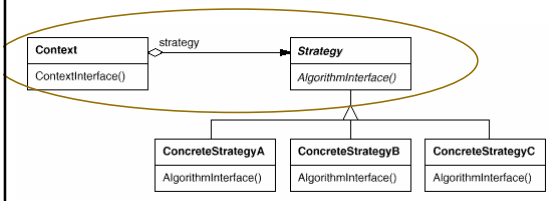
Problema (Strategy)

- 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

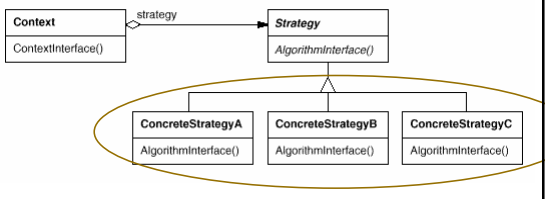
- Nome
  - Strategy
- Descrição do problema
  - Definir uma família de algoritmos
- Descrição da solução
  - próximo slide
- Consequências
  - Permite alternar entre diferentes algoritmos sem o uso de condicionais

Solução do Strategy



A aplicação usa o algoritmos sem conhecer sua real implementação.

Solução do Strategy



Diferentes implementações do algoritmo.

Referências

- E. Gamma, R. Helm, R. Johnson, J. Vlissides. **Padrões de Projeto**, 1a. Edição. Bookman, 2000.
  - Padrões: Factory Method, Singleton, Adapter, Composite, Decorator, Observer e Strategy