SCC0504 - Programação Orientada a Objetos

Design Pattern Pt 1

Prof.: Leonardo Tórtoro Pereira

leonardop@usp.br

Design Pattern?

O Que é um Design Pattern? [1]

- Uma solução genérica, replicável, de um problema de ocorrência comum em *design* de *software*
- Não é um *design* final pra ser transformado diretamente em código
  - Descrição ou *template* de como resolver um problema
  - Pode ser usado em diversas situações

Benefícios dos *Design Patterns* [1]

- Os padrões de *design* podem
  - Aumentar a velocidade de desenvolvimento
    - Paradigmas testados e provados
  - Ajuda a prevenir erros sutis que podem tornar-se grandes problemas
  - Melhora legibilidade para quem é familiar aos padrões
- Não requerem especificidades de problemas particulares
  - Podem ser generalidades facilmente

Críticas aos *Design Patterns* [1]

- Alguns críticos argumentam que só é necessário usar padrões de design caso a linguagem ou técnica não seja abstrata o suficiente
  - Algumas coisas são muito mais simples de fazer em *Lisp* (que permite paradigma funcional além de OO)

Contra-Argumento!

- ➔ Ok, mas cada vez mais faz-se necessário usar linguagens específicas vinculadas à frameworks, engines, etc.
- ➔ Nem sempre é possível escolher a linguagem com a qual vamos programar
- ➔ Ex: não podemos criar jogos com linguagens funcionais E usar todas as facilidades de motores de jogos super robustos e usados pela maioria das empresas de mercado (Unity usa C# e Unreal C++)

Críticas aos *Design Patterns* [1]

- Falta de fundamentação formal
- Pode resultar em solução ineficientes
  - Com certa duplicação de código
  - Muitas vezes é preferível uma implementação bem fatorada
- Alguns também dizem que os padrões não variam significativamente de outras formas de abstração
  - O que tornaria irrelevante sua existência

Os Tipos de Design Pattern

Tipos de *Design Pattern* [1, 2]

- Eles são divididos (principalmente) em 3 tipos:
  - Criacionais (*Creational*)
  - Estruturais (*Structural*)
  - Comportamentais (*Behavioral*)
- Mas também existe uma categoria mais recente
  - Arquiteturais (*Architectural*)
- E outros que não costumam ter categorias
  - Ex: *Dependency Injection, Method Chaining,* etc.

Criacionais

Criacionais (Creational) [3, 4, 5, 6]

- *COMO AS COISAS SÃO CRIADAS*
- Usados quando é preciso decidir algo quando instanciamos um objeto
- Pode ser dividido em padrões de criação de classe de objeto
  - De classe: uso eficiente de herança
  - De Objeto: uso eficiente de delegação

Criacionais (Creational) [3, 4, 5, 6]

- Tenta lidar com problemas como:
  - Código que referencia a criação de um objeto precisa conhecer os construtores dele
    - Aumenta acoplamento
  - O objeto não pode ser criado parcialmente
  - O construtor não consegue controlar o número de instâncias presentes na aplicação

Criacionais (Creational) [3, 4, 5, 6]

- Cria objetos enquanto esconde a lógica
  - Ao invés de instanciá-los diretamente
- Maior flexibilidade em decidir quais objetos precisam ser criados para dado caso de uso
- Padrões:
  - *Abstract Factory, Builder, Factory Method, Object Pool, Prototype, Singleton*

Singleton

Singleton [7]

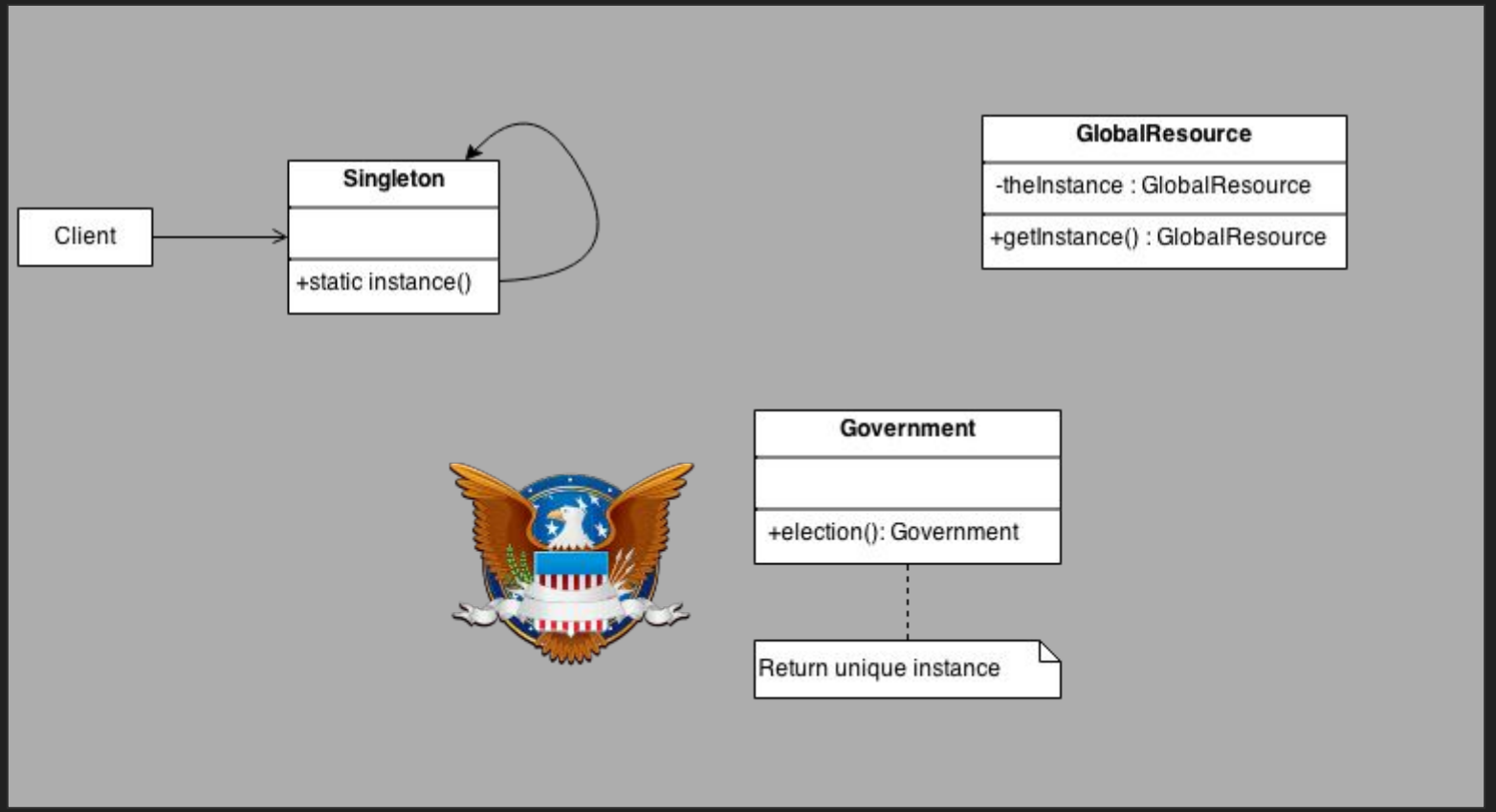
- O Singleton é um dos padrões mais comuns desse tipo e vamos usá-lo como exemplo
- Ele garante que apenas uma instância de uma classe é criada
- Também provê acesso global à esse objeto

Singleton [7]

- Deve ser usado quando é preciso garantir que apenas uma instância da classe pode existir em qualquer lugar do código
- Deve-se tomar cuidado extra em ambientes multi-thread pois todas precisam acessar os recursos de um mesmo objeto

Singleton [7]

- Alguns usos comuns:
  - Classes de log de informações
  - Classes de configuração
  - Acesso de recursos em modo compartilhado
  - Factories, Abstract Factories, Builder e Prototype são implementados como Singleton
  - Em jogos é muito usado para uma classe "*Manager*" que lida com a máquina de estado e dados


Client
Singleton
+static instance()
GlobalResource
-theInstance : GlobalResource
+getInstance() : GlobalResource
Government
+election(): Government
Return unique instance

Fonte: [8]

Singleton [7]

```
//https://www.oodesign.com/singleton-pattern.html#lazy-singleton
class Singleton{
    private static Singleton instance = new Singleton();
    private Singleton(){
        System.out.println("Singleton(): Initializing Instance");
    }
    public static Singleton getInstance(){
        return instance;
    }
    public void doSomething(){
        System.out.println("doSomething(): Singleton does
something!");
    }
}
```

Singleton [3]

- Construtor privado
  - Não pode ser instanciado fora da classe
- Classe é final
  - Não permite subclasses
- Acesso permitido através de método
  - Retorna a única instância da classe
  - Ou cria uma se não existir

Estruturais

Estruturais (Structural) [3, 4, 5, 6]

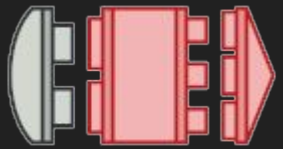
- ➔ *COMPOSIÇÃO DE CLASSES E OBJETOS*
- ➔ Usa herança para compor interfaces
- ➔ Como compor objetos para obter novas funcionalidades
- ➔ Foco em extensibilidade e flexibilidade
- ➔ Garante que quando uma das partes mudar a estrutura toda da aplicação não tenha que mudar

Estruturais (Structural) [3, 4, 5, 6]

- Descrevem os seguintes aspectos entre objetos e classes
  - Elaboração
  - Associação
  - Organização
- Combinam objetos em estruturas mais complexas
- Descrevem como classes são herdadas ou compostas a partir de outras

Estruturais (Structural) [3, 4, 5, 6]

- Padrões:
  - *Adapter, Bridge, Composite, Decorator, Facade, Flyweight, Private Class Data, Proxy*



Fonte: [9]

Fonte: [3]

Fonte: [9]

Decorator

Decorator [10]

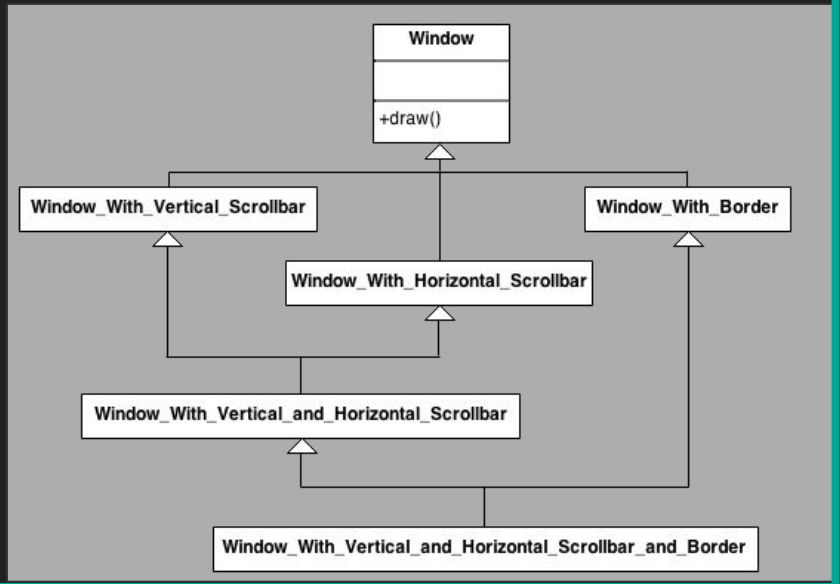
- ➔ Aplica responsabilidades adicionais a um objeto dinamicamente
- ➔ Alternativa flexível à criação de subclasses com o objetivo de estender funcionalidade
- ➔ Embelezamento específico para um cliente de um objeto *core* ao envolver ele recursivamente
- ➔ “Embalar um presente, colocar numa caixa e embalar a caixa”

Decorator [10]

- Quando usar?
  - Quando deseja-se adicionar um comportamento ou estado a um objeto individualmente em tempo de execução
  - Herança não é aplicável pq é estática e aplica à classe toda

Decorator [10]

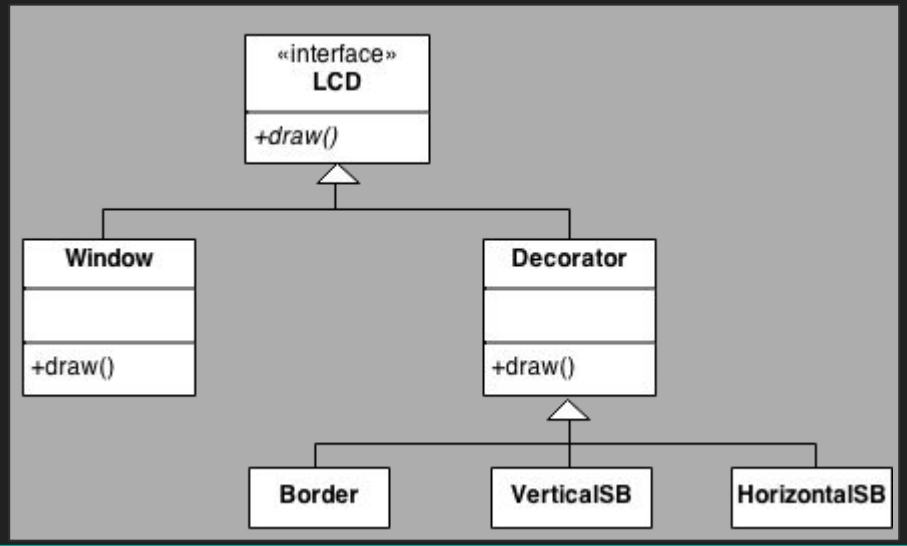
- Imagine que você quer adicionar bordas e barras de rolagem em janelas de uma interface
  - É possível fazer uma herança como ao lado:



Decorator [10]

➔ Mas o padrão Decorator sugere dar ao cliente a habilidade de especificar as combinações de características desejadas

```
Widget* aWidget = new BorderDecorator(
  new HorizontalScrollBarDecorator(
    new VerticalScrollBarDecorator(
      new Window( 80, 24 ))));
aWidget->draw();
```



Decorator [10]

- ➔ A solução envolve encapsular o objeto original dentro de uma interface empacotadora abstrata
- ➔ Ambos os objetos decoradores como o objeto *core* vão herdar da interface abstrata
- ➔ A interface usa composição recursiva para permitir que um número ilimitado de “camadas” decoradoras sejam adicionadas ao objeto *core*

Decorator [10]

- O padrão permite que sejam adicionadas responsabilidades ao objeto
  - Não métodos na interface do objeto
- A interface apresentada ao cliente deve permanecer constante mesmo com as novas camadas
- A identidade do objeto *core* é “escondida” dentro do objeto decorador.
  - Acessar o objeto *core* diretamente é um problema

Comportamentais

Comportamentais (Behavioral) [3, 4, 5, 6]

- ➔ *COMUNICAÇÃO DE OBJETOS DE UMA CLASSE*
- ➔ Padrões que se preocupam com a comunicação entre objetos
- ➔ Costuma reduzir dependências entre os objetos que se comunicam, o que é um design melhor de software
- ➔ Abstraem uma ação que queremos que ocorra num objeto ou classe que recebe a ação

Comportamentais (Behavioral) [3, 4, 5, 6]

- Ao mudar o objeto ou classe podemos mudar o algoritmo usado, os objetos afetados ou o comportamento
  - Enquanto mantemos a mesma interface básica para as classes cliente
- Exemplos:
  - *Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Null Object, Observer, State, Strategy, Template Method, Visitor*

State

State [11]

- Permite a um objeto alterar seu comportamento quando seu estado interno muda
  - O objeto vai aparentar mudar sua classe
- Máquina de estados orientada a objetos
- Empacotador + Empacotado de Polimorfismo + colaboração

State [11]

- Usado quando um comportamento de um objeto monolítico é uma função de seu estado
  - Precisa mudar seu comportamento em tempo de execução dependendo do seu estado

State [11]

- ➔ É preciso definir uma classe “contexto” para apresentar uma única interface ao “mundo exterior”
- ➔ Definir uma classe base abstrata *State*
- ➔ Representar os diferentes “estados” da máquina de estado como classes derivadas da classe base *State*

State [11]

- Definir comportamentos específicos de estados nas classes derivadas apropriadas
- Manter um ponteiro para o *"estado"* atual da classe *"contexto"*
- Para mudar o estado da máquina de estados
  - Mudar o ponteiro atual de "*estado*"

State [11]

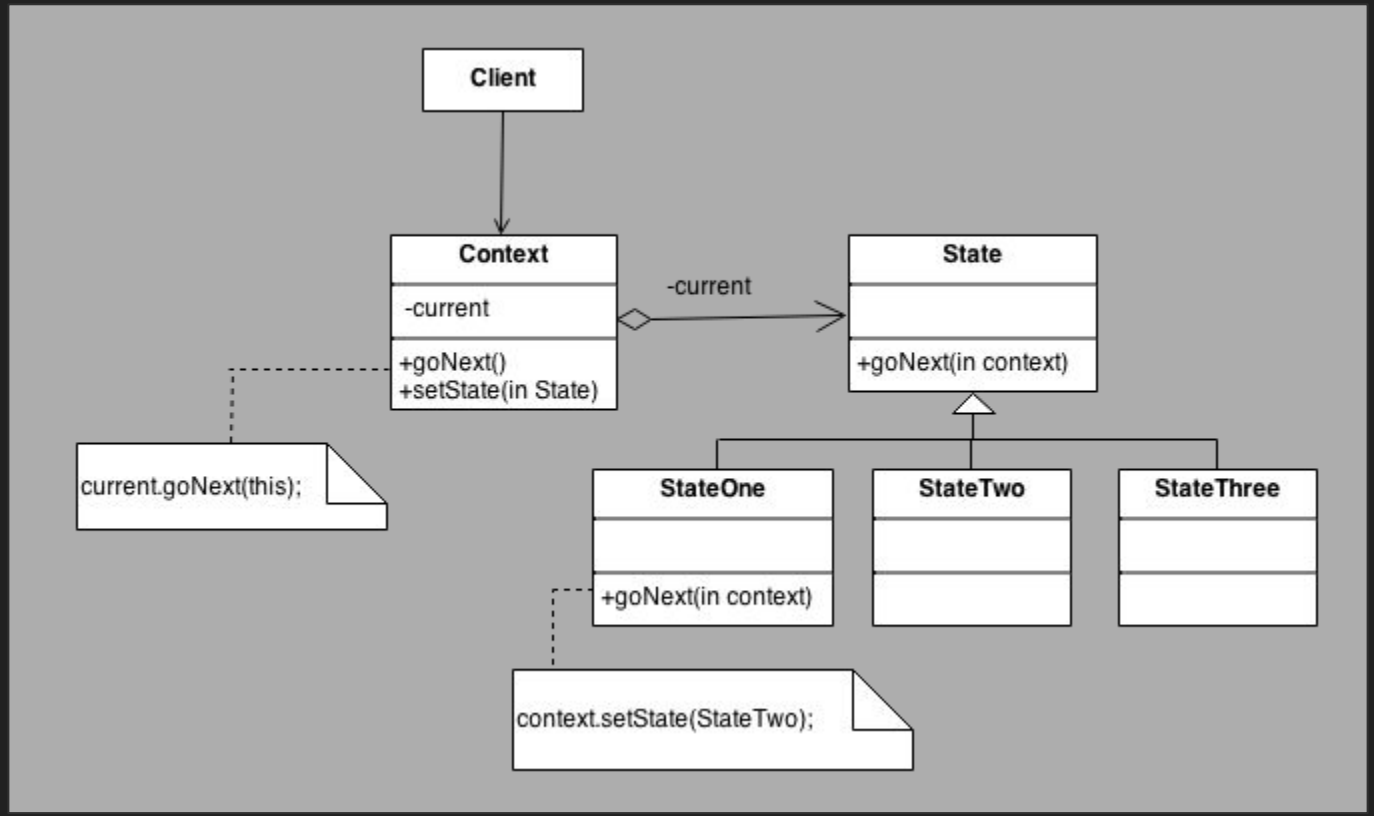
- O padrão não especifica onde as transições são definidas
- Duas possibilidades:
  - No objeto “contexto”
  - Ou cada classe derivada de State
- A segunda opção torna mais fácil adicionar novas classes derivadas de State
  - Mas cada classe derivada tem conhecimento (acoplamento com) suas “irmãs” -> Dependência

State [11]

- ➔ Uma abordagem baseada em tabelas é boa para especificar transições entre estados, mas é difícil adicionar as ações que acompanham essas transições
- ➔ A abordagem desse padrão usa código (ao invés de estruturas de dados) para especificar as transições, mas acomoda bem as ações de transição de estados

State [11]

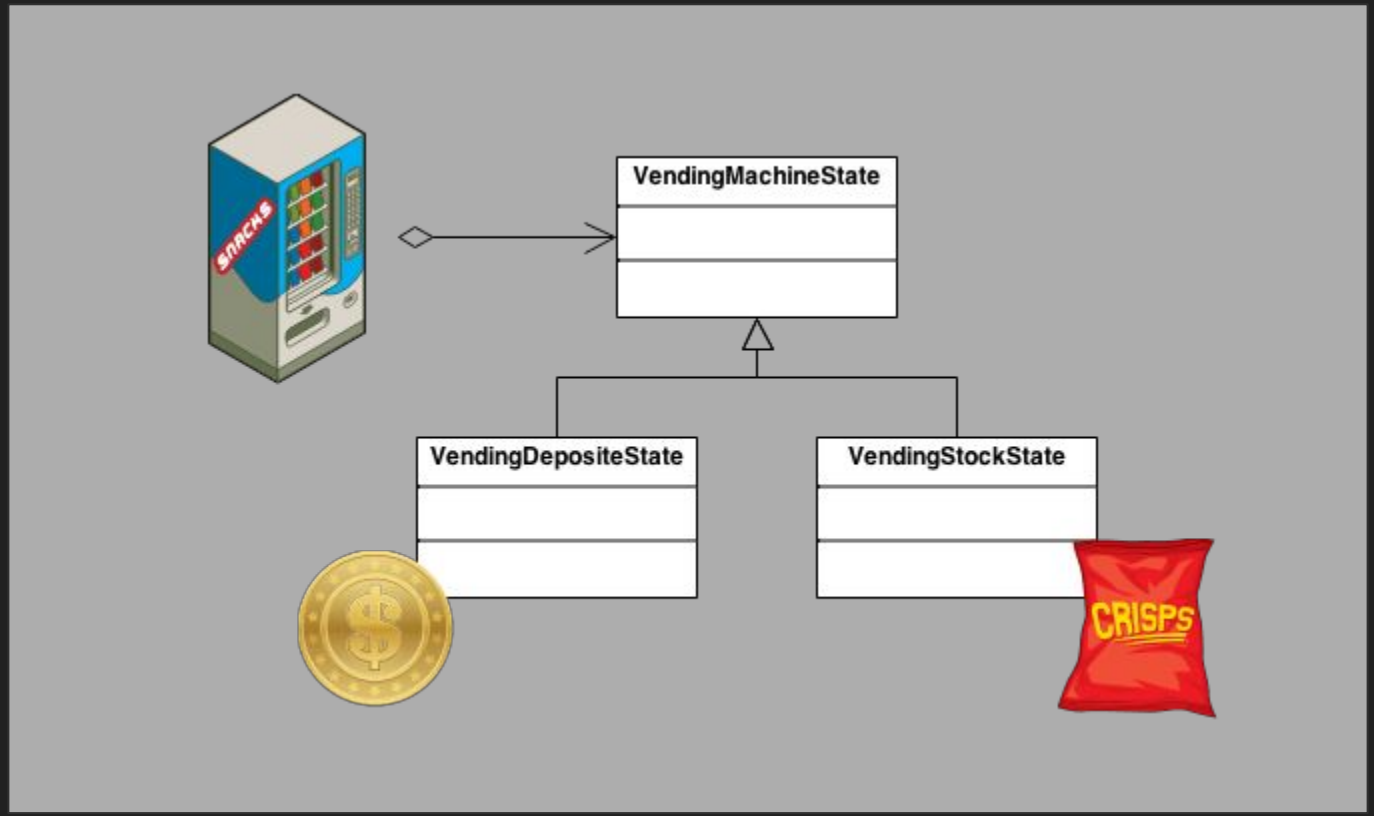
- A interface da máquina de estado é encapsulada na classe “empacotadora”
- A hierarquia do “empacotado” espelha a interface do empacotador com a exceção de um parâmetro extra
  - Permite classes derivadas do “empacotado” chamar de volta a classe “empacotadora” quando necessário
- A complexidade é compartimentada e encapsulada em uma hierarquia polimórfica delegada pelo empacotador


Client
Context
-current
+goNext()
+setState(in State)
-current
State
+goNext(in context)
current.goNext(this);
StateOne
+goNext(in context)
StateTwo
StateThree
context.setState(StateTwo);

Fonte: [11]

State [11]

- ➔ Podemos usar de exemplo uma máquina de vendas
- ➔ Estados baseados em itens, dinheiro depositado, troco, item selecionado, etc.
- ➔ Quando dinheiro é colocado e uma seleção é feita, a máquina vai ou entregar o item e o troco, não entregar um produto por falta de estoque, entregar um produto e nenhum troco ou não entregar nenhum produto por não ter pagamento suficiente.


SNACKS
VendingMachineState
VendingDepositeState
VendingStockState
CRISPS

Fonte: [11]

Os Padrões

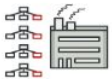
The Catalog of Design Patterns

Scope / purpose	Creational	Structural	Behavioral
Class	Factory Method	Adapter	Interpreter Template Method
Object	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Observer State Strategy Visitor

Fonte: [12]

Fonte: [9]

Creational Patterns



Abstract Factory ★★★

Lets you produce families of related objects without specifying their concrete classes.

Main article Code example

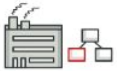
Usage in Java

Builder ★★★

Lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.

Main article Code example

Usage in Java

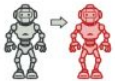


Factory Method ★★★

Provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created.

Main article Code example

Usage in Java



Prototype ★★☆

Lets you copy existing objects without making your code dependent on their classes.

Main article Code example

Usage in Java

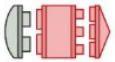
Singleton ★★☆

Lets you ensure that a class has only one instance, while providing a global access point to this instance.

Main article Naïve Singleton

Usage in Java Thread-safe Singleton

Structural Patterns



Adapter ★★★

Allows objects with incompatible interfaces to collaborate.

Main article Code example

Usage in Java

Bridge ★☆☆

Lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.

Main article Code example

Usage in Java

Composite ★★☆

Lets you compose objects into tree structures and then work with these structures as if they were individual objects.

Main article Code example

Usage in Java

Decorator ★★☆

Lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

Main article Code example

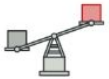
Usage in Java

Facade ★★☆

Provides a simplified interface to a library, a framework, or any other complex set of classes.

Main article Code example

Usage in Java

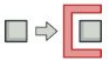


Flyweight ★☆☆

Lets you fit more objects into the available amount of RAM by sharing common parts of state between multiple objects instead of keeping all of the data in each object.

Main article Code example

Usage in Java



Proxy ★☆☆

Lets you provide a substitute or placeholder for another object. A proxy controls access to the original object, allowing you to perform something either before or after the request gets through to the original object.

Main article Code example

Usage in Java

Fonte:
[9]

Behavioral Patterns

Chain of Responsibility ★☆☆

Lets you pass requests along a chain of handlers. Upon receiving a request, each handler decides either to process the request or to pass it to the next handler in the chain.

Main article Code example

Usage in Java

Command ★★★

Turns a request into a stand-alone object that contains all information about the request. This transformation lets you parameterize methods with different requests, delay or queue a request's execution, and support undoable operations.

Main article Code example

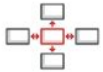
Usage in Java

Iterator ★★★

Lets you traverse elements of a collection without exposing its underlying representation (list, stack, tree, etc.).

Main article Code example

Usage in Java

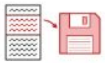


Mediator ★★☆

Lets you reduce chaotic dependencies between objects. The pattern restricts direct communications between the objects and forces them to collaborate only via a mediator object.

Main article Code example

Usage in Java



Memento ★☆☆

Lets you save and restore the previous state of an object without revealing the details of its implementation.

Main article Code example

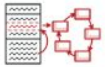
Usage in Java

Observer ★★★

Lets you define a subscription mechanism to notify multiple objects about any events that happen to the object they're observing.

Main article Code example

Usage in Java

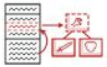


State ★★☆

Lets an object alter its behavior when its internal state changes. It appears as if the object changed its class.

Main article Code example

Usage in Java



Strategy ★★★

Lets you define a family of algorithms, put each of them into a separate class, and make their objects interchangeable.

Main article Code example

Usage in Java

Template Method ★★☆

Defines the skeleton of an algorithm in the superclass but lets subclasses override specific steps of the algorithm without changing its structure.

Main article Code example

Usage in Java

Visitor ★☆☆

Lets you separate algorithms from the objects on which they operate.

Main article Code example

Usage in Java

Referências

1. https://sourcemaking.com/design_patterns
2. https://www.opus-software.com.br/design-patterns/
3. https://www.devmedia.com.br/entendendo-os-conceitos-dos-padroes-de-projetos-em-java/29083
4. https://medium.com/@madasamy/introduction-to-object-oriented-design-patterns-part-i-4e5c7845015b
5. https://howtodoinjava.com/gang-of-four-java-design-patterns/
6. https://www.tutorialspoint.com/design_pattern/design_pattern_overview.htm
7. https://www.oodesign.com/
8. https://sourcemaking.com/design_patterns/singleton
9. https://refactoring.guru/design-patterns/java
10. https://sourcemaking.com/design_patterns/decorator
11. https://sourcemaking.com/design_patterns/state
12. https://pt.slideshare.net/opilo/object-oriented-design-patterns/2