

O DESPERTAR DO CÓDIGO



MESTRES DOS PATTERNS EM JAVA

ÍNDICE



Capítulo
01

Singleton Pattern

Capítulo
02

Factory Pattern

Capítulo
03

Observer Pattern

Capítulo
04

Strategy Pattern

Capítulo
05

Decorator Pattern

Capítulo
06

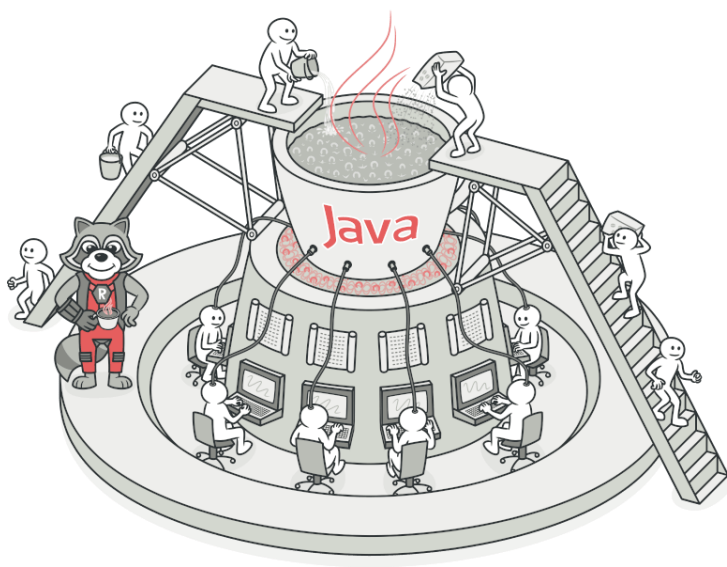
Conclusão

Bem Vindo



Seja bem-vindo ao mundo dos Designer Patterns em Java! Neste ebook, vamos explorar alguns dos principais padrões utilizados na arquitetura de software, com exemplos claros e simples para que seja de fácil entendimento.

Existem muitos padrões de design, mas alguns são mais amplamente reconhecidos e utilizados na prática de desenvolvimento de software. Os padrões selecionados neste ebook foram escolhidos por sua relevância e aplicabilidade em diferentes cenários de desenvolvimento na linguagem Java.



01 Singleton Pattern



O Singleton Pattern é utilizado quando queremos garantir que uma classe tenha apenas uma instância e fornecer um ponto de acesso global a essa instância. É comumente usados em situações em que precisamos de um único objeto para coordenar ações em todo o sistema, como gerenciadores de conexão, caches ou configurações globais.

```
Mestres dos Patterns - Singleton

public class Singleton {
    private static Singleton instance;
    // Construtor privado para evitar instâncias externas
    private Singleton() {}

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

02 Factory Pattern



O Factory Pattern é usado quando queremos criar objetos sem expor a lógica de criação diretamente ao cliente. Isso ajuda a encapsular a criação de objetos e a tornar o código mais flexível para futuras mudanças.

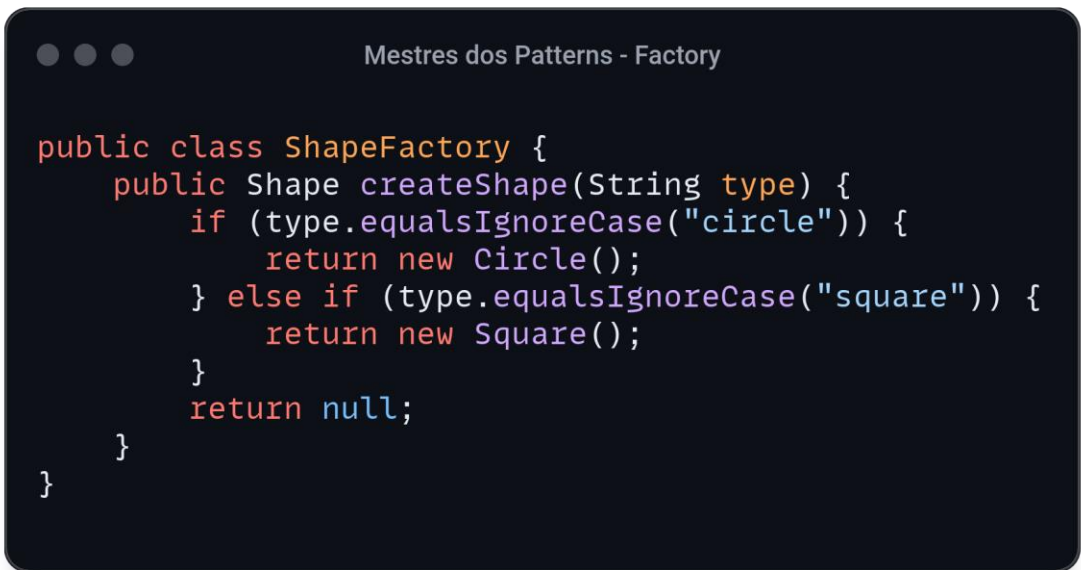
```
Mestres dos Patterns - Factory

public interface Shape {
    void draw();
}

public class Circle implements Shape {
    @Override
    public void draw() {
        System.out.println("Desenhando um círculo.");
    }
}

public class Square implements Shape {
    @Override
    public void draw() {
        System.out.println("Desenhando um quadrado.");
    }
}
```


Aqui temos um exemplo, na qual temos um método declarado em uma interface, onde existem duas classes que a implementam, cada uma sobrescrevendo a lógica do método conforme sua necessidade.



```
public class ShapeFactory {  
    public Shape createShape(String type) {  
        if (type.equalsIgnoreCase("circle")) {  
            return new Circle();  
        } else if (type.equalsIgnoreCase("square")) {  
            return new Square();  
        }  
        return null;  
    }  
}
```

Agora temos um serviço que necessita retornar uma instância de uma lógica específica. Desta forma, ela passa como parâmetro qual é o tipo de instância que necessita. Assim o serviço retorna a instância esperada sem a necessidade de o cliente saber como a lógica é implementada. Isso ajuda a tornar o código mais flexível, fácil de manter e facilita futuras extensões, especialmente em cenários onde a lógica de criação de objetos pode ser complexa ou variável.

03 Observer Pattern



O Observer Pattern é útil quando um objeto precisa notificar outros objetos sobre mudanças de estado. É comumente usados em interfaces gráficas e sistemas de eventos.

O exemplo a seguir demonstra como implementar um sistema de notificação onde um objeto notifica seus observadores sobre mudanças de estado.

```
Mestres dos Patterns - Observer

public interface Observer {
    void update(String message);
}

public class ConcreteObserver implements Observer {
    @Override
    public void update(String message) {
        System.out.println("Nova mensagem recebida: " + message);
    }
}

public interface Subject {
    void attach(Observer observer);
    void detach(Observer observer);
    void notifyObservers(String message);
}
```

Resumindo, o exemplo simula um sistema onde um sujeito (representado pela classe `ConcreteSubject`) pode enviar mensagens e notificar automaticamente todos os observadores (representados pela classe `ConcreteObserver`) registrados sobre a chegada de uma nova mensagem, chamando o método `update` em cada observador. Este padrão é frequentemente utilizado em sistemas onde a comunicação assíncrona entre componentes é necessária, como em interfaces gráficas, sistemas de eventos e notificações em tempo real.

```
Mestres dos Patterns - Observer

public class ConcreteSubject implements Subject {
    private List<Observer> observers = new ArrayList<>();

    @Override
    public void attach(Observer observer) {
        observers.add(observer);
    }

    @Override
    public void detach(Observer observer) {
        observers.remove(observer);
    }

    @Override
    public void notifyObservers(String message) {
        for (Observer observer : observers) {
            observer.update(message);
        }
    }

    public void sendMessage(String message) {
        notifyObservers(message);
    }
}
```


04 Strategy Pattern



O Strategy Pattern é usado quando queremos definir um conjunto de algoritmos, encapsulá-los e torná-los intercambiáveis. Isso permite que o cliente escolha o algoritmo desejado em tempo de execução, sem alterar a estrutura do código.

O exemplo abaixo demonstra como implementar uma família de algoritmos, encapsular esses algoritmos e torná-los intercambiáveis.

```
Mestres dos Patterns - Strategy


public interface Strategy {
    void executeStrategy();
}

public class ConcreteStrategy1 implements Strategy {
    @Override
    public void executeStrategy() {
        System.out.println("Executando estratégia 1.");
    }
}

public class ConcreteStrategy2 implements Strategy {
    @Override
    public void executeStrategy() {
        System.out.println("Executando estratégia 2.");
    }
}
```

Em suma, o Strategy Pattern permite definir um conjunto de algoritmos, encapsular cada algoritmo e fornecer uma forma de trocar dinamicamente entre esses algoritmos. Isso é útil quando queremos que um objeto possa alterar seu comportamento durante a execução sem modificar sua estrutura.

O padrão Strategy é frequentemente usado em situações onde diferentes estratégias podem ser aplicadas a um objeto e a escolha da estratégia pode variar dinamicamente..



```
public class Context {  
    private Strategy strategy;  
  
    public Context(Strategy strategy) {  
        this.strategy = strategy;  
    }  
  
    public void executeStrategy() {  
        strategy.executeStrategy();  
    }  
}
```

05 Decorator Pattern



O Decorator Pattern é usado para adicionar funcionalidades a um objeto dinamicamente, sem modificar sua estrutura. Isso é útil quando precisamos estender o comportamento de objetos de um Sistema Legado, sem precisar modifica-los e implementando novas funcionalidades de forma flexível.

O exemplo a seguir demonstra como adicionar responsabilidades adicionais a um objeto de forma dinâmica, sem alterar sua estrutura básica.

```
Mestres dos Patterns - Decorator

public class LuxuryCar extends CarDecorator {
    public LuxuryCar(Car decoratedCar) {
        super(decoratedCar);
    }

    @Override
    public void assemble() {
        super.assemble();
        System.out.println("Adicionando recursos de luxo.");
    }
}
```


o Decorator Pattern permite adicionar comportamentos ou funcionalidades extras a um objeto de forma dinâmica, mantendo a flexibilidade e a coesão do código.

Isso é útil quando temos múltiplas variações de um objeto e queremos adicionar funcionalidades específicas a cada variação sem criar subclasses excessivas ou modificar diretamente a classe base. O padrão Decorator é comumente utilizado em cenários onde a composição de comportamentos é preferível à herança de classes.

```
Mestres dos Patterns - Decorator

public interface Car {
    void assemble();
}

public class BasicCar implements Car {
    @Override
    public void assemble() {
        System.out.println("Carro básico montado.");
    }
}

public abstract class CarDecorator implements Car {
    protected Car decoratedCar;

    public CarDecorator(Car decoratedCar) {
        this.decoratedCar = decoratedCar;
    }

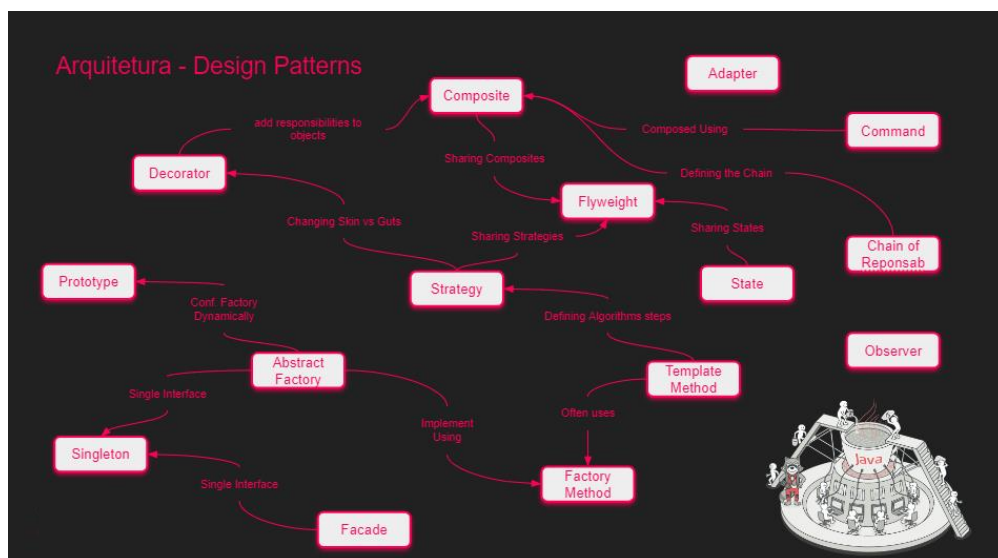
    public void assemble() {
        decoratedCar.assemble();
    }
}
```

06 Conclusão



Esses padrões foram escolhidos por serem fundamentais no desenvolvimento de software orientado a objetos, oferecendo soluções eficientes e eficazes para problemas comuns de design e arquitetura. Ao compreender e aplicar esses padrões em seus projetos, você estará construindo sistemas mais flexíveis, modulares e de fácil manutenção.

Existem inúmeros outros Designer Patterns que se complementam para estruturar a arquitetura, não só com Java. Ao compreender e aplicar esses padrões em seu código, você construirá sistemas mais robustos, flexíveis e de fácil manutenção. Para conhecer mais, acesse o site [Refactoring Guru](https://refactoring.guru) para conhecê-los.





OBRIGADO!

Obrigado por ter chegado até aqui e espero que tenha gostado do conteúdo. Experimente e aprofunde seu conhecimento nesta fascinante área da programação!.

Este ebook foi produzido com referência do [Refactoring Guru](#), em conjunto com ferramentas de IA.



Autor

LinkedIn | GitHub