

Padrões de Design

Definição. Refere-se a soluções reutilizáveis para problemas comuns de design de software.

Os principais tipos de padrões de projeto podem ser categorizados nos seguintes grupos:

- Padrões de Criação:** Aborda a questão da geração de instâncias de objetos e classes, bem como as dependências entre si.
 - Singleton:** Garante a existência de apenas uma instância de uma classe e fornece um ponto global de acesso a essa instância.
 - Eager Singleton:** Instância única é criada no momento em que a classe é carregada na memória, isto é, uma criação antecipada. Pode consumir mais recursos.
 - Lazy Singleton:** Instância única é criada apenas quando houver necessidade, ou seja, de forma preguiçosa. Isso economiza recursos, porém pode gerar sobrecarga na hora de criação da instância.
 - Factory:** Permite a criação de objetos sem especificar a classe exata do objeto que será criado. Isso é realizado por uma interface ou classe base comum que declara um método para criar objetos. Feito isso, são declaradas classes bases que implementam esse método para objetos específicos. As duas principais categorias do padrão "Factory" são:
 - Factory Method:** Define uma interface para criar objetos, permitindo que as subclasses escolham as classes concretas que serão instanciadas.
 - Abstract Factory:** Fornece uma interface para criar famílias de objetos sem especificar as classes concretas. Útil para a criação de objetos pertencentes a um conjunto de classes relacionadas.
- Padrões de Comportamento:** Tratam do comportamento e interação entre objetos e classes. Concentram-se na colaboração de objetos para cumprir responsabilidades e na atribuição das mesmas aos objetos.
 - Observer:** Define uma dependência de um-para-muitos entre objetos de modo que quando um objeto muda de estado, todos os seus dependentes são notificados e atualizados.
- Padrões de Estrutura:** Concentram-se na organização de classes e objetos para compor estruturas maiores e mais complexas.

Singleton.

Como vimos anteriormente, o padrão de design "Singleton" se refere à criação e tratamento de uma única instância de uma classe particular no programa.

Aqui está o código base para a criação de uma classe `Singleton` e sua implementação.

```
public class Singleton {
    private static Singleton uniqueInstance;


    private Singleton() {
        /* Construtor não precisa ser vazio */
    }

    public static synchronized Singleton getInstance() {
        if (uniqueInstance == null) {
            return new Singleton();
        } else return uniqueInstance;
    }
}

public class Principal {
    public static void main(String[] args) {
        Singleton singleton = Singleton.getInstance();

        System.out.println(singleton);
        System.out.println(Singleton.getInstance());
    }
    /* ... */
}
```

Como podemos ver, declaramos uma instância privada e estática de `Singleton` dentro da própria classe juntamente de um **construtor privado**. Feito isso, restringimos a utilização desse construtor ao método estático `getInstance`. Isso impede a criação de instâncias fora da classe, apenas em seu interior.

 A palavra-chave `synchronized` restringe utilização do método por mais de uma **thread** simultaneamente, ou seja, está "sincronizado".

Denominamos essa aplicação de **Lazy Singleton**, ou seja, apenas declaramos uma instância de `Singleton` quando for necessário, mais especificamente no interior do método `getInstance` da classe.

No entanto, podemos também declarar uma única instância de `Singleton` no início do programa e não alterá-la da seguinte forma:

```
public class Singleton {
    public static final Singleton = new Singleton();
    private Singleton() {

    }
    /* ... */
}
```

Chamamos essa implementação de **Eager Singleton**, ou seja, uma única instância é criada e carregada na memória de forma antecipada.

Factory.

Por outro lado, o padrão de criação `Factory` fornece uma interface comum para a criação de objetos de diferentes tipos os quais, por sua vez, implementam uma mesma interface.

Nesse sentido, existem algumas formas de implementar o padrão de design `Factory`, sendo uma delas por meio da definição de uma interface e de uma classe com um método responsável por retornar objetos de classes distintas, porém pertencentes a essa interface.

Aqui está um exemplo de implementação da abordagem mencionada.

```
interface Shape {
    void draw();
}

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

public class Rectangle implements Shape {
    @Override
    public void draw() {
        System.out.println("Eu sou um retângulo.");
    }
}

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

public class ShapeFactory {
    public static Shape getShape(String shapeType) {
        if (shapeType == null) return null;
        return switch(shapeType.toLowerCase()) {
            case "circle" -> new Circle();
            case "rectangle" -> new Rectangle();
            case "square" -> new Square();
            default -> null;
        };
    }
}
```

Como podemos notar, a classe `ShapeFactory`, por si só, é responsável pela criação de todos os objetos que implementam a interface `Shape` segundo o tipo de parâmetro `String shapeType` recebido pelo método estático.

No entanto, também podemos ter uma outra implementação do `Factory`, isto é, o "abstract factory" que possui uma classe cujos métodos criam cada tipo de objeto separadamente.

Aqui estão algumas das mudanças que seriam realizadas para alternar entre as implementações.

```
interface ShapeFactory {
    Shape createCircle();
    Shape createRectangle();
    Shape createSquare();
}

class ConcreteShapeFactory implements ShapeFactory {
    @Override
    public Shape createCircle() {

        return new Circle();
    }

    @Override
    public Shape createRectangle() {
        return new Rectangle();
    }

    @Override
    public Shape createSquare() {
        return new Square();
    }
}
```

Entretanto, essa aplicação do padrão de criação "factory" aparenta ser menos eficiente.

Vantagens de Factory.

- A criação de objetos está concentrada em somente um lugar e qualquer classe que precise criar um objeto específico utiliza a classe `Factory`.
- As classes que precisam de objetos específicos não precisam conhecer todos os objetos que podem ser criados.
- O código é dinâmico, podemos mudar de uma implementação de `Factory` para outra.

Observer.

Por fim, o padrão de projeto Observer é um padrão comportamental que define uma dependência de um para muitos entre objetos, ou seja, quando um objeto muda de estado, todos seus dependentes são notificados e atualizados.

Existem duas classes primordiais neste padrão de design, isto é, a classe que está sendo observada (denominada de `Subject`) e a classe observadora (`Observer`) que será notificada assim que houver alguma mudança no sujeito e atualizada de acordo.

Outrossim, normalmente utilizamos uma interface para a implementação da classe `Observer` que herda o método `update(Object change)` com o parâmetro indicando a mudança que foi realizada.

Aqui está um exemplo da utilização do padrão comportamental `Observer`. Suponha a classe "sujeita" `Counter` e a observadora `CounterObserver`.

```
public class Counter {
    private int count;
    private List<Observer> observers = new ArrayList<>();

    public int getCount() {
        return count;
    }

    public void increment() {
        count++;
        notifyObservers();
    }

    public void addObserver(Observer observer) {
        observers.add(observer);
    }

    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }

    private void notifyObservers() {
        for (Observer observer : observers) {
            observer.update();
        }
    }
}

interface Observer {
    void update();
}

public class CounterObserver {
    private Counter counter;

    public CounterObserver(Counter counter) {
        this.counter = counter;
        counter.addObserver(this);
    }

    @Override
    public void update() {
        System.out.println("O contador foi atualizado");
    }
}
```

Embora o programa não possua uma função `main`, podemos notar que se criarmos um objeto do tipo `Counter` juntamente de alguns observadores e incrementarmos a instância de `Counter` criada, todos os observadores serão notificados e atualizados de acordo.