DES-SIS-II - A5 Texto de apoio

Site: EAD Mackenzie
Tema: DESENVOLVIMENTO DE SISTEMAS II {TURMA 03B} 2023/1
Livro: DES-SIS-II - A5 Texto de apoio

Impresso por: FELIPE BALDIM GUERRA .
Data: sexta, 28 abr 2023, 02:15

Descrição

Índice

1. PADRÕES GoF CRIACIONAIS

Conhecendo os Padrões GoF

Já sabemos que padrões são soluções para problemas encontrados com frequência no desenvolvimento de software – eles representam boas práticas já conhecidas e testadas em diferentes contextos.

Nas aulas passadas, estudamos os nove padrões GRASP (*General Responsibility Assignment Software Patterns* – Padrões de Software para Atribuição Geral de Responsabilidades): *Information Expert*, *Creator, Controller*, *Low Coupling* e *High Cohesion, Polymorphism*, *Pure Fabrication*, *Indirection* e *Protected Variations*. Vimos que são padrões de uso bastante geral, pois se preocupam com o processo de atribuição de responsabilidades como um todo.

Um pouco diferente dos GRASP são os padrões GoF (*Gang of Four*): trata-se de um catálogo de 23 padrões a serem aplicados no momento do projeto de software orientado a objetos (especialmente, na hora de modelar diagramas de classe).

> Este nome curioso – *Gang of Four* ou "Turma dos Quatro" – deve-se aos quatro autores do livro *Design Patterns: Elements of Reusable Object-Oriented Software*, de 1994, que é um marco na Engenharia de Software. Neste livro, os quatro autores: Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides apresentam pela primeira vez o catálogo com os 23 padrões.

Esses padrões trazem soluções prontas para diferentes problemas encontrados no projeto de software, e são divididos em três grupos:

Fonte: Elaborada pelo autor.

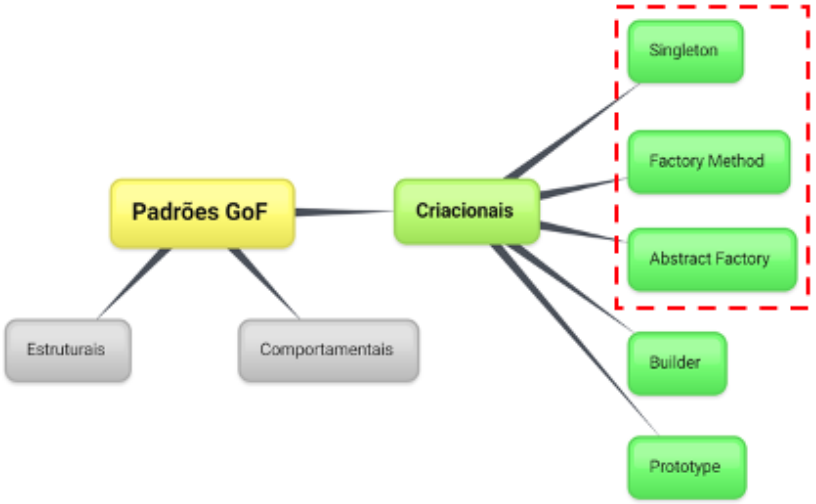
- **Padrões GoF criacionais** relacionam-se a situações envolvendo criação de objetos.
- **Padrões GoF estruturais** referem-se a soluções para a reorganização de determinadas estruturas em um diagrama de classes.
- **Padrões GoF comportamentais** representam soluções prontas para modelar determinados comportamentos no projeto de software.

Nesta aula, conheceremos alguns dos padrões criacionais e trabalharemos exemplos de sua aplicação.

Padrões GoF Criacionais

Todos os padrões desta categoria preocupam-se com diferentes situações nas quais o cerne do problema é a instanciação, ou seja, a criação de objetos. No catálogo GoF, são cinco padrões nesta categoria: *Singleton*, *Factory Method*, *Abstract Factory*, *Builder* e *Prototype*. Destes, estudaremos os três primeiros.

> Como fizemos com refatoração e GRASP, manteremos os nomes originais sem traduzi-los, pois é como são conhecidos no mercado e na academia – afinal, um dos grandes objetivos do uso de padrões é criar uma espécie de “vocabulário comum” entre projetistas de software, e isso só é possível quando se usa um nome único para cada conceito.

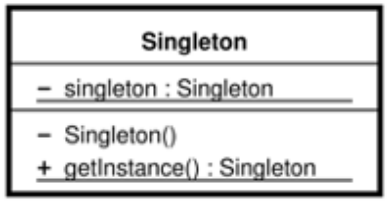


Fonte: Elaborada pelo autor.

Singleton

- **Problema:** Em algumas situações, é necessário restringir que uma classe tenha uma, e apenas uma, instância.
- **Solução:** Implemente esta classe como um *singleton*: uma classe cuja instanciação de objetos é feita de maneira restrita e controlada, de forma a ter sempre apenas uma única instância.

Isso se faz com um construtor *private* – o que acaba proibindo outra classe de usar a construção *new Singleton( )*. A única instância da classe sendo obtida por um método que geralmente é chamado de *getInstance( )*.



Fonte: Wikimedia.

```
public final class Singleton {

    private static final Singleton INSTANCE = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
      return INSTANCE;
    }
}

//Exemplo de uso
Singleton s = Singleton.getInstance();
```

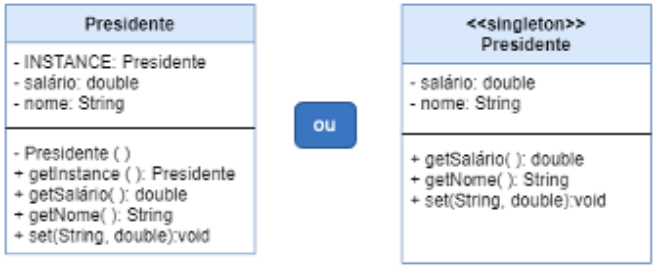
Fonte: Wikipedia.

Pontos a considerar:

- *Singleton* é um dos padrões GoF mais conhecidos, de maneira que, em um diagrama de classes UML, basta você acrescentar um estereótipo <<singleton>> acima do nome da classe que será compreendida.
- *Singleton* é considerado por muitos desenvolvedores um *antipattern*, pois seu uso indiscriminado pode trazer alguns problemas.
- Note que o código acima está bastante simplificado e não suporta, por exemplo, acesso concorrente (com threads, por exemplo).

Exemplo do Mundo Real

Em uma empresa convencional, é comum que haja apenas uma pessoa com o cargo de presidente. Como representar esta situação usando o padrão *Singleton*? Veja a seguir:



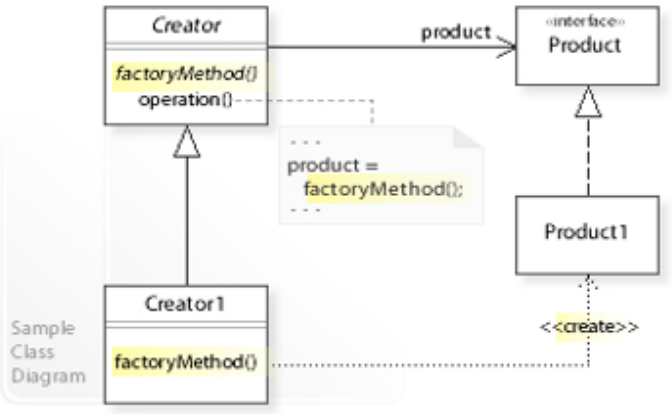
Fonte: Elaborada pelo autor.

Note que, na representação à direita, usamos um estereótipo para indicar que a classe é um *singleton* e nem é mais obrigatório colocar explicitamente o atributo INSTANCE , o construtor privado ou o método getInstance( ), pois todos saberão que, por ser um *singleton*, esta classe terá que ter estes elementos em sua implementação.

> Veja mais em: <https://refactoring.guru/design-patterns/singleton >.
>
> Aplique o que aprendeu agora na primeira atividade Praticando desta aula.

Factory Method

- **Problema:** Como criar objetos sem ter que especificar a classe à qual eles pertencem?
- **Solução:** Crie uma classe de fábrica, com um único método de “fabricação” (instanciação) de objetos. Cada tipo de objeto deverá ter uma fábrica responsável unicamente por sua criação.



Fonte: Wikimedia.

```
abstract class Product {
}

class Product1 extends Product {

}

abstract class Creator {
     protected abstract Product factoryMethod();
}

class Creator1 extends Creator{
     protected Product factoryMethod(){
        return new Product1();}
}

//Exemplo de uso
Creator c = new Creator1();
Product p = c.factoryMethod();
```

Fonte: Elaborado pelo autor.

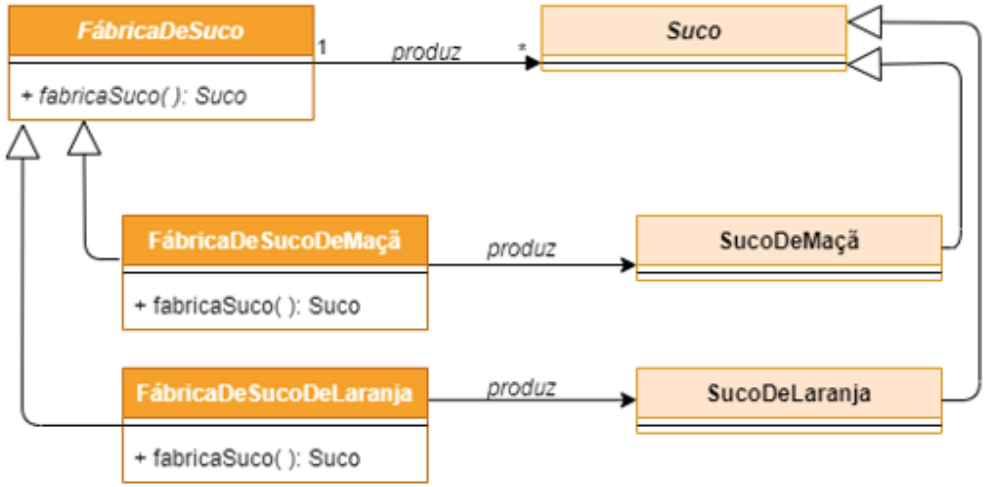
Pontos a considerar:

- Note que a classe Product e a classe Creator são abstratas (em Java, podem ser implementadas como abstract class ou interface), bem como seu método de criação de objetos, o factoryMethod( ) – ele deve ser sobrescrito pelas classes-filhas de Creator (como Creator1).
- Deve-se analisar se não haverá uma "explosão de classes" com a aplicação indiscriminada deste padrão. Se houver n tipos de objetos a serem criados, o uso deste padrão levará à criação de n classes de fábrica. (Neste caso, recomenda-se verificar o próximo padrão, *Abstract Factory*)

- As classes de fábrica podem ser implementadas como *Singleton.*

Exemplo do Mundo Real

Considere uma fábrica de sucos em caixinha automatizada, que produz diferentes sabores de suco. Uma aplicação do uso de *Factory Method* nesta situação seria:



Fonte: Elaborada pelo autor.

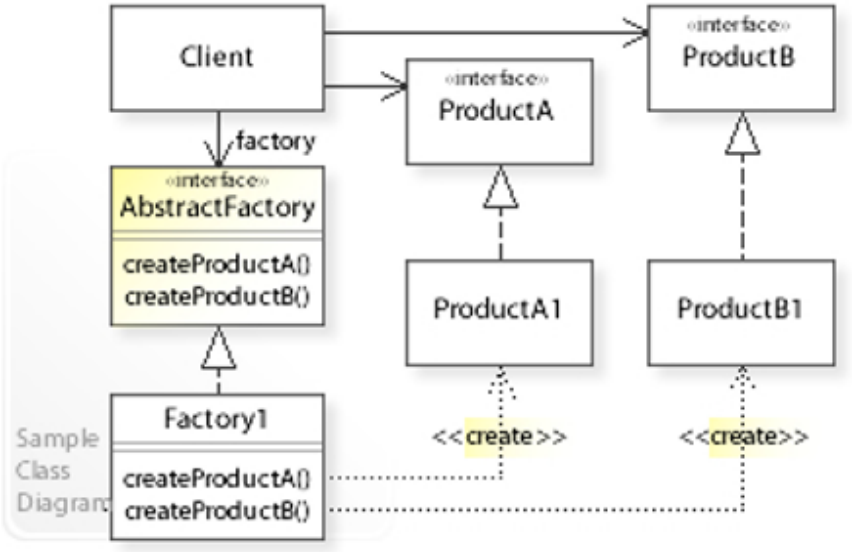
Observe que seria fácil criar um tipo de suco (de uva, por exemplo): basta criar uma classe FábricaDeSucoDeUva, herdando de FábricaDeSuco, que instanciaria objetos de uma outra classe SucoDeUva, herdeira de Suco .

Veja mais em: <https://refactoring.guru/design-patterns/factory-method>.

Aplique o que aprendeu agora na segunda atividade Praticando desta aula.

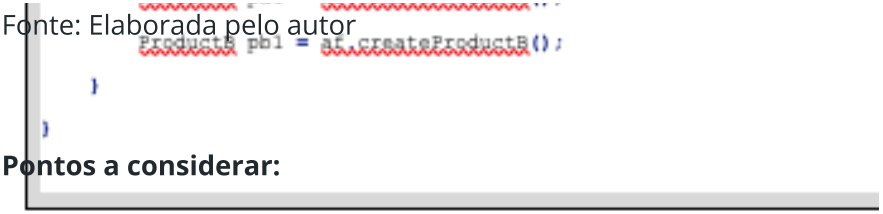
Abstract Factory

- **Problema:** Como criar famílias de objetos sem ter que especificar a classe à qual eles pertencem?
- **Solução:** Crie uma classe de fábrica abstrata, com um método de "fabricação" (instanciação) para cada um dos tipos de objetos. Cada fábrica concreta deverá ser capaz de criar sua própria versão da família de objetos.



Fonte: Wikimedia.

```
abstract class AbstractFactory {
 public abstract ProductA createProductA();
 public abstract ProductB createProductB();
}

interface ProductA {
}

interface ProductB {
}

class ProductA1 implements ProductA{
}

class ProductB1 implements ProductB{
}

class Factory1 extends AbstractFactory{
 public ProductA createProductA(){
    return new ProductA1();
 }
 public ProductB createProductB(){
    return new ProductB1();
 }
}

//Exemplo de uso
class Client
{
    public static void main (String[] args)
    {
        AbstractFactory af = new Factory1();
        ProductA pa1 = af.createProductA();
```



Fonte: Elaborada pelo autor

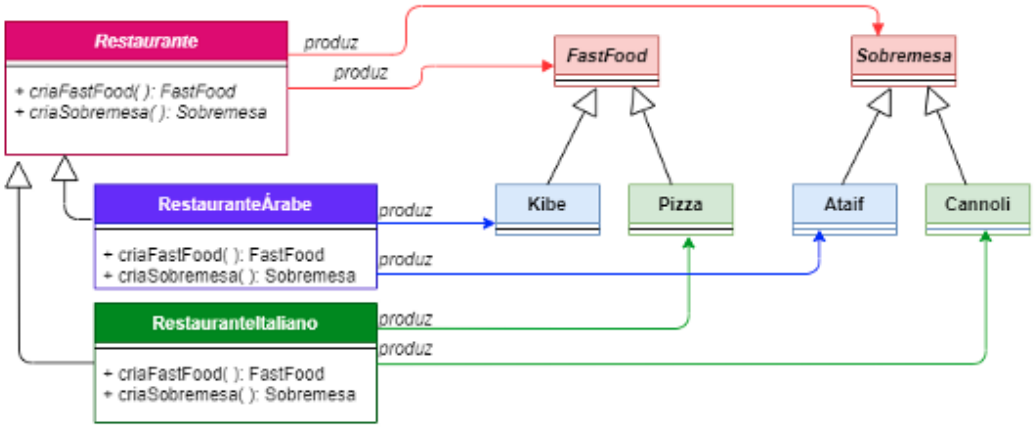
Pontos a considerar:

- Note que a definição da família de objetos que cada fábrica deve criar é bastante importante e não deve variar em diferentes fases de um software (se a família de objetos mudar, mudam todas as fábricas).
- As fábricas “concretas” devem produzir suas versões de todos os objetos previstos na família (a herança nunca é seletiva).
- As fábricas “concretas” podem ser implementadas como *Singleton*.
- No código, usamos interfaces para ProductA e ProductB e classe abstrata para AbstractFactory. No contexto deste código, não há diferença significativa.

Exemplo do Mundo Real

Considere a ideia de um restaurante que deve produzir sempre dois tipos de comida: um tipo de *fast food* e uma sobremesa. Um restaurante árabe que siga esse modelo pode produzir kibes como *fast food* e *ataif*como sobremesa (*ataif* é um doce muito comum nos países do Oriente Médio e é uma massa parecida com a de panqueca, recheada com nozes ou ricota). Já um restaurante italiano pode oferecer pizza como *fast food* e *cannoli* como sobremesa (*cannoli* é um doce italiano que consiste em uma massa doce em forma de tubo que é frita e recheada com creme de ricota).

Veja o diagrama a seguir:



Fonte: Elaborada pelo autor.

Note como seria fácil acrescentar novos restaurantes. Para inserir um restaurante japonês, por exemplo, basta fazê-lo herdar da classe restaurante e implementar um tipo de *fast food* (um bom *temaki*, por exemplo, que herdaria da classe FastFood) e uma sobremesa (pode ser um gostoso *manju* – doce de arroz, trigo e pasta doce de feijão – que herdaria de Sobremesa).

> Veja mais em:<https://refactoring.guru/design-patterns/abstract-factory>.