

**Bacharel em Engenharia de Software**

**Disciplina Modelo, Métodos** **e Técnias de engenharia de Software**

**Visão e Análise de Projeto – Padrões**

**Professor: Rubem Koide**

|  |  |
| --- | --- |
| **Alunos/RA**: |  |
| Caroline Araldi | 172320037 |
| Marcelo Augusto Dos Santos | 172313472 |
| Nicole Karolyne Balão | 172313357 |
| João Victor | 942321478 |

**Curitiba, PR**

**2023**

**1. Sumário**

**2. Introdução............................................................................................**

**3. Teste dos códigos...............................................................................**

**3.1 Classe Criação: Factory Method...........................................**

**3.2 Objeto Criação Builder..........................................................**

**3.3 Estrutura Class adapter........................................................**

**3.4 Estrutura Facade Objeto.......................................................**

**3.5 Comportamento Classe: Template method........................**

**3.6 Comportamento Objeto: Strategy........................................**

**4. Conclusão...........................................................................................**

**5. Referências.........................................................................................**

**2. Introdução**

Padrões de projeto, também conhecidos como **design pattern,** são soluções reutilizáveis para problemas comuns que ocorrem no projeto de software. Eles representam as melhores práticas utilizadas por desenvolvedores ao longo do tempo para resolver desafios específicos durante o processo de design e implementação de software.

Os padrões de projeto oferecem uma abordagem para projetar software que facilita a comunicação eficaz entre desenvolvedores e ajuda a criar sistemas mais flexíveis, escaláveis e fáceis de manter. Eles não são frameworks ou códigos prontos, mas sim modelos que podem ser adaptados para resolver problemas específicos.

Existem vários tipos de padrões de projeto, e eles são geralmente categorizados em três grupos principais:

**1. Padrões de Criação:**

Esses padrões lidam com o processo de criação de objetos. Eles ajudam a criar objetos de uma maneira que seja adequada à situação, promovendo flexibilidade na criação de instâncias de classes.

**Exemplos:** Singleton, Factory Method, Abstract Factory, Builder, Prototype.

**2. Padrões Estruturais:**

Esses padrões se concentram na composição de classes e objetos para formar estruturas maiores. Eles ajudam a garantir que os sistemas sejam flexíveis o suficiente para mudanças na estrutura sem reescrever grandes partes do código.

**Exemplos:** Adapter, Bridge, Composite, Decorator, Facade, Proxy.

**3. Padrões Comportamentais:**

Estes padrões lidam com a interação entre objetos, como a distribuição de responsabilidades e a comunicação entre eles. Eles ajudam a definir como os objetos colaboram e se comunicam entre si.

**Exemplos:** Chain of Responsibility, Command, Interpreter, Iterator, Mediator, Memento, Observer, State, Strategy, Template Method, Visitor.

Ao usar padrões de projeto, os desenvolvedores podem se beneficiar de várias maneiras, incluindo:

**Reusabilidade:** Padrões de projeto promovem a reutilização de soluções eficazes para problemas recorrentes.

**Manutenção:** Eles tornam o código mais fácil de entender e manter, já que seguem princípios de design sólidos.

**Escalabilidade:** Ao aplicar padrões, os sistemas são projetados para serem mais flexíveis e escaláveis.

**3. Teste dos códigos**

**3.1 Classe Criação: Factory Method**

O código exemplifica o padrão de projeto Factory Method:

**Channel (Canal):**

* Classe abstrata que define a interface para os diferentes tipos de canais.

**TCPChannel e UDPChannel (Implementações Concretas de Canal):**

* Classes concretas que implementam a interface **Channel**, representando diferentes tipos de canais (TCP e UDP).

**ChannelFactory (Fábrica de Canais):**

* Classe que implementa o método fábrica estático **create**.
* O método **create** é responsável por criar e retornar uma instância de um canal. Atualmente, está configurado para criar instâncias de **TCPChannel**.

**Main:**

* Métodos **f**, **g** e **h** que demonstram a criação de instâncias de canal usando a fábrica.
* O método estático **main** cria uma instância de **Main** e chama os métodos **f**, **g** e **h** para mostrar a criação de instâncias de canal.

**Execução:**

* O método **main** cria uma instância de **Main** e chama os métodos **f**, **g** e **h**.
* Cada método chama a fábrica para criar uma instância de canal, mas como a fábrica está configurada para criar instâncias de **TCPChannel**, todas as instâncias criadas serão do tipo **TCPChannel**.

**Saída Esperada:**

* O programa imprime a mensagem "Neste momento, estamos trabalhando com TCPChannel" três vezes, indicando que instâncias de **TCPChannel** foram criadas em cada chamada do método **create**.

**Considerações:**

* O padrão Factory Method permite a criação de objetos de uma família de classes, deixando a decisão sobre qual classe concreta instanciar para as subclasses, alterando a criação de objetos sem modificar o código cliente.
* Neste exemplo, a classe **ChannelFactory** age como uma fábrica para criar instâncias de canais, mas pode ser facilmente adaptada para criar instâncias de outros tipos de canais, como **UDPChannel**, conforme necessárias.

**Classe Nomeada:**

* **Sistema de Comunicação:**
* **Descrição:** Representa um sistema de comunicação que utiliza diferentes tipos de canais para troca de dados. O sistema é flexível, permitindo a mudança do tipo de canal conforme necessário.

**3.2 Objeto Criação Builder**

**Classe Livro:**

* Representa um livro com atributos como nome, autores, editora e ano.
* Possui um construtor que aceita esses atributos como parâmetros.
* Contém um método **\_\_str\_\_** para representar o livro como uma string formatada.

**Classe Livro.Builder:**

* Classe interna, pública e estática de **Livro**.
* Implementa o padrão Builder, fornecendo uma interface fluida para a construção de objetos **Livro**.
* Métodos como **set\_nome**, **set\_autores**, **set\_editora** e **set\_ano** permitem configurar os atributos do livro.
* O método **build** cria e retorna uma instância de **Livro** com os atributos configurados.

**Classe Main:**

* Método estático **main** que demonstra a criação de instâncias de **Livro** usando o padrão Builder.
* Cria dois livros (**esm** e **gof**) com diferentes atributos usando o **Livro.Builder**.

**Execução:**

* O método **main** cria dois livros (**esm** e **gof**) usando o padrão Builder.
* Configura os atributos dos livros usando os métodos do **Livro.Builder**.
* Imprime os livros criados.

**Saída Esperada:**

* A saída esperada mostra os detalhes formatados dos livros criados.

**Considerações:**

* O padrão Builder é utilizado para facilitar a criação de objetos complexos, fornecendo uma interface mais flexível e fluente.
* A classe **Livro.Builder** permite a criação de objetos **Livro** passo a passo, configurando os atributos desejados antes de criar a instância final do livro.
* Isso torna o código mais legível e permite uma construção mais expressiva de objetos **Livro** com diferentes conjuntos de atributos.

**Objeto Nomeado**

**Livro de Engenharia de Software Moderna:**

**Atributos:**

**Nome:** Engenharia de Software Moderna

**Editora:** UFMG

**Ano:** 2020

Cada instância do objeto **Livro** é criada usando o padrão Builder, que permite configurar os atributos do livro de maneira fluente e expressiva, resultando em uma instância final de **Livro** com os detalhes desejados.

**3.3 Estrutura Class adapter**

**Classes:**

* **ProjetorSamsung**: Representa um projetor da Samsung.
* **ProjetorLG**: Representa um projetor da LG.
* **Projetor**: Interface abstrata para abstrair o tipo de projetor.
* **AdaptadorProjetorSamsung**: Adaptador para transformar **ProjetorSamsung** em **Projetor**.
* **AdaptadorProjetorLG**: Adaptador para transformar **ProjetorLG** em **Projetor**.
* **SistemaControleProjetores**: Classe que controla os projetores.
* **Main**: Classe principal que contém o método **main** para testar o sistema.

**Estrutura:**

* O sistema usa o padrão de projeto Adaptador para permitir que diferentes tipos de projetores (Samsung e LG) sejam tratados uniformemente através da interface **Projetor**.
* **SistemaControleProjetores** é responsável por controlar os projetores, e utiliza a interface **Projetor** para isso.
* **Main** é a classe principal que cria instâncias de adaptadores e do sistema de controle para testar a integração.

O código é uma implementação do padrão de projeto Adaptador, que permite a integração de diferentes classes de projetores em um sistema uniforme.

**3.4 Estrutura Facade Objeto**

**Objeto:**

* **Luke Skywalker:**
* **Altura:** 172 (cm)
* **C-3PO:**
* **Cor da Pele:** Dourado

**Funções para Obter Dados:**

* **getLukeSkywalkerHeight**: Obtem a altura de Luke Skywalker da API Star Wars.
* **getC3POSkinColor**: Obtém a cor da pele de C-3PO da API Star Wars.

**Métodos:**

* **\_\_init\_\_**: Inicializa a fachada com o endpoint da API.
* **\_make\_request**: Método privado para fazer uma requisição HTTP e retornar os dados brutos.
* **\_parse\_return**: Método privado para de serializar os dados brutos em um formato Python (um dicionário, neste caso).
* **get\_info**: Método público que utiliza os métodos privados para obter informações específicas sobre um personagem.

**Classe Main:**

* **main**: Método estático que exemplifica o uso da fachada.
* Cria uma instância da **StarWarsFacade** e utiliza-a para obter informações específicas sobre personagens da API Star Wars.

**Execução:**

* O método **main** cria uma instância da **StarWarsFacade** e utiliza-a para obter informações específicas sobre personagens da API Star Wars.
* **luke\_height** recebe a altura de Luke Skywalker.
* **c3p0\_skin** recebe a cor da pele de C-3PO.
* As informações são então impressas na tela.

**Considerações:**

* A fachada (**StarWarsFacade**) encapsula a complexidade de interação com a API Star Wars, proporcionando uma interface mais simples e coesa para o usuário da fachada.
* As funções individuais (**getLukeSkywalkerHeight** e **getC3POSkinColor**) representam as operações específicas que são ocultadas pela fachada.
* O uso do padrão de fachada é apropriado quando se deseja simplificar a interação com sistemas ou APIs complexas, isolando detalhes de implementação.

**3.5 Comportamento Classe: Template method**

**Classe Abstrata Funcionario (Template Method):**

* **calcSalarioLiquido**: Método que serve como o template method. Ele define a estrutura geral para o cálculo do salário líquido, chamando métodos abstratos para calcular descontos específicos.
* **calcDescontosPrevidencia**, **calcDescontosPlanoSaude** e **calcOutrosDescontos**: Métodos abstratos que representam etapas específicas no cálculo do salário líquido.

**Classe Concreta FuncionarioCLT (Implementação do Template Method):**

* **calcDescontosPrevidencia**, **calcDescontosPlanoSaude** e **calcOutrosDescontos**: Métodos concretos que implementam as etapas específicas do cálculo para um funcionário com regime CLT.

**Classe Main:**

* **main**: Método estático que exemplifica o uso do padrão. Cria uma instância de **FuncionarioCLT** e calcula o salário líquido usando o método **calcSalarioLiquido**.

**Execução:**

* O método **main** cria uma instância de **FuncionarioCLT** com um salário de 1000.0.
* Em seguida, chama **calcSalarioLiquido** para obter o salário líquido, que envolve a execução do template method.

**Saída:**

* Imprime o salário líquido calculado.

O padrão Template Method é utilizado para definir a estrutura de um algoritmo, deixando algumas etapas específicas para subclasses. Neste exemplo, a estrutura do cálculo do salário líquido é definida na classe abstrata **Funcionario**, e os detalhes específicos para um funcionário com regime CLT são implementados na classe concreta **FuncionarioCLT**.

**3.6 Comportamento Objeto: Strategy**

**Listagem de Funcionários:**

**Lista 1:**

* **Elementos:** [3, 5, 2, 4, 1, 6]
* **Estratégia de Ordenação Padrão:** Bubble Sort

**Lista 2:**

* **Elementos:** [6, 5, 4, 3, 2, 1]
* **Estratégia de Ordenação Alterada:** Selection Sort

**Estrutura:**

**Classe Abstrata SortStrategy:**

* Define uma interface para algoritmos de ordenação.

**Classes Concretas BubbleSortStrategy e SelectionSortStrategy:**

* Implementam estratégias específicas de ordenação.

**Classe MyList:**

* Mantém uma lista de elementos e uma estratégia de ordenação.
* Permite trocar dinamicamente a estratégia de ordenação.
* Utiliza a estratégia para ordenar os elementos e imprimir a lista.

**Classe Main:**

* Demonstração do uso da estratégia de ordenação.
* Cria duas instâncias de **MyList** com diferentes listas e estratégias.
* Mostra como trocar a estratégia de ordenação dinamicamente.

**Execução:**

* Ordena as listas usando diferentes estratégias de ordenação.
* Demonstração da flexibilidade do padrão Strategy na troca dinâmica de algoritmos.

**Objetivos:**

* A **MyList** encapsula a lógica de ordenação e permite que essa lógica seja facilmente substituída por outra.
* A mudança da estratégia de ordenação é transparente para o cliente, pois a interface da **SortStrategy** é mantida consistente.
* O padrão Strategy ajuda a promover a flexibilidade e a manutenção do código, permitindo a variação de algoritmos sem modificar diretamente a classe cliente.

**4. Conclusão**

Os padrões de projeto e comportamentais em POO oferecem soluções eficazes para desafios comuns no desenvolvimento de software. A escolha adequada desses padrões pode resultar em código mais flexível, modular e fácil de entender. Cada padrão exemplificado neste relatório contribui para a melhoria da estrutura, manutenção e extensibilidade de sistemas orientados a objetos.

**5. Referências**