**Curso: Engenharia Software/ADS UC: Modelos, Métodos e Técnicas Engenharia de**

**Software**

**Período: Turma: N Semestre: 2 Ano letivo: 2023-2 Turno: Noite**

**Professor(a): Rubem Koide**

**Aluno(a): Fábio Carvalho de Almeida RA: 172321018**

**Roger Oliveira RA: 172320937**

**Bernardo Gnoatto RA: 172320338**

[**https://github.com/Bnho04/Trabalho**](https://github.com/Bnho04/Trabalho)

**ATIVIDADE 4 – Visao\_Analise\_Projetos\_Design\_e\_Padrões\_2023\_2**

**Nome:**

Factory Method

**Propósito:**

O Factory Method é um padrão de projeto criacional que fornece uma interface para criar objetos em uma superclasse, permitindo que as subclasses alterem o tipo de objetos que serão criados.

**Descrição breve do problema e solução:**

* Problema: Deseja-se criar objetos de um tipo específico, mas não se quer acoplar o código diretamente a esses objetos concretos.
* Solução: O padrão Factory Method resolve esse problema definindo um método (fábrica) na superclasse para criar objetos, enquanto delega a implementação desse método para as subclasses.

**Problema:**

O Factory Method soluciona o problema de criar objetos sem acoplar diretamente o código a esses objetos específicos. Ele é útil quando a lógica de criação de objetos precisa ser flexível e deve ser determinada em tempo de execução, permitindo que diferentes variantes de objetos sejam criadas de maneira controlada.

**Solução:**

O método de fábrica é implementado na classe base (superclasse) e as subclasses decidem qual objeto concreto será criado. Por exemplo, em um sistema de logística, uma classe **Transport** pode ser a superclasse com um método **deliver()** como o Factory Method. As subclasses como **Truck**, **Ship** e **Plane** podem implementar esse método para criar instâncias específicas de entrega conforme o tipo de transporte.

**Estrutura:**

Classes envolvidas:

* Classe Base (**Creator** ou **Product**)
* Classe Concreta (**ConcreteCreator** ou **ConcreteProduct**)
* Método Factory Method (**factoryMethod()**)

**Aplicabilidade:**

O padrão Factory Method é aplicável em situações onde:

* Quer-se delegar a responsabilidade de criação de objetos para subclasses.
* Deseja-se desacoplar o código da classe cliente dos objetos específicos a serem criados.
* Há a necessidade de suportar múltiplas variantes de objetos a serem criados.

**Prós e Contras:**

Prós:

* Flexibilidade na criação de objetos.
* Desacoplamento do código cliente dos objetos concretos.
* Suporte para extensibilidade do código.

Contras:

* Aumenta a complexidade ao introduzir mais classes no código.
* Pode ser excessivo para problemas simples, gerando mais complexidade do que o necessário.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Nome:**

Builder

**Propósito:**

O padrão Builder é um padrão de projeto criacional que separa a construção de um objeto complexo de sua representação, permitindo a criação passo a passo desse objeto. Ele é útil quando a criação de um objeto requer etapas complexas ou quando se deseja isolar o processo de construção do objeto da sua estrutura.

**Descrição breve do problema e solução:**

* Problema: Criar objetos complexos com muitos parâmetros opcionais, sem poluir o construtor com muitos argumentos opcionais.
* Solução: O padrão Builder resolve esse problema permitindo a construção passo a passo de um objeto complexo. Ele usa um construtor com parâmetros obrigatórios e um Builder para definir os parâmetros opcionais, fornecendo uma interface limpa e fluente para criar objetos.

**Problema:**

O padrão Builder soluciona o problema de construir objetos complexos com muitos parâmetros opcionais, fornecendo uma maneira de criar objetos passo a passo sem a necessidade de um construtor com muitos argumentos ou combinações de construtores.

**Solução:**

O Builder separa a criação de um objeto complexo da sua representação, fornecendo um Builder com métodos para configurar os parâmetros opcionais do objeto. No exemplo fornecido, a classe **Livro** possui um Builder interno que permite definir o nome, autores, editora e ano do livro de forma fluente, facilitando a criação de objetos **Livro**.

**Estrutura:**

Classes envolvidas:

* Classe Produto (**Livro**)
* Classe Builder (**Livro.Builder**)

**Aplicabilidade:**

O padrão Builder é aplicável em cenários onde:

* A construção de objetos é complexa e requer muitos passos.
* O objeto a ser construído possui muitos parâmetros opcionais.
* É necessário criar diferentes representações de um mesmo objeto.

**Prós e Contras:**

Prós:

* Separa a construção do objeto da sua representação.
* Facilita a criação de objetos com muitos parâmetros opcionais.
* Melhora a legibilidade do código.

Contras:

* Aumenta a complexidade do código ao introduzir uma classe adicional (Builder).
* Pode ser excessivo para objetos simples, gerando mais código do que o necessário.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Nome:**

Proxy

**Propósito:**

O padrão Proxy é um padrão de projeto estrutural que fornece um substituto ou placeholder para outro objeto. Ele controla o acesso ao objeto original, permitindo adicionar funcionalidades extras sem alterar a estrutura do objeto original.

**Descrição breve do problema e solução:**

* Problema: Controlar o acesso a um objeto ou adicionar funcionalidades extras a esse acesso sem modificar o objeto original.
* Solução: O padrão Proxy resolve esse problema fornecendo um objeto que atua como intermediário entre o cliente e o objeto original, controlando o acesso a esse objeto e permitindo a execução de operações adicionais.

**Problema:**

O padrão Proxy é útil quando se deseja controlar o acesso a um objeto ou adicionar funcionalidades extras, como validações, cache, ou lógica adicional, sem modificar a estrutura do objeto original.

**Solução:**

O Proxy mantém uma referência ao objeto original e implementa a mesma interface desse objeto. Ele intercepta as chamadas feitas ao objeto original e pode executar operações adicionais antes ou depois de repassar a chamada ao objeto real. No exemplo dado, **BookSearch** é o objeto base e **BookSearchProxy** é o Proxy que controla o acesso a esse objeto, permitindo ou não que o cliente acesse o objeto original e adicionando um comportamento adicional.

**Estrutura:**

Classes envolvidas:

* Classe de Interface (**BookSearchInterface**)
* Classe Real (**BookSearch**)
* Classe Proxy (**BookSearchProxy**)
* Classe Cliente (**Main**)

**Aplicabilidade:**

O padrão Proxy é aplicável em situações onde:

* É necessário controlar o acesso a um objeto.
* Deseja-se adicionar funcionalidades extras ao acesso desse objeto sem alterar sua estrutura.
* É necessário gerenciar a criação, destruição e acesso a objetos de maneira transparente para o cliente.

**Prós e Contras:**

Prós:

* Controle de acesso a objetos.
* Adição de funcionalidades extras sem alterar o objeto original.
* Separação de responsabilidades entre o objeto real e o Proxy.

Contras:

* Pode introduzir complexidade adicional no código.
* Aumento do número de classes no sistema.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Nome:**

Adapter

**Propósito:**

O padrão Adapter é um padrão estrutural que permite a interface entre objetos com interfaces incompatíveis, tornando possível a colaboração entre eles.

Descrição breve do problema e solução:

* Problema: Incompatibilidade de interfaces entre objetos existentes.
* Solução: O padrão Adapter resolve esse problema, criando uma interface intermediária entre objetos incompatíveis para permitir sua interação sem modificar sua estrutura.

**Problema:**

O Adapter soluciona o problema de permitir a colaboração entre objetos que têm interfaces incompatíveis. Ele é útil quando se deseja usar uma classe existente, mas sua interface não corresponde à interface desejada.

**Solução:**

O Adapter cria um intermediário entre duas interfaces incompatíveis, permitindo que os objetos que de outra forma não poderiam trabalhar juntos interajam. No exemplo dado, os projetores Samsung e LG têm métodos diferentes para ligar, mas o SistemaControleProjetores espera objetos que implementem a interface Projetor. Os adaptadores (AdaptadorProjetorSamsung e AdaptadorProjetorLG) agem como intermediários, convertendo os métodos dos projetores específicos para a interface Projetor.

**Estrutura:**

Classes envolvidas:

* Interfaces (Projetor)
* Classes Concretas (ProjetorSamsung, ProjetorLG)
* Adaptadores (AdaptadorProjetorSamsung, AdaptadorProjetorLG)
* Cliente (SistemaControleProjetores)

**Aplicabilidade:**

O padrão Adapter é aplicável em situações onde:

* Há a necessidade de interagir com classes existentes que têm interfaces incompatíveis.
* Deseja-se criar uma classe reutilizável para colaborar com classes não relacionadas ou imutáveis.
* É necessário fornecer uma interface unificada para diferentes classes.

**Prós e Contras:**

Prós:

* Permite a colaboração entre objetos com interfaces incompatíveis.
* Ajuda na reutilização de classes existentes sem alterar seu código.
* Fornece flexibilidade ao adaptar diferentes interfaces para uma única interface esperada.

Contras:

* Pode resultar em muitas classes de adaptadores se houver muitos objetos incompatíveis.
* Pode causar complexidade adicional no código devido à introdução de adaptadores intermediários**.**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Nome:**

Iterator

**Propósito:**

O padrão Iterator é um padrão comportamental que fornece uma maneira de acessar sequencialmente os elementos de um objeto sem expor a representação interna desse objeto.

**Descrição breve do problema e solução:**

* Problema: Acesso a elementos de uma coleção sem expor sua implementação interna.
* Solução: O padrão Iterator resolve esse problema fornecendo um objeto que encapsula o acesso sequencial aos elementos de uma coleção sem expor sua estrutura interna.

**Problema:**

O padrão Iterator soluciona o problema de percorrer os elementos de uma coleção sem expor a estrutura interna da coleção. Ele é útil quando se deseja acessar os elementos de uma coleção sem saber sua implementação específica, permitindo varrer os elementos sem depender de sua estrutura subjacente.

**Solução:**

O Iterator permite que os clientes percorram os elementos de uma coleção sem expor detalhes da implementação dessa coleção. Ele fornece um iterador que possui métodos como **\_\_next\_\_()** para acessar sequencialmente os elementos da coleção. No exemplo dado, **IteradorListaPalavras** é um iterador concreto que percorre a coleção de palavras definida em **ListaPalavras**, permitindo iteração normal ou reversa.

**Estrutura:**

Classes envolvidas:

* Classe Iterable (**Iterable**)
* Classe Iterator (**Iterator**)
* Classe Concreta (**IteradorListaPalavras**)
* Classe de Coleção (**ListaPalavras**)

**Aplicabilidade:**

O padrão Iterator é aplicável em situações onde:

* Há a necessidade de percorrer os elementos de uma coleção sem expor sua implementação interna.
* Deseja-se fornecer um método de iteração consistente para diferentes tipos de coleções.
* É necessário permitir múltiplos acessos aos elementos da coleção sem depender da sua implementação.

**Prós e Contras:**

Prós:

* Separa a lógica de iteração da coleção, facilitando a adição de novos tipos de iteração.
* Fornece um método consistente para percorrer elementos em diferentes coleções.
* Encapsula a implementação da coleção, evitando exposição desnecessária.

Contras:

* Pode introduzir complexidade adicional dependendo da implementação do iterador.
* A criação de iteradores personalizados pode ser trabalhosa.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Nome:**

Template Method

**Propósito:**

O padrão Template Method é um padrão comportamental que define o esqueleto de um algoritmo em uma operação, delegando alguns de seus passos para subclasses. Ele permite que as subclasses redefinam certos passos de um algoritmo sem alterar sua estrutura.

**Descrição breve do problema e solução:**

* Problema: Necessidade de definir um algoritmo com etapas fixas, mas com algumas variações em certos passos.
* Solução: O Template Method resolve esse problema definindo uma estrutura fixa (template) para o algoritmo e permitindo que as subclasses substituam ou implementem certos passos conforme necessário.

**Problema:**

O padrão Template Method resolve o problema de definir uma estrutura fixa para um algoritmo, mas com flexibilidade para que certos passos desse algoritmo possam variar entre as subclasses.

**Solução:**

O Template Method implementa um esqueleto de algoritmo na classe base, com métodos definidos (métodos concretos ou abstratos) representando os passos do algoritmo. As subclasses então fornecem implementações para esses métodos específicos, permitindo variações no algoritmo sem alterar sua estrutura geral. No exemplo dado, a classe Funcionario define o Template Method calcSalarioLiquido(), enquanto as subclasses como FuncionarioCLT fornecem implementações para os métodos abstratos calcDescontosPrevidencia(), calcDescontosPlanoSaude(), e calcOutrosDescontos().

**Estrutura:**

Classes envolvidas:

* Classe abstrata (Funcionario)
* Métodos concretos ou abstratos representando etapas do algoritmo
* Subclasses (FuncionarioCLT) que implementam métodos específicos do algoritmo

**Aplicabilidade:**

O padrão Template Method é aplicável em situações onde:

* Há um algoritmo com etapas fixas, mas algumas partes variáveis.
* Deseja-se definir um esqueleto de algoritmo que possa ser reutilizado.
* É necessário evitar a duplicação de código entre classes relacionadas.

**Prós e Contras:**

Prós:

* Promove a reutilização de código ao definir uma estrutura de algoritmo fixa.
* Permite extensão fácil do comportamento através de subclasses.
* Evita a duplicação de código.

Contras:

* Pode introduzir complexidade se muitas subclasses diferentes implementarem muitos métodos variáveis.
* Pode ser difícil de entender se os métodos concretos forem muito diferentes entre subclasses.