Nome: Eduardo Plotchkacz Drozdz - 172311203; André Bordignon de Souza – 172320466

Nome: Método Fábrica Estático (Static Factory Method)

Propósito: O padrão de Método Fábrica Estático é utilizado para centralizar a criação de objetos, fornecendo um método estático em uma classe para criar instâncias de uma interface ou classe abstrata.

Problema: Quando a criação de objetos é uma operação complexa ou precisa ser controlada de maneira centralizada, mas não é necessário instanciar diretamente a classe concreta.

Solução: Introduzir um método estático em uma classe dedicada (frequentemente chamada de "fábrica") para criar instâncias de uma interface ou classe abstrata. Esse método encapsula a lógica de criação, fornecendo uma maneira mais flexível de criar objetos.

Estrutura:

- **Interface Channel:** Define a interface que as classes concretas (TCPChannel e UDPChannel) implementarão.
- **TCPChannel e UDPChannel:** Implementam a interface Channel, representando diferentes tipos de canais.
- **ChannelFactory:** Classe com um método estático (**create**) que cria instâncias da interface Channel. Neste caso, a implementação atual sempre retorna um TCPChannel.
- **Main:** Demonstração da utilização do padrão, onde objetos Channel são criados através do método estático da ChannelFactory.

Aplicabilidade: O padrão de Método Fábrica Estático é aplicável em situações em que a criação de objetos precisa ser centralizada, pode envolver lógica complexa, e quando a escolha do tipo de objeto a ser criado pode mudar dinamicamente.

Prós e Contras:

Prós:

- Encapsula a lógica de criação, proporcionando flexibilidade.
- Evita a exposição direta das classes concretas ao cliente.

Contras:

- Pode tornar-se complexo se a lógica de criação for extensa.
- Menos flexível do que outros padrões de fábrica em termos de extensibilidade.

Custo e Benefício:

Benefícios:

- Centraliza a lógica de criação, facilitando a manutenção.
- Oferece um ponto único de controle para a criação de objetos.

Custo:

- Pode introduzir complexidade desnecessária para casos simples.
- Menos flexível em comparação com algumas alternativas, como o padrão Fábrica Abstrata.

Nome: Builder Pattern

Propósito: O padrão Builder é utilizado para construir um objeto complexo passo a passo, permitindo diferentes configurações de atributos sem a necessidade de vários construtores ou parâmetros opcionais.

Problema: Quando um objeto possui muitos atributos opcionais e criar múltiplos construtores se torna inviável, ou quando a ordem de configuração dos atributos não é crítica.

Solução: Introduzir uma classe interna (Builder) que possui métodos para configurar cada atributo do objeto e, finalmente, um método **build** que cria e retorna a instância do objeto complexo.

Estrutura:

- **Livro:** Classe principal que representa o objeto complexo a ser construído.
- **Livro.Builder:** Classe interna que fornece métodos para configurar cada atributo do Livro e retorna a instância do Livro no método **build**.

Aplicabilidade: O padrão Builder é aplicável quando um objeto tem muitos atributos opcionais, a ordem de configuração não é importante e se deseja fornecer uma interface clara para a construção de objetos complexos.

Prós e Contras:

Prós:

- Flexibilidade na criação de objetos com muitos atributos opcionais.
- Melhora a legibilidade do código quando há muitas opções de configuração.

Contras:

- Introduz uma classe adicional (o Builder) no código.

Custo e Benefício:

Benefícios:

- Facilita a criação de objetos complexos e evita a proliferação de construtores.
- Melhora a manutenção e legibilidade do código.

Custo:

- Introduz uma classe adicional, o que pode ser considerado excessivo para objetos simples.

Nome: Adapter Pattern

Propósito: O padrão Adapter é utilizado para permitir que interfaces incompatíveis possam interagir entre si. Ele atua como uma ponte entre duas interfaces diferentes, permitindo que objetos com interfaces incompatíveis trabalhem juntos.

Problema: Quando uma classe ou interface existente não é compatível com a interface requerida, e o cliente não pode ser modificado para se adaptar a essa interface.

Solução: Introduzir uma classe adaptadora que implementa a interface desejada e contém uma instância da classe incompatível. A classe adaptadora atua como uma camada intermediária que traduz as chamadas da interface desejada para as chamadas apropriadas na interface existente.

Estrutura:

- **ProjetorSamsung e ProjetorLG:** Classes que representam diferentes projetores com interfaces incompatíveis.
- **Projetor:** Interface desejada que define o método **liga()**.
- **AdaptadorProjetorSamsung e AdaptadorProjetorLG:** Classes adaptadoras que implementam a interface **Projetor** e contêm instâncias dos projetores concretos.
- **SistemaControleProjetores:** Cliente que utiliza a interface **Projetor** sem precisar conhecer as implementações concretas.

Aplicabilidade: O padrão Adapter é aplicável quando você precisa integrar classes ou interfaces existentes com diferentes interfaces e não pode modificar diretamente as classes existentes para torná-las compatíveis.

Prós e Contras:

Prós:

- Permite a integração de classes com interfaces incompatíveis.
- Promove reutilização de código existente.

Contras:

- Adiciona uma camada adicional de indireção.
- Pode levar a um código mais complexo se usado em excesso.

Custo e Benefício:

Benefícios:

- Facilita a integração de componentes existentes.
- Permite a reutilização de código sem modificar as classes existentes.

Custo:

- Introduz uma camada adicional de classes adaptadoras.
- Pode tornar o código mais complexo, especialmente quando há muitos adaptadores.

Nome: Proxy Pattern

Propósito: O padrão Proxy é utilizado para controlar o acesso a um objeto por meio de outro objeto intermediário, que atua como um substituto ou representante do objeto original.

Problema: Quando você precisa controlar o acesso a um objeto, seja para implementar lógica adicional ou para adiar a criação ou a execução do objeto original.

Solução: Introduzir um objeto proxy que implementa a mesma interface que o objeto original e encaminha as chamadas para o objeto original, permitindo a execução de lógica adicional antes ou depois das chamadas.

Estrutura:

- **Book:** Classe que representa um livro com um nome.
- **BookSearchInterface:** Interface que define as operações de pesquisa de livros.
- **BookSearch:** Classe concreta que implementa a interface **BookSearchInterface** e realiza a pesquisa de livros.
- **BookSearchProxy:** Classe proxy que implementa a interface **BookSearchInterface** e controla o acesso à classe **BookSearch**.
- **Main:** Demonstração de uso do padrão Proxy.

Aplicabilidade: O padrão Proxy é aplicável quando você precisa controlar ou adiar o acesso a um objeto, como quando deseja realizar ações adicionais antes ou depois de chamar o objeto real.

Prós e Contras:

Prós:

- Controle de acesso a um objeto.
- Permite adiar a criação ou execução do objeto real.
- Pode ser usado para adicionar lógica adicional, como controle de acesso.

Contras:

- Pode introduzir complexidade adicional.

Custo e Benefício:

Benefícios:

- Controle de acesso e adição de lógica sem modificar o objeto real.
- Pode melhorar o desempenho ao adiar a criação do objeto real.

Custo:

- Introduz uma camada adicional de classes proxy.
- Pode ser excessivo para situações simples.

Nome: Template Method Pattern

Propósito: O padrão Template Method é utilizado para definir o esqueleto de um algoritmo em uma classe base, delegando a implementação de etapas específicas para subclasses. Ele permite que as subclasses redefinam ou estendam partes do algoritmo sem alterar sua estrutura geral.

Problema: Quando você tem um algoritmo que possui uma estrutura fixa, mas com partes que podem variar ou serem personalizadas nas subclasses.

Solução: Definir uma classe base que contém um método template (método modelo) que define a estrutura geral do algoritmo usando chamadas a métodos abstratos. As subclasses então fornecem implementações concretas para esses métodos abstratos, personalizando partes específicas do algoritmo.

Estrutura:

- **Funcionario:** Classe abstrata que contém o método template **calcSalarioLiquido** e chama métodos abstratos **calcDescontosPrevidencia**, **calcDescontosPlanoSaude** e **calcOutrosDescontos**.
- **FuncionarioCLT:** Subclasse que herda de **Funcionario** e fornece implementações concretas para os métodos abstratos chamados pelo método template.

Aplicabilidade: O padrão Template Method é aplicável quando você tem um algoritmo com uma estrutura fixa, mas partes que podem variar nas subclasses. Ele é útil quando você deseja evitar duplicação de código e permite que as subclasses personalizem partes específicas do algoritmo.

Prós e Contras:

Prós:

- Evita duplicação de código, pois a estrutura do algoritmo é definida na classe base.
- Permite extensibilidade e flexibilidade, pois as subclasses podem personalizar partes específicas do algoritmo.

Contras:

- Pode ser inflexível se a estrutura do algoritmo precisar ser alterada frequentemente.

Custo e Benefício:

Benefícios:

- Promove a reutilização de código.
- Facilita a manutenção e evita a duplicação de lógica comum.

Custo:

- Pode ser excessivo se a estrutura do algoritmo mudar frequentemente.

Nome: Observer Pattern

Propósito: O padrão Observer é utilizado para definir uma dependência um-para-muitos entre objetos, de modo que quando um objeto muda de estado, todos os seus dependentes são notificados e atualizados automaticamente.

Problema: Quando um objeto (o "sujeito") precisa notificar outros objetos (os "observadores") sobre mudanças em seu estado, mas sem acoplar fortemente esses objetos.

Solução: Define uma interface (**Observer**) que os observadores implementam e uma classe (**Subject**) que mantém uma lista de observadores. O sujeito notifica os observadores sobre mudanças de estado chamando seus métodos de atualização.

Estrutura:

- **Subject:** Classe que mantém uma lista de observadores e fornece métodos para adicionar, remover e notificar observadores.
- **Observer:** Interface que define o método **update** que os observadores implementam para receber notificações do sujeito.

- **Temperatura:** Subclasse de **Subject** que representa um sujeito específico. Possui um estado (a temperatura) que, quando alterado, notifica os observadores.
- **TermometroCelsius:** Implementação concreta de **Observer** que recebe notificações sobre mudanças na temperatura e imprime a temperatura em Celsius.
- **Aplicabilidade:** O padrão Observer é aplicável quando um objeto (o sujeito) precisa notificar outros objetos (os observadores) sobre mudanças em seu estado, e os observadores precisam reagir a essas mudanças.

Prós e Contras:

- *Prós:*
- Desacopla o sujeito dos seus observadores, permitindo fácil adição/remoção de observadores.
- Permite uma comunicação bidirecional entre sujeito e observadores.
- *Contras:*
- Pode resultar em notificações excessivas se não for gerenciado corretamente.
- **Custo e Benefício:**
- *Benefícios:*
- Desacopla sujeito e observadores, facilitando a manutenção e a extensibilidade.
- Suporta comunicação bidirecional.
- *Custo:*
- Introduz uma camada adicional de abstração.