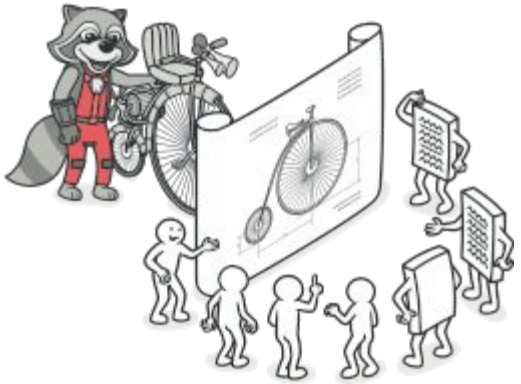
KENZIE ACADEMY

Design Patterns

O que é um Design Pattern?

Design Patterns ou Padrões de projeto são soluções típicas para problemas comuns em projetos de software.

Eles são como plantas de obra pré-fabricadas que você pode customizar para resolver um problema de projeto recorrente em seu código.



História dos padrões

- O conceito de padrões foi primeiramente descrito por Christopher Alexander em Uma Linguagem de Padrões. O livro descreve uma “linguagem” para o projeto de um ambiente urbano. As unidades dessa linguagem são os padrões. Eles podem descrever quão alto as janelas devem estar, quantos andares um prédio deve ter, quão largas as áreas verdes de um bairro devem ser, e assim em diante.
- A ideia foi seguida por quatro autores: Erich Gamma, John Vlissides, Ralph Johnson, e Richard Helm. Em 1994, eles publicaram Padrões de Projeto — Soluções Reutilizáveis de Software Orientado a Objetos, no qual eles aplicaram o conceito de padrões de projeto para programação. O livro mostrava 23 padrões que resolviam vários problemas de projeto orientado a objetos e se tornou um best-seller rapidamente. Devido a seu longo título, as pessoas começaram a chamá-lo simplesmente de “o livro da Gangue dos Quatro (Gang of Four)” que logo foi simplificado para o “livro GoF”

Porque devo aprender Design Patterns?

- Os padrões de projeto são um kit de ferramentas para soluções feitas e testadas para problemas comuns em projeto de software. Mesmo que você nunca os tenha encontrado, saber sobre os padrões é ainda muito útil porque eles ensinam como resolver vários problemas usando princípios de projeto orientado a objetos.
- Eles definem uma linguagem comum que você e seus colegas podem usar para se comunicar mais eficientemente. Você pode dizer, “Ah, não é só usar um Singleton para isso?” e todos vão entender a ideia por trás da sua sugestão. Não é preciso explicar o que um singleton é se você conhece o padrão e seu nome.

Lados negativos dos Design Patterns.

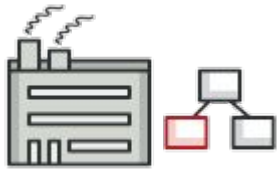
- **Soluções ineficientes:**
  Os padrões tentam sistematizar abordagens que já são amplamente usadas. Essa unificação é vista por muitos como um dogma e eles implementam os padrões “direto ao ponto”, sem adaptá-los ao contexto de seus projetos.

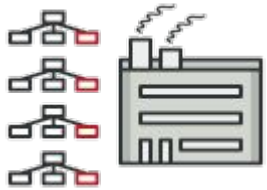
- **Uso injustificável:**
  Esse é o problema que assombra muitos juniores que acabaram de se familiarizar com os padrões. Tendo aprendido sobre eles, tentam aplicá-los em tudo, até mesmo em situações onde um código mais simples seria suficiente.

Tipos de Patterns

- Os **padrões criacionais** fornecem mecanismos de criação de objetos que aumentam a flexibilidade e a reutilização de código.
- Os **padrões estruturais** explicam como montar objetos e classes em estruturas maiores, enquanto ainda mantém as estruturas flexíveis e eficientes.
- Os **padrões comportamentais** cuidam da comunicação eficiente e da assinalação de responsabilidades entre objetos.
- **Padrões idiomáticos** são aqueles que se aplicam a só uma linguagem
- **Padrões arquitetônicos** são os que organizam a arquitetura de um projeto

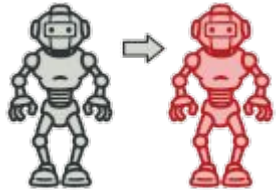
Padrões criacionais

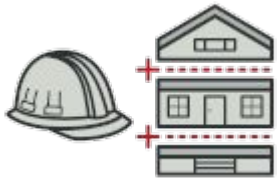


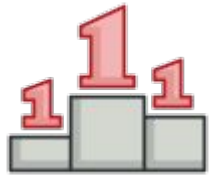


Os padrões criacionais fornecem vários mecanismos de criação de objetos, que aumentam a flexibilidade e reutilização de código já existente.

- Factory method
- Abstract Factory
- Builder
- Prototype
- Singleton



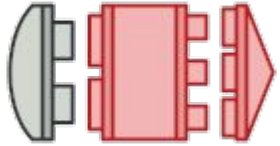


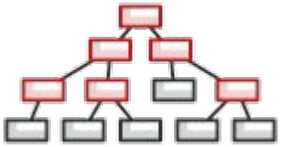


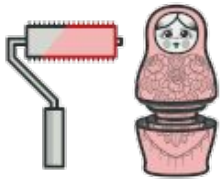
Padrões estruturais

Os padrões estruturais explicam como montar objetos e classes em estruturas maiores, mas ainda mantendo essas estruturas flexíveis e eficientes.

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- ...



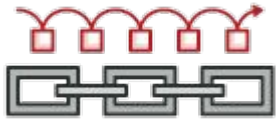


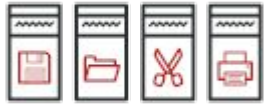


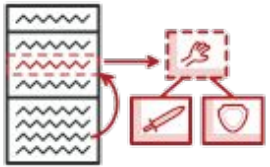
Padrões comportamentais

Padrões comportamentais são voltados aos algoritmos e a designação de responsabilidades entre objetos.

- Chain of responsibility
- Command
- Iterator
- Strategy
- Observer
- ...

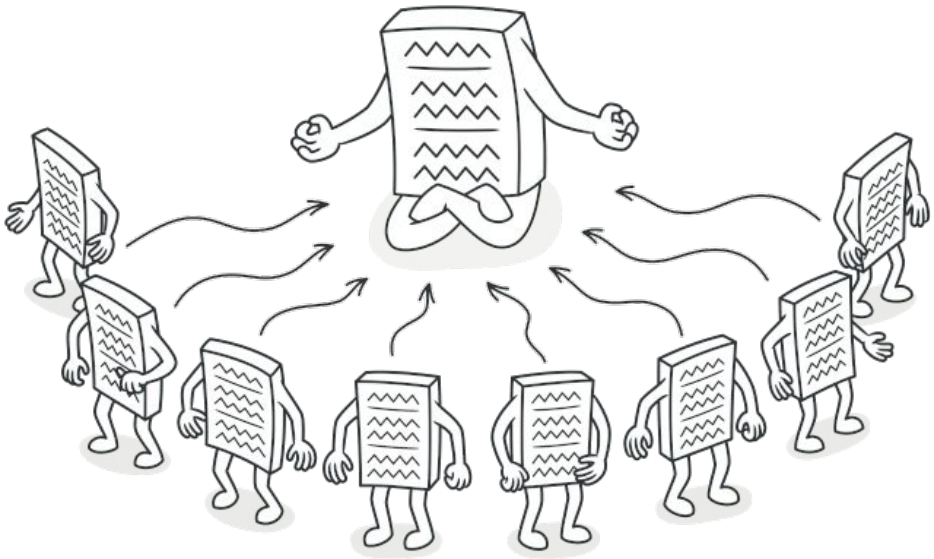






Singleton

O Singleton é um padrão de projeto criacional que permite a você garantir que uma classe tenha apenas uma instância, enquanto provê um ponto de acesso global para essa instância.

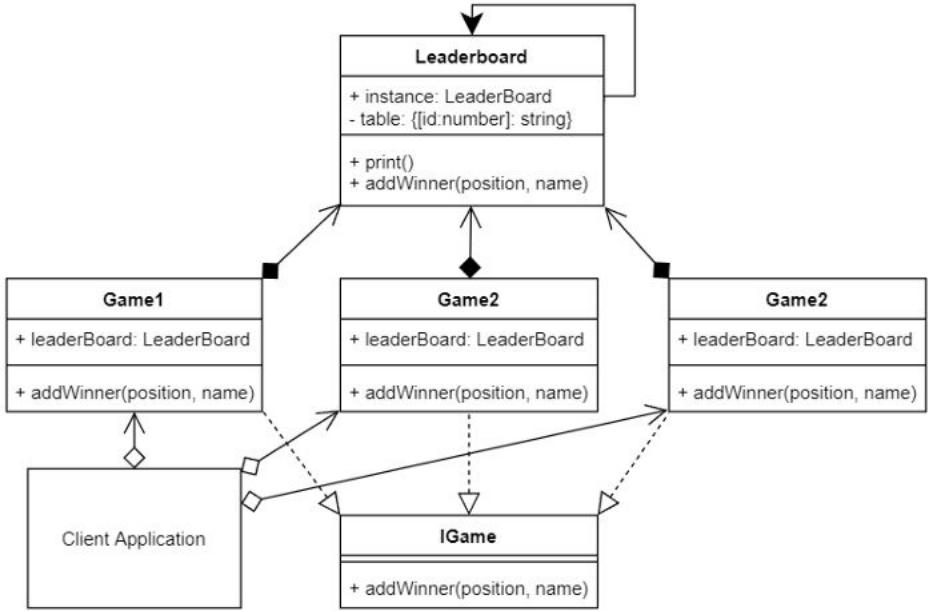


Como utilizar:

- Fazer o construtor padrão privado, para prevenir que outros objetos usem o operador *new* com a classe singleton.
- Criar um método estático de criação que age como um construtor. Esse método chama o construtor privado por debaixo dos panos para criar um objeto e o salva em um campo estático. Todas as chamadas seguintes para esse método retornam o objeto em cache.

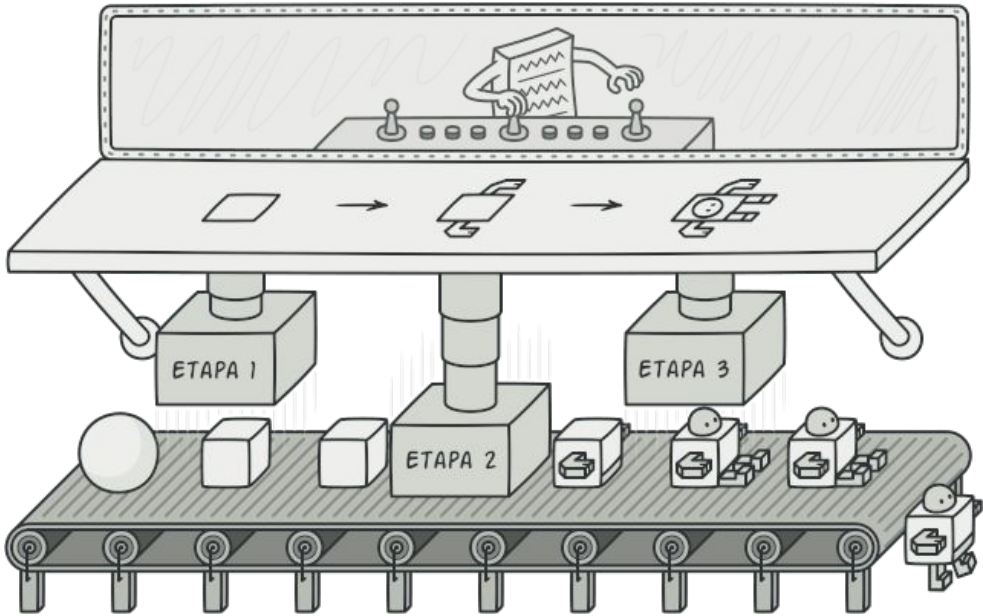
Caso de uso

- No exemplo, há três jogos criados. Todas são instâncias independentes criadas a partir de sua própria classe, mas todas compartilham a mesma tabela de classificação. A tabela de classificação é um Singleton.
- Não importa como os Jogos foram criados, ou como eles fazem referência à tabela de classificação, é sempre um Singleton.
- Cada jogo adiciona independentemente um vencedor, e todos os jogos podem ler a tabela de classificação alterada, independentemente de qual jogo a atualiza.



Builder

O Builder é um padrão de projeto criacional que permite a você construir objetos complexos passo a passo. O padrão permite que você produza diferentes tipos e representações de um objeto usando o mesmo código de construção.

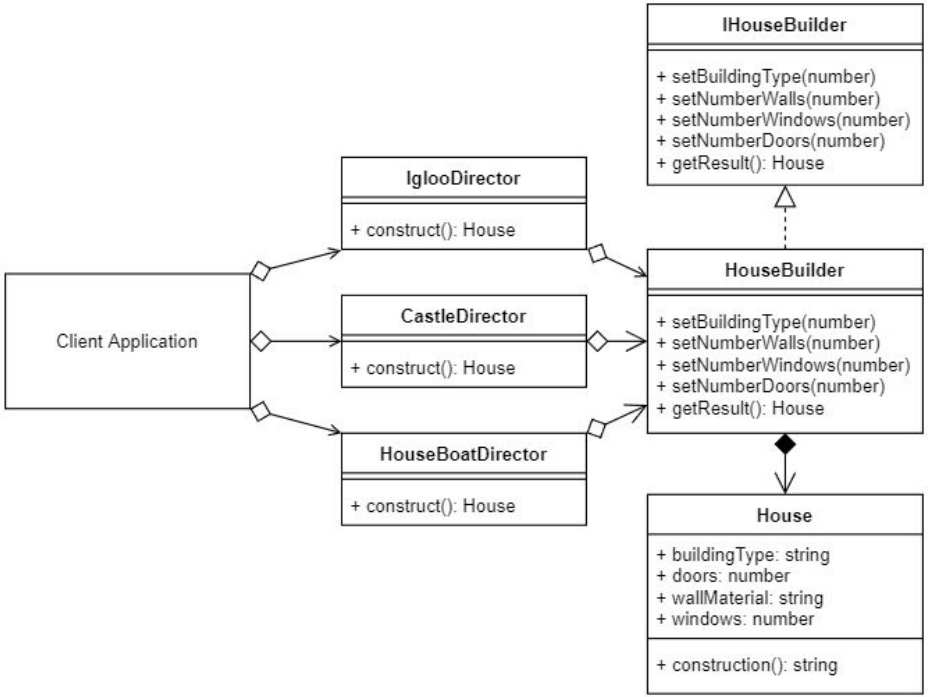


Como utilizar:

- Certifique-se que você pode definir claramente as etapas comuns de construção para construir todas as representações do produto disponíveis.
- Declare essas etapas na interface builder base.
- Crie uma classe builder concreta para cada representação do produto e implemente suas etapas de construção.

Caso de uso

- Usando o Builder Pattern no contexto de um House Builder.
- Existem vários diretores que podem criar seus próprios objetos complexos.
- Observe que na classe IglooDirector, nem todos os métodos do HouseBuilder foram chamados.
- O construtor pode construir objetos complexos em qualquer ordem e incluir/excluir as partes que desejar.

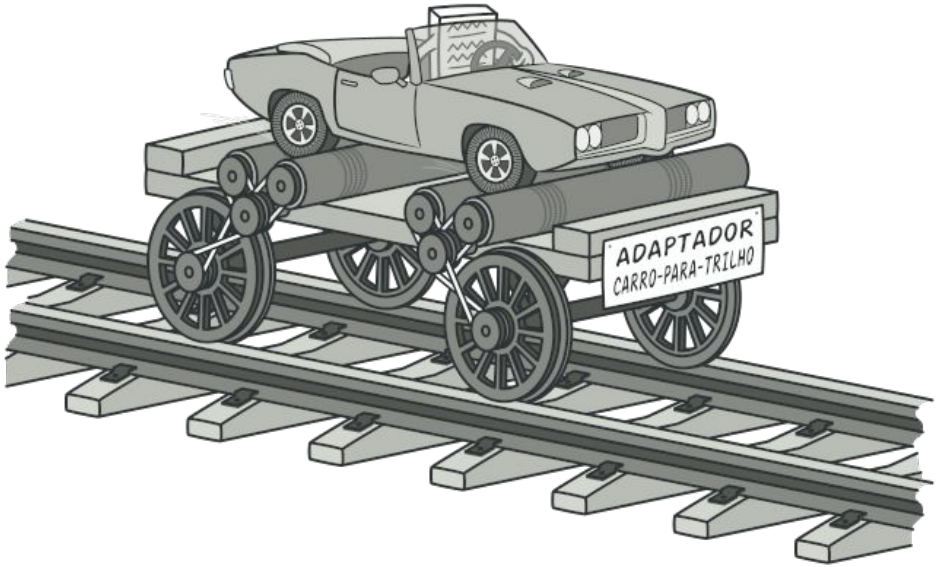


Adapter

O Adapter é um padrão de projeto estrutural que permite objetos com interfaces incompatíveis colaborarem entre si.

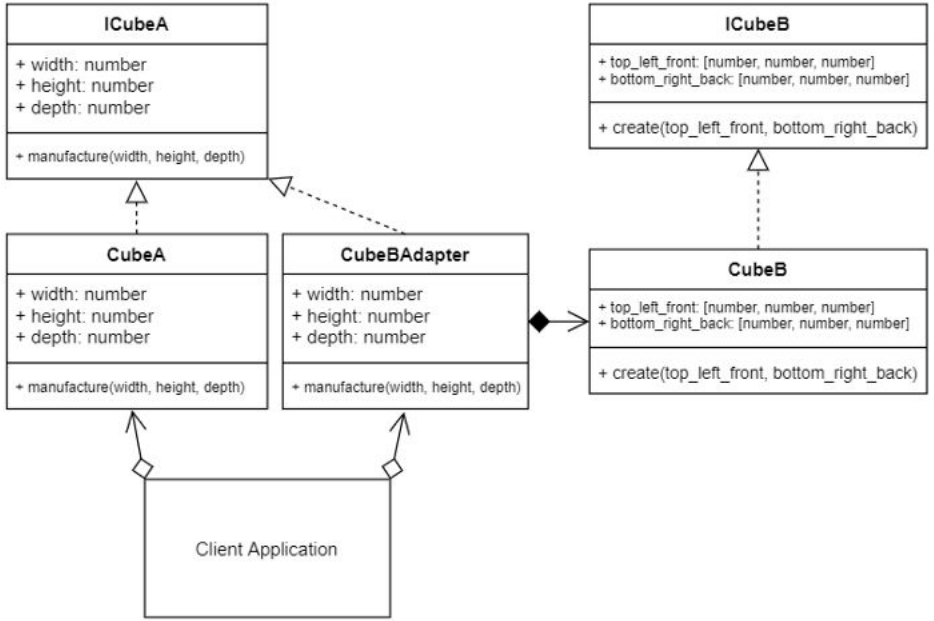
Como utilizar:

- Certifique-se que você tem uma classe serviço útil, que você não pode modificar (quase sempre de terceiros, antiga, ou com muitas dependências existentes).
- Um por um, implemente todos os métodos da interface cliente na classe adaptadora. O adaptador deve delegar a maioria do trabalho real para o objeto serviço, lidando apenas com a conversão da interface ou formato dos dados.



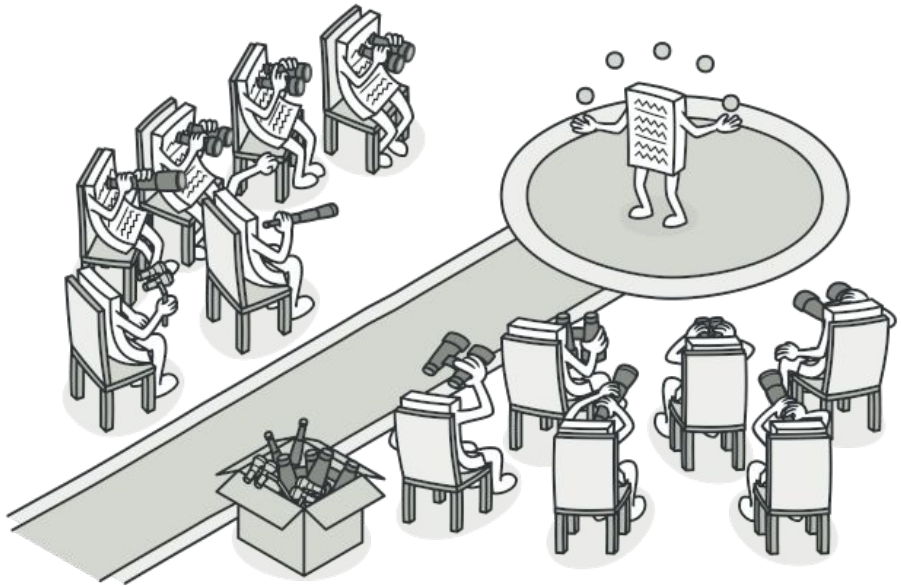
Caso de uso

- Um cliente pode fabricar um Cubo usando diferentes ferramentas. A interface de usuário do cliente gerencia o produto Cubo indicando a largura, altura e profundidade. Isso é compatível com a empresa A que produz a ferramenta Cube, mas não a empresa B que produz sua própria versão da ferramenta Cube que usa uma interface diferente com parâmetros diferentes.
- Neste exemplo, o cliente irá reutilizar a interface para o Cubo da empresa A e criar um Cubo compatível da empresa B.
- Será necessário um adaptador para que a mesma assinatura de método possa ser usada pelo cliente sem a necessidade de solicitar à empresa B que modifique sua ferramenta.
- Minha empresa imaginária precisa usar os dois fornecedores de cubos, pois há uma grande demanda por cubos e, quando um fornecedor está ocupado, posso pedir ao outro fornecedor.



Observer

O Observer/Pus-Sub é um padrão de projeto comportamental que permite que você defina um mecanismo de assinatura para notificar múltiplos objetos sobre quaisquer eventos que aconteçam com o objeto que eles estão observando.

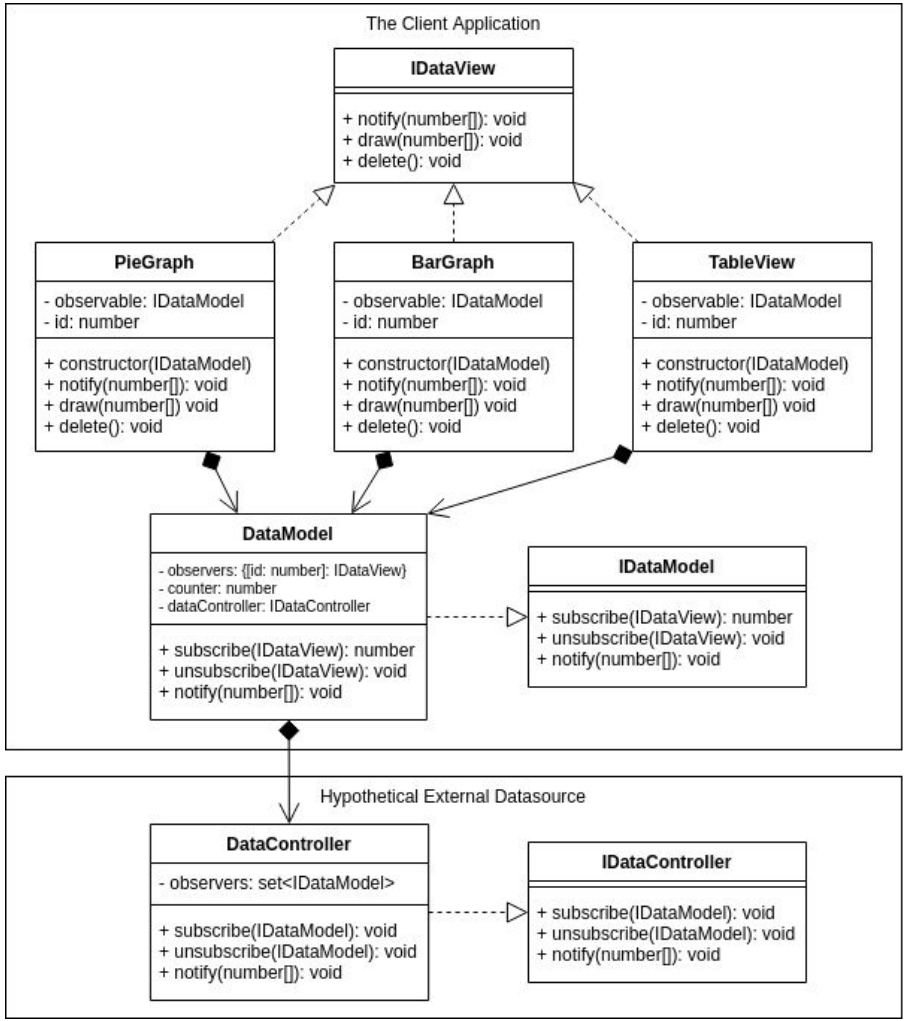


Aplicabilidade:

- Utilize o padrão Observer quando mudanças no estado de um objeto podem precisar mudar outros objetos, e o atual conjunto de objetos é desconhecido de antemão ou muda dinamicamente.
- O padrão Observer permite que qualquer objeto que implemente a interface do assinante possa se inscrever para notificações de eventos em objetos da publicadora. Você pode adicionar o mecanismo de inscrição em seus botões, permitindo que o cliente coloque seu próprio código através de classes assinantes customizadas

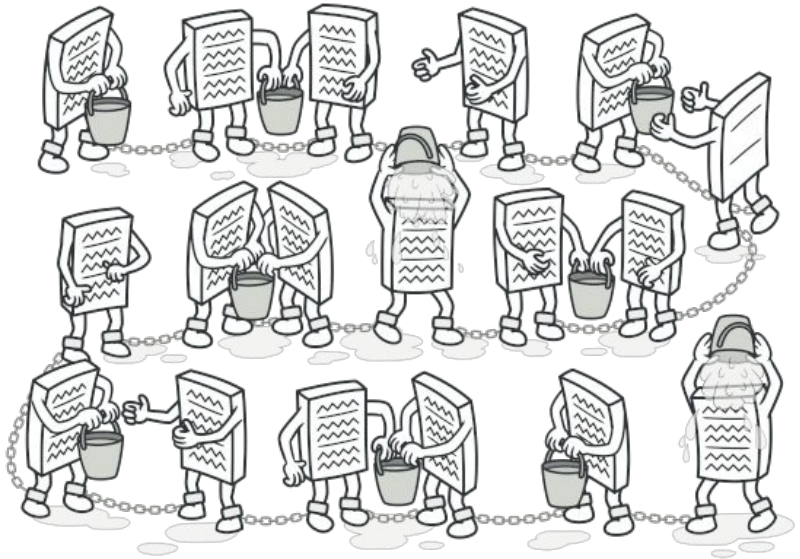
Caso de uso

- O padrão observador pode ser usado para gerenciar a transferência de dados em qualquer camada e até mesmo internamente para adicionar uma abstração. Na estrutura MVC, a View pode ser assinante do Model, que por sua vez também pode ser assinante do controller. Também pode acontecer o contrário se o caso de uso justificar.
- Existe um processo externo chamado DataController e um processo cliente que contém um DataModel e vários DataViews que são um gráfico de pizza, um gráfico de barras e uma visualização de tabela.
- O DataModel assina o DataController e os DataViews assinam o DataModel.
- O cliente configura as várias visualizações com uma assinatura do DataModel.



Chain of Responsibility

O Chain of Responsibility é um padrão de projeto comportamental que permite que você passe pedidos por uma corrente de handlers. Ao receber um pedido, cada handler decide se processa o pedido ou o passa adiante para o próximo handler na corrente.

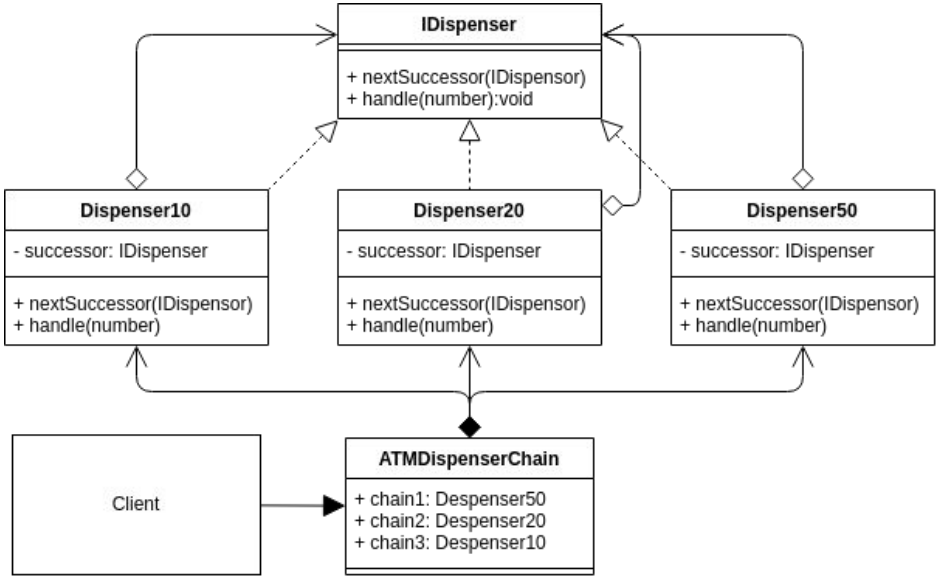


Aplicabilidade:

- Utilize o padrão Chain of Responsibility quando é esperado que seu programa processe diferentes tipos de pedidos em várias maneiras, mas os exatos tipos de pedidos e suas sequências são desconhecidos de antemão.
- O padrão permite que você ligue vários handlers em uma corrente e, ao receber um pedido, perguntar para cada handler se ele pode ou não processá-lo. Dessa forma todos os handlers têm a chance de processar o pedido.

Caso de uso

- No exemplo de caixa eletrônico abaixo, a cadeia é codificada no cliente primeiro para distribuir quantias de R$ 50, depois R$ 20 e, em seguida, R$ 10 em ordem.
- Essa ordem de cadeia padrão ajuda a garantir que o número mínimo de notas seja dispensado. Caso contrário, pode dispensar 5 x R$ 10 quando teria sido melhor dispensar 1 x R$ 50.



Referências

- https://refactoring.guru/pt-br

DÚVIDAS?

PROCURE-NOS

OBRIGADO!

kenzie.com.br

/kenzie-brasil