Design Patterns - Resumo Teórico ;)

Objetivos

- Descrever a importância da necessidade de integração.
- Diferenciar os tipos de integração atuais.
- Explicitar as implicações da integração de sistemas nas organizações.
- Planear a integração de sistemas numa organização.

Programa

- Razões para Integrar um sistema de integração (SI)
- Integração de sistemas de informação
- Geografia da integração
- No computador
- Na empresa
- Entre empresas
- Redes e Middleware
- Características da integração

Tipos de integração

- Orientado aos dados
- Orientado aos métodos
- Orientado às interfaces
- Orientado aos portais
- Orientado aos processos
- Impacto da Internet

Programa Integração no computador

- Ficheiros
- Sockets
- Bases de dados
- Monitores transacionais
- Componentes
- Servidores aplicacionais

Programa Integração na empresa

- Mensagens
- Procedimentos remotos
- Objectos distribuídos
- Código móvel
- Message brokers

Conteúdo

- Conceito de Design Pattern
- História dos Design Patterns
- Pertinência de Uso Críticas aos Design Patterns
- Classificação de
- Cuidados e Boas Práticas

O que é um Design Pattern?

- Design Patterns (Padrões de Projeto) são soluções típicas para problemas comuns em projetos de software.
- São como projetos de obras pré-fabricadas que podemos adaptar para resolver um problema de projeto recorrente no código.

- Não é possível simplesmente encontrar um Design Pattern e copiá-lo para o programa, como se faz com funções e bibliotecas.

- O Design Pattern não é um pedaço de código específico, mas um conceito geral para resolver um problema em particular. Podemos seguir os detalhes do padrão e implementar uma solução que se adeque às realidades do projeto em questão

- Os padrões são frequentemente confundidos com algoritmos, porque ambos os conceitos descrevem soluções típicas para alguns problemas conhecidos.

- Enquanto um algoritmo define sempre um conjunto claro de ações para atingir uma meta, um padrão é mais uma descrição de alto nível de uma solução. O código do mesmo padrão aplicado para dois programas distintos pode ser bem diferente

- A analogia de um algoritmo é uma receita de comida: ambos têm etapas claras para chegar a um objetivo.

Por outro lado, um padrão é mais como uma planta de obras: podemos ver o resultado e as suas funcionalidades, mas a ordem exata de implementação depende de nós

Em que consiste um Design Pattern

A maioria dos padrões são descritos formalmente para que as pessoas possam reproduzi-los em diferentes contextos.
Algumas secções que são geralmente presentes na descrição de um padrão:

- O Propósito do padrão descreve brevemente o problema e a solução.
- A Motivação explica a fundo o problema e a solução que o padrão torna possível.
- As Estruturas de classes mostram cada parte do padrão e como se relacionam. Exemplos de código numa das linguagens de programação populares tornam mais fácil compreender a ideia por trás do padrão.
- Alguns catálogos de padrão listam outros detalhes úteis, tais como a aplicabilidade do padrão, etapas de implementação, e relações com outros padrões.

História de Design Patterns

Quem inventou os Design Patterns? Boa pergunta, mas não muito precisa. Os design patterns não são conceitos obscuros e sofisticados — bem pelo contrário. Os padrões são soluções típicas para problemas comuns em projetos orientados a objetos. Quando uma solução é repetida uma vez e outra e outra... em vários projetos, alguém vai, eventualmente, nomeá-la e descrever a solução em detalhe. É assim que um padrão é descoberto.

O conceito de padrões foi primeiramente descrito por Christopher Alexander no livro 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 projetos orientado a objetos e tornou-se rapidamente um best-seller. Devido ao longo título, as pessoas começaram a chamá-lo simplesmente de "o livro da Gangue dos Quatro (Gang of Four)" que foi simplificado para o "livro GoF".

Porquê aprender Design Patterns?

- A verdade é que conseguimos trabalhar como um programador, por muitos anos, sem saber sobre um único padrão.

- Muitas pessoas fazem exatamente isso. Ainda assim, estaremos a implementar alguns padrões mesmo sem saber. Então, porque gastar tempo a aprender sobre eles?

Os Design Patterns

São um kit de ferramentas para soluções tentadas e testadas para problemas comuns em projetos de software.

Mesmo que nunca tenhamos encontrado esses problemas, saber sobre os padrões é muito útil porque eles ensinam como resolver vários problemas usando princípios de projeto orientado a objetos.

Definem uma linguagem comum que a equipa pode usar para comunicar de forma mais eficiente. Podemos dizer, "Oh, é só usar um Singleton para isso, " e toda a gente vai entender a ideia por trás da sugestão. Não é preciso explicar o que é um singleton se toda a gente reconhecer o padrão.

Críticas aos Design Patterns

"Desenrascanços"
para uma linguagem de programação fraca.

- Geralmente a necessidade de padrões surge quando as pessoas escolhem uma linguagem de programação ou uma tecnologia que tem uma deficiência no nível de abstração.
- Neste caso, os padrões se transformam em desenrascanços que dão à linguagem os superpoderes necessários. Por exemplo, o padrão Strategy pode ser implementado com uma simples função anónima (lambda) na maioria das linguagens de programação modernas

Soluções ineficientes

- Os padrões tentam sistematizar abordagens que já são amplamente usadas. Essa unificação é vista por muitos como um dogma (elefante na sala) e eles implementam os padrões “direto ao ponto” , sem adaptá-los ao contexto dos seus projetos.

Uso Injustificável

- Se tudo que tenho é um martelo, tudo parece um prego.

- Esse é o problema que assombra muitos novatos 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.

Classificação de Design Patterns

- Design Patterns diferem em complexidade, nível de detalhe, e escala de aplicabilidade ao sistema inteiro que está a ser desenvolvido.

- Analogia com a construção de uma estrada: podemos fazer uma intersecção mais segura instalando sinais de trânsito ou construindo nós de vários níveis com passagens subterrâneas para pedestres.

- Os padrões mais básicos e de baixo nível são frequentemente chamados idiomáticos.

- Eles aplicam-se, geralmente, apenas a uma única linguagem de programação.

- Os padrões mais universais e de alto nível são os padrões arquitetónicos. Os programadores podem implementar esses padrões em praticamente qualquer linguagem.

- Ao contrário de outros padrões, podem ser usados para fazer o projeto da arquitetura de uma aplicação na sua integridade

Além disso, todos os padrões podem ser categorizados pelo seu propósito ou intenção
Temos três grupos principais de padrões:

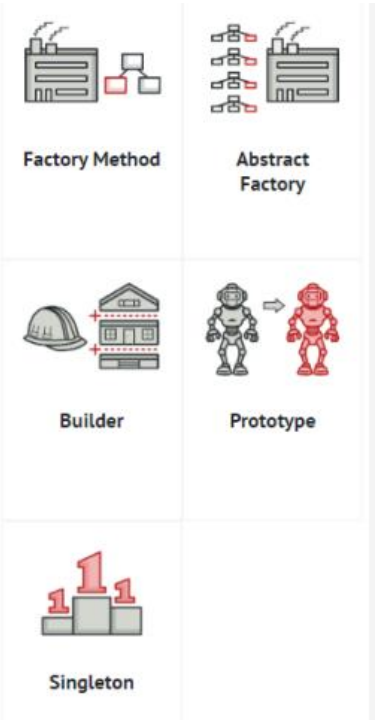
- **Os padrões criacionais (Creational Patterns)** fornecem mecanismos de criação de objetos que aumentam a flexibilidade e a reutilização de código.
- **Os padrões estruturais (Structural Patterns)** explicam como instanciar objetos e classes em estruturas maiores, enquanto ainda mantém as estruturas flexíveis e eficientes.

- **Os padrões comportamentais (Behavioral Patterns)** cuidam da comunicação eficiente e da sinalização de responsabilidades entre objetos.

Creational Patterns

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

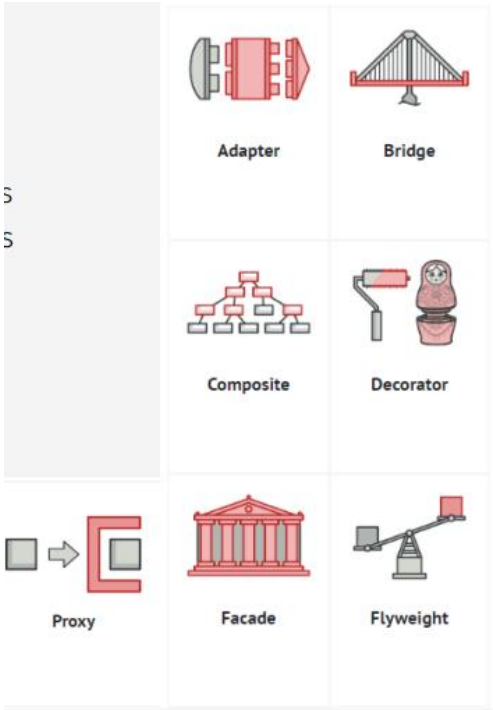
- Factory Method
- Abstract Method
- Builder Prototype
- Singleton



Structural Patterns

Estes padrões explicam como montar objetos e classes em estruturas maiores mas mantendo essas estruturas flexíveis e eficientes.

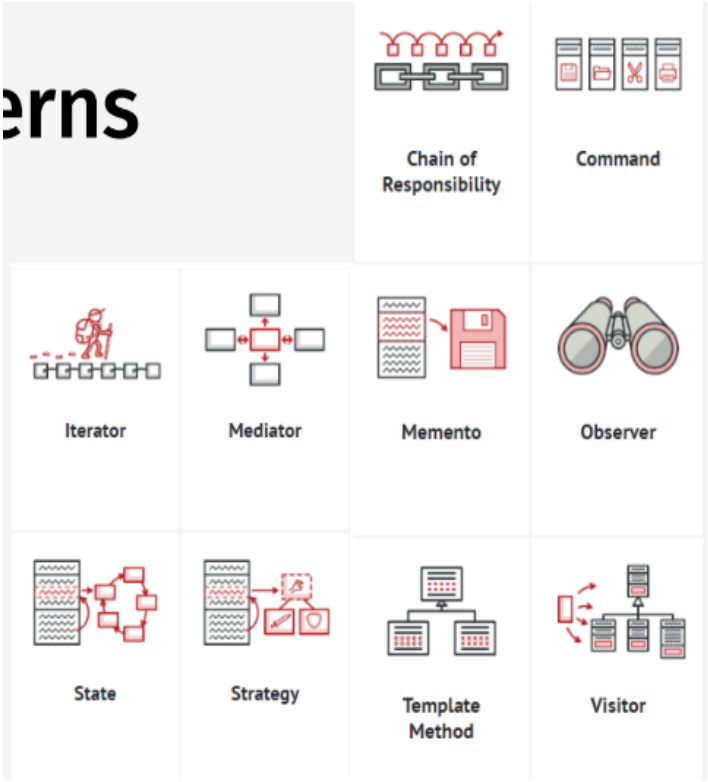
- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy



Behavioral Patterns

Estes padrões são focados em algoritmos e a designação de responsabilidades entre objetos.

- Chain of Responsabilty
- Command
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor



Singleton

O Singleton é um padrão de projeto criacional (Creational Pattern) que garante que uma classe tenha apenas uma instância, enquanto provê um ponto de acesso global para essa instância.

Problema?

O padrão Singleton resolve dois problemas de uma só vez, violando o princípio de responsabilidade única:

Garantir que uma classe tenha apenas uma única instância.

Porquê controlar quantas instâncias uma classe tem?

A razão mais comum para isso é para controlar o acesso a algum recurso compartilhado— por exemplo, uma base de dados ou um arquivo.
Funcionamento: imagine que criou um objeto, mas passado algum tempo decidiu criar um novo.
Ao contrário de receber um objeto criado de fresco, recebe um que já foi criado

- Esse comportamento é impossível implementar com um construtor regular uma vez que uma invocação do construtor retorna sempre um novo objeto, por defeito.

- Clientes podem nem dar conta que estão a lidar sempre com o mesmo objeto.

- Fornece um ponto de acesso global para aquela instância. Podemos utilizar, para isso, variáveis globais... Embora sejam inseguras, pois a qualquer momento podem ser acedidas e modificadas.

- Assim como uma variável global, o padrão Singleton permite aceder um objeto a partir de qualquer lugar no programa. Contudo, ele também protege aquela instância de ser sobrescrita por outro código.

- Há outro lado para esse problema: não é pretendido que o código que resolve o problema #1 fique espalhado pelo programa todo.

- É muito melhor tê-lo dentro de uma classe, especialmente se o resto do código já depende dela. Hoje em dia, o padrão Singleton tornou-se tão popular que as pessoas podem chamar algo de singleton mesmo se ele resolve apenas um dos problemas listados.

Solução

- Todas as implementações do Singleton tem esses dois passos em comum

- Implementar 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, nos bastidores, para criar um objeto e salva num campo estático. Todas as invocações seguintes desse método retornam o objeto em cache.

- Todas as implementações do Singleton tem esses dois passos em comum: Se o código tem acesso à classe singleton, então ele é capaz de chamar o método estático da singleton. Assim, sempre que aquele método é chamado, o mesmo objeto é retornado

Analogia

- Uma analogia comum para o padrão de design Singleton é a de um cofre. Imagina que tens um cofre numa sala, onde guardas um tesouro valioso. Queres ter a certeza de que apenas uma pessoa pode aceder ao tesouro de cada vez, para evitar roubos ou conflitos.

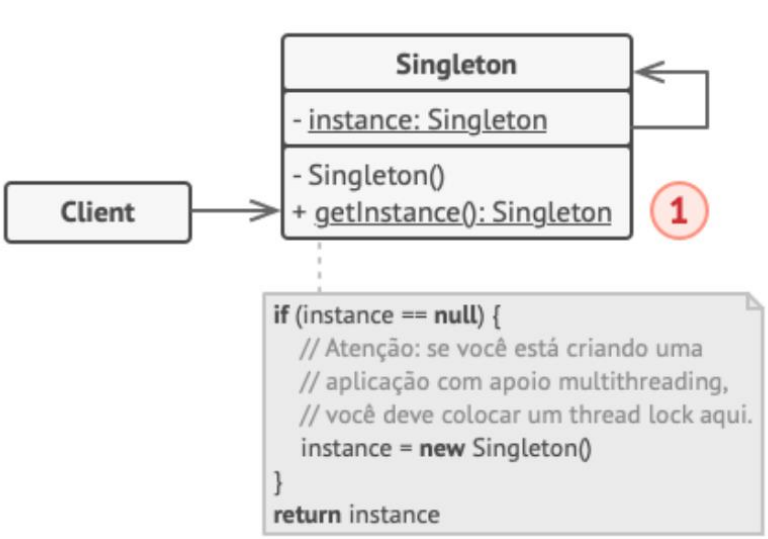
- Neste caso, o cofre é o objeto que possui um estado único (o tesouro) e o acesso a ele precisa de ser controlado. O padrão de design Singleton permite que apenas uma instância desse objeto seja criada, garantindo que todos os acessos ao tesouro sejam feitos através dessa única instância.

- Assim como no padrão Singleton, uma vez que uma pessoa está dentro da sala com o cofre, ela tem acesso exclusivo ao tesouro.

- Se outra pessoa tentar entrar na sala, ela não terá permissão para entrar até que a primeira pessoa saia e liberte o acesso.

- Da mesma forma, quando usas o padrão Singleton num programa, garantes que apenas uma instância do objeto seja criada e que todas as partes do código utilizem essa mesma instância. Isso é útil em situações em que precisas de partilhar um recurso único entre diferentes partes do sistema, como uma conexão com uma base de dados ou configurações globais.

- Estrutura
-
- A classe Singleton declara o método estático getInstance que retorna a mesma instância da sua própria classe. O construtor da singleton deve ser escondido do código cliente. Chamando o método getInstance deve ser o único modo de obter o objeto singleton.



Aplicabilidade

- Utilize o padrão Singleton quando uma classe no programa deve ter apenas uma instância disponível para todos seus clientes, por exemplo, um objeto de base de dados único partilhado por diferentes partes do programa.

- O padrão Singleton desabilita todos os outros meios de criar objetos de uma classe exceto pelo método especial de criação. Esse método tanto cria um novo objeto ou retorna um objeto existente se ele já tenha sido criado.

- Utilize o padrão Singleton quando precisa de um controlo mais restrito sobre as variáveis globais.

- Ao contrário das variáveis globais, o padrão Singleton garante que há apenas uma instância de uma classe. Nada, a não ser a própria classe singleton, pode substituir a instância salva em cache. Observe que podemos sempre ajustar essa limitação e permitir a criação de qualquer número de instâncias singleton

.

- O único pedaço de código que requer mudanças é o corpo do método **getInstance**.

Como Implementar

- Adicione um campo privado estático na classe para o armazenamento da instância singleton.

- Declare um método de criação público estático para obter a instância singleton.

- Implemente a "inicialização preguiçosa" dentro do método estático. Ela deve criar um novo objeto na sua primeira invocação e colocá-lo no campo estático.

- O método deve sempre retornar aquela instância em todas as chamadas subsequentes.

- Faça do construtor da classe privado. O método estático da classe vai ser capaz de chamar o construtor, mas não os demais objetos.

- Vá para o código cliente e substitua todas as chamadas diretas para o construtor do singleton com chamadas para seu método de criação estático.

Prós

- Pode ter certeza que uma classe só terá uma única instância. Ganha um ponto de acesso global para aquela instância. O objeto singleton é inicializado somente quando for pedido pela primeira vez.

Contras

- Viola o princípio de responsabilidade única. O padrão resolve dois problemas de uma só vez.

- O padrão Singleton pode mascarar um mau design, por exemplo, quando os componentes do programa sabem muito sobre cada um.

- O padrão requer tratamento especial num ambiente multithreaded para que múltiplas threads não possam criar um objeto singleton várias vezes. Pode ser difícil realizar testes unitários do código cliente do Singleton porque muitos frameworks de teste dependem de herança quando produzem objetos simulados.

- Já que o construtor da classe singleton é privado e sobrescrever métodos estáticos é impossível na maioria das linguagem, é necessário pensar numa maneira criativa de simular o singleton.

- Ou não escreva os testes. Ou não use o padrão Singleton.

Relações c/ Outros Padrões

- Uma classe fachada pode frequentemente ser transformada numa singleton já que um único objeto fachada é suficiente na maioria dos casos.

- O Flyweight seria parecido com o Singleton se conseguissemos, de algum modo, reduzir todos os estados de objetos compartilhados para apenas um objeto flyweight.

Mas há duas mudanças fundamentais entre esses padrões:

- Deve haver apenas uma única instância singleton, enquanto que uma classe flyweight pode ter múltiplas instâncias com diferentes estados intrínsecos.

- O objeto singleton pode ser mutável. Objetos flyweight são imutáveis. As Fábricas Abstratas, Construtores, e Protótipos podem todos ser implementados como Singletons.

Factory Method

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

Problema

- Imagine que está a criar uma aplicação de gestão de logística. A primeira versão da sua aplicação pode lidar apenas com o transporte de camiões, portanto a maior parte do seu código fica dentro da classe Camião. Depois de um tempo, a aplicação torna-se bastante popular. Todos os dias recebe dezenas de solicitações de empresas de transporte marítimo para incorporar a logística marítima na aplicação.

- Adicionar uma nova classe ao programa não é tão simples se o restante código já tiver acoplado às classes existentes.

- Boa notícia, certo? Mas e o código?

- Atualmente, a maior parte do seu código é acoplada à classe Camião.

- Adicionar Navio à aplicação exigiria alterações por toda a base de código. Além disso, se mais tarde decidir adicionar outro tipo de transporte à aplicação, provavelmente vai ser necessário fazer essas alterações todas novamente.

Como resultado, terá um código bastante "sujo" , repleto de condicionais que alteram o comportamento da aplicação, dependendo da classe de objetos de transporte.

Solução

- O padrão Factory Method sugere que substitua chamadas diretas de construção de objetos (usando o operador new) por chamadas para um método fábrica especial.

- Não se preocupe: os objetos ainda são criados através do operador new, mas esse será invocado dentro do método fábrica. Objetos retornados por um método fábrica geralmente são chamados de produtos.

- As subclasses podem alterar a classe de objetos retornados pelo método fábrica.

- À primeira vista, essa mudança pode parecer sem sentido: apenas mudamos a invocação do construtor de uma parte do programa para outra.
- No entanto, considere o seguinte: agora é possível sobrescrever o método fábrica numa subclasse e alterar a classe de produtos que estão a ser criados pelo método.

- Porém, há uma pequena limitação: as subclasses só podem retornar tipos diferentes de produtos se esses produtos tiverem uma classe ou interface base em comum. Além disso, o método fábrica na classe base deve ter tipo de retorno declarado como essa interface.

- Por exemplo, ambas as classes Camião e Navio devem implementar a interface Transporte, que declara um método chamado entregar.

- Cada classe implementa esse método de maneira diferente: camiões entregam carga por terra, navios entregam carga por mar.

- O método fábrica na classe LogísticaVia retorna objetos de camião, enquanto o método fábrica na classe LogísticaMarítima retorna navios.

- Desde que todas as classes de produtos implementem uma interface comum, você pode passar seus objetos para o código cliente sem quebrá-lo.

- O código que usa o método fábrica (geralmente chamado de código cliente) não vê diferença entre os produtos reais retornados por várias subclasses.

- O cliente trata todos os produtos como um Transporte abstrato.

- O cliente sabe que todos os objetos de transporte devem ter o método entregar, mas como exatamente ele funciona não é importante para o cliente.