PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS
Disciplina: Projeto de Software
Professor: João Paulo Carneiro Aramuni
Aluno: João Pedro Santana Marques

Capítulo 6: Padrões de Projeto

Este capítulo introduz os padrões de projeto como soluções testadas para problemas comuns de design e constituem um vocabulário compartilhado entre desenvolvedores, facilitando a comunicação e a documentação de sistemas. Inspirados nos trabalhos de Christopher Alexander na construção civil, os padrões de projeto foram adaptados para a engenharia de software por Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides, conhecidos como a "Gang of Four". O capítulo detalha alguns padrões de projeto, organizados em três categorias principais:

Padrões Criacionais: padrões que propõem soluções flexíveis para criação de objetos.

Fábrica
Contexto: Sistemas que precisam criar objetos cujos tipos podem variar conforme o contexto.
Problema: Instanciar diretamente classes específicas torna o sistema rígido e difícil de manter quando novos tipos precisam ser adicionados.
Solução: Utilizar uma classe Fábrica que encapsula a lógica de criação dos objetos, permitindo que o sistema opere de forma genérica com interfaces ou classes abstratas, facilitando a adição de novos tipos sem modificar o código existente.

Singleton
Contexto: Aplicações que necessitam garantir a existência de uma única instância de uma classe, como gerenciadores de configuração ou pools de conexões.
Problema: Permitir múltiplas instâncias pode levar a inconsistências e uso ineficiente de recursos.
Solução: Implementar o padrão Singleton, onde a própria classe controla sua instância única, fornecendo um ponto global de acesso e garantindo que apenas uma instância seja criada durante o ciclo de vida da aplicação.

Builder
Facilita a criação de objetos com muitos atributos, alguns opcionais, evitando múltiplos construtores e tornando o código mais legível. O Builder encapsula a configuração dos atributos antes da instanciação do objeto, preservando o princípio de ocultamento da informação.

Padrões Estruturais: padrões que propõem soluções flexíveis para composição de classes e objetos.

Proxy
Contexto: Necessidade de controlar o acesso a um objeto, seja por motivos de segurança, desempenho ou controle de recursos.
Problema: Acesso direto a certos objetos pode ser custoso ou requerer restrições adicionais.
Solução: Introduzir um objeto Proxy que atua como intermediário, controlando o acesso ao objeto real e podendo adicionar funcionalidades como cache, controle de acesso ou criação sob demanda.

Adaptador
Contexto: Integração de classes com interfaces incompatíveis que precisam trabalhar juntas.
Problema: Modificar classes existentes para torná-las compatíveis pode não ser viável, especialmente quando se trata de bibliotecas de terceiros.
Solução: Criar uma classe Adaptador que converte a interface de uma classe em outra esperada pelo sistema, permitindo que classes incompatíveis trabalhem em conjunto sem alterações em seu código original.

Fachada
Contexto: Sistemas complexos com múltiplas interações entre subsistemas.
Problema: A complexidade do sistema torna difícil para os clientes interagirem diretamente com seus componentes.
Solução: Implementar uma classe Fachada que fornece uma interface simplificada e unificada para os subsistemas, facilitando o uso e reduzindo o acoplamento entre o sistema e seus clientes.

Decorador
Contexto: Necessidade de adicionar comportamentos ou responsabilidades a objetos de forma dinâmica e flexível.
Problema: Herança múltipla ou modificações na hierarquia de classes podem levar a sistemas rígidos e difíceis de manter.
Solução: Utilizar o padrão Decorador, onde novos comportamentos são adicionados ao envolver o objeto original em uma série de objetos decoradores, permitindo combinações flexíveis de funcionalidades sem alterar as classes existentes.

Padrões Comportamentais: padrões que propõem soluções flexíveis para interação e divisão de responsabilidades entre classes e objetos.

Strategy
Contexto: Sistemas que requerem a implementação de diferentes algoritmos ou comportamentos intercambiáveis.
Problema: Incorporar múltiplos algoritmos em uma única classe pode torná-la complexa e difícil de manter.
Solução: Definir uma família de algoritmos encapsulados em classes separadas que implementam uma interface comum, permitindo que o comportamento do sistema seja alterado em tempo de execução pela composição de diferentes estratégias.

Observador
Contexto: Cenários onde mudanças no estado de um objeto precisam ser refletidas em outros objetos dependentes.
Problema: Implementar notificações manuais entre objetos pode levar a um alto acoplamento e código redundante.
Solução: Adotar o padrão Observador, onde objetos "observadores" se registram para receber notificações de um objeto "observado", permitindo uma comunicação eficiente e desacoplada entre os componentes.

Template Method
Contexto: Classes que compartilham uma estrutura comum de algoritmo, mas com implementações específicas variando entre subclasses.
Problema: Duplicação de código e dificuldade em manter a consistência entre implementações semelhantes.

Solução: Definir o esqueleto do algoritmo na classe base, delegando passos específicos para as subclasses que podem redefinir esses passos conforme necessário, promovendo reutilização e consistência.

Visitor
Contexto: Estruturas de objetos complexas onde operações sobre esses objetos precisam ser definidas sem alterar suas classes.
Problema: Adicionar novas operações pode exigir modificações nas classes dos objetos, violando o princípio aberto/fechado.
Solução: Implementar o padrão Visitor, onde uma nova classe é criada para cada operação, permitindo que novas funcionalidades sejam adicionadas sem modificar as classes dos elementos sobre os quais operam.

Iterador
Define uma interface padronizada para percorrer estruturas de dados sem conhecer seus detalhes internos. Geralmente inclui métodos como hasNext() e next(), permitindo múltiplos percursos simultâneos sobre a mesma estrutura.

A última seção discute os custos e riscos associados ao uso excessivo de padrões:
Custo dos padrões: Embora aumentem a flexibilidade, padrões como Fábrica e Strategy adicionam complexidade ao exigir a criação de classes adicionais.
Critério de uso: Antes de adotar um padrão, deve-se avaliar se a flexibilidade oferecida é realmente necessária.
Risco de "paternite": O uso indiscriminado de padrões pode tornar o código mais complexo sem benefícios reais.

Aplicação Prática no Mercado: O padrão Singleton pode ser utilizado para gerenciar a conexão com o banco de dados, garantindo que apenas uma instância da conexão seja utilizada ao longo da execução do sistema, otimizando recursos.