PUC Minas

PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS

Projeto de Software

Resenha dos capítulos 5 e 6 do livro Engenharia de Software Moderna

Professor: João Paulo Aramuni
Aluno: Giovanna Ferreira dos Santos de Almeida

É possível definir Projeto de Software como sendo fundamentalmente a decomposição de um problema, isto é, buscar resolver um problema complexo através de sua divisão para que suas partes sejam resolvidas individualmente. Para que sistemas não se tornem complexos, é importante que suas partes sejam abstratas. Para evitar criar sistemas com alta complexidade, é importante criar abstrações: ocultar detalhes desnecessários e destacar apenas os aspectos essenciais para que as entidades sejam mais proveitosas. O conceito de Integridade Conceitual foi exposto em 1975 por Frederick Brooks, que defende a ideia de que a coerência e a coesão de um sistema são as partes mais importantes no projeto de sistemas. Isso faz com que o usuário se sinta confortável ao utilizar todas as partes de um sistema, uma vez que as funcionalidades e a interface da aplicação são mantidas de forma consistente. Um exemplo aplicado à vida real é o GNOME — uma interface gráfica e conjunto de aplicativos desktop para usuários de Linux — que garante que todas as aplicações sigam padrões de design e comportamento, como layout unificado, atalhos consistentes e configurações centralizadas no mesmo painel. O princípio de integridade conceitual também é aplicável ao design e ao código de uma aplicação. A falta desse princípio compromete os desenvolvedores, que terão dificuldade para entender, manter e evoluir o sistema. Para preservar a integridade conceitual em um sistema, é essencial haver consenso na padronização e criação do código. Isso inclui, por exemplo, a escolha conjunta de um padrão para a declaração de variáveis, a uniformidade na utilização de frameworks e a padronização das estruturas de dados utilizadas para solucionar

problemas semelhantes. Em 1972, David Parnas enunciou o princípio de Ocultamento de Informação, capaz de tornar sistemas de software mais flexíveis e fáceis de entender, além de reduzir o tempo de desenvolvimento. Esse conceito traz diversas vantagens para um sistema, pois permite o desenvolvimento em paralelo — classes que ocultam suas principais informações facilitam a implementação por diferentes desenvolvedores —, além de proporcionar flexibilidade a mudanças e facilidade de entendimento. Para que esses benefícios sejam alcançados, as decisões de projeto sujeitas a mudanças devem ser ocultadas. No entanto, uma classe que possui todos os seus métodos privados não é útil. Na vida real, em um sistema de pagamentos, o módulo responsável pelo processamento de pagamentos pode ocultar suas implementações internas dos demais módulos — isso permite alterações na lógica do pagamento sem impactar outras partes do sistema. O conceito de acoplamento refere-se ao grau de dependência entre os componentes de um sistema. Existem dois tipos principais: o acoplamento aceitável, que ocorre quando uma classe A utiliza métodos públicos da classe B que são estáveis tanto do ponto de vista semântico quanto sintático. Já o acoplamento ruim acontece quando alterações na classe B afetam diretamente a classe A. Os princípios SOLID são soluções típicas para problemas comuns na produção de software. O Princípio da Responsabilidade Única estabelece que cada classe deve ter uma única responsabilidade, ou seja, um único motivo para ser modificada. Por exemplo, se uma classe é responsável por gerar relatórios e também por enviá-los por e-mail, ela está violando esse princípio, pois está realizando duas funções distintas. O Princípio da Segregação de Interfaces é uma aplicação do Princípio da Responsabilidade Única, mas focado em interfaces. Isso significa que um cliente não deve ser forçado a implementar métodos de uma interface que ele não vai usar. O princípio defende que as interfaces devem ser pequenas, coesas e específicas para cada tipo de cliente. Por exemplo, se uma classe Impressora tivesse métodos para imprimir tanto documentos quanto fotos, isso violaria esse princípio, já que nem todas as impressoras precisam dos dois tipos de impressão. O princípio de Inversão de Dependências estabelece que uma classe cliente deve depender prioritariamente de interfaces ao invés de classes. Existem dois tipos de herança: a herança de classes, que envolve o reaproveitamento de código, e a herança de interfaces, que não envolve o reúso de código, portanto é mais simples. No início da programação orientada a objetos, o uso de herança era incentivado porque acreditava-se que ela

resolveria o problema do reuso de código. Porém, hoje é sabido que o uso excessivo de herança pode gerar problemas de manutenção e evolução no sistema, devido ao forte acoplamento entre as subclasses e suas superclasses. O Princípio de Demeter estabelece que, ao implementar um método, ele só deve chamar métodos de sua própria classe, de objetos passados como parâmetros, de objetos criados dentro do próprio método e dos atributos da classe do método. Já o Princípio Aberto/Fechado, proposto por Bertrand Meyer, estabelece que uma classe deve ser fechada para modificações, mas aberta para extensões. Esse princípio objetiva a construção de classes mais flexíveis e extensíveis que se adaptam a diferentes cenários de uso sem modificar seu código fonte. As métricas de código fonte permitem avaliar a qualidade de um projeto de forma mais objetiva. Ao analisar características do código fonte, as métricas expressam quantitativamente propriedades como tamanho, coesão, acoplamento e complexidade do código. Atualmente, o método de analisar um projeto através de métricas de código fonte não é uma mais comum, pois as propriedades de um sistema são subjetivas. Além disso, os resultados de métricas de software dependem de contexto — uma determinada métrica pode ser aceitável em um sistema, mas em outro não. A métrica mais popular é a linhas de código (LOC) que mede o tamanho de uma função, classe, pacote ou um sistema inteiro. No entanto, LOC não deve ser usada para medir a produtividade de desenvolvedores. A métrica LCOM é responsável por calcular a falta de coesão de uma classe. Geralmente, quanto maior o valor da métrica, pior a qualidade do código. No entanto, quanto maior a coesão de uma classe, melhor é o projeto. O CBO é a métrica que mede o acoplamento estrutural entre duas classes e a Complexidade Ciclomática (CC) é uma métrica que mede a complexidade de código de uma função ou método. A complexidade nesse caso está relacionada à dificuldade de manter e testar uma função.

Os padrões de projeto foram criados em 1995 por Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides, que elaboraram um catálogo com soluções para problemas comuns na produção de software. Em um sistema distribuído baseado em TCP/IP, existem funções como f, g e h que criam objetos do tipo TCPChannel para comunicação remota. Se, dependendo das configurações do sistema, for necessário usar UDP para a comunicação, o Princípio Aberto/Fechado está sendo violado, pois o sistema não está fechado para modificações e aberto para extensões nos protocolos de comunicação. Para corrigir isso, pode-se usar o

padrão Fábrica. Nesse caso, as funções f, g e h não precisam mais saber qual tipo de objeto — TCPChannel ou UDPChannel — elas estão criando, tornando o sistema mais flexível e permitindo a expansão para novos tipos de canais de comunicação sem modificar o código existente. O padrão Singleton garante que uma classe tenha apenas uma instância e fornece um ponto de acesso global para essa instância. Por exemplo, uma classe Logger, que é usada para registrar eventos em um sistema. Sem o Singleton, diferentes partes do código poderiam criar várias instâncias da classe Logger em métodos distintos. No entanto, o ideal seria que todas as partes do sistema usassem a mesma instância para registrar os eventos. Para isso, a classe Logger deve ser transformada em um Singleton. O o padrão Singleton é frequentemente criticado, pois pode esconder a criação de variáveis e estruturas de dados globais, tornando o código mais difícil de entender e testar. Em geral, o Singleton é mais adequado para modelar recursos que precisam ter, no máximo, uma instância durante a execução do programa. O padrão Proxy estabelece que um objeto intermediário seja inserido entre o objeto base e seus clientes, de modo que os clientes interajam diretamente com o proxy, e não com o objeto base. O objetivo do proxy é mediar o acesso ao objeto real, adicionando funcionalidades extras sem que este saiba. Por exemplo, uma classe BookSearch, que tem um método para pesquisar livros com base no ISBN. Para melhorar o desempenho da pesquisa, podemos adicionar um sistema de cache em uma outra classe, garantindo que o princípio da responsabilidade única seja mantido. Os proxies também podem ser usados para implementar outros requisitos não-funcionais, como stubs, alocação de memória sob demanda ou controle de acesso de múltiplos clientes a um objeto base. O padrão Adaptador (Wrapper) é usado quando é necessário converter a interface de uma classe para outra interface esperada pelo cliente. Por exemplo, em um sistema que controla projetos multimídia, é preciso instanciar classes fornecidas por diferentes fabricantes. O método de uma classe de uma marca específica de projetores pode exigir parâmetros que outra classe não necessita. Para resolver esse problema, uma classe adaptadora é criada para ajustar as diferenças entre as interfaces e garantir que o sistema funcione de forma consistente com todos os fabricantes. O padrão Fachada define uma classe que fornece uma interface simplificada para um sistema complexo. Com isso, o usuário interage apenas com a classe fachada, sem precisar se preocupar com as classes internas do sistema. Por exemplo, a implementação

de um interpretador para a linguagem X, que é executado a partir de Java. O processo para rodar um programa X é complexo, exigindo o uso de várias classes internas do interpretador. Essa complexidade gera dificuldades para os desenvolvedores, que pedem uma interface mais simples para interagir com o interpretador, sem precisar lidar com esses detalhes internos. O Padrão Decorador oferece uma alternativa à herança quando é preciso adicionar novas funcionalidades a uma classe base. Em vez de usar herança, o padrão utiliza composição para adicionar funcionalidades de forma dinâmica às classes. Por exemplo, no exemplo anterior do padrão Fábrica, imagine que os clientes precisem adicionar funcionalidades extras, como buffers ou compactação de mensagens. Usando o padrão Decorador, essas funcionalidades opcionais são configuradas no momento da instância da classe, por meio de uma sequência de instâncias aninhadas com o uso do operador new. O padrão Strategy estabelece como tornar os algoritmos de uma classe parametrizáveis e intercambiáveis. Ele é útil quando uma classe depende de um determinado algoritmo e é necessário flexibilidade para alterá-lo sem modificar o código da classe. Por exemplo, a classe MyList atualmente usa o algoritmo Quicksort para ordenar seus elementos. No entanto, os clientes gostariam de escolher e definir outro algoritmo de ordenação. A implementação da classe não permite essa flexibilidade, o que vai contra o princípio Aberto/Fechado, pois, para adicionar novos algoritmos de ordenação, seria necessário alterar o código da classe. O padrão Observador estabelece uma forma de criar uma relação de um-para-muitos entre um objeto sujeito e seus observadores. Por exemplo, um sistema de controle de estação meteorológica, temos a classe Temperatura, que armazena os dados da temperatura, e a classe Termômetro, que exibe esses dados. Quando a temperatura muda, os termômetros precisam ser atualizados. A classe Temperatura não deve depender diretamente da classe Termômetro, pois a interface pode mudar com frequência. O sistema pode incluir interfaces web, para dispositivos móveis e diferentes tipos de termômetros. Além disso, o sistema possui outras classes e o objetivo é reutilizar o mecanismo de notificação entre esses pares de sujeito-observador. O padrão Observador traz como vantagem o desacoplamento entre os sujeitos e seus observadores, além de permitir que o mecanismo de notificação seja reutilizado entre diferentes pares de sujeitos e observadores. O padrão Template Method estabelece a padronização de algoritmos em uma classe abstrata, deixando algumas partes do processo para serem definidas nas

subclasses. No exemplo de uma folha de pagamento, temos a classe Funcionario com subclasses FuncionarioPublico e FuncionarioCLT. O cálculo do salário é padronizado na classe base por meio de um método que calcula o salário líquido após descontar previdência, plano de saúde e outros descontos. As subclasses, então, implementam os métodos específicos para calcular cada desconto de acordo com o tipo de funcionário. Esse padrão permite que a estrutura do algoritmo seja mantida na classe base, enquanto as variações de implementação ficam nas subclasses. O padrão de projeto Visitor facilita a adição de operações em uma hierarquia de objetos sem precisar alterar suas classes. Por exemplo, em um sistema de estacionamento, temos uma classe Veiculo e suas subclasses Carro, Onibus e Motocicleta. Para realizar operações como imprimir informações dos veículos, sem modificar as classes desses veículos, um método accept deve ser criado em cada classe, que chama o método visit do Visitor, passando o objeto atual como parâmetro. A classe Visitor pode agrupar várias operações, como imprimir ou salvar dados, e pode ser facilmente adaptada para incluir novas operações ou tipos de objetos. No entanto, um ponto importante a considerar é que o uso de Visitors pode comprometer o encapsulamento das classes, pois elas podem precisar expor dados internos para que o Visitor possa acessá-los. Existem vários padrões de projeto, como o Iterator, que define uma forma padronizada de percorrer uma estrutura de dados sem precisar saber seu tipo exato, e o Builder, que facilita a criação de objetos com muitos atributos, alguns deles opcionais. O objetivo dos padrões de projeto é aumentar a flexibilidade dos sistemas, permitindo, por exemplo, trocar tipos de objetos ou adicionar novas funcionalidades às classes. No entanto, eles também podem produzir custos, como a necessidade de criar classes adicionais. Antes de adotar um padrão, é importante avaliar se realmente é necessário. Se a solução for simples o suficiente, como usar o operador new, talvez o padrão não seja necessário. O uso excessivo de padrões pode levar ao uso excessivo sem considerar se eles realmente trazem benefícios significativos.