PUC Minas

PONTIFÍCIA UNIVERSIDADE CATÓLICA DE MINAS GERAIS

Projeto de Software

Resenha dos capítulos 7 e 9 do livro Engenharia de Software Moderna

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

A arquitetura em camadas organiza as classes em módulos maiores, dispostos hierarquicamente. Isso significa que uma camada pode utilizar serviços da camada abaixo, como chamar métodos, instanciar objetos e estender classes. Esse modelo é muito utilizado na implementação de protocolos de rede. Por exemplo, o protocolo HTTP pertence à camada de aplicação e depende dos serviços de um protocolo de transporte, como o TCP. O TCP, por sua vez, usa os serviços de um protocolo de rede, como o IP. Finalmente, a camada IP se baseia em protocolos de comunicação, como o Ethernet. Essa organização em camadas ajuda a reduzir a complexidade do sistema, dividindo-o em partes menores e disciplinando as dependências entre elas. Como cada camada só pode interagir diretamente com a camada imediatamente inferior, a troca de componentes se torna mais simples, e o reuso de camadas por partes superiores do sistema fica mais fácil. Dikjastra propôs a seguinte organização de camadas: multiprogramação, alocação de memória, comunicação entre processos, gerenciamento de entrada/saída e programas dos usuários. A hierarquia de camadas é fundamentalmente importante para sistemas de grande porte. A arquitetura em três camadas é muito utilizada no desenvolvimento de sistemas corporativos. Antigamente, aplicações como folhas de pagamento rodavam em mainframes, com sistemas monolíticos acessados por terminais que não tinham capacidade de processamento e possuíam apenas uma interface textual. Com a evolução tecnológica das redes e do hardware, foi possível migrar esses sistemas para outras plataformas. A arquitetura em três camadas é composta pela interface com o usuário que é responsável pela interação com o cliente, exibindo informações

e coletando dados, lógica de negócio responsável pela implementação das regras do sistema, como validação de dados e automação de processos, garantindo que as operações sigam as regras definidas e o banco de dados responsável por armazenar e gerenciar as informações processadas pelo sistema, garantindo sua persistência e organização. Geralmente, a arquitetura em três camadas é distribuída: a interface é executada na máquina do cliente, a lógica de negócio fica em um servidor, e o banco de dados é gerenciado separadamente. Também existem sistemas em duas camadas, onde a interface e a lógica de negócio ficam juntas em um único programa que roda no cliente, enquanto o banco de dados continua separado. A arquitetura MVC (Model-View-Controller) foi utilizada na implementação da linguagem orientada a objetos Smalltalk-80, que também foi pioneira no uso de interfaces gráficas, como janelas, botões e scrollbars. A arquitetura MVC estabelece que o sistema deve ser organizado em três componentes principais: a Visão, responsável pela interface gráfica e exibição de informações para o usuário; o Controlador, que interpreta e processa os eventos gerados por dispositivos de entrada, como cliques e teclas pressionadas; e o Modelo, que gerencia os dados do sistema, armazenando e manipulando as informações. Em muitos sistemas, a separação entre Visão e Controlador não é tão distinta, sendo mais comum tratá-los como parte da interface gráfica. No entanto, as classes do Modelo devem ser independentes da interface. O padrão MVC traz diversas vantagens, como a especialização do trabalho, a reutilização do Modelo por diferentes interfaces e proporciona maior facilidade de testar o sistema. Os sistemas web chamados Single Page Applications (SPAs) se assemelham mais a aplicações desktop do que a aplicações web tradicionais. As SPAs carregam no navegador todo o código necessário, incluindo páginas HTML, CSS e scripts em JavaScript. Isso faz com que o usuário tenha a impressão de estar usando um aplicativo local, pois a página do navegador não precisa ser recarregada sempre que determinados eventos ocorrem. O Gmail é um dos exemplos mais conhecidos de SPA. Além disso, essas aplicações seguem uma arquitetura semelhante ao modelo MVC (Model-View-Controller). Em um sistema monolítico, mudanças feitas em um módulo podem afetar outros módulos. Para evitar que os clientes enfrentem bugs em seus sistemas, as empresas que utilizam arquiteturas monolíticas adotam processos rigorosos e burocráticos para lançar novas funcionalidades. Nos últimos anos, muitas empresas começaram a migrar seus sistemas monolíticos para uma arquitetura baseada em

microsserviços. Nesse modelo, cada módulo do sistema é executado em um processo independente, sem compartilhar o mesmo espaço de memória. Isso significa que, além de a separação entre módulos ocorrer no desenvolvimento, ela também existe na execução do sistema. Como cada microsserviço roda em um processo separado, ele não pode acessar diretamente os recursos internos de outro microsserviço. Essa abordagem exige que as equipes de desenvolvimento utilizem interfaces públicas para a comunicação entre módulos. A adoção de microsserviços traz diversas vantagens, como a evolução mais rápida e de forma independente, permitindo que diferentes partes sejam escaladas conforme necessário. Além disso, essa arquitetura possibilita o uso de diferentes tecnologias para cada módulo e reduz o impacto de possíveis falhas, uma vez que um erro em um serviço não compromete todo o sistema. O avanço da computação em nuvem foi essencial para a criação dos microsserviços, pois permite que empresas não precisem comprar e manter hardware, software, sistemas operacionais e servidores. Em vez disso, elas alugam máquinas virtuais em plataformas de computação em nuvem e pagam apenas pelo uso dos recursos. Os microsserviços devem ser autônomos do ponto de vista dos dados, ou seja, cada um deve gerenciar as informações necessárias para fornecer seu serviço. Por exemplo, nas equipes de desenvolvimento há um administrador de dados responsável por gerenciar o banco de dados. Qualquer mudança nesse banco precisa ser aprovada por ele, o que exige a conciliação de interesses entre as equipes de desenvolvimento. A adoção de microsserviços traz desafios típicos de sistemas distribuídos. Quando dois módulos rodam no mesmo processo, a comunicação entre eles ocorre por chamadas diretas de métodos. No entanto, quando esses módulos estão em máquinas diferentes, a comunicação precisa ocorrer por meio de protocolos como HTTP/REST, o que exige que os desenvolvedores conheçam tecnologias específicas para comunicação em redes. Outro problema importante é a latência. Se um serviço chamado estiver em outra máquina, por exemplo, no caso de empresas globais, o tempo de resposta pode ser significativamente maior. Além disso, em arquiteturas baseadas em microsserviços, pode ser necessário o uso de protocolos de transações distribuídas, como o Two-Phase Commit, para garantir consistência em operações que envolvem múltiplos bancos de dados. Na arquitetura Orientada a Mensagens, a comunicação entre clientes, que atuam como produtores de informações, e servidores, que atuam como consumidores, é mediada por um serviço responsável por gerenciar uma fila

de mensagens. É importante que a fila de mensagens seja persistente, ou seja, o serviço deve estar instalado em uma máquina estável e com alto poder de processamento para assegurar a confiabilidade da comunicação. Além disso, essa arquitetura proporciona dois tipos de desacoplamento necessários. O desacoplamento no espaço significa que clientes não precisam ter conhecimento dos servidores e vice-versa. O desacoplamento no tempo permite que clientes e servidores não precisem estar disponíveis simultaneamente para se comunicarem, tornando o sistema mais robusto a falhas e garantindo que as mensagens sejam entregues mesmo que um dos lados esteja temporariamente indisponível. Em arquiteturas publish/subscribe, as mensagens são chamadas de eventos, os componentes são os publicadores e assinantes de evento. Os publicadores geram eventos que são publicados em um serviço publish/subscribe, executado em uma máquina separada. Quando um evento é publicado, todos os assinantes desse evento são notificados. No modelo publish/subscribe, um evento gera notificações para todos os seus assinantes, ou seja, o publish/subscribe utiliza um modelo de comunicação em grupo (1 para n). Os assinantes são notificados assincronamente. Eles primeiro assinam determinados eventos e continuam seu processamento normalmente. Quando um evento de interesse ocorre, eles são notificados e executam um método específico para lidar com ele. Já no modelo de filas de mensagens, os servidores precisam buscar ativamente as mensagens da fila para processá-las. Existem também outros padrões arquiteturais, como o pipes e filtros, onde os programas processam os dados recebidos na entrada e geram uma nova saída. A arquitetura cliente/servidor é utilizada para implementar serviços básicos de rede. Nesse modelo, os clientes fazem solicitações a um módulo servidor e aguardam o processamento da resposta. Exemplos comuns incluem serviços de impressão, armazenamento de arquivos e aplicações web. Já a arquitetura peer-to-peer é um modelo distribuído no qual os módulos da aplicação podem atuar tanto como cliente quanto como servidor, dependendo da necessidade. O anti-padrão arquitetural é uma organização não recomendada. Acontece quando os módulos de um sistema se comunicam de forma desorganizada, sem seguir uma estrutura bem definida. Isso significa que qualquer módulo pode interagir diretamente com qualquer outro, resultando em um sistema difícil de manter e propenso a erros. Segundo as Leis de Lehman, um sistema de software precisa ser constantemente mantido para se adaptar ao seu ambiente e esse processo deve

continuar até que se torne mais vantajoso substituí-lo por um novo sistema. Além disso, à medida que um sistema passa por manutenções, sua complexidade tende a aumentar e sua qualidade pode se deteriorar, a menos que sejam tomadas medidas para estabilizá-lo ou evitar esse fenômeno, chamadas de refactoring. As refactorings são mudanças no código que melhoram a manutenibilidade de um sistema. Por exemplo, dividir uma função em duas, renomear variáveis, mover uma função para outra classe, entre outras alterações. Elas ajudam a melhorar a modularidade, a organização do código e a testabilidade do sistema. No entanto, é essencial garantir que essas melhorias não prejudiquem o funcionamento do software. A extração de método é um dos refactorings mais comuns. O objetivo é pegar um trecho de código de um método f e movê-lo para um novo método g. Existem variações do método, como, por exemplo,quando há a extração de vários métodos de uma vez de f ou quando extrai o mesmo código g de vários métodos. Nesse caso, a extração é usada para eliminar duplicação de código. Um estudo realizado em 2016 revelou que a principal razão para extrair um código é facilitar seu reúso. O inline de método é o oposto da extração de método. Por exemplo, um método pequeno que é chamado poucas vezes pode ser removido e seu código incorporado diretamente nos locais onde ele é chamado. O inline de método é uma operação mais rara e menos significativa do que a extração de método. A movimentação de métodos é um dos refactorings mais eficazes para melhorar a modularização de um sistema. Muitas vezes, um método acaba sendo implementado na classe errada. Isso acontece quando um método, apesar de estar em uma classe A, utiliza mais serviços de uma classe B do que da própria classe onde foi definido. Se ele tem mais dependências de B, pode fazer sentido movê-lo para essa classe. Esse refactoring melhora a coesão da classe A, reduz o acoplamento entre A e B. Por não estar restrito apenas a uma única classe, a movimentação de métodos traz vantagens na arquitetura do sistema. O refactoring de extração de classe é recomendado quando uma classe possui muitas responsabilidades e atributos. Se alguns desses atributos estão relacionados entre si e poderiam existir de forma independente, é benéfico extraí-los para uma nova classe. Um dos refactorings mais populares é a renomeação, que consiste em dar nomes mais adequados e significativos para os elementos do código. Segundo Phil Karlton, a invalidação de cache e dar nomes às coisas são as duas tarefas mais difíceis em ciência da computação. A parte mais desafiadora da renomeação não é escolher um novo

nome, mas garantir que todas as referências ao elemento renomeado sejam corretamente atualizadas no código. Há refactorings que ajudam a melhorar o escopo local do código, como por exemplo, a extração de variáveis, que simplifica expressões complexas, tornando-as mais fáceis de ler e entender. Outro exemplo é a remoção de flags, que sugere o uso de comandos como break ou return em vez de variáveis de controle para tornar o fluxo do código mais claro. Alguns refactorings são voltados para a simplificação de comandos condicionais, como o de substituição de condicional por polimorfismo, que reduz a complexidade de estruturas if-else ou switch. Já a remoção de código morto recomenda deletar métodos, classes, variáveis ou atributos que não estão mais sendo utilizados. Por exemplo, um método que não tem mais chamadas no código pode ser removido para evitar confusão e melhorar a manutenção do sistema. Refactorings bem-sucedidos dependem de bons testes, especialmente testes de unidade. Sem testes adequados, alterações no código podem ser arriscadas, principalmente porque os refactorings não adicionam novas funcionalidades nem corrigem bugs, apenas melhoram a estrutura do código. Outra questão importante é saber quando realizar um refactoring. Os refactorings oportunistas acontecem durante o desenvolvimento de uma funcionalidade, quando o programador encontra um trecho de código mal estruturado e decide melhorá-lo no momento. Já os refactorings planejados envolvem mudanças mais profundas, demoradas e complexas, que exigem sessões dedicadas. As IDEs oferecem suporte para automatizar refactorings, permitindo que o usuário selecione um trecho de código e escolha a operação de refactoring desejada. Já os code smells são sinais de que um código tem baixa qualidade, tornando-o difícil de manter, estender, modificar e testar. No entanto, a presença de um código com baixa qualidade não significa que o código deva ser imediatamente refatorado. Antes de tomar essa decisão, é importante avaliar a relevância do trecho de código e com que frequência ele precisará ser mantido. Um exemplo clássico de code smell é o código duplicado, que dificulta a manutenção, pois qualquer alteração precisa ser feita em vários lugares. Isso aumenta o risco de inconsistências, já que pode acontecer de modificar um trecho e esquecer de atualizar outro. Para resolver esse problema, técnicas como extração de método, extração de classe e pull up method podem ser aplicadas para organizar melhor o código e evitar redundâncias. Um estudo realizado em 2013 apontou que, entre os diversos code smells, o código duplicado era o que mais preocupava os

desenvolvedores. Além do código duplicado, existem outros code smells que podem comprometer a qualidade do software. Métodos longos, por exemplo, dificultam a leitura e compreensão do código. Classes grandes podem assumir muitas responsabilidades, o que compromete a coesão e dificulta a manutenção. Outro problema comum é o feature envy, que ocorre quando uma classe acessa mais atributos e métodos de outra classe do que os seus próprios. Métodos com muitos parâmetros também são um tipo de code smells. O uso excessivo de variáveis globais cria um tipo de acoplamento indesejado. Já a utilização de tipos primitivos em vez de classes específicas pode prejudicar a clareza e a extensibilidade do código. Outros exemplos incluem objetos mutáveis; classes de dados, que contêm apenas atributos sem comportamento associado; e até mesmo comentários em excesso, que muitas vezes indicam que o código não está claro o suficiente.