Capítulo 7 -

7. Arquitetura

Introdução

A arquitetura de software refere-se à organização de alto nível de um sistema, focando em grandes unidades como pacotes, componentes, módulos, subsistemas, camadas ou serviços. Essas unidades são conjuntos de classes relacionadas que desempenham papéis essenciais no funcionamento do sistema. A arquitetura também envolve decisões críticas de projeto, como a escolha da linguagem de programação e do banco de dados, que são difíceis de reverter.

Exemplo de Aplicação:

Em um sistema de informações, o módulo de persistência é essencial para armazenar dados. Já em um sistema de diagnóstico de doenças usando inteligência artificial, o módulo de persistência é secundário, pois o foco está no processamento de dados e na tomada de decisões.

Debate Tanenbaum-Torvalds

Em 1992, Andrew Tanenbaum criticou o Linux por seguir uma arquitetura monolítica, onde todas as funções do sistema operam dentro de um único arquivo executável. Ele defendia a arquitetura microkernel, onde apenas funções essenciais permanecem no kernel, enquanto o restante roda como processos independentes.

Linus Torvalds, criador do Linux, rebateu que a arquitetura monolítica era funcional, enquanto a microkernel apresentava problemas. Anos depois, Torvalds reconheceu que o kernel do Linux havia se tornado grande e inchado, confirmando as previsões de Ken Thompson, um dos criadores do Unix.

Exemplo de Aplicação:

O debate ilustra a importância de escolher a arquitetura certa para um sistema. Enquanto a arquitetura monolítica é mais simples de implementar, a microkernel oferece maior modularidade e facilidade de manutenção.

Arquitetura em Camadas

A Arquitetura em Camadas é um dos padrões mais utilizados desde os anos 1960. Nela, as classes são organizadas em camadas hierárquicas, onde cada camada só pode acessar serviços da camada imediatamente inferior. Esse padrão é amplamente usado em protocolos de rede, como HTTP, TCP e IP.

Vantagens:

1. Reduz a complexidade do desenvolvimento.

2. Facilita a manutenção, reuso e substituição de componentes.

Exemplo de Aplicação:

Em um sistema de e-commerce, a arquitetura em camadas pode ser organizada da seguinte forma:

1. Camada de Apresentação: Interface do usuário (UI).
2. Camada de Negócio: Regras de negócio e lógica de processamento.
3. Camada de Dados: Persistência e recuperação de dados.

Arquitetura em Três Camadas

A Arquitetura em Três Camadas é uma variação da arquitetura em camadas, amplamente utilizada em sistemas corporativos. Ela divide o sistema em três camadas principais:

1. Camada de Interface com o Usuário: Responsável por interagir com o usuário.
2. Camada de Lógica de Negócio: Implementa as regras de negócio do sistema.
3. Camada de Dados: Responsável pela persistência e recuperação dos dados.

Exemplo de Aplicação:

Em um sistema bancário, a camada de interface permite que os usuários realizem transações, a camada de negócio valida as regras (como limites de saque), e a camada de dados armazena as informações das contas.

Microsserviços

A arquitetura de Microsserviços surgiu como uma alternativa à arquitetura monolítica, permitindo que grupos de módulos sejam executados independentemente, sem compartilhamento de memória. A comunicação entre os microsserviços ocorre por meio de interfaces públicas.

Vantagens:

1. Evolução Independente: Cada equipe pode liberar novas versões sem depender de outros times.
2. Escalabilidade: Permite escalar partes específicas do sistema conforme necessidade.
3. Flexibilidade Tecnológica: Cada microsserviço pode ser desenvolvido com tecnologias distintas.
4. Maior Resiliência: Falhas em um microsserviço não comprometem todo o sistema.

Exemplo de Aplicação:

Em um sistema de comércio eletrônico, os microsserviços podem ser organizados da seguinte forma:

1. Catálogo de Produtos: Gerencia informações sobre os produtos.

2. Carrinho de Compras: Gerencia os itens selecionados pelos usuários.

3. Pagamento: Processa transações financeiras.

Quando Não Usar Microsserviços?

Sistemas pequenos e de baixa complexidade podem se beneficiar mais de uma abordagem monolítica, evitando a complexidade adicional gerada pela comunicação entre serviços.

Arquitetura Publish/Subscribe (Pub/Sub)

A arquitetura Publish/Subscribe é um modelo de comunicação assíncrona baseado na publicação e assinatura de eventos. Nesse paradigma, os componentes são classificados como publicadores (publishers) e assinantes (subscribers).

Características Principais:

1. Desacoplamento: Publicadores e assinantes não precisam se conhecer diretamente.

2. Comunicação em Grupo: Um evento pode ser entregue a múltiplos assinantes.

3. Notificação Assíncrona: Os assinantes são notificados automaticamente quando um evento relevante ocorre.

4. Organização em Tópicos: Os eventos são categorizados em tópicos, permitindo que os assinantes escolham apenas os eventos de seu interesse.

Exemplo de Aplicação:

Em uma companhia aérea, o sistema de vendas de passagens pode publicar eventos sempre que uma venda é concluída. Esses eventos podem ser assinados por diferentes sistemas:

1. Sistema de Milhagens: Credita pontos na conta do passageiro.

2. Sistema de Marketing: Envia ofertas personalizadas.

3. Sistema de Contabilidade: Registra a transação financeira.

Vantagens do Pub/Sub:

1. Escalabilidade: Permite lidar com um grande volume de eventos.

2. Flexibilidade: Facilita a adição de novos assinantes.

3. Confiabilidade: Eventos podem ser reenviados se um assinante estiver offline.

Arquitetura Pipes & Filtros

A arquitetura Pipes & Filtros é baseada em componentes independentes (filtros) que processam dados e os passam para o próximo componente por meio de canais (pipes). Esse padrão é comum em sistemas de processamento de dados e transformação de informações.

Exemplo de Aplicação:

Em um sistema de processamento de imagens, os filtros podem realizar operações como redimensionamento, aplicação de filtros de cor e compressão. Cada filtro processa a imagem e passa o resultado para o próximo filtro.

Anti-padrão: Big Ball of Mud

O Big Ball of Mud é um anti-padrão arquitetural onde o sistema é uma "bagunça" sem estrutura definida. Esse tipo de arquitetura surge quando o sistema evolui sem planejamento, resultando em código difícil de manter e entender.

Exemplo de Aplicação:

Um sistema legado que foi sendo modificado ao longo dos anos sem uma arquitetura clara pode se tornar um Big Ball of Mud, onde as dependências entre módulos são caóticas e a manutenção é extremamente difícil.

Conclusão

A escolha da arquitetura de software é crucial para o sucesso de um sistema. Padrões como Arquitetura em Camadas, Microsserviços e Publish/Subscribe oferecem soluções para diferentes necessidades, desde sistemas corporativos até aplicações distribuídas. Por outro lado, anti-padrões como o Big Ball of Mud ilustram os riscos de não seguir boas práticas de arquitetura. A aplicação desses conceitos em cenários reais, como sistemas de e-commerce, companhias aéreas e processamento de dados, demonstra sua importância no desenvolvimento de software moderno.

Capítulo 9 -

9. Refactoring

Introdução

Refactoring é o processo de modificar o código para melhorar sua estrutura, legibilidade e manutenibilidade, sem alterar seu comportamento funcional. Essa prática é essencial para garantir a longevidade e evolução do software, especialmente em sistemas que passam por manutenções corretivas, evolutivas e adaptativas.

Exemplo de Aplicação:

Em um sistema de e-commerce, um método que calcula descontos e impostos pode se tornar complexo ao longo do tempo. O refactoring pode dividir esse método em partes menores, como calcularDesconto() e calcularImposto(), tornando o código mais fácil de entender e modificar.

O Envelhecimento do Software

Sistemas de software, como organismos vivos, envelhecem. Meir Lehman formulou as Leis da Evolução de Software, destacando que:

1. Um sistema precisa ser constantemente mantido para se adaptar ao ambiente.

2. A complexidade interna aumenta ao longo do tempo, a menos que medidas sejam tomadas para estabilizá-la.

Exemplo de Aplicação:

Um sistema legado de gestão de estoque pode se tornar difícil de manter devido à complexidade acumulada. O refactoring pode reorganizar o código, reduzindo a complexidade e facilitando a adição de novas funcionalidades.

O Papel do Refactoring

O refactoring melhora a manutenibilidade do software sem alterar seu comportamento. Isso pode incluir:

1. Dividir funções grandes em funções menores.

2. Renomear variáveis e métodos para maior clareza.

3. Mover métodos entre classes para melhor organização.

4. Criar interfaces para facilitar a extensão do sistema.

Exemplo de Aplicação:

Em um sistema de banco, o método processarTransacao() pode ser dividido em validarTransacao(), executarTransacao() e registrarTransacao(), melhorando a organização e a clareza do código.

Extração de Método

A Extração de Método consiste em extrair um trecho de código de um método e movê-lo para um novo método separado. Isso melhora a legibilidade e reduz a complexidade.

Exemplo de Aplicação:

Em um aplicativo Android, o método onCreate() pode ser refatorado para extrair a criação de tabelas em métodos separados:

```
// Antes do Refactoring
void onCreate(SQLiteDatabase database) {
  database.execSQL("CREATE TABLE IF NOT EXISTS ..."); // Tabela 1
  database.execSQL("CREATE TABLE IF NOT EXISTS ..."); // Tabela 2
  database.execSQL("CREATE TABLE IF NOT EXISTS ..."); // Tabela 3
}
```

```
// Após o Refactoring
void createTable1(SQLiteDatabase database) {
  database.execSQL("CREATE TABLE IF NOT EXISTS ...");
}

void createTable2(SQLiteDatabase database) {
  database.execSQL("CREATE TABLE IF NOT EXISTS ...");
}

void createTable3(SQLiteDatabase database) {
  database.execSQL("CREATE TABLE IF NOT EXISTS ...");
}

void onCreate(SQLiteDatabase database) {
  createTable1(database);
  createTable2(database);
  createTable3(database);
}
```

Benefícios:

1. Melhora a organização do código.

2. Facilita a manutenção e a adição de novas tabelas.

Movimentação de Método

A Movimentação de Método envolve mover um método de uma classe para outra, especialmente quando o método utiliza mais serviços da nova classe.

Exemplo de Aplicação:

No IntelliJ, o método averageAmongMedians foi movido da classe PlatformTestUtil para a classe ArrayUtil, que manipula vetores e está mais alinhada com sua funcionalidade.

```
// Antes do Refactoring
class PlatformTestUtil {
  public static long averageAmongMedians(long[] time, int part) {
    // Cálculo da média
  }
}
```

```
// Após o Refactoring
class ArrayUtil {
  public static long averageAmongMedians(long[] time, int part) {
    // Cálculo da média
  }
}
```

Benefícios:

1. Melhora a coesão das classes.

2. Reduz o acoplamento entre classes.

Prática de Refactoring

A prática de refactoring depende de uma sólida base de testes, especialmente testes de unidade. Existem duas abordagens principais:

Refactoring Oportunista:

Realizado durante a implementação de novas funcionalidades ou correções de bugs.

Exemplo:

Ao adicionar uma nova funcionalidade de pagamento, um método complexo pode ser refatorado para facilitar a implementação.

Refactoring Planejado:

Realizado quando são necessárias mudanças estruturais maiores, como reorganizar pacotes inteiros.

Exemplo:

Migrar um sistema monolítico para uma arquitetura de microsserviços.

Refactorings Automatizados

Muitas IDEs oferecem suporte a refactorings automatizados, como renomeação de métodos e movimentação de classes. Essas ferramentas verificam pré-condições para garantir que o refactoring não cause erros.

Exemplo de Aplicação:

No Eclipse, a renomeação de um método atualiza automaticamente todas as referências no código, evitando erros de compilação.

Code Smells

Code Smells são sinais de código de baixa qualidade. Abaixo estão alguns exemplos comuns:

Código Duplicado

A duplicação de código aumenta a complexidade e o esforço de manutenção. Pode ser eliminada com refactorings como Extração de Método ou Extração de Classe.

Exemplo de Aplicação:

Em um sistema de folha de pagamento, o cálculo de descontos pode estar duplicado em vários métodos. A extração de um método calcularDescontos() resolve o problema.

Variáveis Globais

Variáveis globais dificultam o entendimento do código e podem causar bugs inesperados. Devem ser substituídas por injeção de dependência ou encapsulamento.

Exemplo de Aplicação:

Em um sistema de gerenciamento de pedidos, uma variável global totalPedidos pode ser substituída por um atributo encapsulado na classe Pedido.

Obsessão por Tipos Primitivos

O uso excessivo de tipos primitivos (como int, String) pode ser substituído por classes especializadas.

Exemplo de Aplicação:

Em vez de usar String para representar um CEP, pode-se criar uma classe CEP com validações e métodos específicos.

Objetos Mutáveis

Objetos mutáveis podem causar efeitos colaterais inesperados. Sempre que possível, deve-se preferir objetos imutáveis.

Exemplo de Aplicação:

Em Java, a classe String é imutável, garantindo segurança em operações concorrentes.

Classes de Dados

Classes que contêm apenas atributos e getters/setters podem ser melhoradas com a adição de comportamentos.

Exemplo de Aplicação:

Uma classe Cliente pode incluir métodos como validarCPF() e calcularIdade().

Comentários

Comentários excessivos podem indicar código ruim. Em vez de comentar, é melhor refatorar o código para torná-lo autoexplicativo.

Exemplo de Aplicação:

Um método com comentários como "Calcula o desconto" pode ser substituído por um método chamado calcularDesconto().

Dívida Técnica

Dívida técnica refere-se a problemas acumulados no código, como falta de testes e code smells. Eliminar a dívida técnica traz benefícios a longo prazo, como maior facilidade de manutenção e menor custo de desenvolvimento.

Exemplo de Aplicação:

Em um sistema de gestão de estoque, a dívida técnica pode ser reduzida com a refatoração de métodos complexos e a adição de testes automatizados.