Universidade Católica de Minas Gerais

Projeto de Software
2024

Prof. João Paulo Carneiro Aramuni
Aluno: Jhonatan Gutemberg Rosa Ferreira

Resenha do livro:

Engenharia de Software Moderna(Cap. 6 e 7)

Marco Tulio Valente

O livro começa abordando os principais temas que serão abordados como os padrões. Os padrões de projetos foram divididos em três categorias: Criacionais, estruturais e comportamentais. Os criacionais são aqueles que tendem a possibilitar um maior flexibilidade na criação de objetos. Os estruturais buscam proporcionar uma maior facilidade para composição de classes e objetos. Já os comportamentais propõem solução flexíveis para interação e divisão de responsabilidade entre classes e objetos.

O padrão de projeto Fábrica, segundo o autor, a utilização de um método estático que cria e retorna objetos de uma determinada classe e também oculta o tipo desses objetos por trás de uma interface. Existe também, outra forma de aplicação deste padrão que é a com a utilização de classes abstratas que é responsável por armazenar todos os métodos de determinadas. O Singleton é um padrão que usa apenas uma instância de objeto.Alguns autores como os do livro "Gangue dos Quatro", este padrão pode camuflar a criação de variáveis globais, mas o uso adequado possibilita que a instância gerada seja uma variável global e pode ser alterada m qualquer parte do código.

O Singleton o conceito se aplica em ocasiões em que os recursos devem possuir no máximo uma única instância, porém o seu uso expressivo deste padrão pode ocorrer quando ele é adotado como um artifício para criação de variáveis globais.

O Proxy introduz um objeto intermediário entre o objeto base e seus clientes. Com isso, os clientes não referenciam diretamente o objeto base, mas interagem com esse objeto intermediário. Isso permite maior controle sobre o acesso ao objeto base, possibilitando, por exemplo, a implementação de um cache, que reduz o tráfego de dados entre cliente e servidor. Além disso, outros requisitos não funcionais, como autenticação ou logging, podem ser incorporados na classe intermediária, sem alterar o objeto base diretamente.

O Adapter é um padrão de projeto utilizado quando há a necessidade de converter a interface de uma classe para outra interface esperada pelo cliente. Um exemplo comum é o de projetores de diferentes marcas, que geralmente não seguem um código padronizado, apresentando métodos e funcionalidades distintos. O Adapter resolve esse problema criando uma interface intermediária que chama os métodos equivalentes do objeto que está sendo adaptado, permitindo que a funcionalidade solicitada pelo cliente seja executada de forma transparente. Dessa forma, o padrão Adapter facilita a integração entre classes com interfaces incompatíveis.

No padrão Fachada é utilizado uma classe que facilita o uso de classes desconhecidas através do uso de interface mais simples e usual. Com essa facilitação o usuário não precisa conhecer todas as classes internas do sistema, apenas interagir com a classe de

fachada. Observe o exemplo trazido pelo autor:

```
class InterpretadorX {

  private String arq;

  InterpretadorX(arq) {
    this.arq = arq;
  }

  void eval() {
    Scanner s = new Scanner(arq);
    Parser p = new Parser(s);
    AST ast = p.parse();
    CodeGenerator code = new CodeGenerator(ast);
    code.eval();
  }
}
```

Sendo chamado desta forma:

```
new InterpretadorX("prog1.x").eval();
```

O decorador é um padrão é adequado para utilizar quando existe a necessidade de herança, mas em situações onde apenas algumas funcionalidade precisam ser herdadas.O decorador utiliza de composição para adicionar as funcionalidades dinamicamente nas classes base.

O padrão Decorador permite adicionar funcionalidades a objetos de forma dinâmica, sem alterar suas classes originais. No exemplo apresentado, um Channel (como TCPChannel ou UDPChannel) é criado e, em seguida, decorado com funcionalidades adicionais por meio de classes decoradoras, como ZipChannel e BufferChannel. Essas classes decoradoras são subclasses de ChannelDecorator, que implementa a interface Channel e delega as chamadas aos métodos send e receive para o objeto decorado. Assim, cada decorador adiciona sua funcionalidade específica antes de passar a execução para o próximo decorador na cadeia. Por exemplo, ZipChannel compacta a mensagem antes de transmiti-la via TCPChannel, seguindo o fluxo de chamadas que garante a execução correta das funcionalidades decoradas.

O Strategy é utilizado quando se deseja que uma classe tenha comportamentos intercambiáveis sem a necessidade de modificar o código-fonte original. Ele encapsula diferentes algoritmos dentro de classes separadas, permitindo que o cliente escolha qual algoritmo usar no momento da execução. Dessa forma, quando o usuário precisa alterar um método

ou o comportamento de uma classe, ele pode simplesmente fornecer uma nova implementação do algoritmo sem modificar a estrutura da classe, promovendo flexibilidade e manutenção mais fácil no código. Logo abaixo, a imagem ilustra uma aplicação do padrão:

```
class MyList {

  ... // dados de uma lista
  ... // métodos de uma lista: add, delete, search

  private SortStrategy strategy;

  public MyList() {
    strategy = new QuickSortStrategy();
  }

  public void setSortStrategy(SortStrategy strategy) {
    this.strategy = strategy;
  }

  public void sort() {
    strategy.sort(this);
  }
}
```

O padrão Observador é ideal para o cenário descrito, onde temos classes como Temperatura (sujeito) e Termômetro (observador), com a necessidade de desacoplar a lógica de monitoramento dos termômetros das interfaces visuais. Ao adotar esse padrão, os termômetros se registram como observadores de uma instância de Temperatura e são notificados quando a temperatura muda, através do método notifyObservers. Isso permite que o sistema seja extensível para diferentes interfaces (como web ou móvel) e sensores (como pressão e umidade) sem alterar a lógica do modelo, promovendo reutilização e flexibilidade ao implementar novos tipos de observadores para diferentes sujeitos.

O padrão Template Method é uma solução eficiente para padronizar o cálculo de salários em um sistema de folha de pagamento que envolve diferentes tipos de funcionários. No exemplo, a classe abstrata Funcionario define o esqueleto do cálculo salarial através do método calcSalarioLiquido, que executa etapas comuns, como o cálculo de descontos previdenciários, plano de saúde e outros descontos.Os detalhes desses cálculos são específicos para cada tipo de funcionário, com isso ficam delegados às subclasses FuncionarioPublico e FuncionarioCLT através de métodos abstratos. Isso permite flexibilidade para adaptações específicas nas subclasses, promovendo reutilização de código e facilitando futuras customizações.

O padrão de projeto Visitor é uma solução boa para adicionar operações a uma família de objetos sem modificar suas classes, especialmente em linguagens com despacho simples

de métodos, como Java. A solução envolve a implementação de um método accept em cada classe da hierarquia, que recebe um objeto do tipo Visitor. Esse método chama a operação visit do Visitor, passando o próprio objeto como parâmetro. Com isso, o compilador identifica o tipo específico do objeto e chama o método correto. O Visitor centraliza operações relacionadas, como imprimir ou persistir dados, sem alterar a hierarquia das classes originais. Para melhor apresentar as ideia discutidas o autor apresenta o código abaixo:

```
abstract class Veiculo {
  abstract public void accept(Visitor v);
}

class Carro extends Veiculo {
  ...
  public void accept(Visitor v) {
    v.visit(this);
  }
  ...
}

class Onibus extends Veiculo {
  ...
  public void accept(Visitor v) {
    v.visit(this);
  }
  ...
}

// Idem para Motocicleta
```

O padrão Visitor apresenta uma limitação significativa: a introdução de novas classes na hierarquia, como Caminhão, exige a atualização de todos os visitantes com um novo método visit. Além disso, ele pode comprometer o encapsulamento ao exigir que as classes expõem seu estado interno para permitir o acesso do Visitor.

Outros padrões de projetos como o Iterador e Builder são amplamente utilizados. O iterador padroniza uma interface para caminhar sobre uma estrutura de dados, isso permite concorrer um estrutura de dados sem conhecer o seu tipo concreto. Já o builder facilita a instaciação de objetos que possuem muitos atributos, e alguns deles opcionais.

O padrão de projeto Builder facilita a instanciação de objetos que possuem muitos atributos, incluindo aqueles opcionais. Em vez de criar diversos construtores para lidar com combinações diferentes de parâmetros, o Builder permite inicializar os atributos por meio de métodos set, que deixam claro quais atributos estão sendo definidos. Isso torna o processo de construção mais organizado e evita a confusão com a ordem dos parâme-

tros,preserva o encapsulamento, permitindo que os atributos sejam configurados apenas no momento da criação do objeto.

Para finalizar o capítulo 6, o autor faz o seguinte questionamento, "Quando não utilizar padrões de projeto?"Os padrões de projeto têm como objetivo tornar sistemas mais flexíveis e adaptáveis. Porem, seu uso também envolve custos, como a criação de classes adicionais,o que pode tornar o projeto maior. Antes de adotar um padrão como Fábrica ou Strategy, é importante avaliar se a flexibilidade oferecida justifica a complexidade extra. O uso exagerado de padrões, sem uma demanda clara, pode levar ao que se chama de "paternite", onde o excesso de padrões prejudica mais do que ajuda no desenvolvimento do sistema.

No capítulo 7 o autor aborda alguns temas de arquitetura como Arquitetura em três camadas, arquitetura MVC e arquitetura baseada em Microsserviços.Também abordam padrões arquiteturais usados para garantir escalabilidade e desacoplamento em sistemas distribuídos: Filas de Mensagens e Publish/Subscribe.

Boa parte das arquitetura existentes hoje vieram do debate Tanenbaum-Torvalds, ocorrido em 1992, foi uma troca acalorada de ideias sobre a arquitetura de sistemas operacionais entre Andrew Tanenbaum e Linus Torvalds. Tanenbaum criticou a arquitetura monolítica do Linux, argumentando que o futuro dos sistemas operacionais estava nos microkernels, que separavam as funções do sistema em processos independentes. Ken Thompson comentou que, embora mais simples de implementar, kernels monolíticos tendem a se complicar com o tempo, o que se comprovou quando, em 2009, Torvalds admitiu que o kernel do Linux havia se tornado grande e inchado, mostrando que decisões arquiteturais podem ter efeitos negativos que só aparecem anos depois.

A arquitetura em camadas é um padrão amplamente utilizado desde as décadas de 1960 e 1970 para organizar sistemas de software em módulos hierárquicos. Esse padrão facilita o desenvolvimento ao dividir a complexidade em componentes menores e disciplinar as dependências entre as camadas. Um exemplo clássico sitado pelo autour, é o uso em protocolos de rede, como HTTP, TCP e IP. A arquitetura em camadas permite a substituição ou reutilização de camadas, aumentando a flexibilidade e o reúso de componentes. Edsger Dijkstra, em 1968, foi um dos primeiros a propor essa estrutura hierárquica em seu sistema operacional THE.

A arquitetura em três camadas é utilizada na construção de sistemas de informação corporativos. Essa arquitetura é composta por três camadas distintas: a Camada de Interface com o Usuário, que gerencia a interação e exibição de informações ao usuário. A Camada de Lógica de Negócio, responsável por implementar as regras do sistema.Por fim a Camada de Banco de Dados, que armazena e gerencia os dados manipulados.

Essa estrutura promove uma clara separação de responsabilidades, facilita a manutenção e escalabilidade do sistema. As camadas operam de maneira distribuída, com a interface executando nos clientes, a lógica de negócio em um servidor de aplicação e o banco de dados armazenando as informações, proporcionando assim uma arquitetura mais flexível e eficiente para aplicações corporativas.

O padrão arquitetural MVC (Model-View-Controller) surgiu no final da década de 70 e foi implementado em Smalltalk-80, uma das primeiras linguagens orientadas a objetos a utilizar interfaces gráficas. O MVC divide as classes de um sistema em três componentes principais: Modelo, responsável por gerenciar os dados e a lógica de domínio sem depender das classes de interface. Já a de Visão, é responsável pela apresentação gráfica, incluindo janelas, botões e outros elementos de interação. A camada Controladora interpreta os eventos do usuário, como cliques de mouse e teclas, e coordena as interações entre a Visão e o Modelo.

Embora muitas implementações não façam uma distinção clara entre Visão e Controladora, pode-se entender que a interface gráfica é composta por ambos, enquanto o Modelo permanece independente, permitindo uma atualização automática da interface sempre que o estado dos dados mudar. Essa separação de responsabilidades facilita a manutenção e a evolução dos sistemas, promovendo uma arquitetura mais flexível e escalável.

O padrão MVC (Model-View-Controller) favorece a especialização do trabalho de desenvolvimento ao permitir que as classes de Modelo sejam reutilizadas em múltiplas Visões. Isso significa que um único objeto de Modelo, que armazena dados, como horas e minutos, pode ser utilizado em diferentes representações visuais, como um relógio analógico e um relógio digital. Como resultado, equipes de desenvolvimento podem se concentrar em partes específicas do sistema, melhorando a eficiência e a colaboração entre os membros da equipe.

A transição de arquiteturas monolíticas para microsserviços é um tema central na discussão sobre agilidade e escalabilidade no desenvolvimento de software. O autor apresenta de forma clara os desafios enfrentados por sistemas monolíticos, como a interdependência entre módulos e a burocracia nos processos de lançamento, que dificultam a implementação de atualizações rápidas. Mas também, essa abordagem dos microsserviços se destaca ao permitir a execução independente de módulos, reduzindo o risco de efeitos colaterais e possibilitando implantações rápidas e flexíveis.

A escalabilidade fina e a adoção de tecnologias diversificadas para cada serviço fortalecem a eficiência do sistema como um todo. O suporte das plataformas de computação em nuvem é facilitada com a adoção dessa arquitetura e permiti que as empresas se concentrem na inovação, alinhando-se à Lei de Conway.

No gerenciamento de dados a autonomia dos microsserviços em relação ao gerenciamento de dados é um aspecto fundamental para garantir a flexibilidade e a eficiência das arquiteturas modernas. A proposta de que cada microsserviço opere com seu próprio banco de dados elimina a dependência de um administrador central, que muitas vezes atua como um gargalo no desenvolvimento. Em um cenário onde dois microsserviços compartilham um único banco de dados, como destacado no texto, o processo de implementação de mudanças torna-se lento e burocrático, pois qualquer alteração requer a coordenação entre diferentes equipes, o que pode gerar conflitos de prioridades.

Cada equipe pode inovar e evoluir seu serviço sem a necessidade de validações externas, promovendo uma maior agilidade e autonomia no desenvolvimento. O uso de banco de dados independentes não apenas acelera a entrega de novas funcionalidades, mas tam-

bém minimiza o risco de falhas decorrentes de interferências indesejadas entre os serviços, permitindo uma evolução mais harmoniosa e contínua do sistema.

Embora a arquitetura de microsserviços ofereça vantagens significativas, como escalabilidade e flexibilidade, o autor deixa o questionamento "Quando não usar microsserviços?". Cada microsserviço opera de forma independente, o que gera desafios típicos de sistemas distribuídos, como a comunicação entre serviços, que requer o uso de protocolos como HTTP/REST, aumentando a necessidade de conhecimentos técnicos diversos.

A latência se torna uma preocupação, pois as chamadas entre serviços localizados em máquinas diferentes podem resultar em atrasos que afetam o desempenho. Outro aspecto crítico é a dificuldade em gerenciar transações distribuídas, onde garantir a atomicidade das operações que envolvem múltiplos bancos de dados se torna mais complexo, exigindo o uso de protocolos específicos como o two-phase commit. A implementação demanda uma abordagem cuidadosa e uma compreensão profunda dos desafios associados.

Em se tratar de arquitetura orientada a mensagens as filas de mensagens desempenham um papel crucial na comunicação entre clientes e servidores em sistemas distribuídos, atuando como intermediárias que permitem a troca assíncrona de informações. Nessa arquitetura, os clientes se tornam produtores de mensagens, inserindo dados na fila, enquanto os servidores agem como consumidores, retirando e processando essas mensagens em uma estrutura FIFO (first in, first out). Essa abordagem não apenas libera os clientes para continuar seu processamento após a inserção de mensagens, mas também permite um desacoplamento significativo entre os componentes do sistema.

O desacoplamento no espaço assegura que clientes e servidores não precisem conhecer uns aos outros, enquanto o desacoplamento no tempo permite que mensagens sejam armazenadas na fila, mesmo quando um dos componentes está offline. Essa robustez e flexibilidade são essenciais, especialmente em ambientes de desenvolvimento ágeis, onde diferentes equipes podem evoluir seus sistemas de forma independente, desde que o formato das mensagens permaneça consistente.

As arquiteturas Publish/Subscribe oferecem um modelo eficiente para a troca de informações em sistemas distribuídos, caracterizando-se pela utilização de eventos como mensagens. Nesse modelo, os publicadores geram eventos e os disponibilizam em um serviço específico, que geralmente opera em uma máquina separada, enquanto os assinantes, que previamente manifestaram interesse em determinados eventos, são notificados quando esses ocorrem. O desacoplamento tanto no espaço quanto no tempo, semelhante ao que é observado em sistemas baseados em filas de mensagens.

As arquiteturas Publish/Subscribe se distinguem pela comunicação em grupo, onde um único evento pode acionar múltiplas notificações para diferentes assinantes, ao contrário das filas de mensagens, onde cada mensagem é consumida por um único servidor. Além disso, a notificação dos assinantes ocorre de forma assíncrona, permitindo que continuem seu processamento até que o evento de interesse seja disparado. Os eventos podem ser organizados em tópicos, facilitando a gestão e a assinatura de informações relevantes, o que aumenta a eficiência do sistema.

Os padrões arquiteturais Pipes e Filtros é uma arquitetura orientada a dados que se baseia na execução de programas chamados de filtros, responsáveis por processar dados de entrada e gerar uma nova saída. Esses filtros são interconectados por meio de pipes, que funcionam como buffers, armazenando temporariamente a saída de um filtro até que o próximo filtro na sequência a leia. Ela permite que os filtros operem de maneira independente, sem a necessidade de conhecer seus predecessores ou sucessores, o que proporciona flexibilidade e diversas combinações de programas.

A arquitetura possibilita a execução paralela dos filtros, aumentando a eficiência do processamento. Um exemplo clássico dessa arquitetura é encontrado nos sistemas Unix, onde comandos podem ser encadeados usando pipes, como demonstrado na linha de comando ls | grep csv | sort, que ilustra como diferentes filtros podem ser aplicados em sequência.

Um dos anti-padrões arquiteturais mais conhecidos é o big ball of mud (grande bola de lama), que se refere a sistemas com uma organização caótica em que os módulos se comunicam indiscriminadamente, resultando em uma rede complexa de dependências. Essa falta de uma arquitetura definida gera um código desordenado, dificultando a manutenção e aumentando o risco de bugs, pois alterações em um módulo podem impactar outros de maneiras imprevisíveis.

Um exemplo real desse problema é encontrado em um estudo de um sistema bancário que cresceu de 2,5 milhões para mais de 25 milhões de linhas de código ao longo dos anos. Apesar de tentativas de melhorar a situação com documentação e revisões de código, a complexidade da arquitetura desordenada continuou a dificultar o aprendizado de novos desenvolvedores e a implementação de novas funcionalidades, evidenciando como o big ball of mud pode se tornar um sério obstáculo para a evolução de sistemas complexos.