PUC Minas

Pontifícia Universidade Católica de Minas Gerais

Instituto de **C**iências **E**xatas e **I**nformática - **ICEI**

Engenharia de Software

Projeto de Software

Professor: Dr. João Paulo Aramuni

13 out. 2024

Padrões de Projeto e Arquitetura de Software: Uma Análise dos Capítulos 6 e 7 de "Engenharia de Software Moderna"

João Pedro Silva Braga

VALENTE, Marco Tulio. Capítulo 6: *Padrões de Projeto;* Capítulo 7*: Arquitetura*. ***Engenharia de Software Moderna.*** 1. ed. São Paulo: Editora, 2022. Disponível em: https://engsoftmoderna.info/. Acesso em: 13 out. 2024.

Os capítulos 6 e 7 do livro "Engenharia de Software Moderna", de Marco Tulio Valente, abordam os temas de Padrões de Projeto e Arquitetura de Software, respectivamente. Publicados em 2022, os capítulos buscam responder à seguinte pergunta fundamental: como projetar sistemas de software flexíveis, extensíveis e de fácil manutenção, considerando as constantes mudanças inerentes ao desenvolvimento de software?

A ideia central dos capítulos é que a utilização de padrões de projeto e a escolha de uma arquitetura de software adequada são cruciais para alcançar esse objetivo. Através da apresentação de diferentes padrões e estilos arquiteturais, o autor busca fornecer ao leitor um conjunto de soluções e ferramentas para lidar com os desafios do projeto de software moderno.

O capítulo 6 começa introduzindo o conceito de padrões de projeto e sua importância na criação de sistemas flexíveis e extensíveis. Em seguida, o autor detalha dez padrões de projeto: Fábrica, Singleton, Proxy, Adaptador, Fachada, Decorador, Strategy, Observador, Template Method e Visitor. Cada padrão é apresentado dentro de um contexto, mostrando um problema de projeto e a solução proposta pelo padrão. O capítulo conclui com uma breve descrição de outros padrões como Iterador e Builder e discute a importância de evitar o uso excessivo de padrões, a chamada "paternite".

O capítulo 7 foca na arquitetura de software, definindo o conceito e destacando a importância das decisões arquiteturais. O autor apresenta o debate Tanenbaum-Torvalds, ilustrando como a escolha da arquitetura pode impactar o desenvolvimento de um sistema a longo prazo. O capítulo explora diferentes estilos arquiteturais, como a arquitetura em camadas, a arquitetura MVC, microsserviços, arquiteturas orientadas a mensagens e a arquitetura publish/subscribe. Cada estilo é analisado em detalhes, com exemplos e

discussões sobre suas vantagens e desvantagens. O capítulo finaliza apresentando brevemente os padrões Pipes e Filtros e Cliente/Servidor, e descrevendo o anti-padrão "Big Ball of Mud" como um exemplo de arquitetura a ser evitada. Ambos os capítulos incluem seções de exercícios e perguntas frequentes para auxiliar na compreensão dos conceitos.

Na introdução do capítulo, seção 6.1, Valente define Padrões de Projeto como soluções para problemas recorrentes em projetos de software, inspiradas na obra do arquiteto Christopher Alexander. Para o autor, compreender um padrão de projeto exige entender o problema que ele resolve, o contexto em que surge e a solução proposta. Além de fornecerem soluções prontas, os padrões de projeto servem como um vocabulário comum entre desenvolvedores, facilitando a comunicação e a documentação de sistemas.

Valente argumenta que o principal benefício dos padrões de projeto é a criação de sistemas flexíveis e extensíveis, o que ele chama de "design for change". A negligência desse princípio pode levar à necessidade de um "reprojeto" completo do sistema no futuro. O autor defende que, ao utilizar padrões de projeto, o desenvolvedor se beneficia de soluções já testadas e validadas, criando sistemas mais robustos e adaptáveis às mudanças.

A seção 6.2 apresenta o padrão de projeto Fábrica (Factory), que visa flexibilizar a criação de objetos em um sistema. Demonstrando, como o operador "new", presente em linguagens como Java e C++, pode ser inflexível ao exigir o nome literal da classe a ser instanciada, limitando a adaptabilidade do código a diferentes tipos de objetos.

A solução proposta pelo padrão Fábrica é a criação de um método estático que encapsula a instanciação do objeto, ocultando seu tipo concreto e permitindo a criação de objetos de diferentes classes através de uma interface comum.

Valente exemplifica o padrão Fábrica com a criação de canais de comunicação, inicialmente limitados ao protocolo TCP. A introdução do padrão permite que o código seja facilmente adaptado para usar outros protocolos, como UDP, sem a necessidade de modificar as funções que utilizam os canais.

Sendo assim, o método Fábrica funciona como um "aspirador" de "new", centralizando a criação de objetos e facilitando a manutenção do código. Além disso, a variação "Fábrica Abstrata", que usa uma classe abstrata para centralizar múltiplos métodos fábrica, aumentando ainda mais a flexibilidade do sistema, também é apresentada.

A seção 6.3 explora o padrão Singleton, que garante a existência de apenas uma instância de uma classe específica durante a execução do programa. Valente demonstra como essa restrição é útil em cenários como o registro de operações em um sistema (logging), onde múltiplas instâncias podem levar a conflitos e perda de dados. O autor detalha a implementação do padrão Singleton, utilizando um construtor privado para impedir a instanciação direta da classe e um método estático público que retorna a única instância existente, criando-a se ainda não existir.

Apesar de sua utilidade, o autor reconhece que o Singleton é o padrão mais controverso, pois pode ser usado para camuflar variáveis globais, prejudicando o encapsulamento e o acoplamento entre classes. Ele adverte contra o uso indiscriminado do padrão, recomendando sua aplicação apenas em casos onde a unicidade da instância é conceitualmente justificada, como no exemplo do logging. O autor conclui que, embora facilite a modelagem de recursos únicos, o Singleton pode complicar os testes automáticos,

pois o resultado de um método pode depender do estado global armazenado na instância única.

A seção 6.4 apresenta o padrão Proxy, que introduz um objeto intermediário, o "proxy", entre um objeto base e seus clientes. O proxy, que implementa a mesma interface do objeto base, atua como um intermediário, controlando o acesso ao objeto original e adicionando funcionalidades extras de forma transparente. O texto ilustra o padrão com um sistema de busca de livros que se beneficia da adição de um cache para melhorar o desempenho, sem modificar a classe de busca original.

O autor destaca que o proxy permite a separação clara de responsabilidades, delegando a implementação do cache a um objeto dedicado. Essa separação facilita a manutenção e evolução do sistema, permitindo a implementação de requisitos não-funcionais, como o cache, sem impactar o código principal. Dessa forma, o Proxy é um padrão versátil, útil para implementar funcionalidades como comunicação remota, alocação de memória por demanda e controle de acesso, além de promover o baixo acoplamento entre classes.

A seção 6.5 explora o padrão Adaptador (Adapter), também conhecido como Wrapper, que possibilita a comunicação entre classes com interfaces incompatíveis. Valente ilustra o problema com um sistema de controle de projetores multimídia, onde cada fabricante utiliza métodos diferentes para ligar seus projetores. O autor propõe a criação de uma interface unificada para controlar todos os projetores, mas se depara com a incompatibilidade com as classes existentes.

A solução apresentada pelo padrão Adapter é a criação de classes adaptadoras que "traduzem" a interface unificada para os métodos específicos de cada projetor. Cada classe adaptadora implementa a interface unificada e, internamente, delega as chamadas aos métodos correspondentes da classe do projetor específico. O Adapter permite a integração de classes de terceiros, sem necessidade de modificar seu código fonte, promovendo a reutilização de código e a flexibilidade do sistema.

Na seção 6.6, Valente apresenta o padrão Fachada (Facade), que simplifica a utilização de um sistema complexo, oferecendo uma interface unificada e de alto nível para os seus clientes. Ele usa o exemplo de um interpretador para uma linguagem de consulta a dados, onde a execução de um programa exige a interação com várias classes internas do sistema. O autor destaca que essa complexidade pode ser um obstáculo para os usuários, exigindo conhecimento detalhado do funcionamento interno do interpretador.

A solução proposta pelo padrão Fachada é a criação de uma classe que encapsula as classes internas do interpretador, oferecendo métodos simplificados para a execução de programas. Essa classe "fachada" abstrai a complexidade do sistema, facilitando sua utilização pelos clientes. O padrão Fachada promove o baixo acoplamento entre o sistema e seus clientes, protegendo-os de mudanças na implementação interna e tornando o sistema mais fácil de usar e entender.

A seção 6.7 apresenta o padrão Decorator como uma alternativa flexível à herança para adicionar responsabilidades a objetos em tempo de execução. Valente retoma o exemplo do sistema de comunicação, introduzindo a necessidade de adicionar funcionalidades opcionais aos canais de comunicação, como buffers, compactação e logging. O autor demonstra que a utilização da herança para implementar cada combinação

de funcionalidades levaria a uma explosão no número de subclasses, tornando o código complexo e difícil de manter.

O padrão Decorator soluciona esse problema através da composição de objetos. Ele define uma classe base abstrata que encapsula o objeto a ser decorado e delega as chamadas aos seus métodos. As classes decoradoras, que herdam da classe base, adicionam funcionalidades específicas, encapsulando o objeto decorado e chamando o método correspondente do objeto encapsulado. A abordagem permite a combinação dinâmica de funcionalidades, evitando a criação de inúmeras subclasses e tornando o código mais flexível e extensível. O autor conclui que o Decorator oferece uma solução elegante para a personalização de objetos em tempo de execução, respeitando o princípio "Prefira Composição a Herança".

A seção 6.8 apresenta o padrão Strategy como uma forma de encapsular algoritmos e torná-los intercambiáveis, permitindo que uma classe seja configurada com diferentes estratégias em tempo de execução. O exemplo utilizado é a implementação de uma classe de lista ordenada, onde o algoritmo de ordenação (quicksort, mergesort, etc.) pode ser escolhido pelo cliente. O autor demonstra que implementar todos os algoritmos diretamente na classe da lista a tornaria inflexível e violaria o princípio Aberto/Fechado.

O padrão propõe a criação de uma interface abstrata que define o método de ordenação e classes concretas que implementam essa interface para cada algoritmo. A classe da lista, por sua vez, recebe uma instância da estratégia de ordenação através de um método setter e delega a ordenação ao objeto da estratégia. Essa abordagem desacopla a classe da lista dos algoritmos específicos, permitindo a adição de novas estratégias sem modificar o código da lista, tornando o sistema mais flexível e extensível.

O padrão Observador (Observer) na seção 6.9 é apresentado como uma solução para implementar uma relação de dependência um-para-muitos entre objetos, onde um objeto, o "sujeito", notifica automaticamente outros objetos, os "observadores", sobre mudanças em seu estado. O exemplo utilizado é um sistema de monitoramento de temperatura, onde um objeto "Temperatura" precisa notificar vários "Termômetros" quando seu valor é atualizado. O autor destaca que acoplar a classe "Temperatura" às diversas implementações de "Termômetro" seria inadequado, pois limitaria a reutilização da classe "Temperatura" e exigiria modificações constantes em seu código.

O padrão Observer soluciona esse problema definindo uma interface "Observador" com um método "update" e uma classe "Sujeito" que mantém uma lista de observadores e oferece métodos para adicioná-los e removê-los. Quando o estado do sujeito muda, ele chama o método "notifyObservers", que por sua vez invoca o método "update" de cada observador registrado. Valente conclui que o Observer promove o baixo acoplamento entre sujeito e observadores, facilitando a reutilização de código e a evolução do sistema. Além disso, o padrão permite a implementação de diferentes tipos de observadores para um mesmo sujeito, tornando o sistema mais flexível.

A seção 6.10 introduz o padrão Template Method, que define o esqueleto de um algoritmo em uma classe abstrata, permitindo que subclasses redefinam ou implementem etapas específicas do algoritmo sem alterar sua estrutura principal. Valente ilustra o padrão com um exemplo de cálculo de folha de pagamento, onde diferentes tipos de funcionários possuem regras específicas para o cálculo de descontos e do salário líquido. O autor demonstra como o padrão Template Method permite definir um método "template" na classe

abstrata "Funcionario", que descreve a sequência geral do cálculo, delegando a implementação de algumas etapas para métodos abstratos a serem implementados pelas subclasses.

Essa abordagem, conhecida como "inversão de controle", permite que "código antigo" (a classe abstrata) chame "código novo" (métodos das subclasses), tornando o sistema mais flexível e extensível. Template Method é particularmente útil na implementação de frameworks, onde a classe abstrata define a estrutura geral do sistema e as subclasses a customizam de acordo com suas necessidades específicas, sem modificar o código do framework.

A seção 6.11 aborda o padrão Visitor, que permite adicionar novas operações a uma hierarquia de classes sem modificar o código das classes existentes. O padrão é exemplificado utilizando o sistema de estacionamento, onde novas funcionalidades, como imprimir informações dos veículos, persistir os dados ou enviar mensagens aos proprietários, precisam ser adicionadas sem alterar as classes que representam os tipos de veículos (Carro, Ônibus, Motocicleta). O autor demonstra como a implementação direta dessas funcionalidades em cada classe violaria o princípio Aberto/Fechado e tornaria o código repetitivo e difícil de manter.

O padrão Visitor soluciona esse problema criando uma interface "Visitor" com um método "visit" para cada classe da hierarquia. As classes concretas que implementam essa interface definem as operações específicas para cada tipo de veículo. A hierarquia de veículos, por sua vez, recebe um método "accept" que recebe um "Visitor" como parâmetro e chama o método "visit" correspondente ao seu tipo. Valente conclui que o Visitor permite adicionar novas operações de forma organizada, centralizando-as em classes "Visitor" e evitando a modificação das classes da hierarquia original. Apesar de suas vantagens, o autor reconhece que o padrão pode comprometer o encapsulamento, expondo o estado interno dos objetos visitados para permitir a execução das operações.

A seção 6.12 apresenta brevemente dois padrões de projeto adicionais: Iterador e Builder. O padrão Iterador oferece uma interface padronizada para percorrer elementos de uma estrutura de dados sequencialmente, sem expor sua implementação interna. Isso permite a criação de laços de iteração genéricos, que funcionam com diferentes tipos de estruturas de dados, além de possibilitar múltiplas iterações simultâneas sobre a mesma estrutura.

Já o padrão Builder simplifica a criação de objetos com múltiplos atributos, especialmente quando alguns atributos são opcionais. Em vez de criar diversos construtores com diferentes combinações de parâmetros, o Builder utiliza métodos "setter" para configurar os atributos do objeto e um método final "build" para construir o objeto com os valores configurados. Isso torna o código mais legível, evita erros de ordem dos parâmetros e facilita a adição de novos atributos no futuro.

Valente conclui o capítulo 6 na seção 6.13 com um aviso: padrões de projeto não são balas de prata e seu uso indiscriminado pode trazer mais problemas do que soluções. O autor destaca que cada padrão possui um custo em termos de complexidade de código e que a aplicação de um padrão deve ser cuidadosamente analisada, considerando os benefícios em relação aos custos. Ele usa os exemplos de Fábrica e Strategy para demonstrar que a aplicação desses padrões só se justifica se houver a real necessidade de criar objetos de diferentes tipos ou de utilizar diferentes algoritmos, respectivamente.

Além disso, o autor alerta para o problema da "paternite", que é o uso excessivo e desnecessário de padrões de projeto, tornando o código complexo e difícil de entender. Ele cita John Ousterhout, que critica a superaplicação de padrões e defende que a simplicidade e a clareza do código devem ser priorizadas. Valente conclui que a decisão de usar um padrão de projeto deve ser baseada em uma análise cuidadosa do problema e do contexto, buscando o equilíbrio entre flexibilidade, extensibilidade e simplicidade do código.

A introdução do capítulo 7 mergulha no conceito de arquitetura de software, que Valente define como o "projeto em alto nível", focando em unidades maiores como módulos, subsistemas e camadas, e as relações entre eles. O autor destaca que a arquitetura deve refletir as decisões mais importantes do projeto, como a escolha de linguagem de programação e banco de dados, pois essas escolhas são difíceis de reverter posteriormente.

Para ilustrar a relevância dessas decisões, Valente apresenta o famoso debate Tanenbaum-Torvalds sobre a arquitetura de sistemas operacionais. O debate, que colocava em oposição a arquitetura monolítica do Linux e a arquitetura baseada em microkernel defendida por Tanenbaum, demonstra como a escolha da arquitetura impacta o desenvolvimento, a manutenção e a performance de um sistema a longo prazo, com consequências que podem se manifestar anos depois. A conclusão é que as decisões arquiteturais devem ser tomadas com cuidado e levando em consideração o futuro do sistema.

A seção 7.2 apresenta a Arquitetura em Camadas, um padrão clássico que organiza as classes do sistema em módulos hierárquicos chamados camadas. Valente explica que a comunicação entre as camadas se dá de forma restrita: uma camada pode usar serviços da camada imediatamente inferior, mas não de camadas mais abaixo. O autor utiliza a implementação de protocolos de rede como exemplo, mostrando como HTTP, TCP e IP são organizados em camadas que se comunicam de forma hierárquica.

O autor destaca as vantagens da arquitetura em camadas, como a modularidade, que facilita o desenvolvimento e a manutenção, e o encapsulamento, que limita as dependências entre as camadas. Valente também aborda a Arquitetura em Três Camadas, uma variação comum em sistemas de informação, com camadas de Interface com o Usuário, Lógica de Negócio e Banco de Dados. Ele conclui que a arquitetura em camadas, apesar de ser um padrão antigo, continua sendo uma solução eficaz para organizar sistemas complexos, promovendo a modularidade, o reuso e a flexibilidade.

A arquitetura MVC (Model-View-Controller) é explorada na seção 7.3, descrita como um padrão que separa a aplicação em três componentes interconectados: o Modelo, responsável pelos dados e pela lógica de negócio; a Visão, que apresenta os dados ao usuário; e o Controlador, que gerencia a interação do usuário e atualiza o Modelo e a Visão. O autor contextualiza a origem do MVC no desenvolvimento de interfaces gráficas com o Smalltalk-80 e destaca seus benefícios, como a especialização do trabalho, a reutilização de código e a testabilidade.

Valente também discute a aplicação do MVC em sistemas Web, onde a Visão é composta por páginas HTML, o Controlador gerencia as requisições HTTP e o Modelo interage com o banco de dados. Ele esclarece a confusão comum entre MVC e arquitetura em três camadas, explicando que o MVC pode ser usado para implementar a camada de apresentação em uma arquitetura de três camadas. O autor conclui que o MVC, em suas

diferentes variantes, é um padrão fundamental para o desenvolvimento de aplicações interativas, tanto em interfaces gráficas tradicionais quanto na Web.

A seção 7.4 aborda a arquitetura de Microsserviços, uma abordagem que divide um sistema em pequenos serviços independentes que se comunicam através de uma rede. Valente contrasta essa abordagem com a arquitetura monolítica tradicional, onde todos os componentes do sistema executam em um único processo. Ele argumenta que, embora os métodos ágeis tenham acelerado o desenvolvimento, a arquitetura monolítica se torna um gargalo para a entrega frequente de novas releases, devido à dificuldade de testar e implantar um sistema grande e complexo.

O autor elenca as vantagens dos microsserviços, como a evolução independente de cada serviço, a escalabilidade granular, a possibilidade de usar diferentes tecnologias para cada serviço e a tolerância a falhas. Ele também discute os desafios dessa arquitetura, como a complexidade da comunicação entre serviços, o aumento da latência e a necessidade de lidar com transações distribuídas. O texto relaciona o sucesso da arquitetura de microsserviços com a Lei de Conway, que afirma que a arquitetura de um sistema tende a refletir a estrutura organizacional da empresa que o desenvolve. Ele conclui que a arquitetura de microsserviços é uma solução promissora para sistemas complexos e em constante evolução, mas exige uma mudança de mentalidade e a adoção de novas ferramentas e práticas de desenvolvimento.

Na seção 7.5, Valente introduz as Arquiteturas Orientadas a Mensagens, onde a comunicação entre clientes e servidores ocorre de forma assíncrona através de uma fila de mensagens. O autor explica que os clientes, atuando como produtores, enviam mensagens para a fila, enquanto os servidores, atuando como consumidores, leem e processam as mensagens da fila. Ele destaca o conceito FIFO (First In, First Out) como a base do funcionamento da fila, garantindo que as mensagens sejam processadas na ordem em que foram enviadas.

As filas de mensagens permitem o desacoplamento no espaço e no tempo entre clientes e servidores. O desacoplamento no espaço significa que clientes e servidores não precisam se conhecer ou estar localizados na mesma máquina, enquanto o desacoplamento no tempo permite que os clientes enviem mensagens mesmo quando os servidores estão indisponíveis. O autor conclui que essa arquitetura oferece diversas vantagens, como maior flexibilidade, escalabilidade e tolerância a falhas, sendo uma solução robusta para sistemas distribuídos complexos. Ele ainda menciona a existência de soluções prontas no mercado, como brokers de mensagens, que facilitam a implementação dessa arquitetura.

A seção 7.6 explora as Arquiteturas Publish/Subscribe, um modelo de comunicação assíncrona onde os publicadores (publishers) enviam mensagens, chamadas de eventos, para um serviço central, e os assinantes (subscribers) se registram para receber os eventos de seu interesse. Valente compara esse modelo com a arquitetura orientada a mensagens, salientando que, enquanto na fila de mensagens cada mensagem é consumida por apenas um servidor, no modelo publish/subscribe um evento pode ser recebido por múltiplos assinantes, caracterizando uma comunicação um-para-muitos (multicast).

O autor destaca que essa arquitetura, também conhecida como orientada a eventos, oferece desacoplamento no espaço e no tempo, assim como as filas de mensagens. Ele exemplifica o uso da arquitetura publish/subscribe em um sistema de companhia aérea,

onde o evento "venda" é publicado pelo sistema de vendas e assinado por sistemas de milhagens, marketing e contabilidade, permitindo que esses sistemas reajam ao evento de forma independente. Valente conclui que a arquitetura publish/subscribe é uma solução poderosa para integrar sistemas de forma flexível e escalável, facilitando a comunicação assíncrona e a propagação de eventos em sistemas distribuídos complexos.

A seção 7.7, explora brevemente dois padrões arquiteturais adicionais: Pipes e Filtros, e Cliente/Servidor. O padrão Pipes e Filtros, inspirado nos comandos do sistema Unix, consiste em conectar programas (filtros) através de canais de comunicação (pipes), onde a saída de um filtro se torna a entrada do próximo. O autor destaca a flexibilidade dessa arquitetura, que permite a combinação de diferentes filtros e a execução paralela, além de promover o baixo acoplamento entre os componentes.

Já o padrão Cliente/Servidor define dois papéis distintos: o cliente, que solicita serviços, e o servidor, que processa as requisições e fornece os resultados. Valente exemplifica o padrão com serviços comuns de rede, como impressão, acesso a arquivos e bancos de dados, além da própria Web, onde o navegador atua como cliente e o servidor Web provê os recursos. Ele conclui que o padrão Cliente/Servidor é fundamental para a implementação de serviços de rede, permitindo a comunicação entre diferentes sistemas e a distribuição de tarefas em uma rede.

Valente finaliza o capítulo 7 abordando os anti-padrões arquiteturais, focando no "Big Ball of Mud", um anti-padrão que descreve sistemas sem uma arquitetura definida, onde os módulos se comunicam de forma caótica e as dependências entre eles formam um emaranhado complexo, como um "espaguete de código". O autor explica que esse anti-padrão geralmente surge da falta de planejamento arquitetural, da pressão por entregas rápidas e da falta de refatoração constante do código.

Ele cita um estudo de caso sobre a modularização de um sistema bancário com milhões de linhas de código, que exemplifica as consequencias do "Big Ball of Mud", como o aumento do tempo de aprendizado de novos desenvolvedores, a dificuldade de manutenção e a proliferação de bugs. Valente conclui que a falta de uma arquitetura bem definida leva a sistemas difíceis de entender, modificar e testar, tornando o desenvolvimento lento e custoso. A prevenção do "Big Ball of Mud" requer a adoção de boas práticas de projeto, como a definição de uma arquitetura clara, a modularização do código e a refatoração frequente para eliminar as dependências desnecessárias.

Os capítulos 6 e 7 de "Engenharia de Software Moderna" merecem elogios pela abordagem didática e clara com que os conceitos de padrões de projeto e arquitetura de software são apresentados. Valente utiliza linguagem acessível, exemplos práticos e diagramas ilustrativos, tornando o conteúdo compreensível para leitores com diferentes níveis de experiência. O autor demonstra grande cuidado em apresentar um conteúdo abrangente e atualizado, incluindo temas como microsserviços, arquiteturas orientadas a mensagens e publish/subscribe, que são cada vez mais relevantes na construção de sistemas modernos.

Um dos grandes destaques dos capítulos é a ênfase na flexibilidade e extensibilidade dos sistemas. Valente argumenta de forma convincente que a escolha adequada de padrões de projeto e arquitetura é crucial para criar sistemas adaptáveis às constantes mudanças tecnológicas. A utilização de exemplos de código em Java e diagramas UML reforça o caráter prático dos capítulos, facilitando a compreensão e a aplicação dos

conceitos. Além disso, a discussão sobre os perigos da "paternite" e a apresentação do anti-padrão "Big Ball of Mud" demonstram a preocupação do autor em fornecer uma visão crítica e equilibrada sobre o uso de padrões de projeto.

Apesar dos méritos, os capítulos poderiam ser enriquecidos em alguns pontos. A seção sobre "Outros Padrões de Projeto" poderia ser expandida, aprofundando a discussão sobre os padrões Iterador e Builder. A utilização exclusiva de Java nos exemplos de código pode ser uma limitação para leitores que utilizam outras linguagens, e a inclusão de exemplos em linguagens como Python, JavaScript ou C# seria um complemento valioso.

De forma geral, os capítulos 6 e 7 representam uma contribuição valiosa para a área de desenvolvimento de software, servindo como um guia completo e atualizado para estudantes e profissionais. O conteúdo apresentado equipa o leitor com o conhecimento necessário para tomar decisões de projeto mais eficazes, criando sistemas flexíveis, extensíveis e de fácil manutenção. As implicações práticas da pesquisa são evidentes, contribuindo para a formação de desenvolvedores mais capacitados e para a construção de softwares de maior qualidade.

Portanto, o desenvolvimento de software não se resume a escrever código, mas sim a tomar decisões estratégicas que moldam a estrutura, a flexibilidade e a longevidade do sistema. Em um mundo onde a mudança é continua, a escolha da arquitetura e dos padrões de projeto certos pode ser a diferença entre um sistema que evolui com o tempo e um que se torna obsoleto, aprisionado em um "Big Ball of Mud". Cabe a cada desenvolvedor, portanto, dominar as ferramentas do design e da arquitetura para construir sistemas que não apenas atendam às necessidades de hoje, mas que também sejam capazes de se adaptar aos desafios do amanhã.