Design Patterns, Estilos e Padrões Arquiteturais

Bootcamp Arquiteto de Software

Vagner Clementino

2021

Design Patterns, Estilos e Padrões Arquiteturais

Bootcamp Arquiteto de Software

Vagner Clementino

Sumário

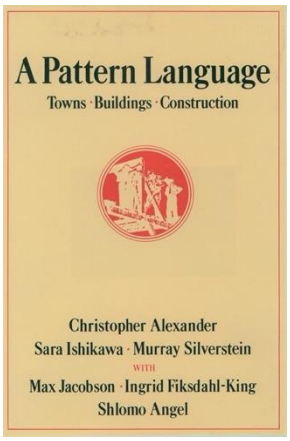
Capítulo 1. Fundamentos

Olá aluno(a)! Seja bem-vindo(a) ao Módulo 3 do nosso Bootcamp de Arquitetura de Software. Nessa parte do curso, vamos apresentar um conjunto de soluções, seja a nível de código (Padrões de Projeto) ou da arquitetura (Padrões de Arquitetura Corporativa/Padrões de Integração). Esse conjunto de soluções, organizados na literatura como catálogos, já foram testadas e se mostram úteis em diversos contextos. Dessa maneira, tal conhecimento, pode ajudar o arquiteto de software na tomada de decisões já provadas e não ficar “reinventando a roda”. Ao invés de reuso de *código*, com padrões você reusa *experiência* (FREEMAN; ROBSON, 2020).

Padrões de Projeto, Estilos e Padrões Arquiteturais

Em diversas áreas do conhecimento identificamos publicações visando documentar práticas que se mostraram úteis em diferentes contextos. Seja um livro de receitas culinárias, um compêndio de soluções para arquitetura (não de software) ou um livro de padrões de projeto, é importante para o profissional ter acesso a esses catálogos de soluções.

Figura 1 – Livro de Cristopher Alexander.



Fonte: www.amazon.com.br.

Em 1977, o arquiteto Alexander lançou um livro chamado *A Patterns Language*, no qual ele documenta diversos padrões para construção de cidades e prédios (VALENTE, 2020). Alguns anos depois, em 1995, quatro autores (Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides) lançaram um livro adaptando as ideias de Alexander para o mundo de desenvolvimento de softwares. Trata-se de catálogo com soluções para resolver problemas recorrentes em projeto de software (VALENTE, 2020). As soluções mapeadas nesse livro acabaram sendo conhecidas como Padrões de Projeto. É importante notar que podem existir outros "padrões de projeto". O que ocorre é que se convencionou a utilizar esse termo para as soluções documentadas no livro.

Figura 2 – Livro de Padrões de Projeto.

Fonte: www.amazon.com.br.

De alguma maneira, os Padrões de Projeto são soluções no nível de código, ou seja, não estão relacionados com a adoção de uma arquitetura de software em específico. Todavia, do ponto de vista da arquitetura de um sistema, a literatura também possui alguns "padrões que se provaram úteis e que, dessa forma, mereciam ser documentados. Alguns autores utilizam os termos "estilos" e "padrões" arquiteturais ao se referirem a essas soluções, contudo, neste texto, não vamos adotar os termos estilos arquiteturais e padrões arquiteturais como sinônimos.

Neste módulo, adotamos as definições em que padrões focam em soluções para problemas específicos de arquitetura, enquanto estilos propõem como os módulos de um sistema devem ser organizados, o que não necessariamente ocorre visando resolver um problema específico (TAYLOR et al, 2009).

Partindo da divisão entre estilo e padrão arquitetural, proposta por Taylor e outros, o Model-View-Controller (MVC) é um padrão arquitetural que resolve o problema de separar apresentação e modelo em sistemas de interfaces gráficas. Por outro lado, Microsserviços, *Pipeline* e *Microkernel* constituem um estilo arquitetural.

Conforme discutido anteriormente, padrões arquiteturais focam em soluções para problemas específicos de arquitetura. Com o tempo, alguns autores perceberam que determinados problemas ocorriam com maior recorrência em sistemas que podem ser classificados como Aplicações Corporativas. Ao falarmos de Aplicações Corporativas, estamos nos referindo a sistemas responsáveis pela exibição, manipulação e armazenamento de grandes quantidades de dados, cujo objetivo é o suporte ou a automação dos processos empresariais com esses dados (FOWLER, 2003). Exemplos incluem sistemas de reserva, sistemas financeiros, sistemas de cadeia de suprimentos e muitos outros que administram negócios modernos.

Apesar das mudanças na tecnologia e do surgimento de novas linguagens e paradigmas de programação, certas ideias básicas do desenho e desenvolvimento de software podem ser adaptadas e aplicadas para resolver problemas comuns nesse tipo de sistema. Dessa forma, Fowler compilou um manual de soluções aplicáveis a grande parte do que chamamos de aplicação corporativa. Na mesma linha, partindo da premissa que aplicações corporativas raramente vivem isoladas, existe uma literatura especializada em documentar soluções para problema de integração. Dentre tais soluções, podemos destacar: transferências de arquivos, banco de dados compartilhados, Remote Procedure Call (*RPC*) e mensageria. Além disso, padrões como Enterprise Service Bus e WebServices tem importância histórica pelo fato de serem largamente utilizados na indústria.

Na próxima seção discutiremos algumas características que o código, ou mesmo, componentes arquiteturais que podem ajudar solucionar ou evitar alguns problemas que são recorrentes no desenho e desenvolvimento de software. É esperado que o arquiteto de software conheça tais padrões de modo a facilitar a comunicação com o time de desenvolvimento e que o ajude a tomar decisões arquiteturais menos arriscadas.

De forma reduzida, podemos pensar que a disciplina de Engenharia de Software consiste na implementação de um sistema que atenda aos requisitos funcionais e não-funcionais definidos por um cliente. Posto de outra forma, estamos falando que um projeto de software é uma solução dado um determinado problema de negócio. Nesse ponto, abrimos aspas para John Ousterhout (2018):

> "O desafio mais fundamental da computação é a decomposição de problemas, ou seja, como pegar um problema complexo e dividi-lo em pedaços que podem ser resolvidos de forma independente". (OUSTERHOUT, 2018)

A arte ou ciência de definir a solução para um problema em projetos de software pode parecer à primeira vista uma atividade simples. Na prática, devemos lidar com a complexidade que caracteriza sistemas modernos de software. Historicamente lidamos com a complexidade por meio de abstrações. Uma abstração — pelo menos em Computação — é uma representação simplificada de uma entidade. Apesar de simplificada, ela nos permite interagir e tirar proveito da entidade abstraída, sem que tenhamos que dominar todos os detalhes envolvidos na sua implementação. Funções, classes, interfaces, pacotes, bibliotecas, etc. são os instrumentos clássicos oferecidos por linguagens de programação para criação de abstrações (VALENTE, 2020).

Conforme discutimos, um dos objetivos do projeto de um software é decompor o problema que o sistema pretende resolver em partes menores. Uma forma de conduzir o processo de decomposição é ser guiado por meio de *Princípios de Projeto.* Princípios de Projeto representam diretrizes para garantir que um projeto de software tenha determinadas propriedades de qualidade. Algumas dessas propriedades estão listadas a seguir:

- **Integridade Conceitual**: propriedade de projeto proposta por Frederick Brooks em 1975 no seu livro o Mítico Homem-Mês (BROOKS, 1975). A ideia é que um sistema não pode ser um amontoado de funcionalidades, sem coerência e

coesão entre elas. Segundo Brooks, a falta de integridade pode ser minimizada por uma autoridade central (um comitê, um arquiteto de software, etc.) responsável por incluir as funcionalidades.

- **Ocultamento de informação**: esse conceito foi discutido inicialmente em 1972 por David *Parnas* (PARNAS, 1972). O autor argumenta que o uso dessa propriedade traz consigo vantagens tais como: desenvolvimento em paralelo, flexibilidade de mudança e a facilidade de entendimento. Em resumo, para se atingir tais benefícios dos módulos elas devem esconder decisões de projeto que são, eventualmente, sujeitas a mudanças.

- **Coesão**: essa propriedade visa garantir que um módulo, seja uma classe ou função, implemente uma única funcionalidade ou serviço. A partir dessa premissa, conseguimos facilitar o desenvolvimento, o entendimento e a manutenção de uma classe.

- **Acoplamento**: pode ser visto como a força e conexão entre dois módulos em um sistema. Na prática, é difícil projetar um sistema com zero acoplamento. Nesse sentido, existe um acoplamento (ruim) de uma classe A para uma classe B quando mudanças em B necessariamente impactam em alterações na classe A. Em resumo, acoplamentos podem ocorrer, mas deveriam ser mediados por alguma interface bem definida.

As propriedades descritas anteriormente podem ser simples em sua definição, contudo, muitas vezes não é fácil garanti-las no código. Por esse motivo, alguns autores vêm propondo alguns princípios que servem como um guia para o desenvolvimento de softwares mais fáceis de entender, manter e evoluir. Um dos mais famoso ficou conhecido com Princípios SOLID que foi introduzido por Robert C. Martin (MARTIN, 2000). O SOLID é o acrônimo de um conjunto de práticas que, quando implementadas em conjunto, tornam o código adaptável à mudança. Cada letra significa os seguintes princípios:

- **Single Responsibility Principle (Responsabilidade Única):** estabelece que uma classe deve fazer apenas uma coisa e, portanto, deve ter apenas uma única razão para mudar.

- **Open Closed/Principle (Aberto/Fechado):** exige que as classes sejam abertas para extensão e fechadas para modificações.

- **Liskov Substitution Principle (Substituição de Liskov):** estabelece que as subclasses devem ser substituíveis por suas classes de base. Isso significa que, dado que a classe B é uma subclasse da classe A, devemos ser capazes de passar um objeto da classe B para qualquer método que espere um objeto da classe A.

- **Interface Segregation Principle (Segregação de Interfaces):** o princípio afirma que criar um número maior de interfaces específicas seria melhor que definir uma única interface de uso geral. Os clientes não devem ser forçados a implementar uma função da qual não precisam.

- **Dependency Inversion Principle (Inversão de Dependências):** afirma que nossas classes devem depender de interfaces ou classes abstratas, em vez de classes e funções concretas.

Nesta seção discutimos como os princípios de projeto permitem o desenho de uma solução mais flexível. Manter esses princípios em mente ao projetar, escrever um sistema permite que um código seja mais limpo, extensível e testável. Uma maneira de obter de forma “automática” tais princípios é através da adoção de Padrões de Projeto. De tal forma, é importante notar que alguns desses princípios ou mesmo propriedades, que descrevemos anteriormente, fazem mais sentido para sistemas que utilizem linguagens orientadas a objeto (Java, C#, Ruby etc.). Caso o projeto utilize uma linguagem de programação baseada em outro paradigma (ex. funcional), o uso de alguns princípios ou padrões acabam não fazendo muito sentido.

Capítulo 2. Padrões de Projeto

Introdução aos Padrões de Projeto

Do ponto de vista histórico, podemos afirmar que os padrões de projeto são inspirados nas ideias de Cristopher Alexander (VALENTE, 2020).

> *"Cada padrão descreve um problema que sempre ocorre em nosso contexto e uma solução para ele, de forma que possamos usá-la um milhão de vezes." - Cristopher Alexander*

Em 1995, a *"Gang of Four",* apelido que foi dado aos quatro autores, escreveram um livro chamado *Padrões de Projeto* que descreve um conjunto de soluções (GAMA, 1995). Os padrões de projeto descrevem objetos e classes que se relacionam para resolver um problema de projeto genérico em um contexto particular (VALENTE, 2020). Cada padrão é estruturado com as seguintes propriedades:

(1) O problema que o padrão pretende resolver.

(2) O contexto em que esse problema ocorre.

(3) A solução proposta.

O conceito de padrão de projeto surgiu da necessidade de documentar soluções amplamente utilizadas. A partir da sua aplicabilidade, os padrões visam aumentar a flexibilidade do projeto do software com o objetivo de deixá-lo mais organizado. Na próxima seção discutiremos os benefícios de conhecer e adotar os Padrões de Projeto.

Benefícios dos Padrões de Projeto

Dentro das expectativas do papel de arquiteto de software, em especial aquela que fala sobre tomar decisões arquiteturais, pode estar fundamentalmente relacionada ao conhecimento dos Padrões de Projeto. Além disso, não menos

importante, cabe ao arquiteto de software entender as situações em que o uso de Padrões de Projeto não é recomendado.

A habilidade de comunicar de maneira efetiva é uma habilidade esperada por todos envolvidos no processo de construção de software. A tendência é que tenhamos um desenvolvimento de software cada vez mais distribuído. Nesse contexto, surge a necessidade de comunicar ideias mais complexas a um grupo maior de pessoas. Por essa razão, o conhecimento de padrões de projeto é fundamental, especialmente porque os padrões transformaram-se em um vocabulário largamente adotado (VALENTE, 2020). Ademais, conhecer padrões permite adotar uma solução testada e validada, algo que em longo prazo traz consigo a possibilidade de que entendamos o comportamento e a estrutura do código que foi desenvolvido.

Todavia, o uso indiscriminado de padrões de projeto é questionável. Em muitos sistemas observa-se um uso exagerado de padrões de projeto, em situações nas quais os ganhos de flexibilidade e extensibilidade são questionáveis (VALENTE, 2020). Existe até um termo para se referir a essa situação: **paternite**, isto é, uma "inflamação" associada ao uso precipitado de padrões de projeto. John Ousterhout tem um comentário relacionado a essa “doença” (VALENTE, 2020):

> O maior risco de padrões de projetos é a sua aplicação em excesso (over-application). Nem todo problema precisa ser resolvido por meio dos padrões de projeto; por isso, não tente forçar um problema a caber em um padrão quando uma abordagem tradicional funcionaria melhor. O uso de padrões de projeto não necessariamente melhora o projeto de um sistema de software; isso só acontece se esse uso for justificado. Assim como ocorre com outros conceitos, a noção de que padrões de projetos são uma boa coisa não significa que quanto mais padrões de projeto usarmos, melhor será nosso sistema. (OUSTERHOUT, 2018).

O texto de John Ousterhout vem reforçar que nem todo o problema precisa ser resolvido por meio de um determinado padrão de projeto. Muitas vezes, seguir a mentalidade proposta pelo princípio KISS (*Keep It Simple* - Mantenha Simples)

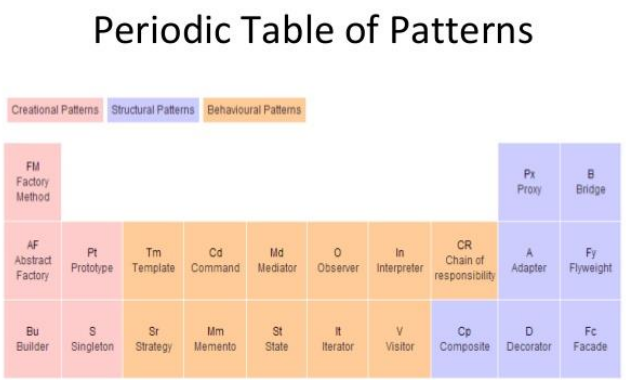
poderia ser o mais adequado. Além disso, o autor comenta que não há relação entre o uso de Padrões de Projeto e qualidade do software produzido.

Organização dos Padrões de Projeto

Em última instância, o livro de Padrões de Projeto é um catálogo (de soluções). Uma das principais características de qualquer catálogo é permitir o fácil acesso ao conteúdo que foi organizado. Os Padrões de Projeto diferem por sua complexidade, nível de detalhes e escala de aplicabilidade. No livro são propostos 23 padrões que são categorizados como criacionais, estruturais e comportamentais.

Os **padrões criacionais** propõem soluções flexíveis para criação de objetos, que aumentam a flexibilidade e o reuso de código. Por outro lado, os **padrões estruturais** facilitam a composição de objetos e classes em estruturas maiores, contudo, sem perda de flexibilidade e eficiência. Por fim, os padrões comportamentais são voltados para o melhor uso de algoritmos e para facilitar a interação e divisão de responsabilidade entre classes e objetos.

Figura 3 – Tabela Periódica de Padrões.



Fonte: https://www.slideshare.net/yinyang581525/javascript-common-design-patterns.

Nos próximos capítulos, vamos apresentar os padrões de projeto propriamente ditos. Eles são exibidos na Tabela 1. Para facilitar a compreensão de cada padrão, optamos por apresentar um contexto e um problema. A partir da análise efetuada, um padrão é apresentado como uma possível solução. Os exemplos foram baseados no livro Engenharia de Software Moderna (VALENTE, 2020). Recomenda-se a leitura do mesmo para um maior detalhamento. Os exemplos de código estão na linguagem Java, contudo, incluímos diagramas UML para facilitar o entendimento. Os diagramas foram baseados nos exemplos disponibilizados no repositório: https://github.com/RafaelKuebler/PlantUMLDesignPatterns.

Tabela 1 - Padrões de Projeto estudados.

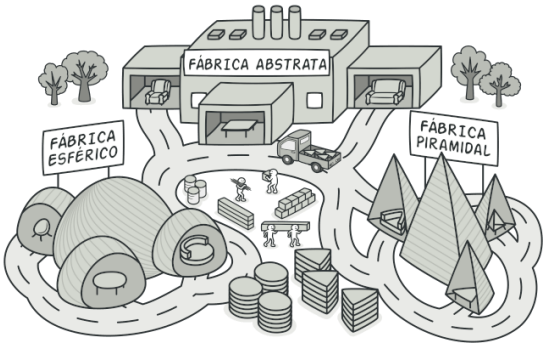
Padrões de Projeto		
Criacionais	**Estruturais**	**Comportamentais**
Abstract Factory	Proxy	Strategy
Singleton	Adapter	Observer
Builder	Facade	Visitor
	Decorator	Iterator

Capítulo 3. Padrões de Projeto Criacionais

Abstract Factory

O Abstract Factory é um padrão de projeto criacional que permite que você produza famílias de objetos relacionados sem ter que especificar suas classes concretas. O padrão permite produzir uma família de objetos relacionados, sem ter que especificar suas classes concretas. Traz para o código os princípios SOLID de Responsabilidade Única e de Aberto/Fechado.

Figura 4 – O padrão Abstract Factory.



Fonte: https://refactoring.guru.

Contexto: Suponha um sistema distribuído em que há três funções f, g e h que instanciam objetos do tipo *TCPChannel* visando realizar algum tipo de comunicação remota, seja ela através do protocolo *TCP* ou *UDP*.

```
void f() {
  TCPChannel c = new TCPChannel();
  ...
}

void g() {
  TCPChannel c = new TCPChannel();
  ...
}

void h() {
  TCPChannel c = new TCPChannel();
  ...
}
```

Problema: suponha que precisaremos usar o protocolo UDP para comunicação. Para deixar o código mais flexível, seria interessante "parametrizar" a criação dos objetos que representem canais de comunicação capazes de trabalhar com protocolos diferentes, nesse caso, que consigam trabalhar ou com TCP ou UDP. Além disso, não gostaríamos de alterar o código existente quando os clientes exigirem novas maneiras de comunicar, como, por exemplo, através de chamadas *gRPC*. Fazendo uma ligação com os princípios SOLID que estudamos anteriormente, dizemos que queremos um código fechado para *modificação* e aberto para *extensão*.

Solução: a solução passa, inicialmente, por encapsular a criação dos objetos para um método que cria e retorna objetos de uma determinada classe, de modo que o tipo específico que foi criado acaba sendo ocultado pelo uso de uma interface, no exemplo em questão, estamos falando da interface ***Channel***. Nesse caso, o tipo específico do canal de comunicação (TCP ou UDP) fica oculto, o que não é um problema, tendo em vista que o cliente da interface **Channel** sabe, ou deveria saber, qual tipo de canal de comunicação gostaria de utilizar e deveria fazer uso de algum método genérico da interface; como por exemplo um método **send()** para realizar a comunicação. No código a seguir, vemos que a classe **ChannelFactory** que só tem a responsabilidade de criar objetos concretos por meio do método **create()**. O método é estático é retorna uma interface do tipo **Channel**. Nesse exemplo, os método f, g e h fazem uma chamada para o método **create()** para receber o tipo **TCPChannel**. Caso fosse necessário incluir o novo canal de comunicação (exemplo UDP), basta alterar o método **create()** da classe **ChannelFactory**.

```
class ChannelFactory {
  public static Channel create() {// método fábrica estático
    return new TCPChannel();
  }
}

void f() {
  Channel c = ChannelFactory.create();
  ...
}

void g() {
  Channel c = ChannelFactory.create();
  ...
}
void h() {
  Channel c = ChannelFactory.create();
  ...
}
```

Vantagens:

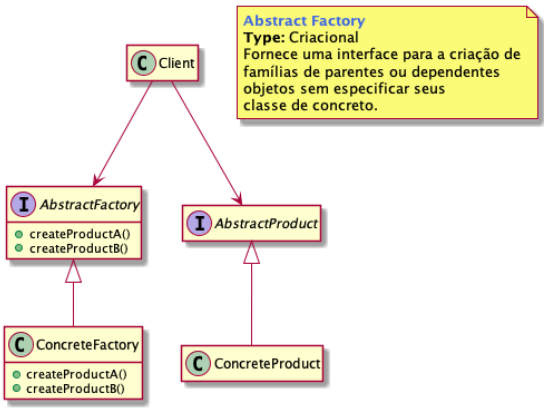
- Os objetos que você obtém de uma fábrica são compatíveis entre si.
- Permite adotar no código o Princípio de Responsabilidade Única já que a criação de objetos fica contida apenas no método de factory.
- Atende ao Princípio Aberto/Fechado já que, para incluir um novo comportamento, bastaria alterar o metódo de fábrica para retornar o tipo concreto. O resto do código deveria funcionar tendo em vista que todos os tipos retornados são compatíveis entre si.

Desvantagens:

- O código pode tornar-se complexo por causa de novas interfaces e classes necessárias para criar o padrão.

Na Figura 2, temos um diagrama UML representando o padrão Abstract Factory. É possível identificar que um cliente precisa instanciar uma família de produtos e, para isso, delega a responsabilidade para a interface (**AbstractFactory**) que é implementada por uma fábrica concreta (**ConcreteFactory**). A fábrica concreta possui métodos para criação da família de objetos (**concreteProductA**, **concreteProductB**) cujo retorno deve ser um tipo abstrato através da interface **AbstractProduct**.

Figura 2 - Diagrama UML do padrão Abstract Factory.

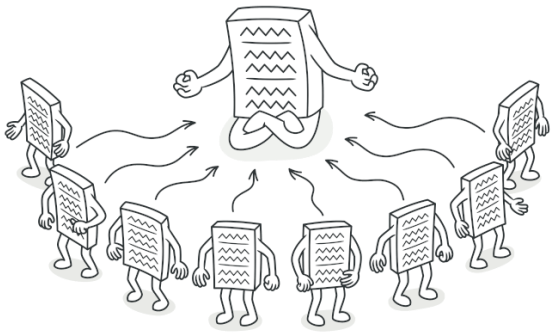


Fonte: https://github.com/RafaelKuebler/PlantUMLDesignPatterns.

Singleton

O Singleton é um padrão de projeto criacional que permite a você garantir que uma classe tenha apenas uma instância, enquanto provê um ponto único de acesso global.

Figura 6 – O padrão Singleton.



Fonte: https://refactoring.guru.

Contexto: em um determinado sistema, toda lógica para registar as operações relevantes do sistema, conhecido como o processo de *logging*, está em uma classe chamada **Logger**. Na imagem a seguir, vemos as funções f, g e h que instanciam a classe **Logger** para fazer o log de sua execução.

```
void f() {
  Logger log = new Logger();
  log.println("Executando f");
  ...
}
void g() {
  Logger log = new Logger();
  log.println("Executando g");
  ...
}
void h() {
  Logger log = new Logger();
  log.println("Executando h");
  ...
}
```

Problema: A capacidade de registrar o seu comportamento é uma habilidade que deve ser compartilhada por múltiplos módulos de um sistema. Diante disso, uma boa prática é ter um único ponto no sistema responsável por registar o log. Isso evitaria, por exemplo, problemas de concorrência ao escrever esse registro em um arquivo. Posto de outra maneira, seria interessante se todos os usos da classe

Logger tivessem como alvo a mesma instância (VALENTE, 2020). Dessa maneira, o ideal é que tivéssemos um único objeto de **Logger** em todo o sistema, de modo que todas as chamadas para a classe compartilhassem a mesma instância.

Solução: a classe **Logger** é uma boa candidata para o uso do padrão Singleton. Ao adotarmos esse padrão de projeto garantimos que uma classe terá, como o próprio nome indica, no máximo uma instância (VALENTE, 2020). Na prática, o padrão define uma classe que tem a responsabilidade de gerir uma única instância de si mesma e que impede qualquer outra classe de criar uma nova instância por si só. Por outro lado, o Singleton fornece um ponto de acesso global à instância: sempre que precisar de uma, basta consultar a classe e ela irá lhe devolver a instância única (FREEMAN; ROBSON, 2020).

```
class Logger {

  private Logger() {} // proíbe clientes chamar new Logger()

  private static Logger instance; // instância única

  public static Logger getInstance() {
    if (instance == null) // 1a vez que chama-se getInstance
      instance = new Logger();
    return instance;
  }

  public void println(String msg) {
    // registra msg na console, mas poderia ser em arquivo
    System.out.println(msg);
  }
}
```

Vantagens:

- Você pode ter certeza que uma classe terá apenas uma única instância.
- Você ganha um ponto de acesso global para a instância.
- O objeto *Singleton* é inicializado somente quando for pedido pela primeira vez.

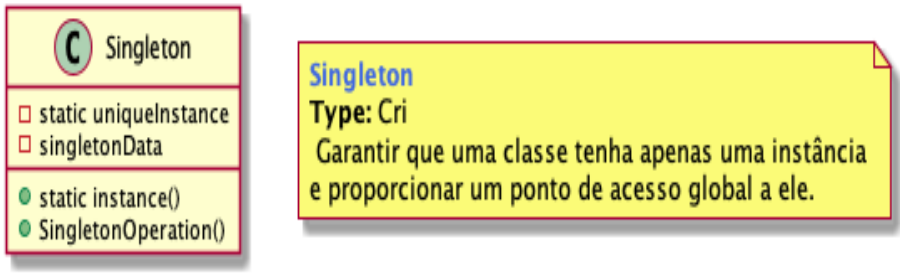
Desvantagens:

- Pode camuflar a criação de variáveis e estruturas de dados globais (VALENTE, 2020).
- Tornam os testes automáticos de determinados métodos mais complexos.

- Pode mascarar um design ruim.

A Figura 3 exibe o diagrama UML do padrão Singleton. O padrão é composto, naturalmente, de uma única classe que possui um atributo privado e estático (**uniqueInstance**) que armazena a única instância da classe. O método **instance()** garante o ponto único de acesso à classe. Os comportamentos oferecidos pela classe que se tornou um **Singleton**, pode ser acessado pelo método **SingleOperation().**

Figura 3 - Diagrama UML do padrão Singleton

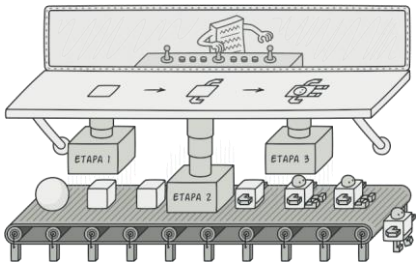


Fonte: https://github.com/RafaelKuebler/PlantUMLDesignPatterns.

Builder

O Builder é um padrão de projeto criacional que encapsula a construção de objetos complexos, permitindo que a sua criação seja feita em etapas. Nesse padrão, os atributos apenas podem ser definidos em tempo de instanciação da classe (VALENTE, 2020), o que na prática tira a necessidade de definir os famosos métodos **"set"** nas classes.

Figura 8 – O padrão Builder.



Fonte: https://refactoring.guru.

Contexto: Imagine um objeto complexo com uma inicialização trabalhosa, seja porque possui muitos campos e seus atributos são objetos agrupados (ex.: um objeto representando uma resposta no formato JSON). Diante desse tipo de classe, a instanciação é realizada através de um construtor monstruoso com vários parâmetros. Caso alguns desses parâmetros tenham valores padrões (default), acabamos por criar constantes que são passadas como argumentos do construtor.

Problema: Imagine um objeto **Livro** com diversos atributos, nem todos obrigatórios. Se durante o processo de instanciação não for informado o calor dos atributos opcionais, eles devem ser inicializados com um valor padrão. Uma possível solução para a classe **Livro** seria criar diversos construtores: um para o caso dos tributos obrigatório e diversos outros para cada cenário com atributos opcionais.

Solução: O padrão *Builder* sugere que você extraia o código de construção de um objeto complexo fora de sua própria classe e mova ele para objetos separados chamados **builders**. A partir de então, a criação do objeto passa pela chamada encadeada de cada um dos **builders**, contudo sem necessariamente ser obrigatório chamar todas as etapas. O desenvolvedor tem a liberdade de utilizar apenas aquelas etapas que são necessárias para a produção de uma configuração específica do objeto. Na imagem a seguir, visualizamos a utilização dos métodos **setNome**, **setEditora**, **setAno** e **setAutores** que são os “builders” da classe **Livro**.

```
Livro esm = new Livro.Builder().
                setNome("Engenharia Soft Moderna").
                setEditora("UFMG").setAno(2020).build();

Livro gof = new Livro.Builder().setName("Design Patterns").
                setAutores("GoF").setAno(1995).build();
```

Vantagens:

- Possibilidade de construir objetos por etapas que possibilitam obter a configuração desejada.
- Permite o reuso do código de construção para diferentes representações do objeto.

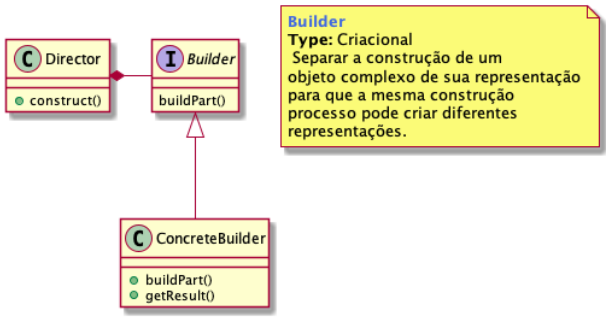
- Garante o Princípio de Responsabilidade Única já que temos uma única classe responsável para construção de um objeto complexo..

Desvantagens:

- A complexidade do código aumenta devido ao padrão criar múltiplas classes e métodos para a criação do objeto.

A Figura 4 exibe o diagrama UML do padrão Builder. O diagrama apresenta uma classe **Director** que tem como dependência uma interface **Builder**, como o método **buildPart()** que representa uma etapa de construção do objeto. A classe **ConcreteBuilder** implementa a interface **Builder** e tem o método **getResult()** que retorno o objeto final já pronto para o uso.

Figura 4 - Diagrama UML do padrão Builder.



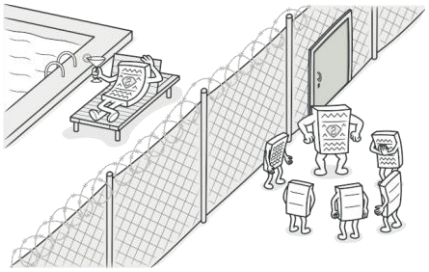
Fonte: https://github.com/RafaelKuebler/PlantUMLDesignPatterns.

Capítulo 4. Padrões de Projeto Estruturais

Proxy

O Proxy é um padrão de projeto estrutural que permite que você forneça um substituto ou um espaço reservado para outro objeto. Um proxy controla o acesso ao objeto original, permitindo que você faça algo ou antes ou depois da chamada chegar ao objeto original. Devemos utilizar um Proxy para controlar o acesso ao objeto quando ele é remoto, dispendioso de criar ou porque ele necessite ser protegido (FREEMAN; ROBSON, 2020).

Figura 10 – O padrão Proxy.



Fonte: https://refactoring.guru.

Contexto: Suponha um sistema de uma biblioteca que tenha uma classe **BookSearch**, cujo principal objetivo é pesquisar por um livro por seu ISBN. A classe tem o método **getBook()** que retorna o **Book** encontrado por meio de uma chamada a um serviço externo.

```
class BookSearch {
  ...
  Book getBook(String ISBN) { ... }
  ...
}
```

Problema: por uma demanda externa, por exemplo, aumento repentino do número de usuários o arquiteto do sistema sugeriu introduzir um sistema de cache. Cache é um dispositivo de acesso rápido, interno a um sistema, que serve de intermediário entre um operador de um processo e o dispositivo de armazenamento ao qual esse operador de alguma maneira tenta acessar. Ao implantar uma camada

de cache no sistema de biblioteca, implica que antes de pesquisar por um novo livro é necessário verificar se seus dados estão no cache. Caso a busca no cache resulte em sucesso o livro será imediatamente retornado. Caso contrário, a pesquisa prosseguirá por meio da chamada do método **getBook()**. Inicialmente poderíamos pensar em adicionar essa lógica na classe **BookSeach**. Contudo, isso não seria uma boa prática. O interessante seria conseguir separar, em classes distintas, o interesse "pesquisar livros por ISBN" (que é um requisito funcional) do interesse "usar um cache nas pesquisas por livros" (que é um requisito não-funcional) (VALENTE, 2020).

Solução: por meio do padrão Proxy é possível criar um componente intermediário entre um objeto base e seus clientes. Assim, os clientes não terão mais uma referência direta para o objeto base, mas dependerão de um intermediário (o proxy) que fornece a mesma interface do objeto base, contudo, agregando novos comportamentos. O Proxy, por ter como atributo o objeto base (no nosso exemplo a classe **BookSearch**), sabe como realizar a operação necessária, no caso em questão, como buscar um livro. Na imagem a seguir, vemos como o proxy pode ser implementado. A classe **BookSearchProxy** implementa a interface **BookSearchInterface** que possui o método **getBook** que retorna o tipo **Book**. A diferença é que o **BookSearchProxy**, ao implementar a interface, faz o uso da classe **BookSearch.** Dessa forma, o método **getBook** adiciona a lógica de incluir um cache do livro recuperado.

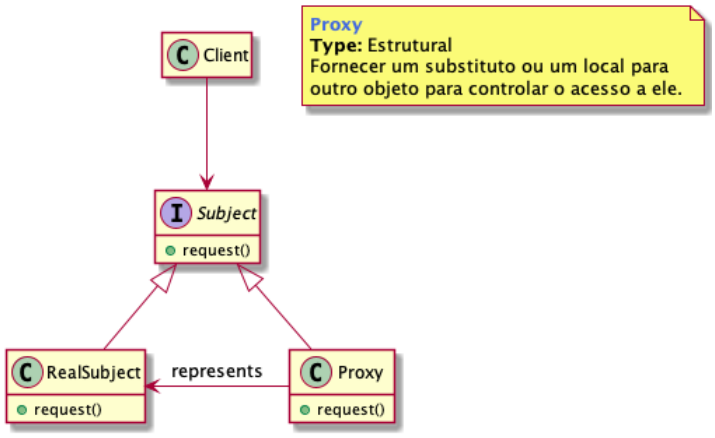
```
class BookSearchProxy implements BookSearchInterface {

 private BookSearchInterface base;

 BookSearchProxy (BookSearchInterface base) {
   this.base = base;
 }

 Book getBook(String ISBN) {
   if("livro com ISBN no cache")
     return "livro do cache"
   else {
     Book book = base.getBook(ISBN);
     if(book != null)
       "adicione book no cache"
     return book;
   }
 }
 ...
}
```

O método **main**, a seguir, mostra como podemos utilizar a classe de proxy. Inicialmente, temos a instanciação da classe base **BookSearch** que é passada para a classe proxy **BookSearchProxy**. Como ambas as classes implementam a mesma interface, a classe proxy pode ser passada como atributo para a classe **View** que anteriormente foi utilizada a classe **BookSearch**.

```
void main() {
  BookSearch bs = new BookSearch();
  BookSearchProxy pbs;
  pbs = new BookSearchProxy(bs);
  ...
  View view = new View(pbs);
  ...
}
```

A Figura 5 exibe o diagrama UML do padrão Proxy. O diagrama apresenta uma classe que representa um cliente que necessita acessar um recurso que é abstraído por meio da interface **Subject**. Como as classes **RealSubject** e **Proxy** implementam a interface **Subject**, a classe **Client** pode agora obter o recurso que necessita por meio de um proxy.

Figura 5 - Diagrama UML do padrão Proxy.

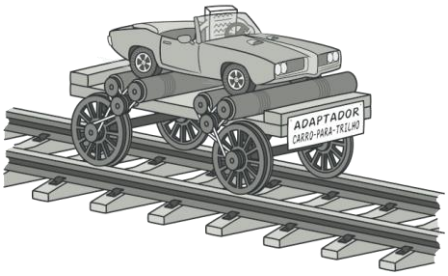


Fonte: https://github.com/RafaelKuebler/PlantUMLDesignPatterns.

Adapter

O Adapter é um padrão de projeto estrutural que permite objetos com interfaces incompatíveis colaborarem entre si. O padrão converte a interface de uma classe em outra que os clientes esperam, de modo que as classes possam trabalhar em conjunto.

Figura 12 – O padrão Adapater.



Fonte: https://refactoring.guru.

Contexto: Suponha que você esteja projetando um sistema para controlar projetores multimídia. O mercado de fabricantes de projetores é bastante diverso e sem qualquer tipo de padronização. Cada fabricante oferece o seu kit de desenvolvimento (SDK) com seus respectivos métodos e comportamentos. O sistema de gerenciamento de projetores deve instanciar classes para cada projetor, conforme definido pelo SDK do fabricante. No exemplo a seguir temos duas classes que representam os projetores das marcas Samsung e LG.

```
class ProjetorSamsung {
  public void turnOn() { ... }
  ...
}

class ProjetorLG {
  public void enable(int timer) { ... }
  ...
}
```

É importante notar que os métodos utilizados para ligar os projetores são diferentes. No caso de um projetor Samsung, basta chamar o método **turnOn**. Para um projetor da marca LG o método chama **enable** e espera um número inteiro que

representa o tempo em segundos que o produto deveria desligar caso ficasse sem uso. Em um cenário real teríamos diversos outros fabricantes, cada um com sua própria maneira de ligar, por exemplo.

Problema: para o cenário com diversos tipos de projetores ficaria inviável saber, por exemplo, os detalhes de como ligar cada um deles. O ideal seria definir uma maneira única, padronizada, para ligar os projetores, independentemente de marca (VALENTE, 2020). Uma solução simples, seria criarmos em cada classe o "nosso" método de ligar, contudo, estaríamos ferindo o princípio SOLID de Aberto/Fechado. Outra solução seria fazer com que as classes implementem uma mesma interface, mas é dificultado pelo fato das classes de cada projetor terem sido implementadas pelos seus fabricantes e não ser possível acessar o código fonte. O código a seguir demostra como seria o cenário em que há uma interface comum chamada **Projetor** possui o método **liga** que é uma maneira de adaptar essa funcionalidade para todos os tipos de projetores.

```
interface Projetor {

  void liga();

}
...
class SistemaControleProjetores {

   void init(Projetor projetor) {
     projetor.liga();  // liga qualquer projetor
   }

}
```

Solução: Para que consigamos facilitar o processo de ligar qualquer tipo de projetor, podemos utilizar o padrão *Adapter*. Para isso, o primeiro passo é criar uma interface chamada **Projetor** que exige a implementação de um método chamado **liga**.

Dessa forma, para ligar um projetor Samsung, criamos um adaptador chamado **AdaptadorProjetorSamsung** que implementa a interface **Projetor**. Como o adaptador possui um atributo privado do tipo **ProjetorSamsung** o processo de ligar um projetor consiste em chamar o método **liga**. Internamente, a classe adaptadora **AdaptadorProjetoSamsung** sabe como ligar o objeto adaptado que, no exemplo em questão, consiste em fazer uma chamada ao método **turnOn**. A partir dessa mudança

não usaremos mais as classes dos fabricantes, mas sim a interface **Projetor**, passando o respectivo objeto adaptado conforme a necessidade.

```
class AdaptadorProjetorSamsung implements Projetor {

  private ProjetorSamsung projetor;

  AdaptadorProjetorSamsung (ProjetorSamsung projetor) {
    this.projetor = projetor;
  }

  public void liga() {
    projetor.turnOn();
  }
}
```

Vantagens:

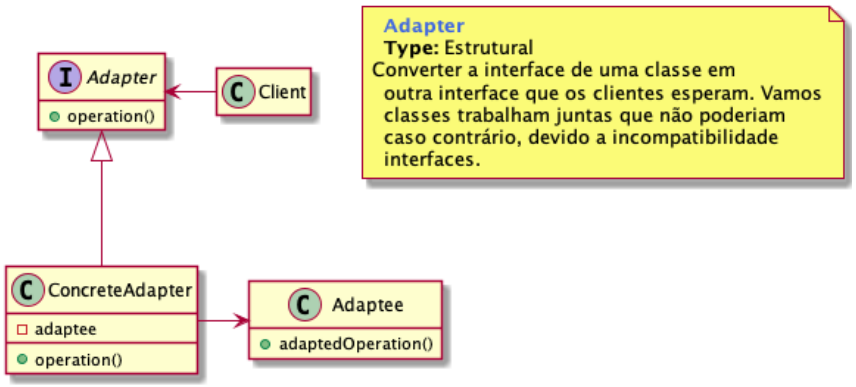
- Princípio de responsabilidade única.
- Princípio aberto/fechado.

Desvantagens:

- O código pode tornar-se mais por causa de novas interfaces e classes necessárias para criar o padrão.

A Figura 6 exibe o diagrama UML do padrão Adapter. Na imagem, podemos visualizar a classe **Client** que precisa acessar um recurso fornecido pela classe **Adaptee.** De alguma maneira, o recurso fornecido pelo **Adaptee** tem uma interface incompatível para a classe **Client**. Para isso, o cliente utiliza da interface **Adapter** que possui o método **operation** que possibilita o acesso ao recurso desejado. A classe **ConcreteAdapter** implementa a interface **Adapter** e por meio do atributo privado **adaptee** consegue entregar o recurso que a classe **Client** precisa por meio da chamada do método **adaptedOperation**.

Figura 6 - Diagrama UML do padrão Adapter.

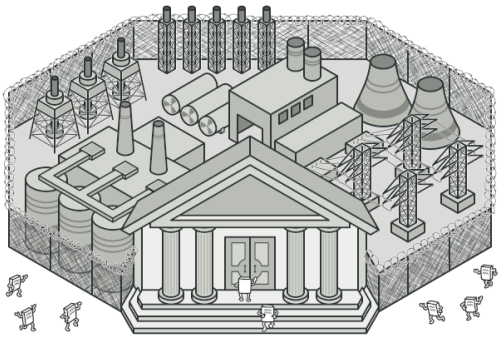


Fonte: https://github.com/RafaelKuebler/PlantUMLDesignPatterns.

Facade

O Facade é um padrão de projeto estrutural que fornece uma interface simplificada para uma biblioteca, um framework, ou qualquer conjunto complexo de classes. Para utilizar o padrão Facade, criamos uma classe que simplifica e unifica um conjunto de classes mais complexas que pertencem a algum subsistema (FREEMAN; ROBSON, 2020).

Figura 14 – O padrão Facade.



Fonte: https://refactoring.guru.

Contexto: alguns problemas de recuperação da informação podem ser resolvidos pela criação de uma linguagem de busca específica. Na computação

chamamos isso de criar uma *domain-specific language* (DSL), ou seja, uma linguagem especializada para um domínio específico. Um exemplo bastante conhecido é a SQL que é uma linguagem especializada para realizar consultas em banco de dados relacionais. Suponha que estamos criando um interpretador para uma linguagem X especializada em realizar consultas nos nossos sistemas. Após definida a sintaxe da nossa linguagem X, a primeira versão do interpretador que executa programas escritos na linguagem X tem os comandos exibidos a seguir. A partir da classe **Scanner** o arquivo com o código fonte é lido. Posteriormente é feito um *parsing*, cujo resultado é uma *Abstract Syntax Tree (AST)* do código. O código em Java é criado pela classe **CodeGenerator** e em seguida é executado pelo método **eval**.

```
Scanner s = new Scanner("prog1.x");
Parser p = new Parser(s);
AST ast = p.parse();
CodeGenerator code = new CodeGenerator(ast);
code.eval();
```

Problema: O uso da linguagem X para realizar buscas foi um sucesso. Diversos desenvolvedores gostariam de incluir nos seus sistemas o código capaz de interpretar um arquivo com o código fonte escrito em X. Contudo, os desenvolvedores acharam complexo o código acima: disseram que é necessário diversos passos para executar e que ele requer conhecimento de classes internas do interpretador da linguagem X.

Solução: O padrão de projeto Facade seria uma alternativa por permitir criar uma interface mais simples para um subsistema. Uma interface mais simples significa oferecer um ponto único em que os clientes podem usar as mesmas funcionalidades que eram oferecidas anteriormente. O objetivo é evitar que os usuários tenham que conhecer classes internas desse sistema; em vez disso, eles precisam interagir apenas com a classe de Facade (VALENTE, 2020). Na prática, conforme imagem a seguir, o padrão Facade é implementado pela criação de um único objeto chamado **InterpretadorX** e pela chamada do método **eval**. Dessa forma, para interpretar um código escrito na linguagem X basta uma linha de código:

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

```
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();
  }
}
```

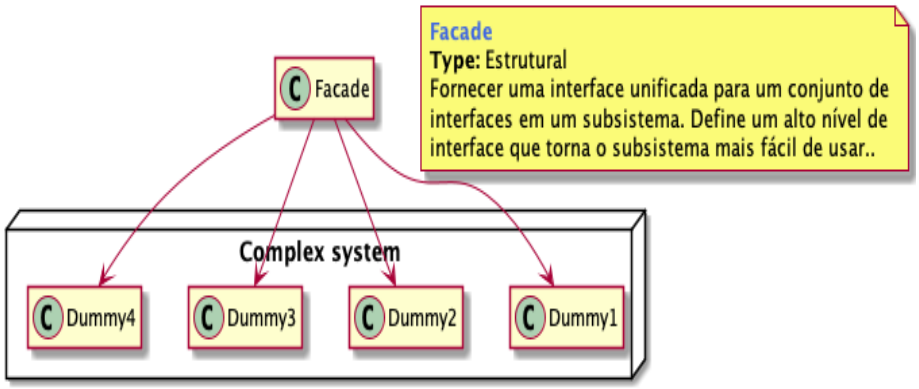
Vantagens:

- Permite isolar seu código da complexidade de um subsistema.

Contras:

- Uma fachada pode se tornar um objeto Deus acoplado a todas as classes do projeto.

Figura 7 - Diagrama UML do padrão Facade.

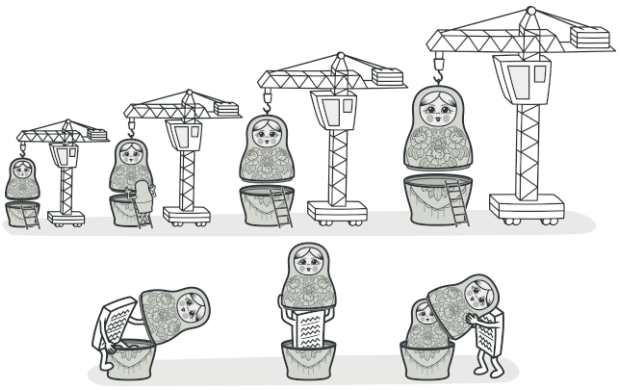


Fonte: https://github.com/RafaelKuebler/PlantUMLDesignPatterns.

Decorator

O Decorator é um padrão de projeto estrutural que permite acoplar novos comportamentos para objetos ao colocá-los dentro de invólucros que já contém aqueles comportamentos.

Figura 16 – O padrão Decorator.



Fonte: https://refactoring.guru.

Contexto: Voltemos ao nosso sistema distribuído que utiliza os canais *TCP* e *UDP* para comunicar. No Capítulo 3 discutimos o uso do padrão Abstract Factory para criar uma família de objetos que representassem canais de comunicação. Não que o padrão tenha causado algum efeito colateral, mas precisamos revisitar o sistema porque os requisitos mudaram. Lembre-se: os requisitos sempre mudam. Na última mudança que fizemos no sistema foi criada a interface **Channel** e as classes **TCPChannel** e **UDPChannel** que a implementam, conforme demonstrado a seguir.

```
interface Channel {
   void send(String msg);
   String receive();
}

class TCPChannel implements Channel {
   ...
}

class UDPChannel implements Channel {
   ...
}
```

Problema: Os clientes das classes **TCPChannel** e **UDPChannel** gostariam que elas ofertassem funcionalidades extras tais como buffers, compactação das mensagens, *logging*, etc. Contudo, nem todas as funcionalidades precisam ser utilizadas ao mesmo tempo, ou seja, em alguns cenários pode ser necessário um canal TCP com apenas compactação, outras vezes, gostaria do mesmo canal só que agora com algum tipo de *buffer* e *compactação*. Esse mesmo cenário pode ocorrer quando for necessário um canal do tipo UDP. Uma primeira solução consiste no uso de herança para criar subclasses com cada possível seleção de funcionalidades (VALENTE, 2020). O uso de herança pode resolver o problema, contudo, traz consigo a necessidade de criar diversas classes, deixando o código muito maior e naturalmente mais difícil de manter. Por exemplo, para ter um canal TCP com apenas compactação e outro com compactação e buffer, teríamos que criar duas classes. Imagine o quanto esse código poderia crescer dado à necessidade de novas funcionalidades.

Solução: o padrão Decorator representa uma alternativa à herança, quando é necessário adicionar novas funcionalidades em uma classe base (VALENTE, 2020). O padrão opta por utilizar composição ao invés de herança para adicionar funcionalidades para uma classe. Dessa forma, para configurar um **Channel**, basta ao cliente encadear as funcionalidades se faz necessário. Por exemplo, na figura a seguir temos a criação, respectivamente de um canal TCP com compactação, um canal TCP com buffer, um canal UDP buffer e para finalizar um canal TCP que proporciona ao mesmo tempo compactação e um buffer.

```
 channel = new ZipChannel(new TCPChannel());
// TCPChannel que compacte/descompacte dados

channel = new BufferChannel(new TCPChannel());
// TCPChannel com um buffer associado

channel = new BufferChannel(new UDPChannel());
// UDPChannel com um buffer associado

channel= new BufferChannel(new ZipChannel(new TCPChannel()));
// TCPChannel com compactação e um buffer associado
```

No exemplo mostrado a configuração de um **Channel** é realizada em tempo de execução por uma sequência aninhada de operadores **new**. É possível observar

que o **new** mais interno está sempre criando uma instancia das classes base **TCPChannel** ou **UDPChannel**. Em seguida, as classes mais externas têm o objetivo de decorar, ou seja, adicionar novas funcionalidades.

Os decoradores propriamente ditos, como **ZipChannel** e **BufferChannel,** devem ser subclasses da classe **ChannelDecorator**. Tal classe implementa a interface **Channel** e possui um atributo privado também do tipo **Channel** que funciona como um histórico das funcionalidades já adicionadas ao tipo concreto. O código da classe **ChannelDecorator** é exibido a seguir.

```
class ChannelDecorator implements Channel {

  private Channel channel;

  public ChannelDecorator(Channel channel) {
    this.channel = channel;
  }

  public void send(String msg) {
    channel.send(msg);
  }

  public String receive() {
    return channel.receive();
  }

}
```

Na solução proposta, todo decorador tem que ser subclasses de **ChannelDecorator**. Quando o nosso decorador recebe, por meio do seu construtor, alguma classe que implementa a interface **Channel**, ele chama o construtor da classe **ChannelDecorator**. Na prática essa abordagem encapsula um determinado canal de comunicação que já possui algum conjunto de funcionalidades. A partir disso, a classe decoradora sobrescreve (*override*) os métodos da interface **Channel** para adicionar as funcionalidades que precisa. No exemplo mostrado a seguir, a classe **ZipChannel**, ao sobrescrever o método **send**, previamente realiza a compactação e chama o mesmo método da classe que herdou. Nesse cenário a classe herdada já pode ter “recebido” novas funcionalidades, como um buffer, por exemplo.

```
class ZipChannel extends ChannelDecorator {

   public ZipChannel(Channel c) {
    super(c);
   }

   public void send(String msg) {
    "compacta mensagem msg"
    super.send(msg);
   }

   public String receive() {
    String msg = super.receive();
    "descompacta mensagem msg"
    return msg;
   }

}
```

Vantagens:

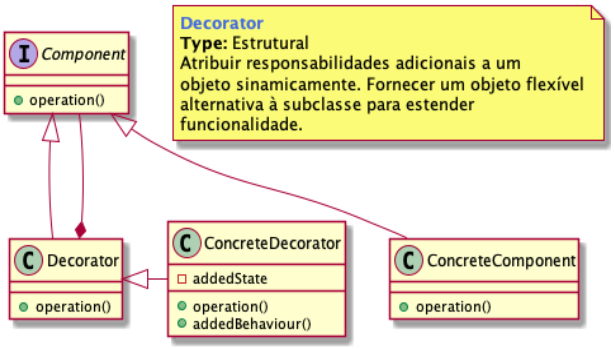
- Estender o comportamento de um objeto sem fazer uma nova subclasse.
- Adicionar ou remover responsabilidades de um objeto em tempo de execução.
- Combinar diversos comportamentos ao envolver o objeto com múltiplos decoradores.

Desvantagens:

- Pode ser difícil implementar um decorador que seu comportamento não dependa da ordem da pilha de decorador

A Figura 8 exibe um diagrama UML do padrão Decorator. O diagrama apresenta a interface **Component** que descreve o comportamento para os objetos que terão responsabilidades adicionadas para eles dinamicamente. Em seguida, vemos a classe **ConcreteComponent** que define um objeto para o qual responsabilidades adicionais poderiam ser anexadas. A classe **Decorator** mantém uma referência para o objeto **Component** e ao mesmo tempo implementa o método **operation** que está em conformidade com a interface **Component**. Os novos comportamentos são adicionados pela classe **ConcreteDecorator** que herda da classe **Decorator**.

Figura 8 - Diagrama UML do padrão Decorator.



Fonte: https://github.com/RafaelKuebler/PlantUMLDesignPatterns.

Capítulo 5. Padrões de Projeto Comportamentais

Strategy

O Strategy é um padrão de projeto comportamental que permite definir uma família de algoritmos, que podem ser utilizados de forma intercambiável. O padrão permite que os algoritmos possam variar independentemente dos clientes que o usam.

Figura 18 – O padrão Strategy.

Fonte: https://refactoring.guru.

Contexto: suponha que você decida criar uma biblioteca com novas estruturas de dados que serão utilizados por diversos projetos dentro da sua organização. Uma dessas estruturas de dados é chamada **MyList**, que possui métodos para adicionar e remover um item na lista, além de permitir ordenar os elementos por meio do método **sort.** Você realizou algumas pesquisas e escolheu o *Quicksort* como o algoritmo padrão de ordenação. Um exemplo do código da estrutura **MyList** é exibido a seguir.

```
class MyList {

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

  public void sort() {
    ... // ordena a lista usando Quicksort
  }
}
```

Problema: alguns desenvolvedores, ao utilizar a sua estrutura de dados, identificam cenários em que o Quicksort não tinha um desempenho satisfatório. Dessa forma, foi demandado que o cliente da estrutura de dados tivesse a opção de alterar e definir, por conta própria, o algoritmo de ordenação que deseja utilizar (VALENTE, 2020). Ao analisarmos a modelagem da classe **MyList**, verificamos que ela não segue o princípio Aberto/Fechado, considerando o algoritmo de ordenação, ou seja, a classe não permite estender o seu comportamento de ordenar itens sem necessariamente alterar o código da própria classe.

Solução: ao adotarmos o padrão Strategy na classe **MyList**, podemos parametrizar os algoritmos de ordenação. Em síntese, o padrão nos permite "deixar aberto" comportamentos, ou seja, no nosso exemplo, podemos encapsular uma família de algoritmos de ordenação e alteração como acharmos conveniente. Na figura a seguir temos uma nova versão da classe **MyList** com o seu comportamento de ordenar parametrizado.

```
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);
  }
}
```

Nessa nova versão, o algoritmo de ordenação transformou-se em um atributo da classe utilizando a classe abstrata **SortStrategy**. A partir de agora, uma classe que represente um algoritmo de ordenação deve necessariamente herdar da classe **SortStrategy** e implementar o método **sort**. Para alterar a maneira de ordenar da classe **MyList,** basta atribuir o respectivo algoritmo por meio do método **set.** A seguir mostramos o código das classes que implementam as estratégias de ordenação:

```
abstract class SortStrategy {
  abstract void sort(MyList list);
}

class QuickSortStrategy extends SortStrategy {
  void sort(MyList list) { ... }
}

class ShellSortStrategy extends SortStrategy {
  void sort(MyList list) { ... }
}
```

Vantagens:

- Trocar algoritmos usados dentro pelo objeto durante sua execução.
- Isolar os detalhes de implementação de um algoritmo do código que usa ele.
- Você pode introduzir novas estratégias sem mudar o contexto.

Desvantagens:

- Deve ser utilizado quando há diversos algoritmos para o mesmo problema.
- Os clientes devem estar cientes das diferenças entre as estratégias.
- Acaba não sendo muito útil em linguagens com suporte ao paradigma funcional.

A Figura 9 exibe o digrama UML do padrão Strategy. Na imagem, temos a classe **Context** que possui uma família de algoritmos que é oferecida pela interface **Strategy** através do método execute. Cada estratégia é implementada pelas classes concretas **ConcreteStrategyA** e **ConcreteStrategyB**.

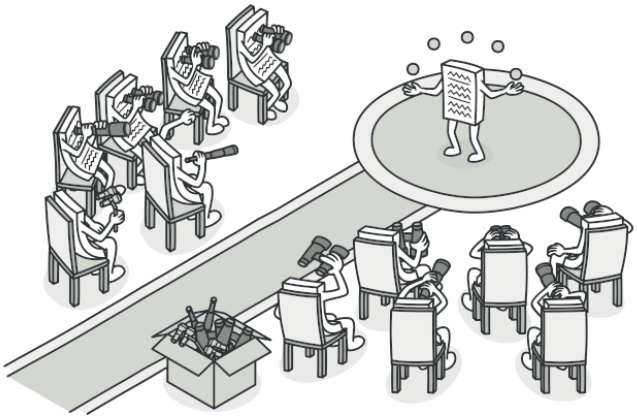
Figura 9 - Diagrama UML do padrão Strategy.

Fonte: https://github.com/RafaelKuebler/PlantUMLDesignPatterns.

Observer

O Observer é um padrão de projeto comportamental que permite definir um mecanismo de assinatura para notificar múltiplos objetos sobre quaisquer eventos que aconteçam com o objeto que eles estão observando.

Figura 20 – O padrão Observer.



Fonte: https://refactoring.guru.

Contexto: suponha que estamos desenvolvendo um sistema de controle de uma estação meteorológica. A modelagem do problema resultado na criação de duas

classes: **Temperatura** e **Termômetro**. A responsabilidade da classe Temperatura é armazenar as temperaturas monitoradas na estação meteorológica. Por outro lado, a classe Termômetro tem a responsabilidade de exibir as temperaturas sob monitoramento. A exibição pode ser feita em diferente formados (Celsius, Fahrenheit, etc.) e plataformas (na web, no console ou exportada para um arquivo). Um requisito necessário ao sistema em questão é os diferentes tipos de termômetros atualizem sua exibição assim que as temperaturas mudem.

Problema: gostaríamos de evitar um acoplamento entre as classes Temperatura (modelo) e Termômetro (visualização). O motivo é simples: classes de visualização mudam com frequência. É bem possível que no futuro possam ter modificações na classe Termômetro, para, por exemplo, exibir a temperatura em celulares ou mesmo em diferentes formatos (digital, analógico, etc.). Além disso, gostaríamos que a classe Temperatura pudesse notificar outras classes do sistema que tenham algum tipo de interesse na mudança de temperatura.

Solução: o padrão Observer é a solução recomendada para o problema descrito (VALENTE, 2020). O objetivo desse padrão é criar um mecanismo de comunicação entre um determinado sujeito e seus observadores. Trata-se de uma relação um-para-muitos que consiste no sujeito notificar os observadores previamente configurados quando o seu estado seja alterado. No nosso sistema de controle meteorológico a classe Temperatura é o sujeito (Subject) que permite adicionar as classes que estão interessadas na mudança do seu estado, que no caso é a mudança da temperatura. No exemplo exibido a seguir, temos dois observadores (**TemometroCelsius** e **TermometroFahrenheit**), que são adicionados por meio do método **addObserver**. Logo em seguida a classe temperatura tem o seu estado alterado com a chamada do método **setTemp**

```
void main() {
  Temperatura t = new Temperatura();
  t.addObserver(new TermometroCelsius());
  t.addObserver(new TermometroFahrenheit());
  t.setTemp(100.0);
}
```

. A seguir exibimos como seria a implementação das classes **Temperatura** e **TermometroCelsius.** A classe **Temperatura** herda da classe **Subject** que possui dois métodos básicos:

- **addObserver:** adiciona observadores em uma instância de Temperatura.
- **notifyObservers**: notifica os observadores quando o estado for alterado

A implementação **notifyObservers** consiste basicamente em percorrer todos os observadores cadastrados chamando o método **update** de cada um deles**.** Veja que, com base na classe **TermometroCelsius,** o método recebe o tipo **Subject,** logo na implementação do método **notifyObservers** a classe **Subject** pode-se passar a própria referência com a palavra reserva **this.**

```
class Temperatura extends Subject {

  private double temp;

  public double getTemp() {
    return temp;
  }

  public void setTemp(double temp) {
    this.temp = temp;
    notifyObservers();
  }
}

class TermometroCelsius implements Observer {

  public void update(Subject s){
    double temp = ((Temperatura) s).getTemp();
    System.out.println("Temperatura Celsius: " + temp);
  }
}
```

Por outro lado, a classe **TermometroCelsius** implementa a interface **Observer** que possui o método **update**. Veja que quando a classe é notificada, ela simplesmente imprime no terminal a temperatura. Contudo, dependendo do tipo do observador, o comportamento poderia ser distinto, como, por exemplo convertendo a temperatura para Fahrenheit.

Vantagens:

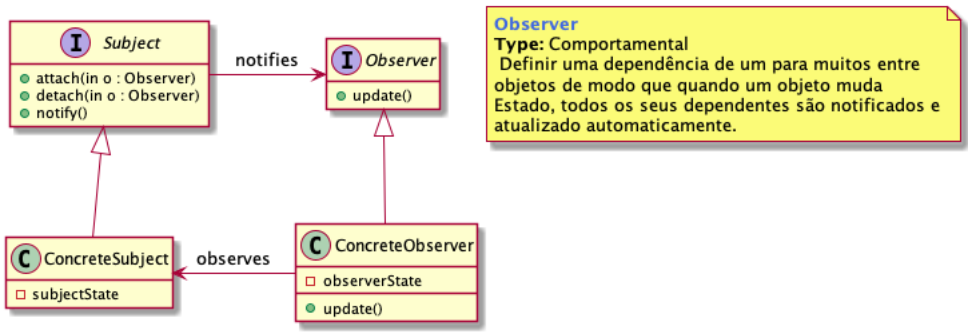
- Não acopla os sujeitos a seus observadores.
- O padrão Observador disponibiliza um mecanismo de notificação que pode ser reusado por diferentes pares de sujeito-observado (VALENTE, 2020).

Desvantagens:

- Assinantes são notificados em ordem aleatória.

Na Figura 10 vemos o diagrama UML do padrão Observer. Na imagem vemos os dois papeis que compõem o padrão: **Subject** e **Observer**. Ambos são interfaces que são implementadas respectivamente pelas classes **ConcreteSubject** e **ConcreteObserver**. A classe **ConcreteSubject** implementa os métodos para adicionar (**attach**) e remover (**detach**) os observadores. Além disso, o sujeito deve implementar o método **notifiy** que notifica os observadores. A classe **ConcreteObserver** deve implementar o método **update** que define como o observador gostaria de ser notificado.

Figura 10 - Diagrama UML do padrão Observer.

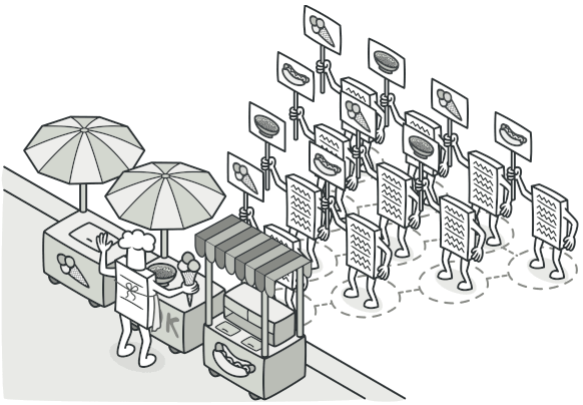


Fonte: https://github.com/RafaelKuebler/PlantUMLDesignPatterns.

Visitor

O Visitor é um padrão de projeto comportamental que permite separar os algoritmos dos objetos nos quais eles operam. Utilize o padrão para adicionar capacidades a um conjunto de objetos em cenários em que o encapsulamento não seja importante.

Figura 22 – O padrão Visitor.



Fonte: https://refactoring.guru.

Contexto: suponha o sistema de estacionamentos que uma classe Veículo com subclasses de Carro, Ônibus e Motocicleta. Nesse sistema todos os veículos poderiam estar armazenados em uma lista polimórfica, que recebem os tipos que sejam subclasses de Veículo (VALENTE, 2020).

Problema: diante da necessidade de realizar uma operação em todos os veículos estacionados, precisamos implementar essas operações fora das classes de Veículo por meio de uma interface, de modo a manter o princípio da Responsabilidade Única.

O código do nosso sistema possui uma classe **PrintVisitor** responsável por imprimir os dados de um Carro, Ônibus e Motocicleta. Como cada classe possui dados distintos, **PrintVisitor** implementa a interface **Visitor** que possui um método **visit** para cada tipo de Veículo. Logo, se queremos imprimir os dados dos veículos armazenados, basta percorrer uma lista que armazena o tipo Veículo e chamar o respectivo método **visit** da classe **PrintVisitor**. Contudo, um código assim não funciona em linguagens como Java, C++ ou C#, porque apenas o tipo do objeto alvo da chamada é considerado na escolha do método a ser chamado (VALENTE, 2020).

```
interface Visitor {
   void visit(Carro c);
   void visit(Onibus o);
   void visit(Motocicleta m);
}

class PrintVisitor implements Visitor {
   public void visit(Carro c) { "imprime dados de carro" }
   public void visit(Onibus o) { "imprime dados de onibus" }
   public void visit(Motocicleta m) {"imprime dados de moto"}
}
```

Solução: Diante da impossibilidade de usar a versão anterior de **PrintVisitor** precisamos implementar o padrão Visitor. O padrão permite "adicionar" uma mesma operação em uma família de objetos, sem que seja preciso modificar as respectivas classes (VALENTE, 2020). A primeira parte da implementação consiste em criar um método **accept** na classe Veículo e nas demais classes da hierarquia. Esse método recebe como parâmetro a interface **Visitor** e consiste basicamente em chamar o método **visit** passando **this,** ou seja, o próprio objeto como parâmetro. A implementação do método **visit** nas classes **Carro** e **Onibus** é mostrado a seguir**.**

```
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 próximo é criar um laço da lista de veículos estacionados, mas dessa vez chamamos o método **accept** passando a classe **PrintVisitor**, conforme imagem a seguir. Veja que ao invés de alterar as subclasses de **Veículo** para imprimir os seus dados, estamos passando para cada classe um comportamento de imprimir os dados do respectivo tipo.

```
PrintVisitor visitor = new PrintVisitor();
foreach (Veiculo veiculo: listaDeVeiculosEstacionados) {
  veiculo.accept(visitor);
}
```

Vantagens:

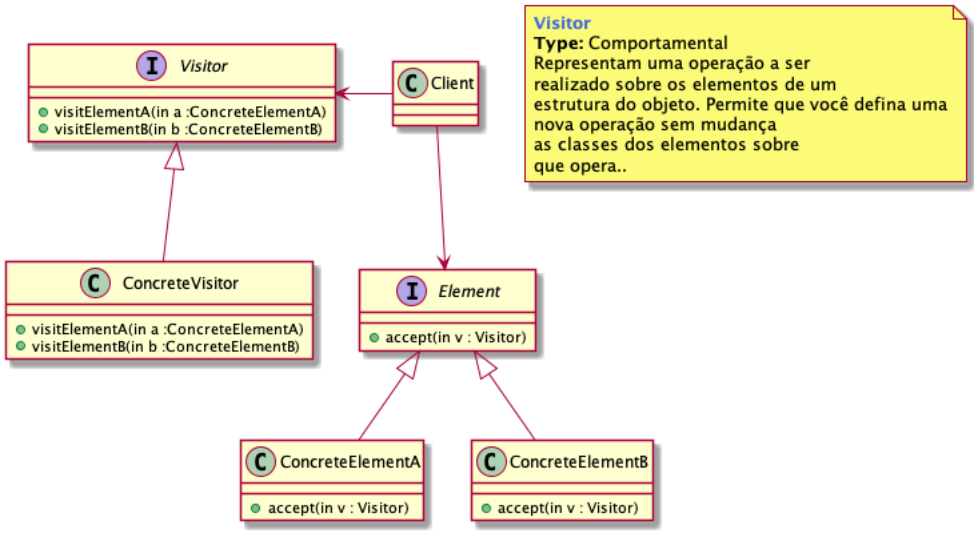
- Facilitam a adição de um método em uma hierarquia de classes.
- Um objeto Visitante pode acumular informações úteis enquanto trabalha com vários objetos.

Desvantagens:

- Podem forçar uma quebra no encapsulamento das classes que serão visitadas.
- Necessário atualizar todos os visitantes a cada vez que a classe é adicionada ou removida da hierarquia de elementos.

A Figura 11 exibe o diagrama UML para o padrão Visitor. O padrão é representado pela interface **Visitor** que possui os métodos **visiElementA**, **visitiElementB** etc. que descrevem como será feita a visita em cada tipo concreto. Veja que os métodos recebem como parâmetro os tipos concretos, fazendo analogia com o nosso exemplo anterior dos veículos, os métodos poderiam ser da seguinte forma **visitCarro(Carro c), VisitOnibus(Onibus o)**, etc. Por sua vez, os tipos concretos **ConcreteElementA** e **ConcreteElement** implementam a interface Visitor, em específico o seu método **accept**. O nosso Visitor, propriamente dito, é representado pela classe **ConcreteVisitor** que acaba implementando a interface **Visitor** que poderia ser visto como a classe **PrintVisitor** do exemplo anterior.

Figura 11 - Diagrama UML do padrão Visitor.

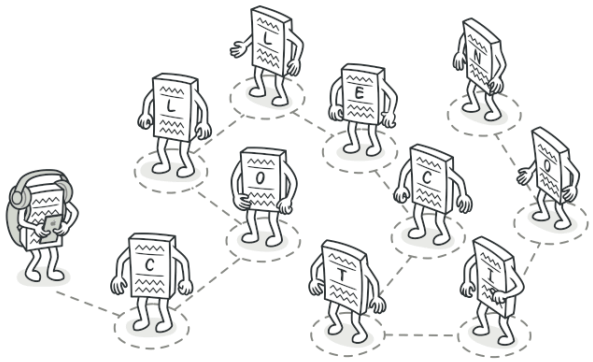


Fonte: https://github.com/RafaelKuebler/PlantUMLDesignPatterns.

Iterator

Um Iterator permite percorrer uma estrutura de dados sem conhecer o seu tipo concreto. Em vez disso, basta conhecer os métodos da interface Iterator (VALENTE, 2020). O padrão Iterator também permitem que múltiplos caminhamentos sejam realizados de forma simultânea em cima da mesma estrutura de dados.

Figura 24 – O padrão Iterator.



Fonte: https://refactoring.guru.

Contexto: suponha que você tenha um sistema com diferentes tipos de coleções. Tais coleções estão baseadas em diferentes tipos de estruturas de dados, tais como pilhas, árvores, grafos e outras estruturas complexas de dados.

Problema: muitas vezes precisamos percorrer todos os itens da coleção. A tarefa é fácil se você tem uma coleção baseada em uma lista. Contudo, em estruturas de dados mais complexas, como uma árvore, determinar o próximo elemento pode ser complicado. Além disso, para a classe de estrutura de dados implementar um algoritmo de travessia resulta em atribuir à coleção outra responsabilidade diferente de armazenar dados de maneira eficiente que deveria ser sua principal função.

Solução: a ideia principal do padrão Iterator é extrair o comportamento de travessia de uma coleção para um objeto separado de mesmo nome. Em geral, a classe fornece dois métodos: **hasNext()** e **next()** para, respectivamente, validar se tem um próximo item e obter o próximo item. É um padrão bastante popular que já foi implementado na biblioteca padrão do Java, como podemos visualizar no código a seguir em que obtemos um iterator de uma lista.

```
List<String> list = Arrays.asList("a","b","c");
Iterator it = list.iterator();
while(it.hasNext()) {
  String s = (String) it.next();
  System.out.println(s);
}
```

Vantagens:

- Possibilidade de extrair o código da travessia com o de armazenamento.
- Possibilidade de iterar na mesma coleção de forma paralela.
- É possível implementar novos tipos de coleções e Iterators sem quebrar o código.

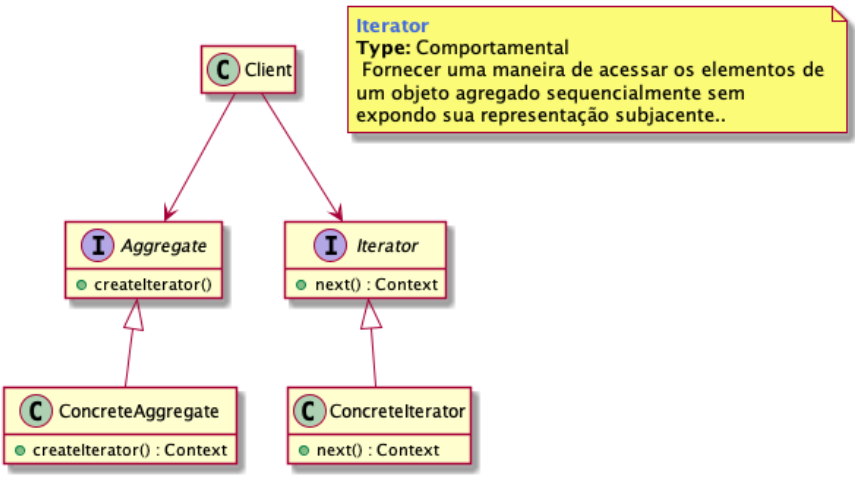
Desvantagens:

- Não faz muito sentido implementar para coleções simples.

- Usar um iterator pode ser menos eficiente que percorrer elementos de algumas coleções.

A Figura 12 exibe o diagrama UML do padrão Iterator. A classe **Client** depende de uma interface do tipo **Iterator** que tem o método **next**. A interface **Aggregate** que define como um novo **Iterator**. A classe **ConcreteAggregate** implementa a interface **Aggregate** para retornar a instância adequada do **ConcreteIterator**.

Figura 12 - Diagrama UML do padrão Iterator.



Fonte: https://github.com/RafaelKuebler/PlantUMLDesignPatterns.

Capítulo 6. Estilos Arquiteturais

Visão Geral

Nesta seção, voltamos à discussão sobre estilos e padrões arquiteturais. Conforme discutido anteriormente, não existe na literatura de arquitetura de software um consenso quanto ao uso dos termos estilo e padrão arquitetural. Alguns autores VALENTE (2020), RICHARDS e FORD (2020) utilizam os termos de forma intercambiáveis. Outros, como TAYLOR *et al* (2009), fazem questão de diferenciá-los. Neste texto, por questões didáticas, optamos por trata-los de maneira distinta, ou seja, padrões focam em soluções para problemas específicos de arquitetura; enquanto estilos propõem como os módulos de um sistema devem ser organizados. A seguir, apresentamos as definições de estilo e padrão arquitetural que nos guiará durante esse módulo.

Um padrão arquitetural é coleção nomeada de decisões arquiteturais que são aplicáveis para um problema de desenho recorrente, no qual deve ser parametrizado para responder diferentes contextos de desenvolvimento de software em que o problema aparece (TAYLOR et al, 2009). Com base nessa definição, podemos notar que o foco de um padrão arquitetural está na solução de um problema específico, algo que não necessariamente existe ao se propor um determinado estilo arquitetural.

Por sua vez, estilos de arquitetura descrevem uma relação nomeada de componentes cobrindo uma variedade de características de arquitetura (RICHARDS; FORD, 2020). Importante notar que um nome de estilo de arquitetura, semelhante ao que vimos anteriormente ao discutirmos os Padrões de Projeto, cria um vocabulário único que atua como abreviação e facilita a comunicação entre arquitetos, desenvolvedores, pessoas de infraestrutura etc. Nos próximos capítulos, iremos focar nos estilos arquiteturais, contudo, antes, precisamos entender como eles podem ser classificados.

Os estilos de arquitetura podem ser classificados em dois tipos principais: **monolítico**, em que é possível identificar um único *deployment* (unidade de implantação) de todo o código fonte; e **distribuído**, onde o processo de deployment resulta em dois ou mais “executáveis” que estão conectados por meio de protocolos de acesso remoto, seja uma requisição HTTP ou mesmo uma chamada remota de métodos (RPC).

Embora nenhum esquema de classificação seja perfeito, todas as arquiteturas distribuídas compartilham um conjunto comum de desafios e problemas não encontrados nos estilos de arquitetura monolítica, tornando este esquema de classificação uma boa separação entre os vários estilos de arquitetura. A tabela a seguir apresenta exemplos concretos de cada um dos tipos de estilos arquiteturais que iremos discutir neste módulo.

Tabela 2 – Classificação dos Estilos Arquiteturais.

Tipos de Estilos Arquiteturais	
Monolítico	**Distribuído**
▪ Arquitetura em Camada ▪ Pipeline ▪ Microkernel	▪ Arquitetura Orientada a Eventos ▪ Arquitetura Orientada a Serviço ▪ Microsserviços

Ao analisar um estilo arquitetural, devemos descrever sua topologia, suas características básicas e as vantagens e desvantagens do seu uso. Nos próximos capítulos, vamos aprofundar um pouco mais nesses estilos sob esse ponto de vista. Os estilos arquiteturais apresentados foram baseados no livro de RICHARDS e FORD (2020), sugerimos que utilizem essa referência para uma discussão mais aprofundada.

Capítulo 7. Estilos Arquiteturais Monolíticos

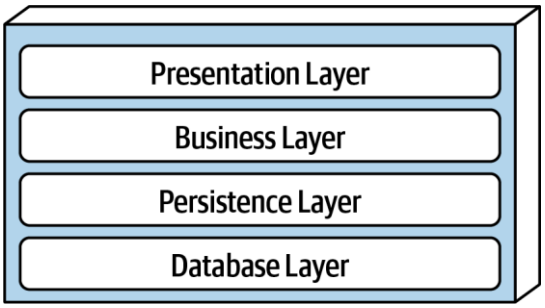
Arquitetura em Camadas

A arquitetura em camadas, também conhecida como estilo de arquitetura de n-camadas, é um dos estilos de arquitetura mais comuns (RICHARDS; FORD, 2020). Esse estilo de arquitetura é o padrão de fato para a maioria das aplicações, principalmente por causa de sua simplicidade, familiaridade e baixo custo.

Existem organizações em que há uma clara separação da equipe de desenvolvimento entre desenvolvedores frontend e backend, desenvolvedores de regras de negócio e especialistas em banco de dados (DBAs). Essas camadas organizacionais se encaixam bem nos níveis de uma arquitetura tradicional em camadas, tornando-a uma escolha natural para muitas aplicações comerciais.

Quanto a sua topologia, os componentes são organizados em camadas horizontais lógicas, com cada camada desempenhando um papel específico dentro da aplicação (como lógica de apresentação ou lógica de negócio). Embora não haja restrições específicas em termos de número e tipos de camadas que devem existir, a maioria das arquiteturas em camadas consiste em quatro camadas padrão: apresentação, negócios, persistência e banco de dados.

Figura 26 – A divisão padrão em camadas lógicas na Arquitetura em Camadas.



Fonte: RICHARDS e FORD, 2020.

Na prática, existem diversas variantes da topologia padrão. Existem casos em que se combinam as camadas de apresentação, negócios e persistência em uma única unidade de implantação, com a camada de banco de dados separada. A

segunda variante separa fisicamente a camada de apresentação em seu próprio "*deployment*", deixando as camadas de negócios e persistência combinadas em uma segunda unidade de implantação. Uma terceira variante combina todas as quatro camadas padrão em uma única implantação, incluindo a camada de banco de dados.

Ao analisarmos a topologia de uma arquitetura em camadas, podemos nos questionar se uma requisição do usuário, por exemplo, precisa sempre passar por cada camada. Conceitualmente essa pergunta se resume em saber se cada camada nesse estilo arquitetural pode ser fechada ou aberta. Uma camada fechada significa que uma solicitação (ex. uma requisição HTTP) não pode saltar nenhuma camada, ou seja, deve passar pela camada imediatamente abaixo para chegar à camada seguinte. Inversamente, ao definirmos uma camada como aberta, uma solicitação pode ser atendida sem o uso desta camada.

Buscando maior flexibilidade em uma arquitetura de camadas, é importante que as mudanças feitas em uma camada da arquitetura geralmente não impactem ou afetem componentes em outras camadas. Tal propriedade é conhecida como camadas de isolamento que exige que as camadas sejam fechadas. Em determinados cenários, como na inclusão de uma nova camada em um sistema, é útil que algumas camadas sejam abertas, de modo a dar uma maior flexibilidade em fazer chamadas ou não para essa nova camada.

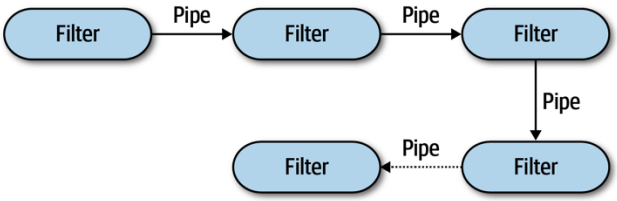
O estilo de arquitetura em camadas é uma boa escolha para aplicações pequenas e simples ou websites. É também uma boa escolha de arquitetura, particularmente, como ponto de partida para situações com orçamento muito apertado e restrições de tempo. À medida que as aplicações que utilizam o estilo de arquitetura em camadas crescem, características como a capacidade de manutenção, agilidade, testabilidade e capacidade de implantação são adversamente afetadas.

Pipeline

Caso algum dia você tenha utilizado o operador | (pipe) em terminais de sistemas operacionais Unix-like, você já tirou proveito do estilo arquitetural pipeline. Ao executar o comando $echo bootcamp ast | tr a-z A-Z em algum terminal, o resultado seria o texto BOOTCAMP AST. Esse é um exemplo de uso de um dos estilos fundamentais na arquitetura de software chamado pipeline, também conhecido como arquitetura de *pipes* (dutos) e *filters* (filtros).

Na solução de problemas que consiste em dividir as funcionalidades em partes discretas, os desenvolvedores e arquitetos decidem adotar este estilo. Esse mesmo estilo é adotado em ferramentas que utilizam o modelo de programação *MapReduce*. A topologia da arquitetura *pipeline* consiste basicamente em dutos (pipe) e filtros (filters), conforme ilustrado na figura a seguir.

Figura 27 – Topologia básica da arquitetura pipeline.



Fonte: RICHARDS; FORD, 2020.

Os dutos (*pipes*) formam o canal de comunicação entre os filtros. Cada duto é tipicamente unidirecional e ponta-a-ponta, ou seja, ele aceita uma carga de trabalho (*payload*) de uma fonte e tem como função direcionar a saída para outra fonte distinta. O *payload* pode ser qualquer formato de dados, mas o ideal é utilizar um formato de menor tamanho - JSON ao invés de XML, por exemplo - visando um alto desempenho.

Os filtros são autocontidos, independentes uns dos outros e, em geral, sem estado. Os filtros deveriam realizar apenas uma tarefa de modo que tarefas mais complexas deveriam ser organizadas como uma sequência de filtros. Existem quatro tipos de filtros dentro desse estilo de arquitetura (RICHARDS e FORD, 2020):

- **Produtor (Producer):** O ponto de partida de um processo, às vezes chamado de fonte. Pode receber a informação a ser processada assinando algum serviço de processamento de streams (Ex. Kafka).

- **Transformador (Transformer):** Aceita uma entrada e executa uma transformação em alguns ou todos os dados, depois os encaminha para um duto de saída**.**

- **Testador (Tester):** Aceita uma entrada, contudo, diferente do tipo anterior, realiza um teste baseado em um ou mais critérios e, opcionalmente, produzir uma saída, com base no teste.

- **Consumidor (Consumer):** O ponto de término para o fluxo da pipeline. Os consumidores muitas das vezes persistem em um banco ou exibem em uma tela o resultado final do processo da pipeline.

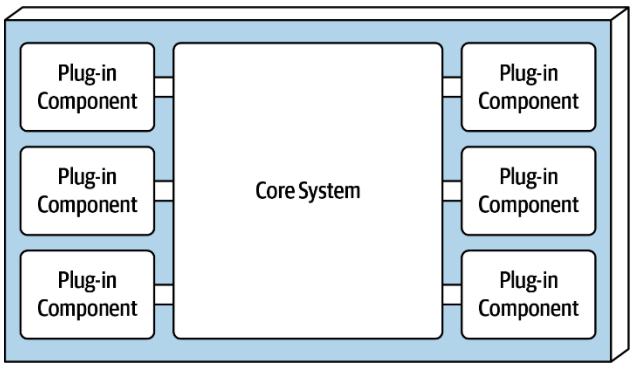
O custo geral e a simplicidade, combinados com a modularidade são os principais pontos fortes da arquitetura pipeline. Sendo de natureza monolítica, as arquiteturas de pipeline não têm as complexidades associadas aos estilos de arquitetura distribuída, são simples e fáceis de entender e têm um custo relativamente baixo para construir e manter. Por outro lado, a arquitetura pipeline possui baixa ou nenhuma tolerância a falhas, especialmente por conta de sua natureza monolítica e da falta de modularidade. Posto de outra forma, se uma pequena parte de uma arquitetura (pipes ou *filters*) tem algum problema (ex. falta de memória) toda aplicação é afetada e poderia parar de funcionar.

Microkernel

O estilo de arquitetura do *microkernel* (também chamado de arquitetura plug-in) é uma escolha natural para aplicações baseadas em produtos. Como uma aplicação baseada em produtos, estamos falando sobre um sistema que tem em seus requisitos ser empacotado e disponibilizado para download e instalação como “executável”, normalmente instalada no site do cliente como um produto de terceiros.

Quanto a sua topologia, microkernel é uma arquitetura monolítica relativamente simples que consiste em dois componentes; um sistema central (core) e os plug-ins. A lógica de aplicação é dividida entre os plug-ins e o core, o que permite o isolamento das funcionalidades e a implementação de lógicas de processamento personalizadas.

Figura 28 – Componentes básicos do estilo microkernel.



Fonte: RICHARDS e FORD, 2020.

Para ilustrar uma arquitetura no estilo microkernel, usaremos a IDE Eclipse. Nesse tipo de arquitetura, o sistema central (core) deve ser responsável pelo conjunto mínimo de funcionalidades para executar o sistema. No caso do Eclipse, o seu core consiste em um editor de texto básico oferecendo a possiblidade de abrir um arquivo, alterar o seu conteúdo e salvá-lo. Além do mínimo possível para funcionar, uma outra maneira de visualizar o que pode representar o core é o chamado “caminho feliz”, ou seja, o fluxo básico de processamento da aplicação com pouco ou nenhum tipo de customização.

Os componentes definidos como plug-ins devem ser autônomos e independentes. Seu comportamento deve refletir um processamento especializado e de características adicionais de modo a melhorar ou ampliar o sistema principal. Além disso, eles podem ser usados para isolar códigos altamente voláteis, criando uma melhor manutenção e testabilidade dentro da aplicação.

A comunicação entre os plug-ins e o core é geralmente ponto-a-ponto, ou seja, existe um ponto único no código, também chamado de pipe, que conecta o plug-

in ao core. Em geral, essa integração é feita por meio de uma invocação de método ou chamada de função de uma classe que funciona como um ponto de entrada (adapter).

Um plug-in pode ser adicionado ao sistema core em tempo de compilação ou em tempo de execução. Os plug-ins, quando adicionados ou removidos em tempo de execução, não necessitam de um novo “deploy” da aplicação, contudo, necessitam ser gerenciados por *frameworks* como o Open Service Gateway Initiative (OSGi) para Java ou Prism para .NET. Por outro lado, plug-ins adicionais em tempo de compilação são mais simples, entretanto, necessitam que aplicação seja reiniciada quando modificados, adicionados ou removidos.

A arquitetura *microkernel* compartilha das mesmas vantagens e desvantagens da arquitetura em camadas. Sua simplicidade e custo geral são os principais pontos fortes, contudo, a escalabilidade, tolerância a falhas e extensibilidade são suas principais fraquezas. Essas fraquezas são devidas às típicas implementações monolíticas encontradas com a arquitetura do *microkernel*.

Uma importante característica do *microkernel* é que ele é o único estilo que pode ser tanto particionado por domínio quanto tecnicamente, o que dá uma maior flexibilidade ao arquiteto. Ademais, por conta da sua modularidade e extensibilidade, novos comportamentos podem ser adicionados, removidos e alterados através de componentes plug-in independentes e autônomos. Finalmente, as arquiteturas que utilizam *microkernel* podem ser simplificadas desligando uma funcionalidade desnecessária, fazendo com que a aplicação seja executada mais rapidamente.

Capítulo 8. Estilos Arquiteturais Distribuídos

Arquitetura Orientada a Eventos

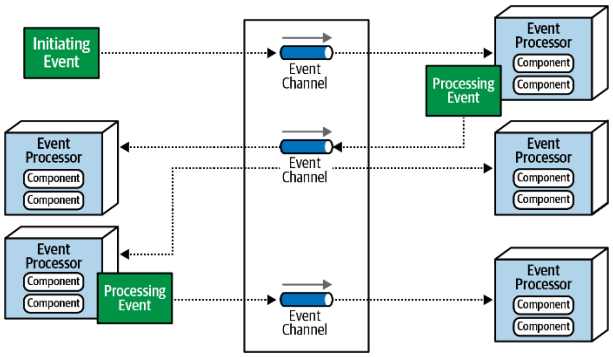
A arquitetura orientada a eventos é um estilo assíncrono e distribuído, utilizada para produzir aplicações altamente escaláveis e de alto desempenho. Nesse tipo de arquitetura, componentes de processamento especializados recebem e executam eventos de forma assíncrona.

Quando utilizamos uma aplicação síncrona, seja interagindo com uma interface gráfica ou fazendo uma requisição para uma API, as solicitações passam por algum tipo de orquestrador. O papel do orquestrador é direcionar determinística e sincronamente uma requisição para vários processadores da solicitação. Esse modelo é conhecido como request-based. Por outro lado, um modelo baseado em eventos, reage a uma situação particular e toma medidas com base nesse evento. Esse modelo é a base do estilo orientado a eventos.

Diferentemente dos outros estilos estudados até o momento, a arquitetura orientada a eventos possui duas topologias primárias: mediator e broker. A topologia mediator é comumente usada quando necessitamos de algum tipo de controle sobre o fluxo de trabalho (sequência de etapas) do processamento de um evento. Em contrapartida, a topologia de broker é usada quando se requer um alto grau de responsividade e controle dinâmico sobre o processamento de um evento.

Na topologia *broker*, não há uma entidade central responsável por mediar os eventos, ao contrário, o fluxo de mensagens é distribuído para os componentes responsáveis por processar os eventos, como sistemas de mensagerias (*RabbitMQ, ActiveMQ, HornetQ* e etc.). Na topologia *broker* é possível identificar quatro componentes principais: initiating event (evento inicial), event broker (despachante de eventos), event processor (processador de eventos) e processing event (evento de processamento).

Figura 29 – Topologia básica do estilo broker.



Fonte: RICHARDS e FORD, 2020.

Com base nesses componentes, o fluxo na topologia broker é o seguinte:

i. Um evento inicial é enviado a um canal de eventos por meio de um event broker para processamento;

ii. Como não há um componente de mediação (mediator) gerenciando e controlando o evento, um único processador de eventos (event processor), aceita o evento inicial e inicia o processamento do mesmo;

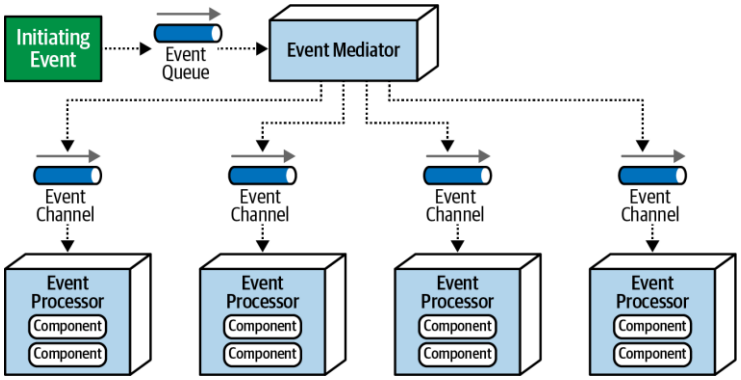
iii. O processador de eventos que aceitou o evento realiza uma tarefa específica associada ao processamento daquele evento;

iv. Em seguida, ele anuncia assincronamente o que fez ao resto do sistema, criando o que é chamado de evento de processamento (processing event). Em geral, o evento de processamento, que comunica que um evento foi processado com sucesso, é enviado para o event broker para que novos processadores de eventos possam realizar uma tarefa como enviar um e-mail ou SMS.

A topologia mediator foi proposta para tratar alguns problemas relacionados com a topologia broker. No centro da topologia mediator existe um componente responsável por gerenciar e controlar o fluxo de trabalho para eventos iniciais (initial event) que requerem a coordenação de vários processadores de eventos (event processor). Na topologia mediator, os principais componentes são: initiating event

(evento inicial), uma fila de eventos (event queue), um mediador de eventos (event mediator), canais de eventos (event channels) e processadores de eventos (event processor).

Figura 30 – Topologia básica do estilo mediator.



Fonte: RICHARDS e FORD, 2020.

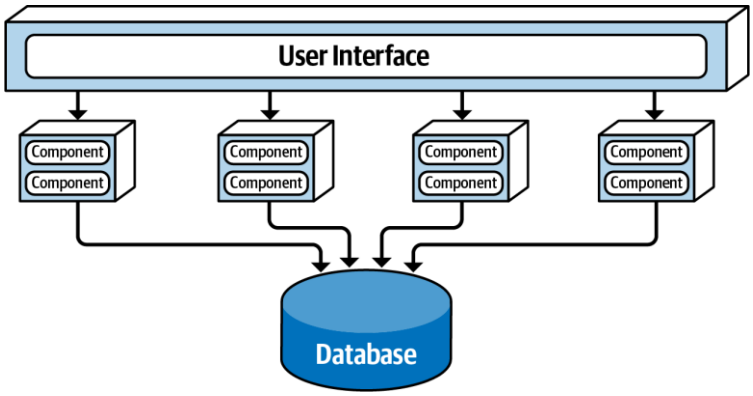
Ao contrário da topologia *broker*, o evento inicial é enviado para uma fila de eventos iniciadores, que é aceita pelo mediador do evento. O mediador do evento conhece apenas as etapas envolvidas no processamento do evento e, portanto, gera eventos de processamento correspondentes que são enviados para canais de eventos dedicados (geralmente filas de espera). Os processadores de eventos então ouvem os canais de eventos dedicados, processam o evento e geralmente respondem ao mediador que completaram seu trabalho. Ao contrário da topologia do *broker*, os processadores de eventos não anunciam o que fizeram para o resto do sistema.

Ao analisarmos os motivos de adotar uma arquitetura orientada a eventos, é importante notar que ela é tecnicamente dividida, na medida em que qualquer domínio específico está espalhado por vários processadores de eventos e amarrado através de mediadores, filas e tópicos. Tal estilo arquitetural é bastante útil em contextos em que desempenho, escalabilidade e tolerância a falhas são necessários. Finalmente, as arquiteturas orientadas por eventos são altamente evolutivas. Adicionar novas características através de processadores de eventos existentes ou novos é relativamente simples, particularmente na topologia *broker*.

A arquitetura baseada em serviços é um híbrido do estilo de arquitetura dos microsserviços. Embora a arquitetura seja distribuída, ela não tem o mesmo nível de complexidade e custo de outras arquiteturas distribuídas, tais como microsserviços ou arquitetura orientada a eventos, tornando-a uma escolha muito popular para muitas aplicações corporativas.

A topologia básica da arquitetura baseada em serviços segue uma estrutura distribuída em macro camadas que consistem em uma interface de usuário implantada e serviços remotos, ambos implantados separadamente, além de um banco de dados (relacional ou não).

Figura 31 – Topologia básica da arquitetura orientada a serviços.



Fonte: RICHARDS e FORD, 2020.

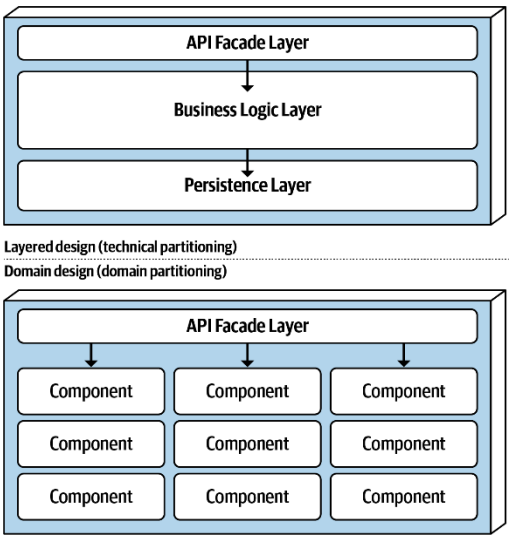
Os serviços dentro desse estilo de arquitetura são tipicamente "partes de uma aplicação" (geralmente chamados de serviços de domínio) que são independentes e implantados separadamente. Eles são acessados remotamente a partir de uma interface de usuário, usando um protocolo de acesso remoto. Enquanto chamadas REST é tipicamente usado para acessar serviços a partir da interface do usuário, mensagens, chamada RPC ou SOAP também podem ser utilizados.

Um aspecto importante da arquitetura baseada em serviços é que ela normalmente utiliza um banco de dados compartilhado. Isso permite que os serviços

otimizem as consultas SQL da mesma forma que uma arquitetura tradicional monolítica em camadas poderia fazer.

Nesse estilo arquitetural, cada serviço é tipicamente organizado usando um estilo em camadas (uma fachada API, uma camada de negócios e uma camada de persistência). Outra abordagem bastante utilizada é a de organizar cada serviço internamente como subdomínios similares ao estilo de arquitetura monolítica. Esses dois estilos de organização dos serviços são mostrados na imagem a seguir. Independentemente de como o serviço está organizado (por camada ou domínio), ele deve fornecer algum tipo de fachada de acesso (*API Facade Layer*) com a qual a interface do usuário possa interagir.

Figura 32 – Variação da organização dos serviços.



Fonte: RICHARDS e FORD, 2020.

A arquitetura baseada em serviços é uma arquitetura particionada por domínio, o que significa que a estrutura é impulsionada pelo domínio em vez de uma consideração técnica. Como cada serviço pode ser implantado separadamente, o uso desse estilo de arquitetura permite mudanças mais rápidas (agilidade), melhor cobertura de teste (testabilidade), e a capacidade de implantações mais frequentes.

As arquiteturas baseadas em serviços tendem a ser mais confiáveis do que outras arquiteturas distribuídas, devido à natureza da divisão dos serviços em

domínios bem definidos. Como na prática cada serviço fica “maior”, isso significa menos tráfego de rede e entre os serviços. Além disso, temos menos transações distribuídas e menos largura de banda utilizada, aumentando, portanto, a confiabilidade geral com relação à rede.

Finalmente, a arquitetura baseada em serviços é uma boa escolha para alcançar um bom nível de modularidade arquitetônica sem ter que se enredar nas complexidades e nas armadilhas da granularidade, ou seja, como melhor dividir serviços em domínios, associadas com a arquitetura de micorsserviços. Da mesma forma, como os serviços dentro de uma arquitetura baseada em serviços tendem a ser maiores, eles não requerem coordenação como outras arquiteturas distribuídas.

Microsserviços

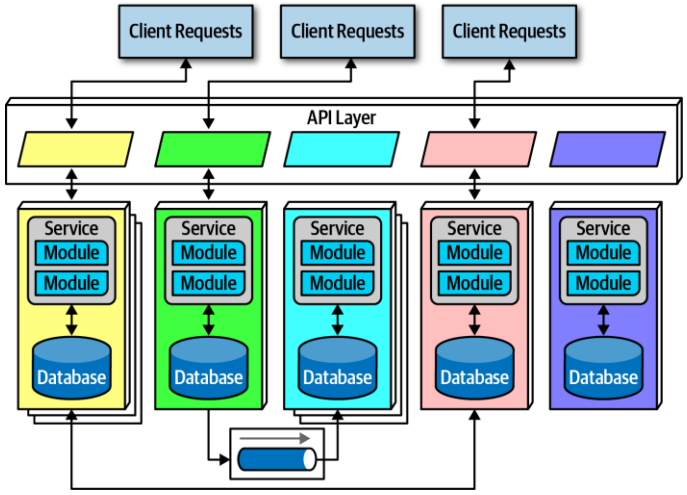
Microsserviços é um estilo de arquitetura extremamente popular que ganhou um impulso significativo nos últimos anos que é fortemente inspirada pelas ideias do *domain-driven design* (DDD). Um conceito em particular do DDD, *bounded context* (contexto delimitado), inspirou decisivamente os microsserviços. A ideia é que, ao arquiteto definir um domínio é que este domínio deverá incluir muitas entidades e comportamentos, identificados em artefatos, tais como códigos e esquemas de bancos de dados.

Ao modelarmos um sistema por domínio, um efeito colateral pode ocorrer: a duplicação de código. O objetivo principal dos microsserviços é um alto desacoplamento, o que acaba favorecendo a duplicação em detrimento da reutilização. Quando um arquiteto projeta um sistema que favorece a reutilização, ele também favorece o acoplamento para conseguir essa reutilização, seja por herança ou por composição. Esse é trade-off a ser considerado ao escolher a arquitetura de microsserviços.

Um serviço nesse tipo de arquitetura deve ser pensado para atender um único propósito. Nesse sentido, o seu tamanho deveria ser muito menor do que outras arquiteturas distribuídas, como por exemplo na arquitetura orientada a serviços. A

premissa básica é que cada serviço seja uma entidade autônoma, ou seja, inclua todas as partes necessárias para operar de forma independente, incluído o seu próprio banco de dados. A topologia típica de uma arquitetura de microsserviços é mostrada na figura a seguir.

Figura 33 – Topologia básica da arquitetura de microsserviços.



Fonte: RICHARDS e FORD, 2020.

Os microsserviços são essencialmente distribuídos de forma que cada serviço é executado em seu próprio processo. Como *processo*, queremos dizer uma máquina física ou virtual. A separação de cada serviço em seu próprio processo é facilitada atualmente com o uso de recursos em nuvem e das tecnologias de contêineres. Dessa forma, as equipes podem colher os benefícios de um maior nível de desacoplamento, tanto a nível de domínio como a nível operacional.

O desempenho é muitas vezes o efeito colateral negativo da natureza distribuída dos microsserviços. As chamadas de rede levam muito mais tempo do que as chamadas de método, e a verificação de segurança em cada ponto final acrescenta tempo adicional de processamento, exigindo que os arquitetos pensem cuidadosamente sobre as implicações da granularidade ao projetar o sistema.

Quando um arquiteto adota a arquitetura de microsserviços, um problema subjacente é definir a granularidade correta de cada serviço. Nessa tarefa muitas vezes erros são cometidos em tornar os serviços muito pequenos, o que os obriga a

construir diversas integrações entre os serviços para que seja possível fazer um trabalho útil. Como o objetivo de definir um escopo ideal para cada serviço recomenda-se utilizar diretrizes como propósito, transações e coreografia para ajudar a encontrar os limites apropriados (RICHARDS e FORD, 2020).

Outro requisito dos microsserviços, movidos pelo conceito *bounded context*, é o isolamento de dados. Muitos outros estilos de arquitetura utilizam um único banco de dados para persistência. Entretanto, os microsserviços tentam evitar todos os tipos de acoplamento, incluindo esquemas de dados compartilhados ou a utilização de banco de dados como ponto de integração. Como o estilo desaconselha o uso de um banco de dados centralizado, os arquitetos devem decidir como querem lidar com esse problema: identificar um domínio com o objetivo de ser a única fonte de verdade ou utilizar algum tipo de cache ou replicação entre os bancos de dados para distribuir informações.

Ao analisarmos as vantagens da adoção da arquitetura de microsserviços, destacamos o apoio às modernas práticas de engenharia, tais como implantação (*deployment*) automatizada e testabilidade. Nesse sentido, os microsserviços favorecem a adoção das práticas de *DevOps* dentro da organização. Além disso, os pontos altos desta arquitetura são a escalabilidade, a elasticidade e a facilidade de evolução.

Por outro lado, o desempenho é frequentemente um problema nas arquiteturas distribuídas por microsserviços, especialmente pelo fato de necessitarem fazer muitas requisições na rede (*requests*) para concluir o trabalho. Nesse caso pode ocorrer perda de desempenho, especialmente em cenários em que se tem a necessidade de realizar validações de segurança para cada acesso aos *endpoints*. Existem padrões para aumentar o desempenho, incluindo o cache de dados e replicação para evitar um excesso de requisições.

Capítulo 9. Padrões de Aplicação Corporativa – EAP

Introdução

Em última instância, as pessoas envolvidas no processo de desenvolvimento estão construindo software. Todavia, sempre é importante salientar que existem diferentes tipos de software, cada qual com seus respectivos desafios e complexidades. Independente do seu papel no processo de desenvolvimento é possível que em algum momento da sua carreira você esteve envolvido com a construção de um tipo de software chamado de Aplicação Corporativa.

Não existe uma definição formal na literatura sobre o que seria uma aplicação corporativa. Talvez seja mais fácil trazer exemplo do que consideramos como esse tipo de software. Inclui-se no conjunto das aplicações corporativas sistemas responsáveis por folha de pagamento, registros de pacientes, rastreamento de remessas, análise de custos, pontuação de crédito, seguros, cadeia de suprimentos, contabilidade, atendimento ao cliente, etc. Por outro lado, não se encaixam como esse tipo de aplicação: sistemas de injeção de combustível em automóveis, processadores de texto, controladores de elevadores, controladores de plantas químicas, interruptores telefônicos, sistemas operacionais, compiladores e jogos (FOWLER, 2003).

Conforme estamos discutindo nesse módulo, a literatura em computação tem diversos textos documentando padrões que podem solucionar problemas relacionados ao desenho e construção de software. Da mesma maneira, durante o desenvolvimento de aplicações corporativas, percebeu-se que algumas soluções poderiam ser reaproveitadas e por isso mereciam ser documentadas. Nesse contexto, surge o conceito de Padrões de Aplicação Corporativa que está relacionado ao conjunto de padrões documentados por Martin Fowler em seu livro de mesmo nome (FOWLER, 2003).

A premissa básica foi de criar um vocabulário comum, assim como nos demais padrões, ao mesmo que descreve como a solução funciona e quando o padrão deveria ser utilizado. Caso você tenha experiência no desenvolvimento desse

tipo de aplicação, é possível que já tenha usado muito desses padrões. De qualquer maneira, é importante conhecê-los de modo a padronizar a nomenclatura de práticas que você já tem no dia a dia.

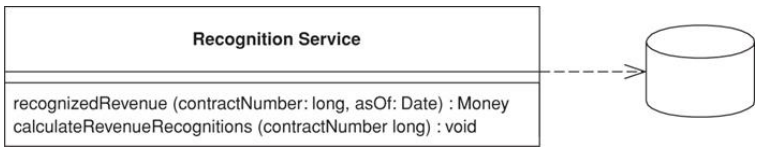
Os padrões são organizados pelo tipo de problema que tenta solucionar. Neste capítulo, optamos por apresentar os padrões de lógica de domínio, fonte de dados e de apresentação web. Para uma lista abrangente de padrões recomendamos a leitura do livro de Fowler, que deve ser utilizado como uma referência e não um texto para ser lido de “capa a capa”.

Padrões de Lógica de Domínio

Uma das primeiras etapas no desenho de um software é decidir qual abordagem devemos adotar ao especificar a lógica de domínio. Como lógica de domínio, queremos dizer sobre a definição das classes que representam e armazenam os dados do domínio de negócio. Para essa atividade, apresentamos dois padrões: *Transaction Scripts* e *Domain Model.*

O *Transaction Script* organiza a lógica de negócio relacionada com armazenamento de dados como um único procedimento, fazendo chamadas diretamente para o banco de dados ou através de uma camada “fina” de acesso a dados. Nesse contexto, cada transação com o banco de dados terá seu próprio *Transaction Script,* embora tarefas comuns possam ser divididas em procedimentos menores. Devido a sua simplicidade, o padrão deveria ser usado para aplicações com uma pequena quantidade de lógica de acesso aos dados, especialmente por ter um bom desempenho e proporcionar uma fácil compreensão.

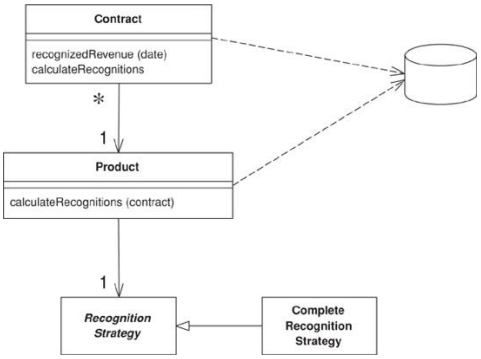
Figura 34 – Padrão *Transaction Scripts*.



Fonte: FOWLER, 2003.

Em determinados contextos, a lógica de negócio associadas ao armazenamento e recuperação de informações em um sistema pode ser muito complexa. As regras descrevem muitos cenários de uso, muitos deles com comportamentos distintos. De maneira geral, foi para essa complexidade que componentes da programação orientada a objetos (classes/objetos, herança etc.) foram projetados para trabalhar. O padrão *Domain Model* cria uma rede de objetos interligados, onde cada um deles representa alguma entidade significativa no contexto do negócio, seja representando uma corporação ou uma única linha em um formulário de pedido.

Figura 35 – Padrão *Domain Model*.



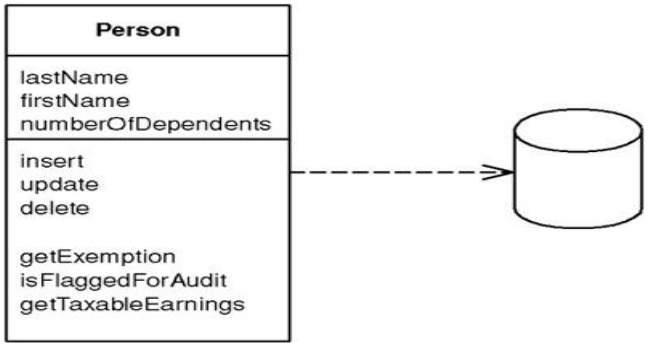
Fonte: FOWLER, 2003.

A utilização de um *Domain Model* em uma aplicação envolve inserir uma camada inteira de objetos que modelam a área de negócios em que você está trabalhando. Tais objetos imitam os dados no negócio e capturam as regras que o negócio utiliza. Nesses casos, pode acontecer que o modelo orientado a objeto seja similar ao modelo de dados (relacional). Em contrapartida, quando um *Domain Model* modela uma entidade mais complexa, ele pode se tornar muito diferente do desenho do banco de dados, por causa do uso de herança, padrões de projeto etc. Esse tipo de *Domain Model* é melhor para uma lógica mais complexa, contudo, acaba sendo mais complexo em mapear para o banco de dados.

Uma vez escolhida sua camada de domínio, você tem que descobrir como conectá-la às suas fontes de dados. A escolha do melhor padrão de acesso aos dados pode depender de outras decisões, por exemplo, como você optou em modelar o seu domínio. Vamos apresentar dois padrões desconsiderando qualquer escolha prévia do modelo do domínio.

Em um sistema que segue uma implementação orientada a objetos, é bem capaz que os objetos sejam responsáveis tanto por armazenar dados quanto por realizar comportamentos. Muitos desses dados devem ser persistidos por meio de um banco de dados. O *Active Record* define um objeto que abstrai uma linha de uma tabela no banco de dados, dessa maneira, encapsulando o acesso ao banco de dados e adicionando uma lógica de domínio sobre esses dados. Dessa forma, todas as pessoas sabem como ler e escrever seus dados no banco de dados.

Figura 36 – Padrão *Active Record*.



Fonte: FOWLER, 2003.

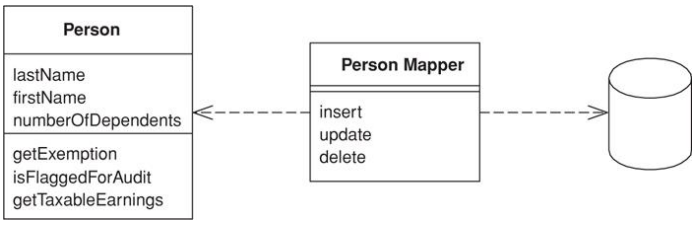
A essência de um *Active Record* é definir um padrão no qual as classes correspondem à estrutura do registro do banco de dados. Cada objeto do tipo *Active Record* é responsável por salvar e carregar no banco de dados e também por qualquer lógica de domínio que atue sobre os dados. O padrão *Active Record* é uma boa escolha para a lógica de domínio que não é muito complexa, tal como criar, ler, atualizar e excluir informações em um banco de dados.

Os objetos e bancos de dados relacionais têm diferentes mecanismos para estruturar os dados. Muitas partes de um objeto, tais como coleções e herança, não estão presentes nos bancos de dados relacionais. Essa diferença originou os bancos não relacionais (NoSQL) e diversos frameworks para mapeamento objeto relacional (ORM).

Quando se constrói ou modela um objeto com muita lógica de negócio envolvida, pode ser valioso adotar algum mecanismo para separar os dados e o comportamento que os acompanha. Apesar de melhorar a organização, mesmo com essa separação, ainda é preciso transferir dados entre os dois esquemas (de negócio e de dados), o que pode algo complexo dependendo do objeto. Uma possível solução para esse problema é a adoção do padrão Data Mapper.

O padrão *Data Mapper* é uma camada de software que separa os objetos em memória daqueles armazenados no banco de dados. Ele move dados entre objetos e um banco de dados, mantendo-os independentes um do outro e do próprio *mapper*. Com o *Data Mapper* os objetos não precisam saber nem mesmo que há um banco de dados presente e nem do respectivo modelo de dados que o banco utiliza.

Figura 37 – Padrão *Data Mapper*.



Fonte: FOWLER, 2003.

Padrões de Apresentação Web

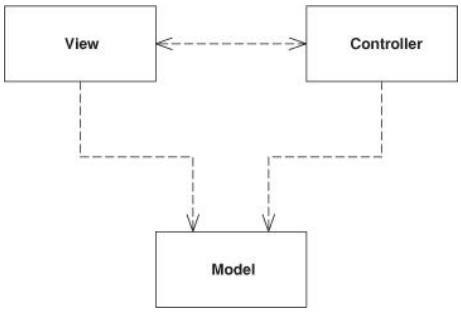
O padrão Model View Controller (MVC) divide a interação da interface do usuário em três funções distintas. O MVC considera três papéis:

i. O modelo (model) é um objeto que representa algumas informações sobre o domínio;

ii. A visão (view), representa a exibição do modelo na interface gráfica;

iii. O controlador (controller), responsável por tratar quaisquer mudanças nas informações.

Figura 38 – Padrão Data Mapper.



Fonte: FOWLER, 2003.

O MVC pode ser pensado como dois objetivos principais: separar a apresentação do modelo e separar o controlador da visão. A segunda divisão, a separação da visão e do controlador, é menos importante, contudo, a separação da apresentação do modelo é uma das heurísticas mais fundamentais de um bom projeto de software. Essa separação é importante por várias razões:

- A apresentação e o modelo são sobre diferentes preocupações;
- Dependendo do contexto, os usuários querem ver a mesma informação de diferentes maneiras;
- Objetos não-visuais são geralmente mais fáceis de testar do que os visuais.

O valor da MVC está em suas duas separações. Dessa maneira, seu uso é sempre recomendado, exceto em sistemas muito simples, onde o modelo não altera o comportamento visual do sistema.

Capítulo 10. Padrões de Integração - EAI

A Necessidade e os Desafios da Integração

Aplicações (corporativas) raramente vivem isoladas, sua existência depende de algum tipo de integração. No desenvolvimento de aplicações corporativas, a integração vai além da criação de uma única aplicação distribuída com uma arquitetura de *n* camadas. O envio de mensagens permite que múltiplas aplicações troquem dados ou comandos através da rede, de forma síncrona ou assíncrona. As Chamadas assíncronas podem tornar um projeto mais complexo do que uma abordagem síncrona, mas uma chamada assíncrona pode ser refeita até ter sucesso, o que torna a comunicação muito mais confiável.

Muitas vezes, dois sistemas a serem integrados são separados por continentes, e os dados entre eles têm que viajar através de linhas telefônicas, segmentos LAN, roteadores, switches, redes públicas e links de satélite. Cada passo pode causar atrasos ou interrupções. O envio de dados através de uma rede é de múltiplas ordens de magnitude mais lenta do que fazer uma chamada pelo método local.

Projetar uma solução distribuída da mesma forma que você abordaria uma única aplicação, poderia ter implicações desastrosas em termos de desempenho. As soluções de integração ajudam a transmitir informações entre sistemas que utilizam diferentes linguagens de programação, sistemas operacionais e formatos de dados. Uma solução de integração deve ser capaz de interagir com essas diferentes tecnologias e minimizar as dependências de um sistema para outro usando um acoplamento fraco entre as aplicações. Com o tempo, os desenvolvedores têm superado esses desafios com quatro abordagens principais (HOHPE e WOOLF, 2004):

- Transferência de arquivos;
- Banco de dados compartilhado;
- Remote Procedure Call;

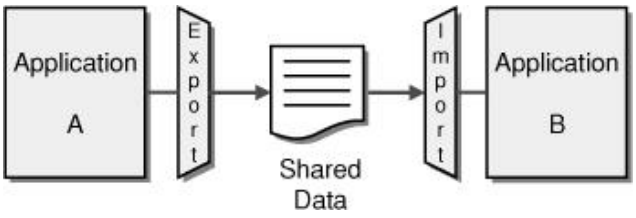
- Mensageria.

As quatro abordagens resolvem essencialmente o mesmo problema, contudo, cada estilo tem suas vantagens e desvantagens distintas, como veremos nas próximas seções.

Transferência de Arquivos

Uma aplicação escreve um arquivo que outra mais tarde lê. As aplicações precisam concordar sobre o nome e localização do arquivo, o formato do arquivo, o momento em que ele será escrito e lido, e quem irá apagar o arquivo.

Figura 39 – Transferência de Arquivos.

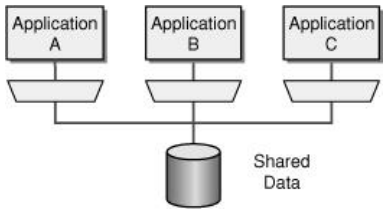


Fonte: HOHPE e WOOLF, 2004.

Banco de Dados Compartilhados

As aplicações múltiplas compartilham o mesmo esquema de banco de dados, localizado em um único banco de dados físico. Como não há duplicação de armazenamento de dados, nenhum dado tem que ser transferido de uma aplicação para outra.

Figura 40 – Banco de Dados Compartilhado.

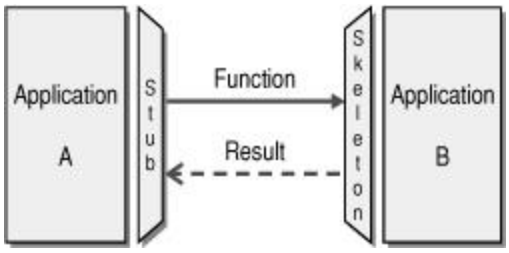


Fonte: HOHPE e WOOLF, 2004.

Remote Procedure Call

Uma aplicação expõe algumas de suas funcionalidades para que possa ser acessada remotamente por outras aplicações como um procedimento remoto. A comunicação ocorre em tempo real e de forma sincronizada.

Figura 41 – Remote Procedure Call.

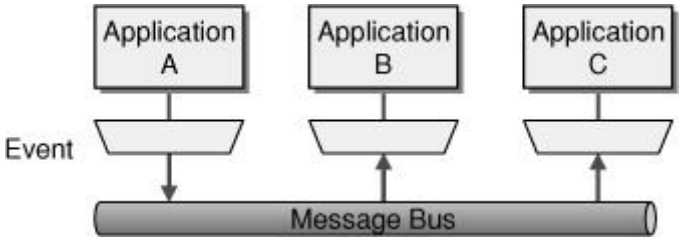


Fonte: HOHPE e WOOLF, 2004.

Mensageria

Em geral, uma organização tem múltiplas aplicações que estão sendo construídas de forma independente, com diferentes linguagens, frameworks e plataformas. A empresa precisa compartilhar dados e processos de uma maneira responsiva. Nesse contexto, uma aplicação poderia publicar uma mensagem para um canal de comum.

Figura 42 – Mensageria.



Fonte: HOHPE e WOOLF, 2004.

O uso de mensageria permite a transferência de dados com frequência e de forma imediata, confiável e assíncrona, usando formatos personalizáveis. O envio assíncrono é, fundamentalmente, uma reação pragmática aos problemas dos sistemas distribuídos, tendo em vista que não requer que ambos os sistemas estejam prontos e em funcionamento ao mesmo tempo.

Capítulo 11. Enterprise Service Bus - ESB

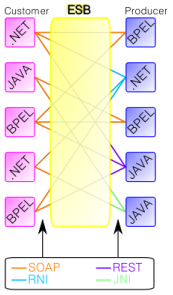
Introdução ao ESB

Nos últimos anos, surgiram algumas tecnologias focadas na melhoria da integração entre as aplicações corporativas, tais como, Arquitetura Orientada a Serviços (SOA), Integração de Aplicações Empresariais (EAI), Business-to-Business (B2B) e Web Services. Essas tecnologias tentaram enfrentar os desafios de melhorar os resultados e aumentar o valor dos processos comerciais integrados, e atraíram a atenção generalizada dos líderes de TI, fornecedores e analistas do setor. Nessa onda, o Enterprise Service Bus (ESB) se apresenta como a nova geração de integração, que se diz utilizar a melhor parte de cada uma daquelas tecnologias.

O conceito ESB é uma nova abordagem de integração que pode fornecer os alicerces para uma rede de integração altamente distribuída e de baixo acoplamento. Um ESB é uma plataforma de integração baseada em padrões que combinam mensagens, serviços web, transformação de dados e roteamento inteligente para conectar e coordenar de forma confiável a interação de um número significativo de aplicações. Nesse contexto, surge a ideia de uma extended enterprise.

Uma extended enterprise representa uma organização e seus parceiros de negócio que, mesmo separados por fronteiras, sejam físicas ou comerciais, podem usufruir de sistemas que estejam separados por dispersão geográfica, firewalls corporativos e políticas de segurança interdepartamental.

Figura 43 – Visão geral de um ESB.



Fonte: Wikipédia.

Em uma arquitetura fazendo uso de um ESB, a aplicação irá comunicar via um barramento (bus), que atua como um message broker entre as aplicações. A principal vantagem é a redução de conexões ponto a ponto, ou seja, entre as aplicações. Isso, por sua vez, afeta diretamente na simplificação nas mudanças nos sistemas. Por reduzir o número de conexões ponto a ponto para uma aplicação específica, o processo de adaptar um sistema às mudanças em um de seus componentes torna-se mais fácil.

Características do ESB

Apesar do ESB servir como uma solução de integração, talvez fique mais claro entender o padrão ESB por meio de suas principais características. A partir do entendimento dessas características é possível delinear se um determinado produto pode ser utilizado como esse tipo de solução.

Pervasividade (Pervasivenes): Uma solução ESB pode ser adaptada para atender às necessidades de projetos de integração de propósito geral através de uma variedade de situações de integração. Ela é capaz de construir projetos de integração que podem abranger toda uma organização e seus parceiros comerciais.

Altamente distribuído, orientado a serviços: Componentes de integração fracamente acoplados, utilizando SOA por exemplo, podem ser implantados no barramento (bus) através de topologias de implantação geográfica distribuídas, mas que são acessíveis como serviços compartilhados a partir de qualquer ponto.

Deployment seletivo de componentes de integração: Adaptadores, serviços distribuídos de transformação de dados e serviços de roteamento baseados em conteúdo podem ser implantados seletivamente quando e onde forem necessários, e podem ser escalados de forma independente.

Segurança e confiabilidade: Todos os componentes que se comunicam através do canal (bus) podem tirar proveito de mensagens confiáveis, integridade transacional e comunicações seguras.

Suporte XML: Uma solução ESB pode tirar proveito do XML como seu tipo de dados "nativo".

Visão em tempo real: Uma solução ESB deve permitir uma visão em tempo real dos dados do negócio.

Adoção do ESB

Muitas tecnologias têm o desafio de serem adotadas pela indústria enquanto provam ser capazes de solucionar o problema para o qual foram desenvolvidas. Os conceitos que fazem parte da arquitetura ESB, por outro lado, evoluíram da necessidade de arquitetos que trabalham junto com os fornecedores de soluções ESB, de modo que a ESB foi adotada na medida em que foi construída. A arquitetura ESB já está sendo utilizada em diversos setores, incluindo serviços financeiros, seguros, manufatura, varejo, telecomunicações, energia, distribuição de alimentos e governo.

Capítulo 12. Web Services

Introdução aos Web Services

Os serviços acessíveis pela Web, chamados Webservices, são parte integrante dos serviços modernos de tecnologia da informação, desde dispositivos móveis até a computação em nuvens. A Internet das Coisas (IoT), Big Data e redes sociais contam com interfaces baseadas na Web para permitir a conectividade com sistemas distribuídos, que nos permite oferecer soluções inovadoras em todos os setores do mercado. Já se foi o tempo em que os desenvolvedores tinham que codificar cada serviço necessário para a execução do sistema. Ao invés disso, os serviços Web estão impulsionando a rápida criação de software. Hoje, com algumas linhas de código, é possível explorar dados em uma rede de pagamentos como MasterCard.

Damos o nome de serviço a unidade fundamental de uma solução orientada a serviços. Um serviço é um sistema autocontido, autodescritivo e modular, que realiza uma função comercial, como a validação de um cartão de crédito ou a geração de uma fatura. O termo autocontido implica que os serviços incluem tudo o que é necessário para que eles funcionem. Autodescrição significa que eles têm interfaces que descrevem suas funcionalidades comerciais. Modular significa que os serviços podem ser agregados para formar aplicações mais complexas.

Os serviços são definidos para serem baseados em padrões e devem ser independentes de plataforma e protocolo a fim de abordar as interações em ambientes heterogêneos. Um único serviço fornece um conjunto de capacidades, muitas vezes agrupadas dentro de um contexto funcional, conforme estabelecido pelas exigências do negócio.

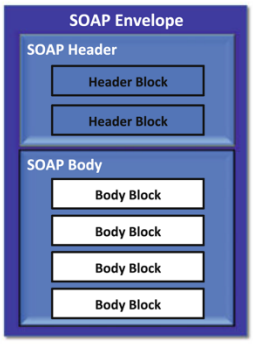
Simple Object Access Protocol – SOAP

SOAP é um protocolo utilizado pelos Webservices para construir e compreender as mensagens que eles trocam. O SOAP está no coração desse tipo de

arquitetura, na medida em que permite os serviços se comunicarem uns com outros usando um formato de mensagem padrão e bem compreendido.

A especificação e a evolução da SOAP são mantidas pelo W3C. A especificação define um formato de mensagem padrão baseado em XML, descrevendo como a mensagem, os metadados e o *payload* devem ser empacotados em um documento XML. O layout básico do formato de uma mensagem SOAP é mostrado a seguir.

Figura 44 – Exemplo de uma mensagem SOAP.




```
<?xml version="1.0"?>
<soap:Envelope
  xmlns:soap="http://
  www.w3.org/2003/05/soap-envelope/"
  soap:encodingStyle="http://
  schemas.xmlsoap.org/soap/encoding/">
  <soap:Body xmlns:m="http://
                www.test.org/stock">
    <m:GetStockPrice>
      <m:StockName>IBM</m:StockName>
      <m:StockPrice>126</m:StockPrice>
    </m:GetStockPrice>
  </soap:Body>
</soap:Envelope>
```

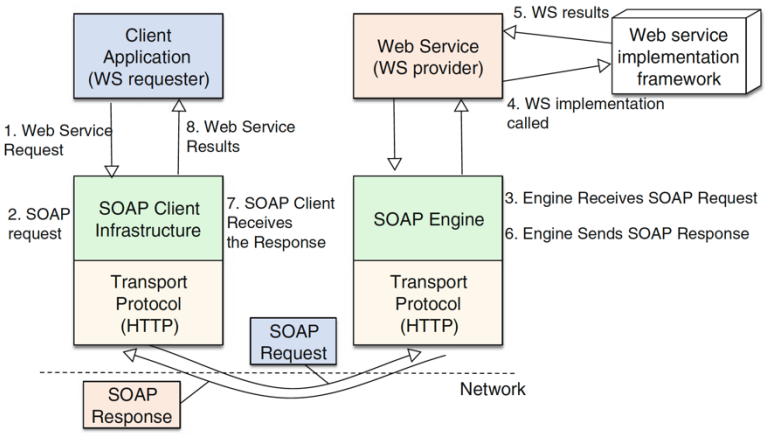
Fonte: PAIK, 2017.

A tag **soap;Envelope** sinaliza o início de uma mensagem. Cada mensagem consiste em duas seções: **soap:Header** e **soap:Body.** O payload (informação a ser enviada) é incluído no corpo da mensagem. Os detalhes adicionais com as instruções de processamento, bem como o protocolo de transação ou as políticas de segurança, estão no cabeçalho da mensagem.

A próxima figura ilustra a interação (requisição e resposta) baseada em SOAP através da Internet. Primeiro, uma solicitação de cliente (serviço requerente) constrói uma mensagem SOAP (solicitação) e a transmite através da rede via HTTP. No lado do servidor, um servidor SOAP - software especial que escuta as mensagens SOAP e atua como distribuidor e intérprete de documentos SOAP - aceita a mensagem e a envia para o destinatário pretendido (prestador de serviços). O serviço é executado baseado na requisição e sua resposta é gerada. A resposta é novamente construída

como uma mensagem SOAP (resposta) e transmitida por HTTP de volta para o cliente.

Figura 45 – Serviços comunicando através de uma mensagem SOAP.



Fonte: PAIK, 2017.

Web Services Description Language - WSDL

Como sabemos, uma interação de serviço web normalmente envolve dois papéis: um cliente que inicia a interação, enviando uma mensagem de solicitação, e um provedor do serviço, que dá seguimento com uma resposta ao pedido. Na seção anterior, explicamos o SOAP como o padrão de formato de mensagem a ser utilizado durante essa interação. Agora que sabemos como formatar uma mensagem, vamos entender o que determina o conteúdo da comunicação.

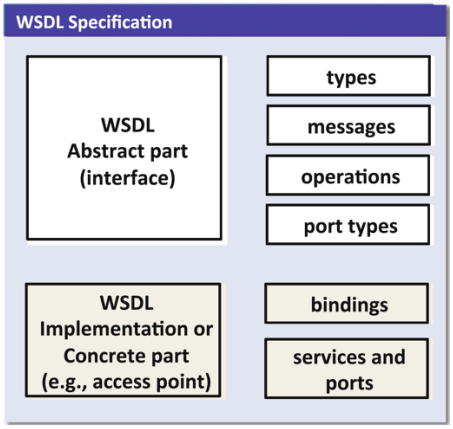
A WSDL é uma especificação, capaz de ser processada (entendia) por máquina, que define o contrato de um Web Service. Trata-se de um documento que o provedor de serviços define para informar aos clientes que tipos de serviços são oferecidos pelo fornecedor e como eles podem ser utilizados. Tipicamente, em um documento WSDL, você encontrará uma lista de operações, ou seja, a funcionalidade de serviço oferecida pelo serviço. Para cada operação, têm-se detalhados os dados de entrada esperados e a saída prevista.

Assim como a SOAP, também é baseada em XML, e descreve um serviço em termos das operações que o compõem, as mensagens que cada operação exige, e as partes a partir das quais cada mensagem é composta. É importante ressaltar que ele é utilizado pelo cliente para gerar um proxy para um Web Service. O proxy atua então como intermediário entre o serviço e o cliente. Essa atividade geralmente é suportada por uma ferramenta, tornando-se uma tarefa quase automática.

Um documento WSDL contém duas partes principais: abstrata e concreta. A parte abstrata define operações e mensagens trocadas através delas (ex. o projeto conceitual do serviço em termos do que ele oferece funcionalmente). A parte concreta contém informações sobre a implantação de redes específicas e a vinculação de formatos de dados.

A divisão entre partes abstratas e concretas é útil para separar os detalhes de projeto do ambiente de implantação dos Web Services. Ou seja, a mesma definição de mensagem desenhada na parte abstrata pode ser vinculada a um transporte HTTP ou transporte SMTP, dependendo dos detalhes da parte concreta de sua WSDL.

Figura 46 – As duas parte de um documento WSDL.



Fonte: PAIK, 2017.

Referências

BROOKS, Frederick P. *The mythical man-month*. Addison-Wesley Longman Publishing Co., Inc. 1975.

CHAPPELL, David A. *Enterprise Service Bus*. O'Reilly Media, Inc., 2004.

FOWLER, Martin. *Padrões de Arquitetura de Aplicações Corporativas*. Bookman, 2003.

FREEMAN, Eric, ROBSON, Elisabeth. *Head First Design Patterns*. 2. ed. O'Reilly Media, Incorporated, 2020.

Gamma E. *Design patterns:* elements of reusable object-oriented software. Pearson Education India, 1995.

HOHPE, Gregor; WOOLF, Bobby. *Enterprise integration patterns:* Designing, building, and deploying messaging solutions. Addison-Wesley Professional, 2004.

MARTIN, R. C. *Design principles and design patterns*. Object Mentor. 2000.

PAIK, Hye-young, et al. *Web Service Implementation and Composition Techniques*. Springer International Publishing, v. 256. 2017.

PARNAS, David L. On the criteria to be used in decomposing systems into modules. In: *Pioneers and Their Contributions to Software Engineering*. Springer, Berlim: Heidelberg. 1972.

RICHARDS, Mark; FORD, Neal. *Newton*. O'Reilly, 2020

TAILOR, R. N.; MEDVIDOVIC, N.; DASHOFY, E. M. *Software Architecture:* Foundations, Theory, and Practice. Wiley Publishing. 2009.

VALENTE, Marco Túlio. Engenharia de Software Moderna: Princípios e Práticas para Desenvolvimento de Software com Produtividade. Leanpub, 2020. Disponível em: <https://engsoftmoderna.info>. Acesso em: 17 fev. 2021.