Design Patterns, Estilos e Padrões Arquiteturais

Bootcamp Arquiteto de Software

Vagner Clementino

2020

Design Patterns, Estilos e Padrões Arquiteturais

Bootcamp Arquiteto de Software

Vagner Clementino

Sumário

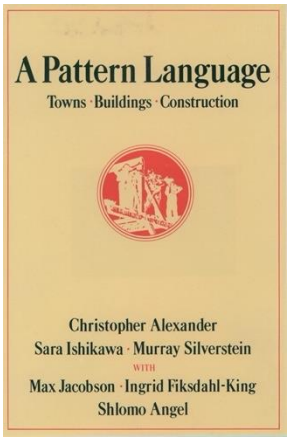
Capítulo 1. Fundamentos

Olá aluno(a)! Seja bem-vindo(a) ao Módulo 3 do nosso Bootcamp de Arquitetura de Software. Nesta 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”.

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 culinários, 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, 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. As soluções mapeadas nesse livro acabaram sendo conhecidas como Padrões de Projeto. É importante notar que podem existem outros "padrões de projeto", o que ocorre é que se convencionou a utilizar esse tempo 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 focam mais na solução, ou seja, no código para solucionar um determinado problema. Todavia, do ponto de vista da arquitetura de um sistema, a literatura também documenta algumas soluções que se provaram uteis. Neste texto, vamos utilizar os termos "estilos" e "padrões arquiteturais" ao nos referirmos a esse tipo solução. Contudo, é importante salientar que não vamos adotar os termos estilos arquiteturais e padrões arquiteturais como sinônimos.

Neste módulo, fazemos uma distinção entre padrões e estilos arquiteturais (TAYLOR et al, 2009). Segundo Taylor e outros, padrões focam em soluções para problemas específicos de arquitetura, enquanto estilos arquiteturais propõem que os módulos de um sistema devem ser organizados de um determinado modo, o que não necessariamente ocorre visando resolver um problema específico.

Partindo da divisão 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 Aplicações Corporativas. Ao falarmos de Aplicações Corporativas, estamos nos referido a sistemas responsáveis pela a exibição, manipulação e armazenamento de grandes quantidades de dados muitas vezes complexos e o suporte ou 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, ideias básicas de design podem ser adaptadas e aplicadas para resolver problemas comuns nesse tipo de sistema. Dessa forma, Fowler compilou manual de soluções aplicáveis a qualquer plataforma 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.

Nesta seção discutimos como a literatura em Engenharia de Software documenta soluções que se provaram capazes de solucionar problemas específicos. Essas soluções focam desde o código até aos componentes arquiteturais. É esperado que um arquiteto de software conheça tais padrões de modo a facilitar a comunicação com o time e tomar decisões arquiteturais menos arriscadas.

Princípios de Projeto

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. 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. Alguns desses princípios são largamente discutidos na literatura de Engenharia de Software (ex. SOLID). Dentre as propriedades desejadas em um projeto de software, podemos destacar:

- **Integridade Conceitual**: propriedade de projeto proposta por Frederick Brooks em 1975 no seu livro o Mítico Homem-Mês. 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) 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 sujeitas a mudanças.

- **Coesão**: de maneira geral, uma classe deve implementar uma única funcionalidade ou serviço. A partir dessa premissa, conseguimos facilitar a implementação, o entendimento e manutenção de uma classe. Além disso, essa propriedade facilita a alocação de um único responsável por manter uma classe, o reuso e o teste da 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 podem facilmente impactar a classe A. Em resumo, acoplamentos podem ocorrer, mas deveriam ser mediados por alguma interface bem definida.

Os Princípios SOLID é um acrônimo para:

- **Single Responsibility Principle (Responsabilidade Única):** estabelece que uma classe deve fazer 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. Isto 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 e o método não deve dar nenhuma saída estranha nesse caso.

- **Interface Segregation Principle (Segregação de Interfaces):** o princípio afirma que muitas interfaces específicas do cliente são melhores do que uma 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 seja de código ou de arquitetura mais flexível. Manter estes princípios em mente ao projetar, escrever um sistema permite 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 tem relação com projetos que utilizem linguagem 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 em uma ideia proposta por Cristopher Alexander em seu livro publicado em 1977.

> *"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, adaptaram as ideias de Alexander para o mundo de desenvolvimento de software. Eles deram o nome de Padrões de Projeto às soluções propostas no livro. Segundo o livro (GAMA, 1995), padrões de projeto descrevem objetos e classes que se relacionam para resolver um problema de projeto genérico em um contexto particular. Visando facilitar a organização os padrões são estruturados da seguinte maneira:

(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 tornar o projeto de um sistema mais flexível. De maneira a deixa-lo mais bem organizado, os padrões são estruturados por meio do problema, contexto e solução.

Benefícios dos Padrões de Projeto

Dentro das expectativas do papel de arquiteto de software, é fundamental que ela entenda como o domínio de Padrões de Projeto pode ajudá-lo na tomada de

decisões. 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 desenvolvimento 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 por desenvolvedores. Ademais, conhecer padrões permite adotar uma solução testada e validada, além de ser possível entender o comportamento e a estrutura do código sendo utilizado.

Todavia, o uso indiscriminado de padrões de projeto é questionável. No entanto, 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":

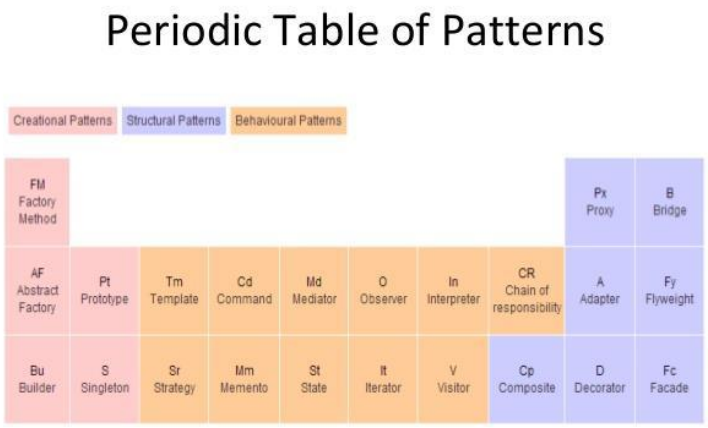
> O maior risco de padrões de projetos é a sua super-aplicação (over-application). Nem todo problema precisa ser resolvido por meio dos padrões de projeto; por isso, não tente forçar um problema a caber em um padrão de projeto quando uma abordagem tradicional funcionar melhor. O uso de padrões de projeto 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).

A premissa básica é que nem todo o problema precisa ser resolvido por meio de um padrão de projeto. Além disso, não há garantia de que quanto mais Padrões de Projeto usarmos, melhor será nosso sistema.

Em uma instância o livro de padrões de projeto é um catálogo (de soluções). Uma das principais características de um catálogo é o fácil acesso. Os Padrões de Projeto diferem por sua complexidade, nível de detalhes e escala de aplicabilidade. No livro são propostos 23 padrões que são categorizados da seguinte forma:

- **Criacionais**: padrões que propõem soluções flexíveis para criação de objetos.
- **Estruturais**: padrões que propõem soluções flexíveis para composição de classes e objetos.
- **Comportamentais**: padrões que propõem soluções flexíveis para interação e divisão de responsabilidades entre classes e objetos.

Figura 3 – Tabela Periódica de Padrões.



Fonte: Google Imagens.

Nos próximos capítulos, vamos apresentar os padrões de projeto propriamente ditos. Para facilitar a compreensão, vamos entender como os padrões estão estruturados por meio de um contexto, de um problema e da solução proposta. Os exemplos foram baseados no livro Engenharia de Software Moderna (VALENTE, 2020). Recomenda-se a leitura do mesmo para um maior detalhamento. Além disso,

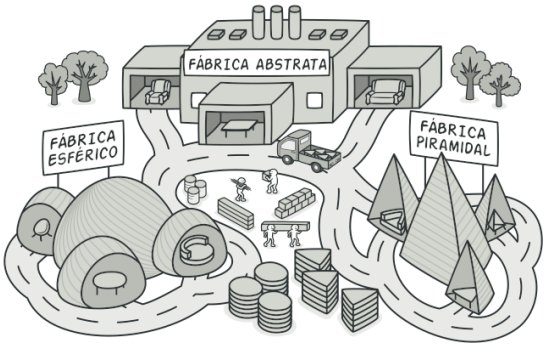
utilizamos alguns diagrama em UML baseado no repositório https://github.com/RafaelKuebler/PlantUMLDesignPatterns.

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 baseado em TCP/IP. Nesse sistema, três funções f, g e h criam objetos do tipo *TCPChannel* para comunicação remota.

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

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

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

Problema: Suponha que precisaremos usar UDP para comunicação. Sendo mais claro, gostaríamos de "parametrizar" o código acima para criar objetos dos tipos TCPChannel ou UDPChannel, dependendo dos clientes. O problema consiste em parametrizar a instanciação dos canais de comunicação de forma que ele consiga trabalhar com protocolos diferentes.

Solução: A solução adota um método que cria e retorna objetos de uma determinada classe e oculta o tipo desses objetos por trás de uma interface.

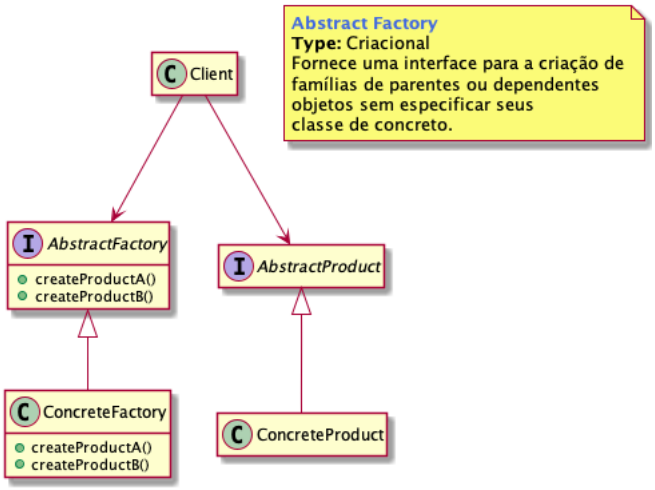
```
class ChannelFactory {
  public static Channel create() {// método fábrica estático
    return new TCPChannel();
  }
}

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

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

Vantagens:

- Os produtos que você obtém de uma fábrica são compatíveis entre si.
- Princípio de responsabilidade única.
- Princípio aberto/fechado.

Desvantagens:

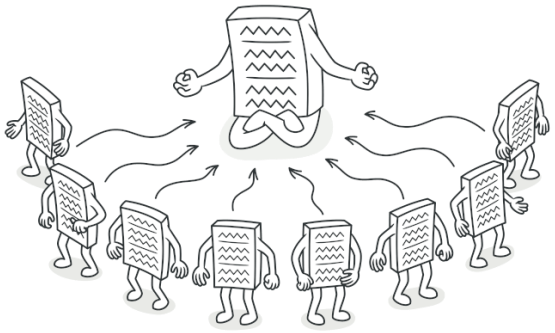
- O código pode tornar-se mais por causa de novas interfaces e classes pelo padrão.



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 de acesso global.

Figura 5 – O padrão Singleton.



Fonte: https://refactoring.guru.

Contexto: Suponha uma classe Logger, usada para registrar as operações realizadas em um sistema.

```
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: Seria interessante se todos os usos da classe Logger tivessem como alvo a mesma instância da classe. Em outras palavras, gostaríamos que existisse, no máximo, uma única instância dessa classe e que ela fosse usada em todas as partes do sistema.

Solução: A solução para esse problema consiste em transformar a classe Logger em um Singleton. Esse padrão de projeto define como implementar classes que terão, como o próprio nome indica, no máximo uma instância.

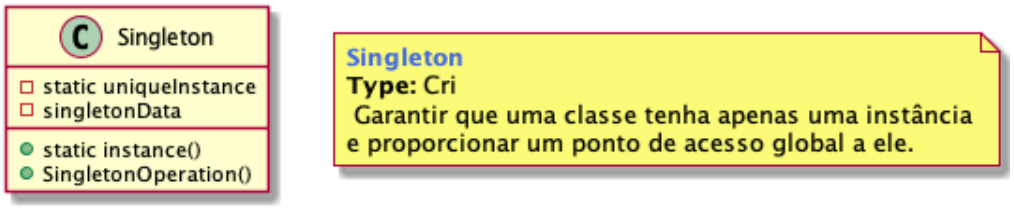
```
class Logger {

  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:

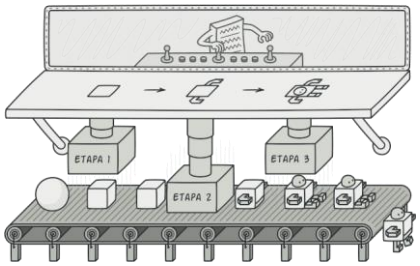
- Pode camuflar a criação de variáveis e estruturas de dados globais.
- Tornam o teste automático de métodos mais complicado.
- Pode mascarar um design ruim.



Builder

O Builder é um padrão de projeto criacional que permite a você construir objetos complexos passo a passo. Os atributos somente podem ser definidos em tempo de instanciação da classe.

Figura 6 – O padrão Builder.



Fonte: https://refactoring.guru.

Contexto: Imagine um objeto complexo com uma inicialização trabalhosa, seja porque possui muitos campos e objetos agrupados. Muitas das vezes, a inicialização é realizada através de um construtor monstruoso com vários parâmetros. Para um exemplo mais concreto, imagine um objeto *Livro* com diversos atributos, nem todos obrigatórios. Caso não seja informado o valor dos atributos opcionais, eles devem ser inicializados com um valor default.

Problema: Para o exemplo do objeto livro seria criar diversos construtores, por exemplo, uma para o caso dos atributos obrigatórios, outros para os cenários com atributos opcionais. Na prática, o desenvolvedor teria que conhecer exatamente a ordem dos diversos parâmetros, o que poderia deixar o desenvolvimento mais complexo e o código difícil de manter.

Solução: O padrão Builder sugere que você extraia o código de construção do objeto para fora de sua própria classe e mova ele para objetos separados chamados builders. Para criar um objeto você executa uma série de etapas em um objeto builder. A parte importante é que você não precisa chamar todas as etapas. Você chama apenas aquelas etapas que são necessárias para a produção de uma configuração específica de um objeto. Na imagem a seguir, visualizamos a utilização dos métodos de “builder”.

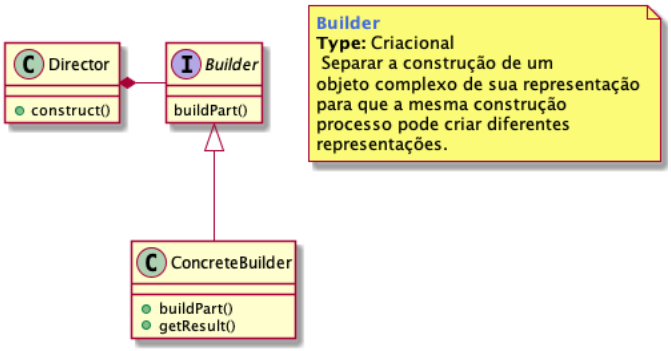
```
Livro esm = new Livro.Builder().
                setNome("Engenharia Soft Moderna").
                setEditora("UFMG").setAno(2020).build();

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

Vantagens:

- Possibilidade de construir objetos passo a passo.
- Reuso do código de construção para diferentes representações do produto.
- Princípio de responsabilidade única.

Desvantagens:

- A complexidade do código aumenta devido ao padrão criar múltiplas classes novas.

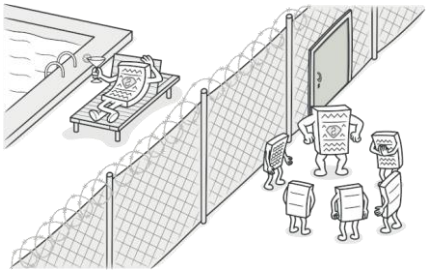

Director
construct()
Builder
buildPart()
ConcreteBuilder
buildPart()
getResult()
Builder
Type: Criacional
Separar a construção de um objeto complexo de sua representação para que a mesma construção processo pode criar diferentes representações.

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 do pedido chegar ao objeto original.

Figura 6 – 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:

```
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 acede. Sendo assim, antes de pesquisar por um livro, iremos verificar se ele está no cache. Se sim, o livro será imediatamente retornado. Caso contrário, a pesquisa prosseguirá segundo a lógica normal do método **getBook()** gostaríamos de 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).

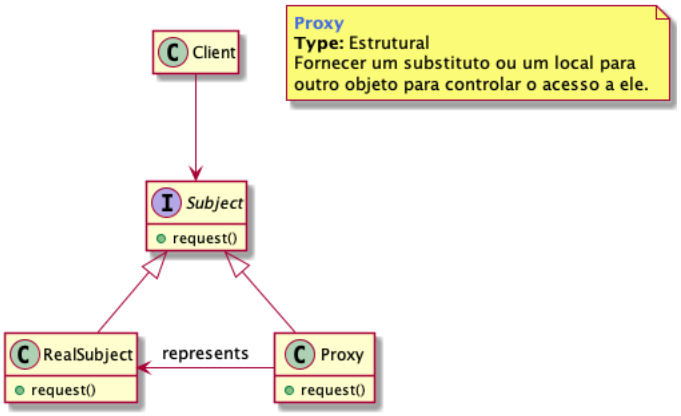
Solução: Por meio do padrão de projeto Proxy, criamos um objeto intermediário entre um objeto base e seus clientes. Assim, os clientes não terão mais uma referência direta para o objeto base, mas sim para o proxy. Por sua vez, o proxy possui uma referência para o objeto base. Além disso, o proxy deve implementar as mesmas interfaces do objeto base.

```
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 objetivo de um proxy é mediar o acesso a um objeto base, agregando-lhe funcionalidades, sem que ele tome conhecimento disso. No nosso caso, o objeto base é do tipo **BookSearch**; a funcionalidade que pretendemos agregar é um cache; e o proxy é um objeto da seguinte classe:

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

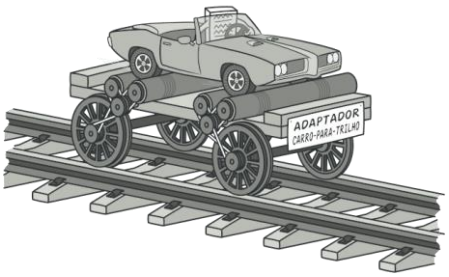
.



Adapter

O Adapter é um padrão de projeto estrutural que permite objetos com interfaces incompatíveis colaborarem entre si.

Figura 7 – O padrão Adapater.



Fonte: https://refactoring.guru.

Contexto: Suponha um sistema que tenha que controlar projetores multimídia. Para isso, ele deve instanciar objetos de classes fornecidas pelos fabricantes de cada projetor, como ilustrado a seguir:

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

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

Em um cenário real, pode envolver classes de outros fabricantes de projetores. Importante notar que métodos e atributos podem ser diferentes para cada fabricante. Particularmente, o método mostrado é responsável por ligar o projetor. No caso dos projetores da Samsung, esse método não possui parâmetros. No caso dos projetores da LG, podemos passar um intervalo em minutos para ligação do projetor.

Problema: No sistema de controle de projetores multimídia, queremos usar uma interface única para ligar os projetores, independentemente de marca. As classes de cada projetor foram implementadas pelos seus fabricantes e não temos acesso ao código dessas classes para fazer com que elas implementem uma interface, por exemplo.

```
interface Projetor {

  void liga();

}
...
class SistemaControleProjetores {

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

}
```

Solução: O padrão de projeto permite converter a interface de uma classe para outra interface, esperada pelos seus clientes. No nosso exemplo, ele pode ser usado para converter a interface Projetor para as interfaces (métodos públicos) das classes implementadas pelos fabricantes dos projetores.

```
class AdaptadorProjetorSamsung implements Projetor {

   private ProjetorSamsung projetor;

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

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

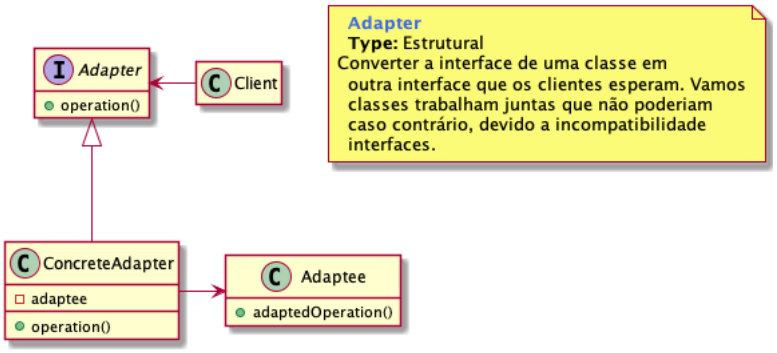
A classe **AdaptadorProjetorSamsung** implementa a interface **Projetor** e possui um atributo privado do tipo **ProjetorSamsung**. Para ligar o projeto um cliente chama o método chama **liga()** da classe adaptadora que em seguida chama o método equivalente — no caso, **turnOn()** — do objeto que está sendo adaptado.

Vantagens:

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

Desvantagens:

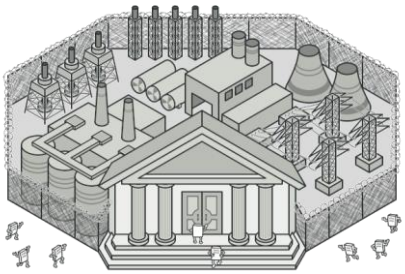
- O código pode tornar-se mais por causa de novas interfaces e classes pelo padrão.



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.

Figura 8 – O padrão Facade.



Fonte: https://refactoring.guru.

Contexto: Suponha que implementamos um interpretador para uma linguagem X. Esse interpretador permite executar programas X a partir de uma linguagem hospedeira, no caso Java. Se quiser tornar o exemplo mais real, imagine que X é uma linguagem para consulta a dados, semelhante a SQL. Para executar programas X, a partir de um código em Java, os seguintes passos são necessários:

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

Problema: Como a linguagem X está ficando popular, os desenvolvedores estão reclamando da complexidade do código acima, pois ele requer conhecimento de classes internas do interpretador de X. Logo, os usuários frequentemente pedem uma interface mais simples para chamar o interpretador da linguagem X

Solução: O padrão de projeto Fachada é uma solução para o nosso problema. Uma Fachada é uma classe que oferece uma interface mais simples para um sistema. 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 Fachada. As classes internas ficam encapsuladas por trás dessa Fachada.

```
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();
  }
}
```

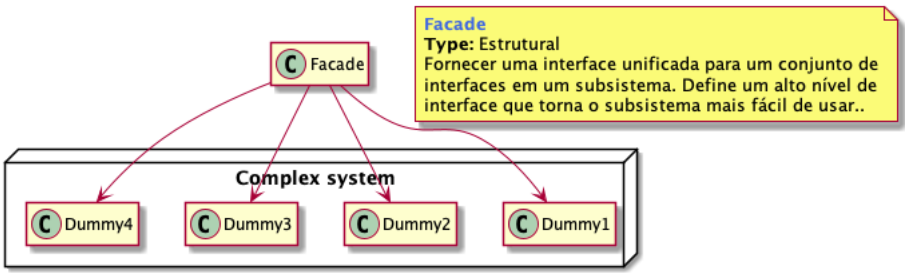
Antes de implementar a fachada, os clientes precisavam criar três objetos de tipos internos do interpretador e chamar dois métodos. Agora, basta criar um único objeto e chamar eval.

Vantagens:

- Permite isolar seu código da complexidade de um subsistema.

Contras:

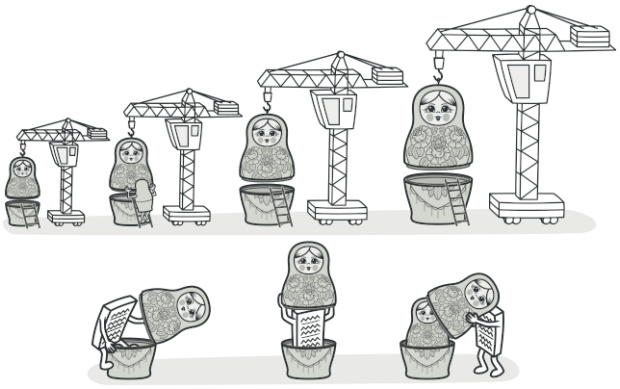
- Uma fachada pode se tornar um objeto Deus acoplado.



Decorator

O Decorator é um padrão de projeto estrutural que permite que você acople novos comportamentos para objetos ao colocá-los dentro de invólucros de objetos que contém os comportamentos.

Figura 9 – O padrão Decorator.



Fonte: https://refactoring.guru.

Contexto: Vamos voltar ao sistema de comunicação remota usado para explicar o Padrão Fábrica. Suponha que as classes **TCPChannel** e **UDPChannel** implementam uma interface **Channel**:

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

class TCPChannel implements Channel {
   ...
}

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

Problema: Os clientes dessas classes precisam adicionar funcionalidades extras em canais, tais como buffers, compactação das mensagens, log das mensagens trafegadas, etc. Mas essas funcionalidades são opcionais: dependendo do cliente precisamos de apenas algumas funcionalidades ou, talvez, nenhuma delas. Uma primeira solução consiste no uso de herança para criar subclasses com cada possível seleção de funcionalidades. Nessa solução, usamos herança para implementar subclasses para cada conjunto de funcionalidades. Suponha que o usuário precise de um canal UDP com buffer e compactação.

Solução: O Padrão Decorator representa uma alternativa a herança quando se precisa adicionar novas funcionalidades em uma classe base. Em vez de usar

herança, usa-se composição para adicionar tais funcionalidades dinamicamente nas classes base. No nosso problema, ao optarmos por decoradores, o cliente poderá configurar um Channel da seguinte forma:

```
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
```

Portanto, em uma solução com decoradores, a configuração de um **Channel** é feita no momento da sua instanciação, por meio de uma sequência aninhada de operadores new. O new mais interno sempre cria uma classe base, no nosso exemplo **TCPChannel** ou **UDPChannel**. Feito isso, os operadores mais externos são usados para "decorar" o objeto criado com novas funcionalidades. Os decoradores propriamente ditos, como **ZipChannel** e **BufferChannel** são subclasses da classe **ChannelDecorator** que é fundamental para o funcionamento do padrão Decorador. Porém, as classes de cada projetor — mostradas anteriormente — foram implementadas pelos seus fabricantes e estão prontas para uso. Ou seja, não temos acesso ao código dessas classes para fazer com que elas implementem a interface Projetor.

```
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();
  }

}
```

Por fim, chegamos aos decoradores reais. Eles são subclasses de **ChannelDecorator**, como no código a seguir, que implementa um decorador que compacta e descompacta as mensagens trafegadas pelo canal:

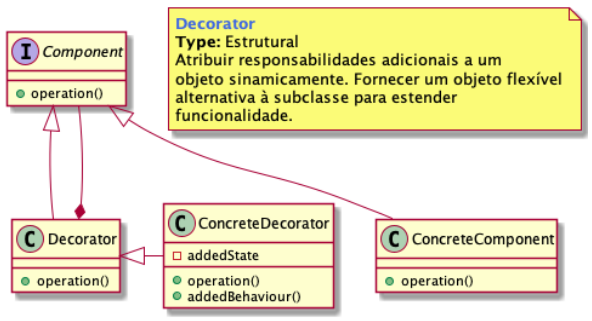
```
class ZipChannel extends ChannelDecorator {

   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 no momento da 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



Capítulo 5. Padrões de Projeto Comportamentais

Strategy

O Strategy é um padrão de projeto comportamental que permite que você defina uma família de algoritmos, coloque-os em classes separadas, e faça os objetos deles intercambiáveis.

Figura 10 – O padrão Strategy.

Fonte: https://refactoring.guru.

Contexto: Suponha que estamos implementando um pacote de estruturas de dados, com a seguinte classe lista:

```
class MyList {

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

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

Problema: os nossos clientes estão solicitando que novos algoritmos de ordenação possam ser usados para ordenar os elementos da lista. Explicando melhor, eles querem ter a opção de alterar e definir, por conta própria, o algoritmo de ordenação. No entanto, a versão atual da classe sempre ordena a lista usando o algoritmo Quicksort. Se lembrarmos dos princípios de projeto que estudamos no

capítulo anterior, podemos dizer que a classe **MyList** não segue o princípio Aberto/Fechado, considerando o algoritmo de ordenação.

Solução: o Padrão Strategy é a solução para o nosso problema de "abrir" a classe MyList para novos algoritmos de ordenação, mas sem alterar o seu código fonte. O objetivo do padrão é parametrizar os algoritmos usados por uma classe com o objetivo de encapsular uma família de algoritmos e como torná-los intercambiáveis.

```
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 **MyList** e um método set foi criado para configurar esse algoritmo. 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) { ... }
}
```

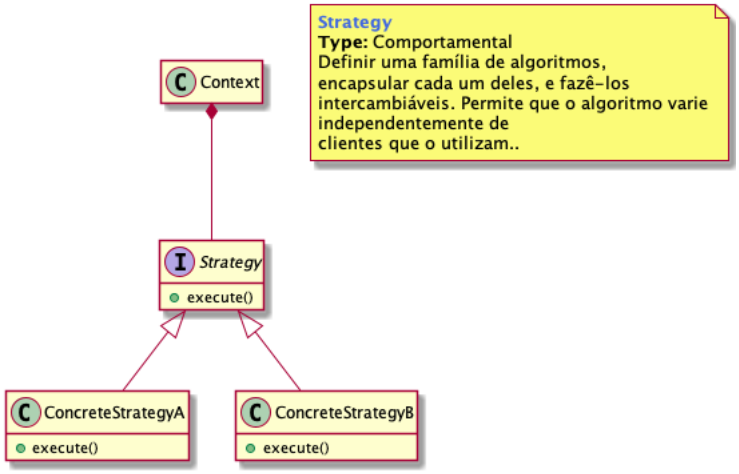
Vantagens:

- Trocar algoritmos usados dentro de um objeto durante a execução.
- Isolar os detalhes de implementação de um algoritmo do código que usa ele.

- Você pode introduzir novas estratégias sem mudar o contexto.

Desvantagens:

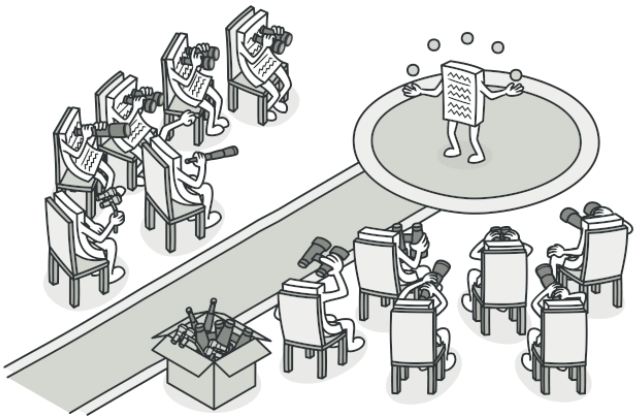
- Deve ser utilizado quando há diversos algoritmos para o mesmo problema.
- Os clientes devem estar cientes das diferenças entre as estratégias.
- Acaba não sendo muito útil em linguagens com suporte ao paradigma funcional.



Observer

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

Figura 11 – O padrão Observer.



Fonte: https://refactoring.guru.

Contexto: Suponha que estamos implementando um sistema para controlar uma estação meteorológica. Nesse sistema, temos que manipular objetos de duas classes: Temperatura, que são objetos de "modelo" que armazenam as temperaturas monitoradas na estação meteorológica; e Termômetro, que é uma classe usada para criar objetos visuais que exibem as temperaturas sob monitoramento. Termômetros devem exibir a temperatura atual que foi monitorada. Se a temperatura mudar, os termômetros devem ser atualizados.

Problema: Não queremos acoplar Temperatura (classe de modelo) a Termômetro (classe de interface). O motivo é simples: classes de interface mudam com frequência. Na versão atual, o sistema possui uma interface textual, que exibe temperaturas em Celsius no console do sistema operacional. Mas, em breve, pretendemos ter interfaces Web, para celulares e para outros sistemas. Pretendemos também oferecer outras interfaces de termômetros, tais como digital, analógico, etc. Por fim, temos mais classes semelhantes a Temperatura e Termômetro em nosso sistema, tais como: PressaoAtmosferica e Barômetro, UmidadeDoAr e Higrômetro, VelocidadeDoVento e Anemômetro, etc. Logo, na medida do possível, gostaríamos de reusar o mecanismo de notificação também nesses outros pares de classes.

Solução: O padrão Observador é a solução recomendada para o nosso contexto e problema. Esse padrão define como implementar uma relação do tipo um-

para-muitos entre objetos sujeito e observadores. Quando o estado de um sujeito muda, seus observadores devem ser notificados.

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

Esse programa cria um objeto do tipo Temperatura (um sujeito) e então adiciona dois observadores nele: um TermometroCelsius e um **TermometroFahrenheit**. Como a classe **Temperatura** herda de Subject acaba herdando os métodos **addObserver** para adicionar dois termômetros como observadores de uma instância de Temperatura e **notifyObservers** para notificar seus observadores de que o seu valor foi alterado no método **setTemp**.

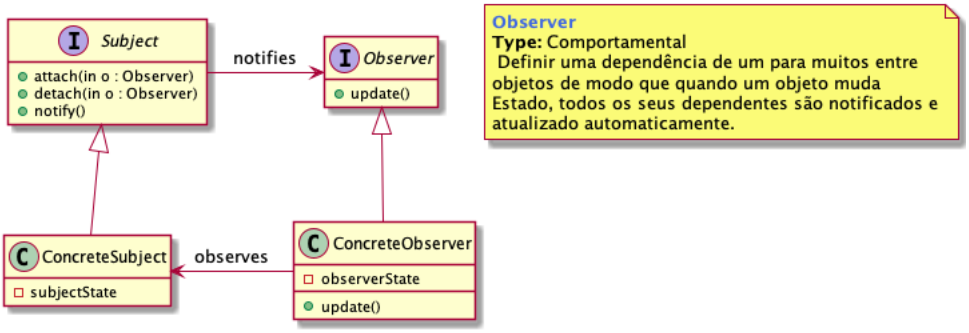
```
class Temperatura extends Subject {

  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);
  }
}
```

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.

Desvantagens:

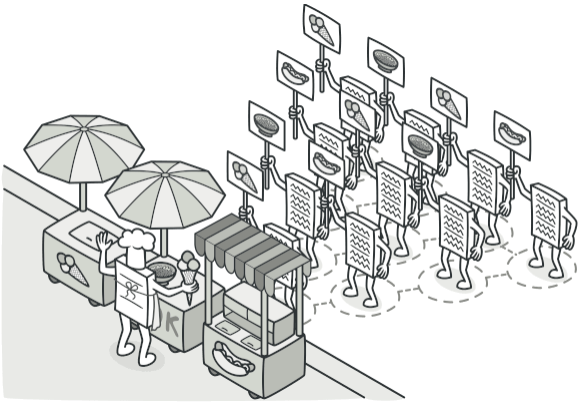
- Assinantes são notificados em ordem aleatória.



Visitor

O Visitor é um padrão de projeto comportamental que permite que você separe algoritmos dos objetos nos quais eles operam.

Figura 12 – 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 esses veículos estão armazenados em uma lista polimórfica que armazena objetos que sejam subclasses de Veículo.

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.

```
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"}
}
```

A classe **PrintVisitor** inclui métodos que imprimem os dados de um Carro, Ônibus e Motocicleta, como mostrado no código a seguir. É importante notar que o código a seguir 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.

Solução: A solução reside em usar o padrão de projeto Visitor que define como "adicionar" uma operação em uma família de objetos, sem que seja preciso modificar as respectivas classes. O primeiro passo consiste um método **accept** em cada classe da hierarquia. Nas subclasses, ele recebe como parâmetro um objeto do tipo **Visitor**. E a sua implementação apenas chama o método **visit** desse Visitor, passando **this,** ou seja, o próprio objeto como parâmetro.

```
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
```

Por fim, temos que modificar o laço que percorre a lista de veículos estacionados, conforme imagem a seguir.

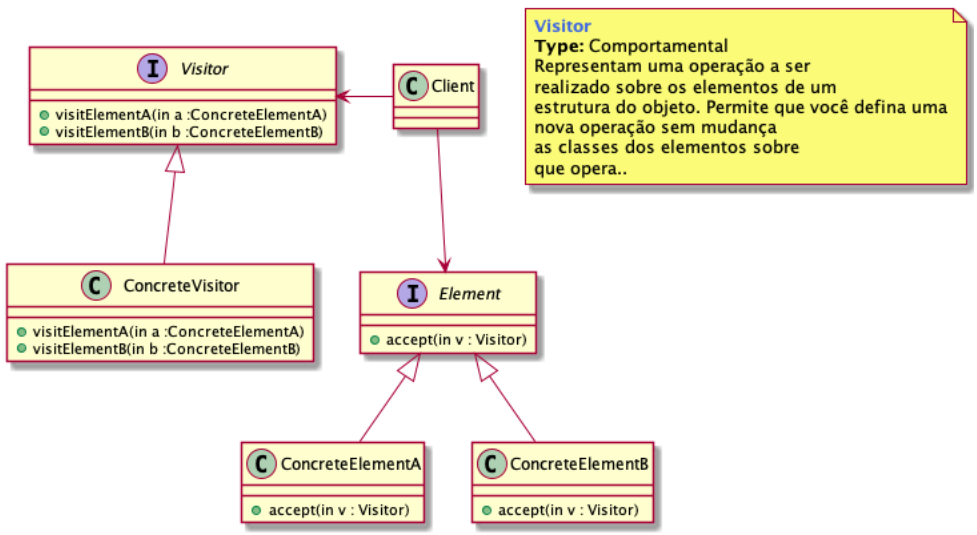
```
PrintVisitor visitor = new PrintVisitor();
foreach (Veiculo veiculo: listaDeVeiculosEstacionados) {
  veiculo.accept(visitor);
}
```

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:

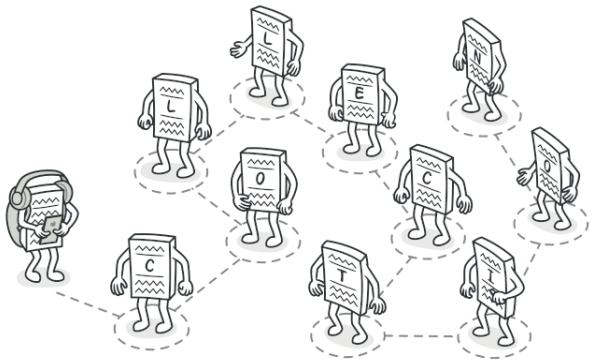
- Podem forçar uma quebra no encapsulamento das classes que serão visitadas.
- Necessário atualizar todos os visitantes a cada vez que a classe é adicionada ou removida da hierarquia de elementos.



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. Iterators também permitem que múltiplos caminhamentos sejam realizados de forma simultânea em cima da mesma estrutura de dados.

Figura 13 – O padrão Iterator.



Fonte: https://refactoring.guru.

Contexto: Suponha um sistema com diferentes tipos de coleções. Tais coleções podem estar baseadas em pilhas, árvores, grafos e outras estruturas complexas de dados.

Problema: Muitas vezes precisamos todos os itens da coleção. A tarefa é fácil se você tem uma coleção baseada em lista. Contudo, em estruturas como árvores determinar, o próximo elemento pode ser complexo. Além disso, para a classe de estrutura de dados implementar, um algoritmo de travessia desfoca a coleção da sua responsabilidade de armazenamento eficiente.

Solução: A ideia principal do padrão Iterator é extrair o comportamento de travessia de uma coleção para um objeto separado chamado um iterator. Em geral, a classe fornece dois métodos: **hasNext()** e **next()** para, respectivamente, validar se tem um próximo item e obter o próximo item.

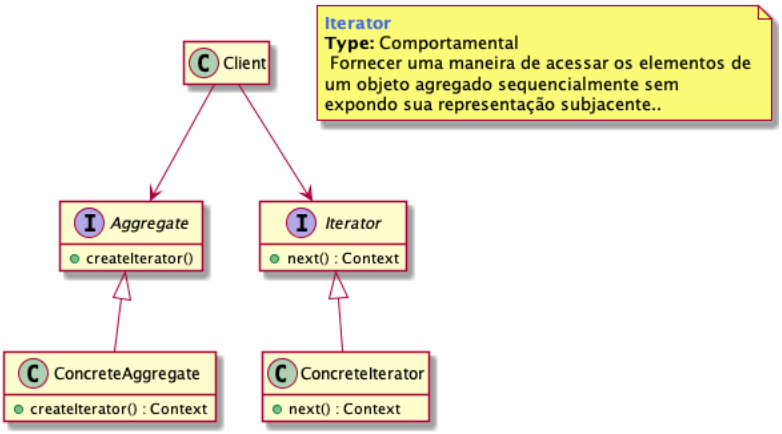
```
List<String> list = Arrays.asList("a","b","c");
Iterator it = list.iterator();
while(it.hasNext()) {
  String s = (String) it.next();
  System.out.println(s);
}
```

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.



Referências

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

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.

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. Disponível em: <https://engsoftmoderna.info>. Acesso em: 16 out. 2020.