

Orientação a Objetos e Padrões de Projeto.

Professor: Luan Felipe da Silva knebel



Metodologia de Ensino

- ▶ A metodologia de ensino que será utilizada nas aulas serão com as explicações teóricas de cada tópico com interações e exemplos práticos utilizando uma linguagem de programação orientada a objetos.



POO

O que é Orientação a Objetos

- ▶ Orientação a Objetos (OO) é um paradigma de programação que organiza o software em torno de "objetos", em vez de funções e lógica.
- ▶ Esses objetos são instâncias de "classes", que servem como modelos ou moldes para criar objetos com características e comportamentos específicos.

Qual o principal objetivo da POO?

O principal objetivo da programação **orientação a objetos (POO)** é facilitar o desenvolvimento de software mais modular, reutilizável, fácil de manter e evoluir.

Isso é alcançado ao modelar o software com base em "objetos", que são instâncias de classes que representam entidades do mundo real ou conceitos abstratos, com atributos (dados) e métodos (comportamentos).

Classes

- ▶ **Classe:** É um modelo ou que define as características (atributos) e comportamentos (métodos) comuns para um tipo de objeto. Por exemplo, uma classe "Carro" pode definir atributos como cor e modelo, e métodos como "acelerar" e "frear".

- ▶ **Características Importantes:**
 1. **Encapsulamento:** As classes encapsulam dados e comportamentos. Isso significa que você pode esconder a complexidade do objeto e expor apenas o que é necessário para a interação com ele.
 2. **Modularidade:** Classes ajudam a organizar o código em módulos distintos, cada um responsável por uma parte específica do comportamento do programa.
 3. **Reusabilidade:** Uma vez que você define uma classe, você pode criar múltiplos objetos com base nela, sem precisar reescrever o código para cada novo objeto.
 4. **Hierarquia e Herança:** Classes podem herdar de outras classes, permitindo criar novas classes que são especializações ou extensões de classes existentes.

Pacote (package)

- ▶ Em programação, especialmente em linguagens como Python, Java e C#, um **package** (ou pacote) é uma forma de organizar e estruturar o código em módulos relacionados. Um package ajuda a agrupar classes, funções e outros recursos de forma que facilite a modularidade, reutilização e manutenção do código.

Objetos

- ▶ Um **objeto** é uma instância de uma classe. Ele representa uma entidade concreta que possui estado e comportamento. O estado de um objeto é armazenado em seus atributos, e seu comportamento é definido pelos métodos da classe da qual ele é instanciado.

- ▶ **Principais Características dos Objetos**
 1. **Estado:** O estado de um objeto é representado pelos seus atributos. Esses atributos são variáveis associadas ao objeto que armazenam informações sobre o objeto. Por exemplo, em um objeto Carro, os atributos podem incluir modelo, cor, e ano.
 2. **Comportamento:** O comportamento de um objeto é definido pelos métodos da classe. Métodos são funções que operam sobre os atributos do objeto e podem realizar diversas ações, como acelerar o carro ou frear. Eles definem o que o objeto pode fazer.
 3. **Identidade:** Cada objeto tem uma identidade única, o que significa que mesmo que dois objetos tenham o mesmo estado e comportamento, eles são instâncias distintas e podem ser diferenciados.

Construtores

- ▶ Na programação orientada a objetos (POO), um **construtor** é um método especial utilizado para inicializar objetos de uma classe. Ele é chamado automaticamente quando um novo objeto da classe é criado e é responsável por configurar o estado inicial desse objeto.
- ▶ Isso significa que você pode passar valores para o construtor, que então atribuirá esses valores às propriedades do objeto.

Assinatura de métodos

- ▶ A **assinatura de um método** é a parte da declaração de um método que inclui seu nome e os parâmetros que ele recebe. Ela define como o método pode ser chamado e quais argumentos precisam ser passados para ele.
- ▶ **Componentes da Assinatura de um Método:**
 1. **Nome do Método:** O identificador que é usado para chamar o método.
 2. **Lista de Parâmetros:** Os tipos e nomes dos parâmetros que o método aceita, geralmente especificados entre parênteses.

Modificadores de acesso

- ▶ Os modificadores de acesso **public**, **private**, e **protected** são conceitos importantes em programação orientada a objetos, usados para controlar a visibilidade e o acesso aos membros de uma classe (atributos e métodos).
- ▶ Os modificadores ajudam a implementar o princípio de encapsulamento, permitindo que você proteja o estado interno do objeto e controle como e onde os dados podem ser acessados e modificados.

Encapsulamento

- ▶ **Encapsulamento** é um dos conceitos fundamentais da programação orientada a objetos e refere-se à prática de esconder os detalhes internos de uma classe e expor apenas o que é necessário para a interação com outros objetos. Ele permite que você crie objetos que têm uma interface pública bem definida, enquanto mantém a implementação interna escondida e protegida.
- ▶ **Benefícios do Encapsulamento**
 1. **Proteção de Dados:** Ao ocultar os detalhes internos e permitir o acesso apenas através de métodos controlados, você protege os dados de modificações indesejadas e garante que o estado do objeto permaneça consistente.
 2. **Modularidade:** O encapsulamento ajuda a dividir o código em módulos distintos e independentes. Isso facilita a manutenção e a atualização do código, pois mudanças internas em uma classe não afetam diretamente outras partes do sistema.
 3. **Facilidade de Manutenção:** Com o encapsulamento, você pode alterar a implementação interna de uma classe sem afetar o código que a usa, desde que a interface pública permaneça a mesma. Isso reduz o risco de introduzir erros ao modificar o código.
 4. **Reusabilidade:** A interface pública bem definida e o comportamento encapsulado permitem que as classes sejam reutilizadas em diferentes partes do sistema ou em diferentes projetos sem a necessidade de entender seus detalhes internos.

Composição

- **Composição** é um conceito fundamental na programação orientada a objetos (POO), que se refere à construção de classes complexas a partir da combinação de objetos de outras classes. Em vez de criar uma nova classe com todas as funcionalidades desde o zero, a composição permite que uma classe seja composta por instâncias de outras classes, delegando funcionalidades a esses componentes.

Herança

- ▶ **Herança** é um conceito fundamental na programação orientada a objetos que permite que uma classe (chamada de classe derivada ou subclasse) herde atributos e métodos de outra classe (chamada de classe base ou superclasse). Isso promove a reutilização de código e a criação de uma hierarquia de classes que compartilham comportamentos e características comuns.
- ▶ **Benefícios da Herança**
 1. **Reutilização de Código:** Permite que você reutilize o código existente da classe base sem ter que reescrevê-lo na subclasse. Isso reduz a duplicação de código e facilita a manutenção.
 2. **Extensão e Especialização:** Classes derivadas podem adicionar novos atributos e métodos ou modificar o comportamento dos métodos herdados, permitindo que você crie especializações e extensões de funcionalidades.
 3. **Hierarquia de Classes:** Herança ajuda a criar uma hierarquia de classes que reflete relações do mundo real. Isso torna o código mais intuitivo e organizado.

Polimorfismo

- ▶ **Polimorfismo** é um dos pilares fundamentais da programação orientada a objetos (POO). O termo vem do grego e significa "muitas formas". Em POO, o polimorfismo permite que objetos de diferentes classes sejam tratados como objetos de uma classe base comum, permitindo que uma única interface funcione com objetos de tipos diferentes.

- ▶ **Tipos de Polimorfismo:**

- ▶ **Polimorfismo de Sobrecarga (Overloading):**

Ocorre quando várias funções ou métodos com o mesmo nome são definidos, mas com diferentes assinaturas (número ou tipo de parâmetros).

- ▶ **Polimorfismo de Inclusão (Herança ou Subtipagem):**

Permite que uma classe derivada (ou filha) substitua um método da classe base (ou pai), podendo ser usada no lugar da classe base.

Abstração (Interfaces e Classes Abstratas)

- ▶ **Abstração** é um dos quatro pilares fundamentais da programação orientada a objetos (POO), ao lado de encapsulamento, herança e polimorfismo. A abstração refere-se ao processo de ocultar os detalhes complexos de implementação e mostrar apenas as funcionalidades essenciais de um objeto. Em outras palavras, a abstração permite que você se concentre no "o que" um objeto faz, em vez de "como" ele faz.
- ▶ **Conceitos-Chave da Abstração:**
 1. **Ocultação de Complexidade:** A abstração permite esconder os detalhes internos de funcionamento de um objeto ou sistema, expondo apenas a interface necessária para interagir com ele. Isso facilita o uso dos objetos, já que os usuários não precisam entender como eles funcionam internamente.
 2. **Interfaces e Classes Abstratas:** Em muitas linguagens orientadas a objetos, a abstração é implementada usando **interfaces** e **classes abstratas**.
 1. **Interface:** Define um contrato que outras classes devem seguir. Uma interface só declara métodos, sem implementá-los.
 2. **Classe Abstrata:** Similar a uma interface, mas pode conter tanto métodos abstratos (sem implementação) quanto métodos concretos (com implementação).

Próximos passos...

- ▶ Após termos vistos os conceitos básicos da programação orientada a objetos, devemos também conhecer os 5 princípios da programação orientada a objetos chamado de SOLID, estes princípios trazem as melhores práticas de desing de código para a POO.
- ▶ SOLID é um acrônimo criado por Michael Feathers e Robert C. Martin (Uncle Bob) após observar que cinco princípios da orientação a objetos e design de código ajudam o programador a escrever códigos mais limpos, separando responsabilidades, diminuindo acoplamentos, facilitando na refatoração e estimulando o reaproveitamento do Código.

Princípios da Programação Orientada a Objetos - SOLID

S	Single Responsibility Principle (SRP) A class should have only one reason to change, meaning it should have a single, well-defined responsibility.
O	Open/Closed Principle (OCP) Software entities (e.g., classes, modules) should be open for extension but closed for modification. This promotes the idea of extending functionality without altering existing code.
L	Liskov Substitution Principle (LSP) Subtypes (derived classes) must be substitutable for their base types (parent classes) without altering the correctness of the program.
I	Interface Segregation Principle (ISP) Clients should not be forced to depend on interfaces they don't use. This principle encourages the creation of smaller, focused interfaces.
D	Dependency Inversion Principle (DIP) High-level modules should not depend on low-level modules; both should depend on abstractions. This promotes the decoupling of components through abstractions and interfaces.

Single Responsibility Principle - (SRP)

- ▶ O **Princípio da Responsabilidade Única (SRP)** é um dos cinco princípios SOLID do design orientado a objetos. Ele afirma que uma classe deve ter apenas uma razão para mudar, ou seja, deve ter apenas uma responsabilidade ou função. Esse princípio visa tornar o sistema mais fácil de entender, manter e expandir, garantindo que cada classe ou módulo em um sistema faça apenas uma coisa.
- ▶ **Conceitos-chave do SRP:**
 1. **Responsabilidade Única:** Uma classe deve ser responsável por apenas uma parte da funcionalidade oferecida pelo software. Se uma classe tem mais de uma responsabilidade, ela se torna mais complexa, difícil de manter e mais propensa a erros.
 2. **Razão para Mudar:** Se uma classe tem múltiplas responsabilidades, mudanças em uma responsabilidade podem afetar ou quebrar as outras responsabilidades. Garantindo que uma classe tenha apenas uma responsabilidade, mudanças nessa responsabilidade são isoladas e não impactam outras partes do sistema.

Single Responsibility Principle - (SRP)

- ▶ Exemplo do problema:
- ▶ Muitas responsabilidades expostas em uma única classe (God Class).
- ▶ Como resolver:
- ▶ Separar as responsabilidades por contextos em classes diferentes.

```
PedidoVenda.java X
1 package com.unochapeco.treinamento;
2
3 public class PedidoVenda {
4
5     public void salvarPedido() {
6     }
7     public void excluirPedido() {
8     }
9     public void atualizarPedido() {
10    }
11    public void imprimirPedido() {
12    }
13    public void exportarPedido() {
14    }
15    public void calcularPedido() {
16    }
17    public void removerItemPedido() {
18    }
19    public void adicionarItemPedido() {
20    }
21
22 }
23
```

Open Closed Principle - (OCP)

- ▶ O sistema deve ser **aberto para extensão**, mas **fechado para modificação**. Isso significa que você deve poder adicionar novas funcionalidades ao sistema sem alterar o código existente.
- ▶ **Explicação:**
 1. **Aberto para Expansão:**
 1. O código deve ser escrito de forma que novos comportamentos e funcionalidades possam ser adicionados com facilidade. Isso geralmente é feito através de abstrações, como interfaces ou classes base, que permitem que novas classes sejam criadas para estender o comportamento existente.
 2. **Fechado para Modificação:**
 1. O código existente não deve ser modificado quando novas funcionalidades são necessárias. Alterações no código existente podem introduzir bugs ou quebrar o comportamento que já está funcionando. Ao invés de modificar o código existente, você deve criar novos componentes que estendam a funcionalidade.

Open Closed Principle - (OCP)

- ▶ **Exemplo do problema:**
- ▶ Para que novos funcionarios possam ser adicionados, a classe que calcula o salário precisa ser alterada, expondo risco de modificar um comportamento já existentes. Impossibilitando que a classe seja expansível.
- ▶ **Como resolver:**
- ▶ Abstrair a classe funcionário e criar subtipos de funcionários para isolar o código de cada calculo de salário.

```
SalarioFuncionario.java X Funcionario.java
1 package com.unochapeco.treinamento;
2
3 import java.math.BigDecimal;
4
5 public class SalarioFuncionario {
6
7     public BigDecimal calcularSalario(Funcionario funcionario) {
8         if(funcionario.tipoFuncionario == "Analista") {
9             return new BigDecimal(1500.00);
10        }
11        if(funcionario.tipoFuncionario == "Programador") {
12            return new BigDecimal(2500.00);
13        }
14        return BigDecimal.ZERO;
15    }
16
17 }
18 }
```

Liskov Substitution Principle - (LSP)

- ▶ O Princípio de Substituição de Liskov (LSP - *Liskov Substitution Principle*) é o terceiro princípio dos cinco que compõem o SOLID. Ele foi introduzido por Barbara Liskov em 1987 e afirma que **subtipos devem ser substituíveis por seus tipos base** sem alterar as propriedades corretas do programa.
- ▶ **Definição do LSP:**
- ▶ O Princípio de Substituição de Liskov afirma que se S é um subtipo de T, então os objetos do tipo T em um programa devem poder ser substituídos por objetos do tipo S sem que isso quebre o funcionamento correto do programa.
- ▶ Em termos mais simples, as classes derivadas (ou subclasses) devem ser substituíveis por suas classes base (ou superclasses) sem que o comportamento do programa seja alterado.

Liskov Substitution Principle - (LSP)

- ▶ Para que o LSP seja respeitado:
 1. A subclasse não deve enfraquecer a funcionalidade da superclasse. Todos os métodos e propriedades que existem na superclasse devem funcionar de maneira consistente quando aplicados a instâncias da subclasse.
 2. A subclasse não deve lançar exceções inesperadas ou adicionar pré-condições que não existiam na superclasse. Se a superclasse não requer uma verificação específica antes de executar uma operação, a subclasse também não deve requerer.
 3. A subclasse deve cumprir as promessas feitas pela superclasse. Isso significa que a subclasse deve manter o comportamento da superclasse, inclusive garantindo que os métodos retornem valores esperados e que as postcondições (resultados) permaneçam válidas.
- ▶ Exemplos de violação do LSP:
 - Sobrescrever/implementar um método que não faz nada;
 - Lançar uma exceção inesperada;
 - Retornar valores de tipos diferentes da classe base;

Liskov Substitution Principle - (LSP)

- ▶ Exemplo do problema:
- ▶ O método sacar em ContaPoupanca tem um comportamento diferente do método sacar na classe Conta.
- ▶ Se um código for projetado para funcionar com objetos da classe Conta, ele pode não funcionar corretamente com objetos da classe ContaPoupanca devido à restrição adicional de saldo mínimo.
- ▶ Por exemplo, se o saldo mínimo for negativo, o valor sacado pode ser menor que o saldo da conta violando a regra da classe Conta.

```
Conta.java X ContaPoupanca.java
1 package com.unochapeco.treinamento;
2
3 public class Conta {
4
5     protected double saldo;
6
7     public Conta(double saldoInicial) {
8         this.saldo = saldoInicial;
9     }
10
11     public void sacar(double quantia) {
12         if (quantia > 0 && quantia <= saldo) {
13             saldo -= quantia;
14         } else {
15             throw new IllegalArgumentException("Quantia inválida ou saldo insuficiente.");
16         }
17     }
18
19     public double getSaldo() {
20         return saldo;
21     }
22 }

Conta.java X ContaPoupanca.java X
1 package com.unochapeco.treinamento;
2
3 public class ContaPoupanca extends Conta {
4
5     private double saldoMinimo;
6
7     public ContaPoupanca(double saldoInicial, double saldoMinimo) {
8         super(saldoInicial);
9         this.saldoMinimo = saldoMinimo;
10    }
11
12    @Override
13    public void sacar(double quantia) {
14        if ((saldo - quantia) >= saldoMinimo) {
15            saldo -= quantia;
16        } else {
17            throw new IllegalArgumentException("Saldo insuficiente para manter o saldo mínimo.");
18        }
19    }
20
21 }
22 }
```


Liskov Substitution Principle - (LSP)

- ▶ **Como resolver:**
- ▶ Para respeitar o LSP, precisamos garantir que a subclasse ContaPoupanca possa ser substituída por Conta sem alterar o comportamento esperado do programa.
- ▶ Uma abordagem seria não adicionar restrições adicionais no método sacar, mas sim tratar essas restrições de forma externa ou através de métodos adicionais.

```
Conta.java  ContaPoupanca.java X
1 package com.unochapeco.treinamento;
2
3 public class ContaPoupanca extends Conta {
4
5     private double saldoMinimo;
6
7     public ContaPoupanca(double saldoInicial, double saldoMinimo) {
8         super(saldoInicial);
9         this.saldoMinimo = saldoMinimo;
10    }
11
12    @Override
13    public void sacar(double quantia) {
14        if (podeSacar(quantia)) {
15            super.sacar(quantia);
16        } else {
17            throw new IllegalArgumentException("Saldo insuficiente para manter o saldo mínimo.");
18        }
19    }
20
21    public boolean podeSacar(double quantia) {
22        return (saldo - quantia) >= saldoMinimo;
23    }
24
25 }
26
```

Interface Segregation Principle - (ISP)

- ▶ O Princípio da Segregação de Interfaces (ISP - *Interface Segregation Principle*) é o quarto princípio do SOLID. Ele diz que uma classe não deve ser forçada a implementar interfaces que ela não usa. Em outras palavras, é melhor ter várias interfaces específicas e pequenas, em vez de uma única interface grande e geral.
- ▶ **Definição do ISP:**
- ▶ "Os clientes não devem ser forçados a depender de métodos que não utilizam."
- ▶ Isso significa que quando uma interface é muito grande e tem muitas responsabilidades, qualquer classe que a implemente será forçada a implementar métodos que não são relevantes para seu propósito específico. Isso viola a coesão e pode levar a um design de código rígido e difícil de manter.

Interface Segregation Principle - (ISP)

- ▶ Exemplo de problema:
- ▶ Implementação Forçada:
 - A classe ImpressoraBasica é forçada a implementar métodos que não fazem sentido para ela (escanearDocumento e enviarFax). Isso leva à necessidade de lançar exceções ou deixar métodos sem implementação, o que é uma má prática.
- ▶ Baixa Coesão:
 - A interface Impressora é grande e faz mais do que uma coisa, o que viola a coesão. Como resultado, as classes que a implementam são obrigadas a lidar com responsabilidades que não lhes pertencem.

```
Impressora.java X ImpressoraBasica.java
1 package com.unochapeco.treinamento;
2
3 public interface Impressora {
4     void imprimirDocumento();
5     void escanearDocumento();
6     void enviarFax();
7 }
8
```

```
Impressora.java ImpressoraBasica.java X
1 package com.unochapeco.treinamento;
2
3 public class ImpressoraBasica implements Impressora {
4     @Override
5     public void imprimirDocumento() {
6         System.out.println("Imprimindo documento..");
7     }
8
9     @Override
10    public void escanearDocumento() {
11        throw new UnsupportedOperationException("Esta impressora não suporta escaneamento.");
12    }
13
14    @Override
15    public void enviarFax() {
16        throw new UnsupportedOperationException("Esta impressora não suporta fax.");
17    }
18 }
19
```

Interface Segregation Principle - (ISP)

- ▶ Como resolver:
- ▶ Podemos corrigir esse problema dividindo a interface Impressora em interfaces menores e mais específicas, para que cada classe implemente apenas os métodos que realmente precisa.

```
interface Impressora {  
    void imprimirDocumento();  
}
```

```
interface Escaneadora {  
    void escanearDocumento();  
}
```

```
interface Fax {  
    void enviarFax();  
}
```

Dependency Inversion Principle - (DIP)

- ▶ **Princípio da Inversão de Dependência (DIP)**
- ▶ O Princípio da Inversão de Dependência afirma que:
 - Módulos de alto nível não devem depender de módulos de baixo nível. Ambos devem depender de abstrações.
 - Abstrações não devem depender de detalhes. Detalhes devem depender de abstrações.

Dependency Inversion Principle - (DIP)

► Explicação Detalhada:

1. Módulos de Alto Nível vs. Módulos de Baixo Nível:

1. Módulos de alto nível são aqueles que contêm a lógica de negócio principal da aplicação. Eles lidam com a funcionalidade essencial.
2. Módulos de baixo nível, por outro lado, são aqueles que implementam detalhes específicos, como interações com bancos de dados, manipulação de arquivos ou comunicação com APIs.

► Dependência de Abstrações:

- Em vez de módulos de alto nível dependerem diretamente dos módulos de baixo nível, ambos devem depender de uma abstração, como uma interface ou uma classe abstrata.
- Isso significa que as dependências entre os componentes do sistema são baseadas em contratos (interfaces) e não em implementações concretas.

Dependency Inversion Principle - (DIP)

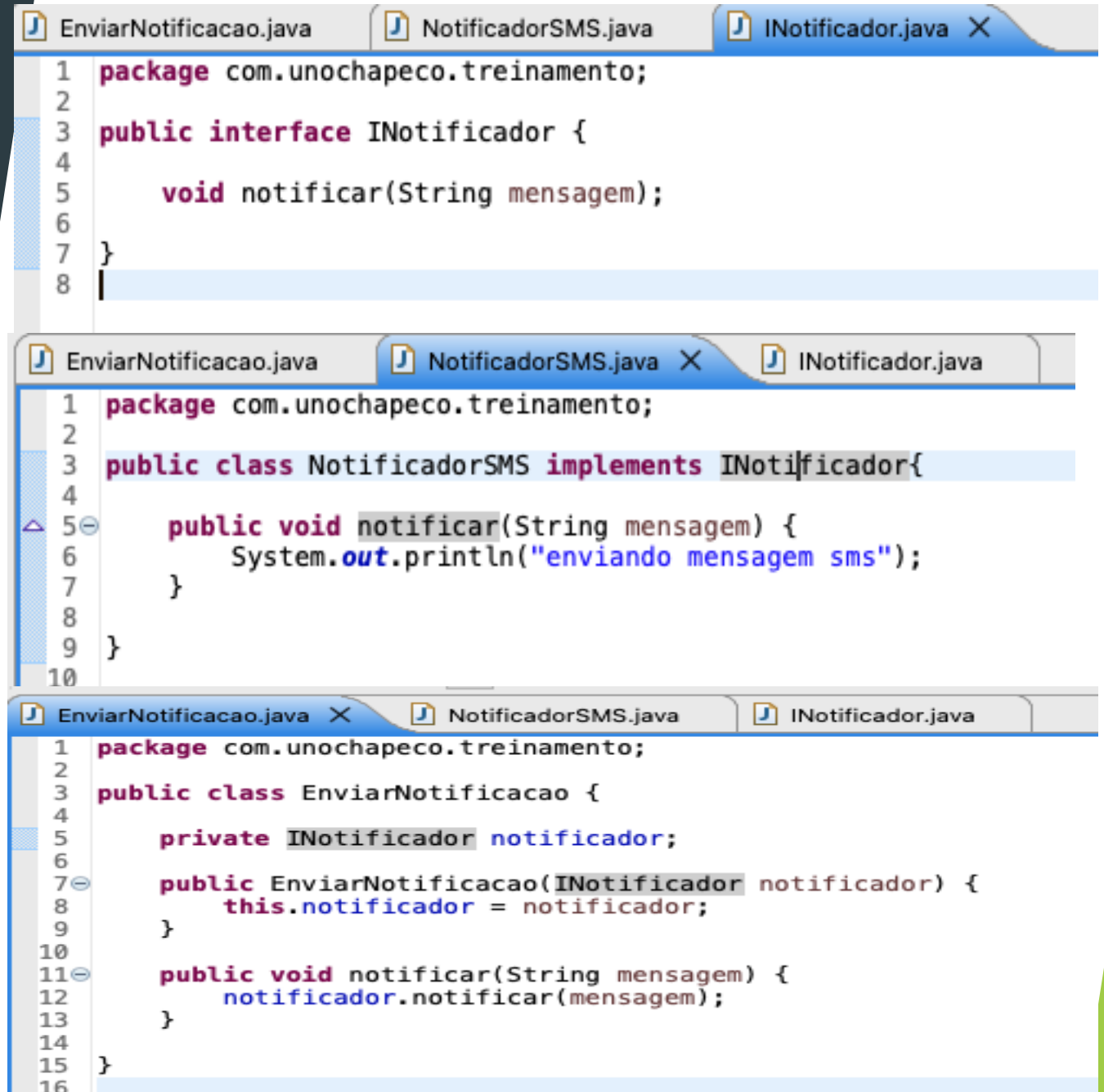
1. Exemplo de Problema:
2. Uma classe de alto nível que envia notificação possui acoplamento com uma classe de baixo nível para envio de SMS.
3. Como resolvermos o problema utilizando DIP para removermos esse acoplamento forte?

```
EnviarNotificacao.java X NotificadorSMS.java
1 package com.unochapeco.treinamento;
2
3 public class EnviarNotificacao {
4
5     public void notificar(String mensagem) {
6         new NotificadorSMS().notificar(mensagem);
7     }
8
9 }
10
```

Dependency Inversion Principle - (DIP)

Como resolver:

1. Criamos uma interface como um contrato para que o módulo de alto nível não dependa diretamente do módulo de baixo nível, removendo o acoplamento e possibilitando múltiplas formas de envio de notificação.



```
EnviarNotificacao.java | NotificadorSMS.java | INotificador.java X
1 package com.unochapeco.treinamento;
2
3 public interface INotificador {
4     void notificar(String mensagem);
5 }
6
7
8

EnviarNotificacao.java | NotificadorSMS.java X | INotificador.java
1 package com.unochapeco.treinamento;
2
3 public class NotificadorSMS implements INotificador{
4
5     public void notificar(String mensagem) {
6         System.out.println("enviando mensagem sms");
7     }
8
9 }
10

EnviarNotificacao.java X | NotificadorSMS.java | INotificador.java
1 package com.unochapeco.treinamento;
2
3 public class EnviarNotificacao {
4
5     private INotificador notificador;
6
7     public EnviarNotificacao(INotificador notificador) {
8         this.notificador = notificador;
9     }
10
11     public void notificar(String mensagem) {
12         notificador.notificar(mensagem);
13     }
14
15 }
16
```


Padrões de Projeto (Design Patterns)

Todos os exemplos a seguir foram obtidos pelo site de referência Refactoring Guru.

Fonte: <https://refactoring.guru>

Padrões de Projeto (Design Patterns)

Padrões de projeto (design patterns) são soluções típicas para problemas comuns em projeto de software. Cada padrão é como uma planta de construção que você pode customizar para resolver um problema de projeto particular em seu código.

► Fonte: <https://refactoring.guru/pt-br/design-patterns>

Benefícios dos Padrões

Padrões são um conjunto de ferramentas para soluções de problemas comuns em design de software. Eles definem uma linguagem comum que ajuda sua equipe a se comunicar mais eficientemente.

Fonte: <https://refactoring.guru/pt-br/design-patterns>

Classificação

Padrões de projeto diferem por sua complexidade, nível de detalhamento e grau de aplicabilidade. Além disso, eles podem ser categorizados por seu propósito e divididos em três grupos.

Padrões Criacionais, Estruturais e Comportamentais.

Fonte: <https://refactoring.guru/pt-br/design-patterns>



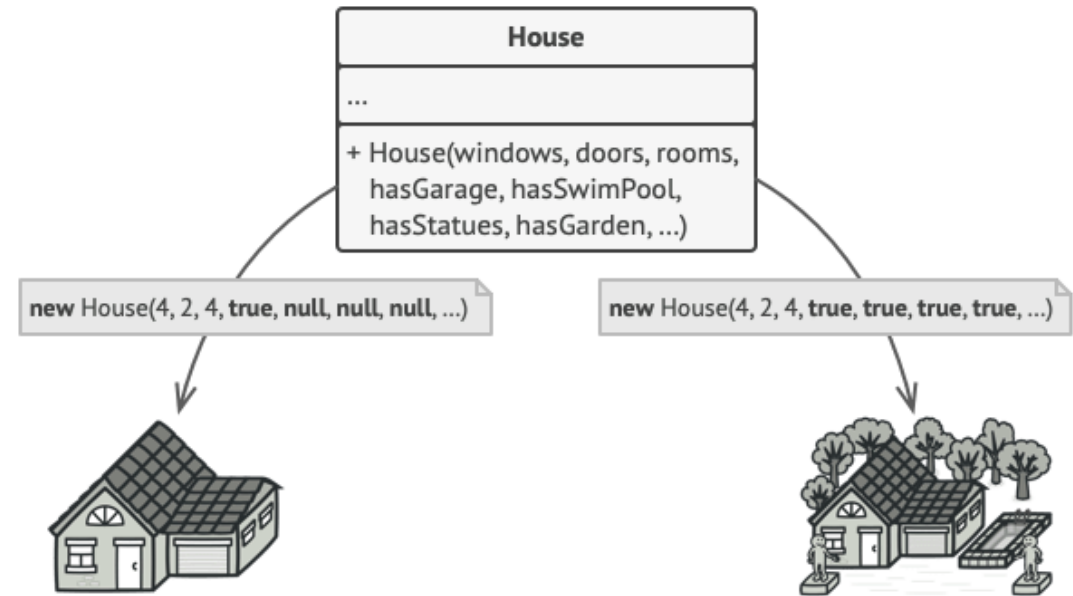
Exemplos de Padrões de Projetos

- ▶ O Catálogo de Padrões de Projeto é extenso e de certo modo complexo, com conceitos e soluções para diferentes necessidades.
- ▶ Os próximos slides demonstram alguns exemplos dos padrões mais básicos e utilizados na programação.

Builder

- ▶ O **Builder** é um padrão de projeto criacional que permite a você construir objetos complexos passo a passo.
- ▶ O padrão permite que você produza diferentes tipos e representações de um objeto usando o mesmo código de construção.

▶ Fonte: <https://refactoring.guru/pt-br/design-patterns/builder>



O construtor com vários parâmetros tem um lado ruim: nem todos os parâmetros são necessários todas as vezes.

Na maioria dos casos a maioria dos parâmetros não será usada, tornando as chamadas do construtor em algo feio de se ver. Por exemplo, apenas algumas casas têm piscinas, então os parâmetros relacionados a piscinas serão inúteis nove em cada dez vezes.

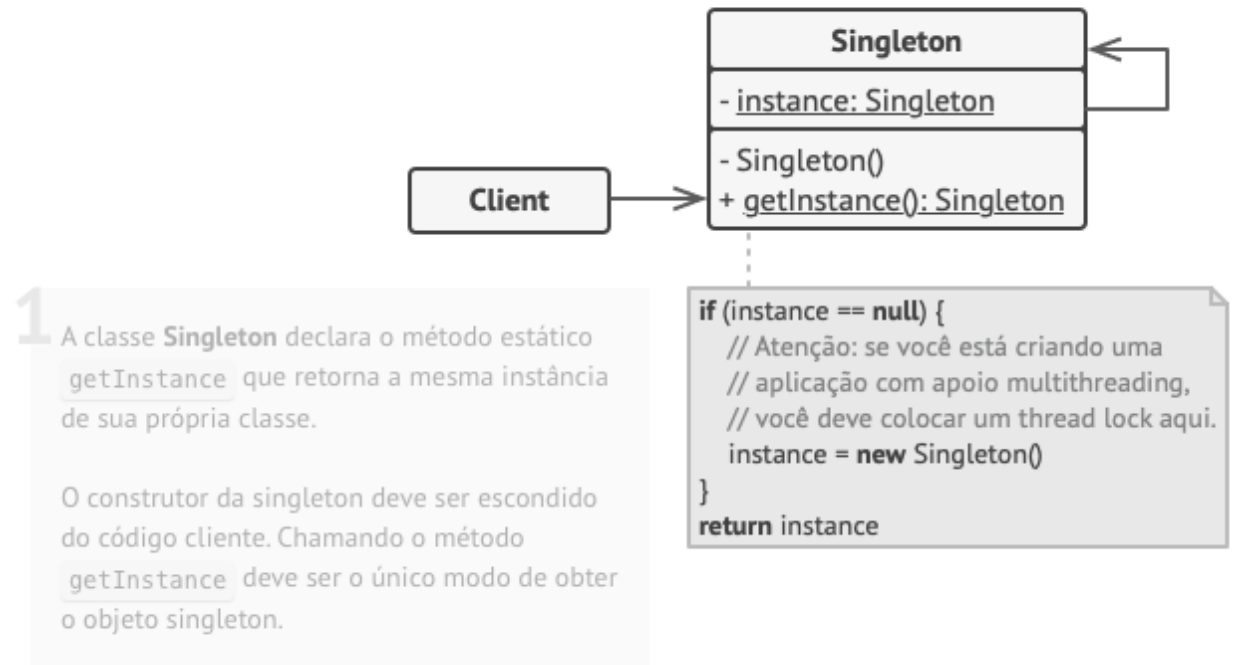
Singleton

- ▶ 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 para essa instância.
- ▶ O padrão Singleton resolve dois problemas de uma só vez, violando o *princípio de responsabilidade única*:
- ▶ **1 - Garantir que uma classe tenha apenas uma única instância.** Por que alguém iria querer controlar quantas instâncias uma classe tem? A razão mais comum para isso é para controlar o acesso a algum recurso compartilhado—por exemplo, uma base de dados ou um arquivo.
- ▶ **2 - Fornece um ponto de acesso global para aquela instância.** Se lembra daquelas variáveis globais que você (tá bom, eu) usou para guardar alguns objetos essenciais? Embora sejam muito úteis, elas também são muito inseguras uma vez que qualquer código pode potencialmente sobrescrever os conteúdos daquelas variáveis e quebrar a aplicação.
- ▶ Fonte: <https://refactoring.guru/pt-br/design-patterns/singleton>

Singleton

- ▶ O padrão Singleton permite que você acesse algum objeto de qualquer lugar no programa. Contudo, ele também protege aquela instância de ser sobrescrita por outro código.

- ▶ Fonte: <https://refactoring.guru/pt-br/design-patterns/singleton>



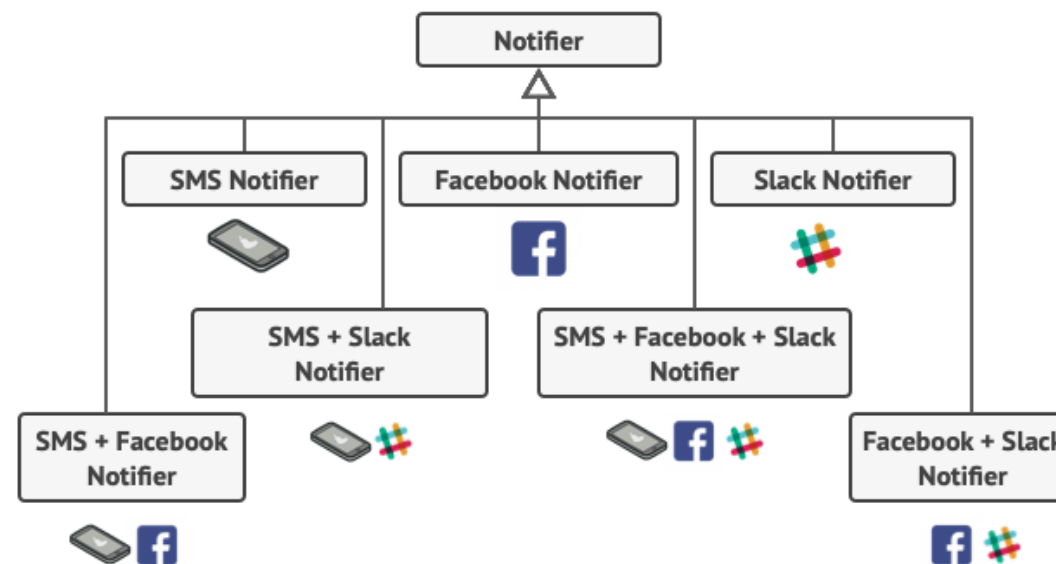
Decorator

- ▶ 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.

- ▶ Fonte: <https://refactoring.guru/pt-br/design-patterns/decorator>

Decorator

- Imagine que você está trabalhando em um biblioteca de notificação que permite que outros programas notifiquem seus usuários sobre eventos importantes.
- Em algum momento você se dá conta que os usuários da biblioteca esperam mais que apenas notificações por email.
- Muitos deles gostariam de receber um SMS acerca de problemas críticos. Outros gostariam de ser notificados no Facebook, e, é claro, os usuários corporativos adorariam receber notificações do Slack.
- Fonte: <https://refactoring.guru/pt-br/design-patterns>

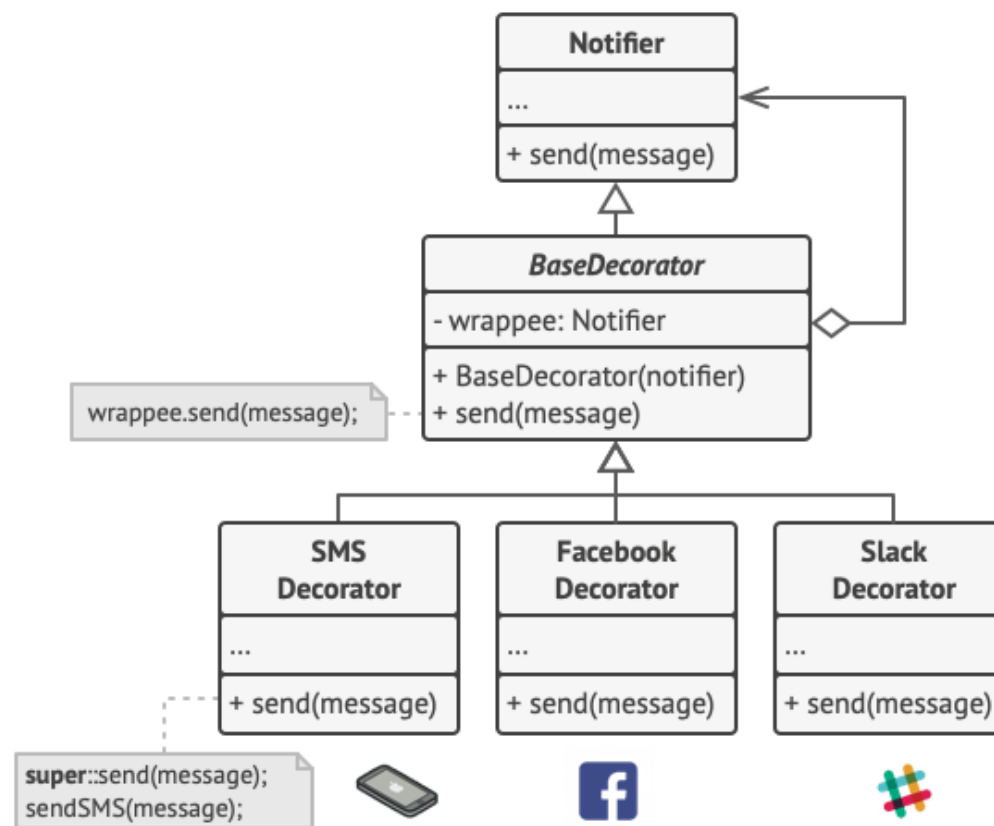


Combinação explosiva de subclasses.

Decorator

- ▶ Na solução utilizando o Decorator para o exemplo de notificações vamos deixar o simples comportamento de notificação por email dentro da classe Notificador base, mas transformar todos os métodos de notificação em decoradores.

- ▶ Fonte: <https://refactoring.guru/pt-br/design-patterns/decorator>



Vários métodos de notificação se tornam decoradores.

Decorator

- ▶ O código cliente vai precisar envolver um objeto notificador básico em um conjunto de decoradores que coincidem com as preferências do cliente. Os objetos resultantes serão estruturados como uma pilha.
- ▶ O último decorador na pilha seria o objeto que o cliente realmente trabalha. Como todos os decoradores implementam a mesma interface que o notificador base, o resto do código cliente não quer saber se ele funciona com o objeto “puro” do notificador ou do decorador.

- ▶ Fonte: <https://refactoring.guru/pt-br/design-patterns/decorator>

```
stack = new Notifier()
if (facebookEnabled)
    stack = new FacebookDecorator(stack)
if (slackEnabled)
    stack = new SlackDecorator(stack)

app.setNotifier(stack)
```



```
notifier.send("Alerta!")
// Email → Facebook → Slack
```

As aplicações pode configurar pilhas complexas de notificações decoradores

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.
- ▶ Imagine que você precisa fazer seu código funcionar com um amplo conjunto de objetos que pertencem a uma sofisticada biblioteca ou framework. Normalmente, você precisaria inicializar todos aqueles objetos, rastrear as dependências, executar métodos na ordem correta, e assim por diante.
- ▶ Como resultado, a lógica de negócio de suas classes vai ficar firmemente acoplada aos detalhes de implementação das classes de terceiros, tornando difícil compreendê-lo e mantê-lo.

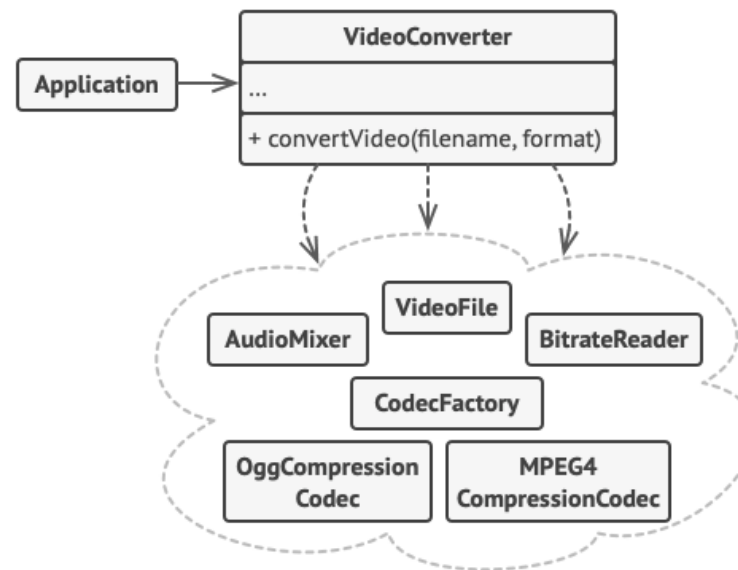
Facade

- ▶ Uma fachada é uma classe que fornece uma interface simples para um subsistema complexo que contém muitas partes que se movem. Uma fachada pode fornecer funcionalidades limitadas em comparação com trabalhar com os subsistemas diretamente. Contudo, ela inclui apenas aquelas funcionalidades que o cliente se importa.

Facade

- ▶ Ao invés de fazer seu código funcionar com dúzias de classes framework diretamente, você cria a classe fachada que encapsula aquela funcionalidade e a esconde do resto do código. Essa estrutura também ajuda você a minimizar o esforço usando para atualizar para futuras versões do framework ou substituí-lo por outro.

Neste exemplo, o padrão **Facade** simplifica a interação com um framework complexo de conversão de vídeo.



Um exemplo de isolamento de múltiplas dependências dentro de uma única classe fachada.

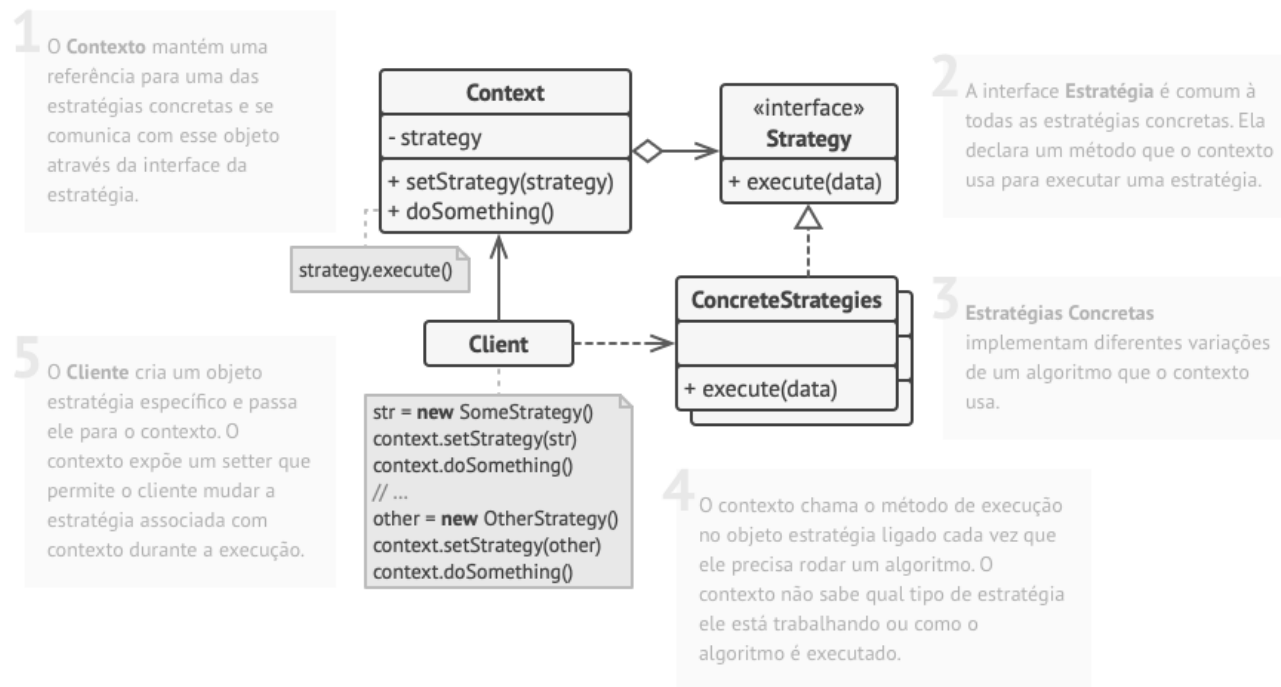
Strategy

- ▶ 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.

Strategy

- O padrão Strategy sugere que você pegue uma classe que faz algo específico em diversas maneiras diferentes e extraia todos esses algoritmos para classes separadas chamadas *estratégias*.

🏗️ Estrutura

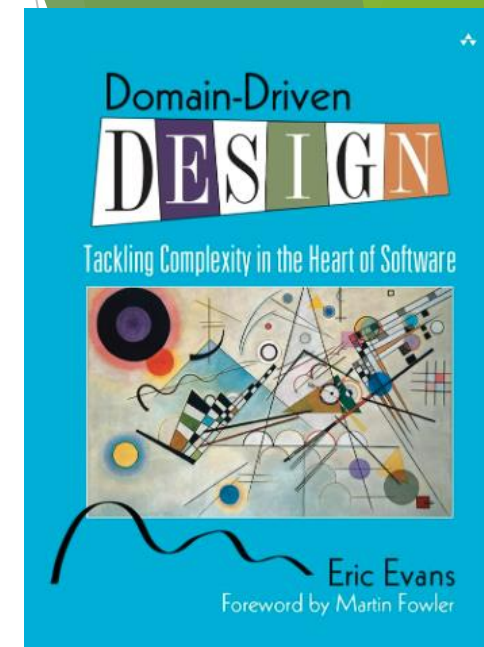
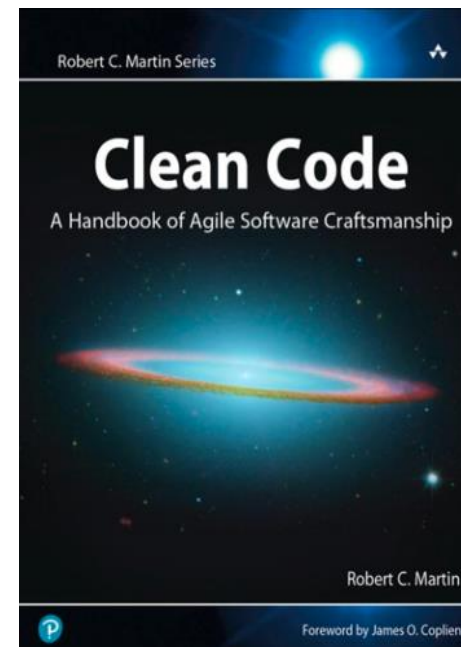


Muito Obrigado!

- ▶ A partir dos conceitos básicos, princípios da programação ao objetos e padrões de projetos podemos desenvolver aplicações profissionais de alto nível com baixo acoplamento, alta coesão e uma boa manutenibilidade.
- ▶ Os demais padrões podem ser estudados diretamente no site Refactoring Guru no link: <https://refactoring.guru>



Indicações de Livros



Design Patterns: <https://refactoring.guru/design-patterns/book>

Clean Code: <https://www.amazon.com.br/Código-limpo-Robert-C-Martin/dp/8576082675>

Domain Driven Design: <https://www.amazon.com.br/Domain-Driven-Design-Atacando-Complexidades-Software/dp/8550800651>