

# INE5404

# Padrões de Projeto

Prof. Jônata Tyska  
Prof. Mateus Grellert



UNIVERSIDADE FEDERAL  
DE SANTA CATARINA

# Desenvolvendo Sistemas Complexos

---

## Já sabemos:

- modelar sistemas utilizando o paradigma OO
- adicionar uma interface gráfica sistemas
- serializar os dados para reuso/compartilhamento
- Desenvolver sistemas seguindo uma arquitetura padrão com MVC
- Utilizar os princípios SOLID para tornar nosso código mais limpo e reusável

Vamos ver agora algumas técnicas de **padrões de projeto**

# Parte 1: Padrões de Projeto

# Padrões de Projeto

---

São **protótipos** de soluções recorrentes no desenvolvimento de SW

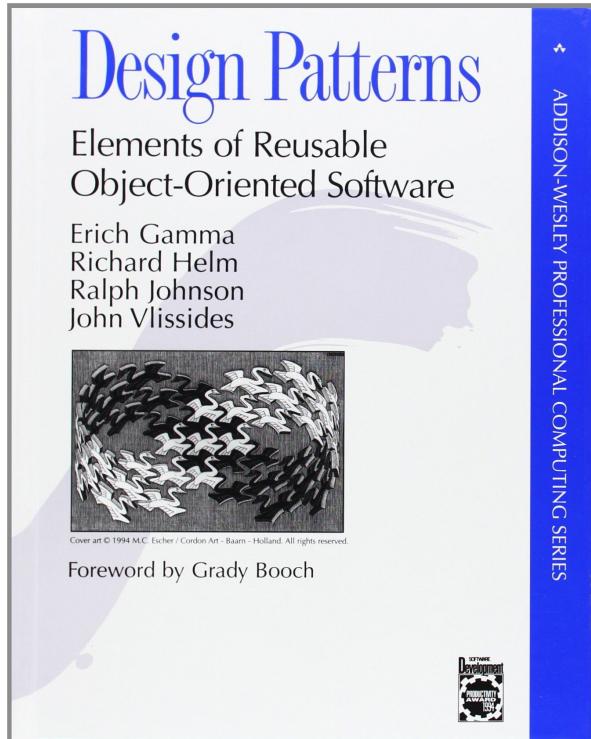
São modelados como **abstrações/templates**

Conhecer os principais padrões é **requisito básico** para ser um bom programador

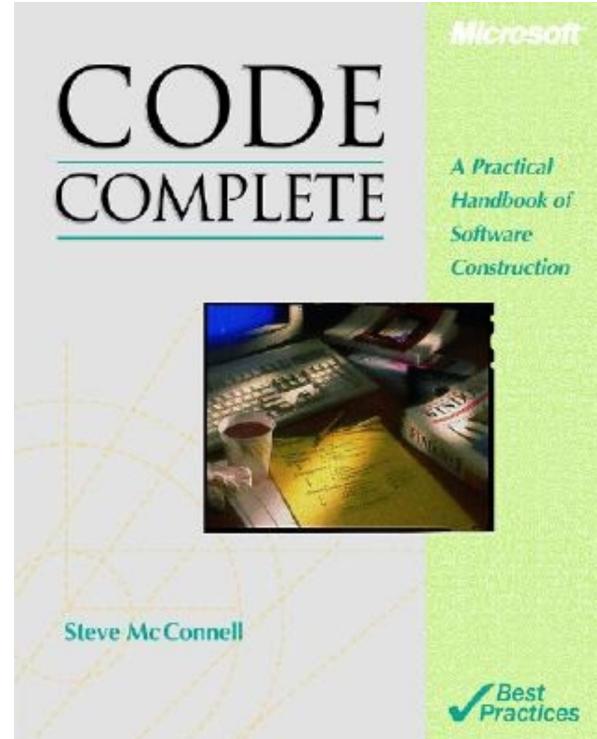
Não utilizar padrões leva ao que chamamos **Big Ball of Mud**

# Criadores dos Padrões de Projeto

---



1994



1993

# Padrões de Projeto - A Bíblia

---

Padrões de Projeto — Soluções Reutilizáveis de Software  
Orientado a Objetos

- Gang of Four (GoF) - Eric, John, Ralph, Richard
- Contém 23 padrões que resolvem problemas de projeto orientado a objetos
- Passou a ser conhecido como **“the GoF book”**

# Gang of Four (GoF)

---

Autores do livro Design Patterns



(L-R) Ralph, Erich, Richard and John

Ralph Johnson  
Erich Gamma  
Richard Helm  
John Vlissides †

# Por que devo aprender Padrões de Projeto?

---

1. Soluções já **consolidadas** para problemas recorrentes
2. **Linguagem comum** entre desenvolvedores para buscar uma solução

# Classes de Padrões

---

Os padrões que vamos estudar são divididos em três classes:

**1 - Padrões criacionais:** fornecem mecanismos de **criação** de objetos de forma a facilitar a flexibilidade e reutilização do código

**2 - Padrões estruturais:** explicam como montar classes para facilmente **estender** suas funcionalidades

**3 - Padrões comportamentais:** cuidam da **comunicação** eficiente e da definição de responsabilidades entre objetos

# Como Entender Padrões de Projeto?

---

Em cada padrão estudado, vamos tentar mapear:

**1 - o propósito:** descrição breve do objetivo do padrão e da razão para usá-lo

**2 - a motivação:** cenário descrevendo um problema e como o padrão pode ser aplicado

**3 - as estruturas:** diagrama de classes

**4 - exemplos práticos**

# Outros Padrões e Princípios

---

Novos padrões de projeto surgiram com o nascimento de novos paradigmas e de novas necessidades

## Exemplos:

- Padrões de interface gráfica
- Padrões arquiteturais
- Padrões para processamento paralelo
- Padrões para visualização de informação
- GRASP

**Software design patterns**

<b>Creatational</b>	Abstract factory · Builder · Dependency injection · Factory method · Lazy initialization · Multiton · Object pool · Prototype · RAII · Singleton
<b>Structural</b>	Adapter · Bridge · Composite · Decorator · Delegation · Facade · Flyweight · Front controller · Marker interface · Module · Proxy · Twin
<b>Behavioral</b>	Blackboard · Chain of responsibility · Command · Interpreter · Iterator · Mediator · Memento · Null object · Observer · Servant · Specification · State · Strategy · Template method · Visitor
<b>Functional</b>	Monoid · Functor · Applicative · Monad · Comonad · Free monad · HOF · Currying · Function composition · Closure · Generator
<b>Concurrency</b>	Active object · Actor · Balking · Barrier · Binding properties · Coroutine · Compute kernel · Double-checked locking · Event-based asynchronous · Fiber · Futex · Futures and promises · Guarded suspension · Immutable object · Join · Lock · Messaging · Monitor · Nuclear · Proactor · Reactor · Read write lock · Scheduler · STM · Thread pool · Thread-local storage
<b>Architectural</b>	ADR · Active record · Broker · Client-server · CBD · DAO · DTO · DDD · ECB · ECS · EDA · Front controller · Identity map · Interceptor · Implicit invocation · Inversion of control · Model 2 · MOM · Microservices · MVA · MVC · MVP · MVVM · Monolithic · Multitier · Naked objects · ORB · P2P · Publish-subscribe · PAC · REST · SOA · Service locator · Specification
<b>Cloud Distributed</b>	Ambassador · Anti-Corruption Layer · Bulkhead · Cache-Aside · Circuit Breaker · CQRS · Compensating Transaction · Competing Consumers · Compute Resource Consolidation · Event Sourcing · External Configuration Store · Federated Identity · Gatekeeper · Index Table · Leader Election · MapReduce · Materialized View · Pipes · Filters · Priority Queue · Publisher-Subscriber · Queue-Based Load Leveling · Retry · Scheduler Agent Supervisor · Sharding · Sidecar · Strangler · Throttling · Valet Key
<b>Other</b>	Business delegate · Composite entity · Intercepting filter · Lazy loading · Mangler · Mock object · Type tunnel · Method chaining
<b>Books</b>	<i>Design Patterns</i> · <i>Enterprise Integration Patterns</i> · <i>Code Complete</i> · <i>POSA</i>
<b>People</b>	Christopher Alexander · Erich Gamma · Ralph Johnson · John Vlissides · Grady Booch · Kent Beck · Ward Cunningham · Martin Fowler · Robert Martin · Jim Coplien · Douglas Schmidt · Linda Rising
<b>Communities</b>	The Hillside Group · The Portland Pattern Repository

[Authority control](#)

GND: 4546895-3 · LCCN: sh98003823

**Até a  
próxima!**

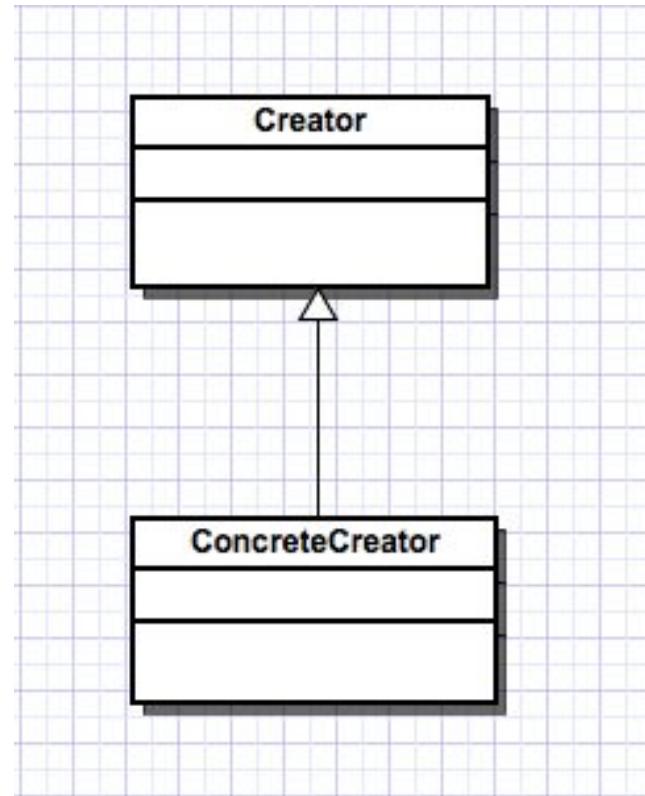
# Parte 2: Padrões Criacionais

Agradecimentos: [Refactoring Guru](#)

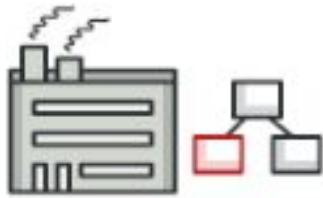
# Padrões Criacionais

---

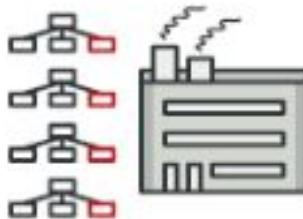
Define mecanismos de criação de objetos que permitem reusar e estender esse código facilmente



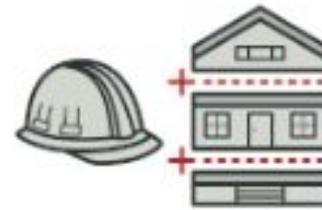
Fonte:[^wikipedia](#)



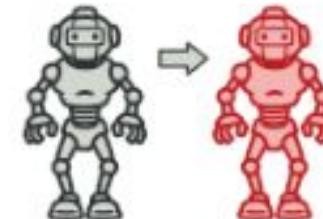
Factory Method



Abstract Factory



Builder



Prototype



Singleton

# Padrões Criacionais

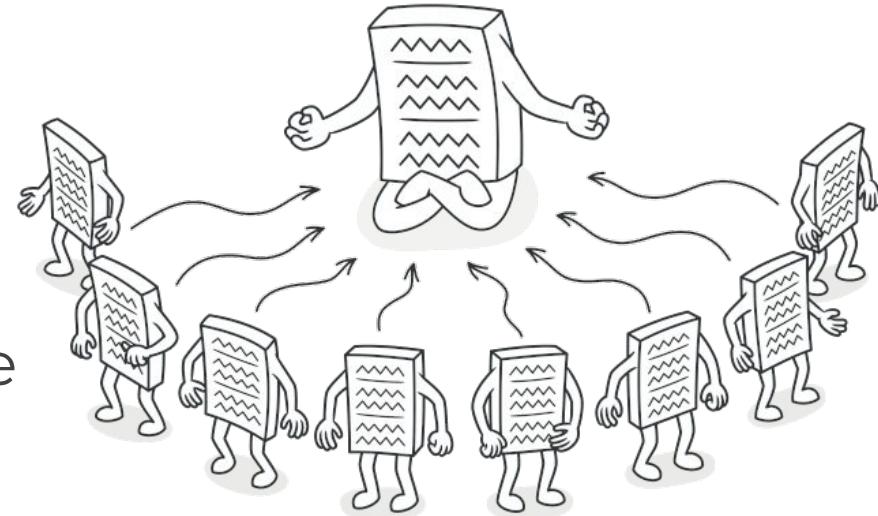
[Fonte](#)

# Singleton

---

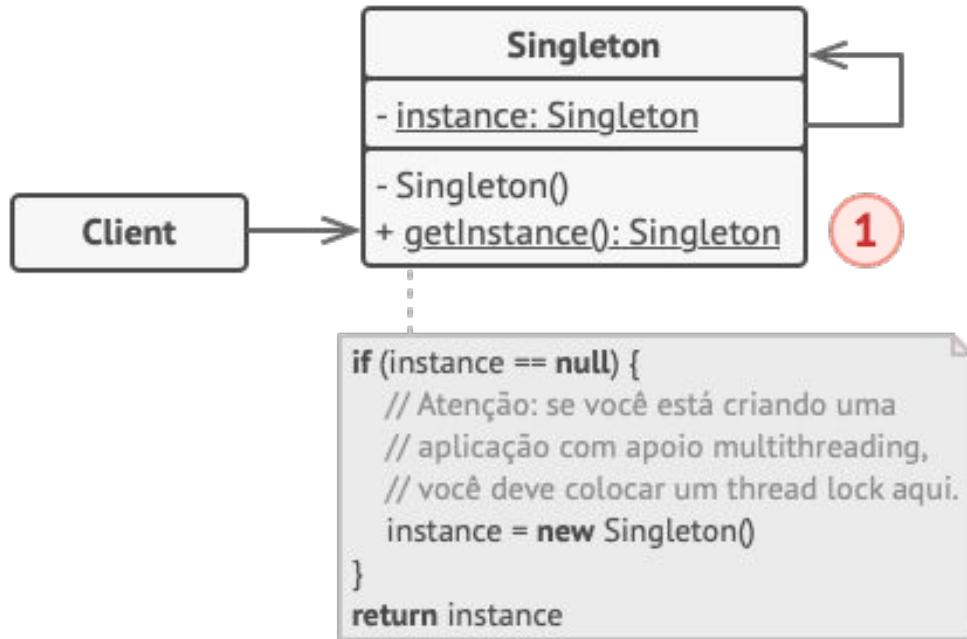
**Propósito** - garantir que uma classe tenha apenas uma instância e de acesso global

**Motivação** - acessos a um recurso compartilhado (Base de Dados/Arquivo) em diferentes partes do sistema



# Singleton

---



A classe Singleton declara o método estático `getInstance`, que retorna a **mesma instância** de sua própria classe.

Reparem que o construtor passa a ser um método **privado**

# Singleton

---

```
2 # exemplo de singleton
3 # a classe abaixo serve como classe base para outras
4 class Singleton(object):
5     __instance = None
6
7     # sobrescrita do magic method __new__ para
8     # ser nosso getInstance
9     def __new__(cls, *args):
10         if cls.__instance is None:
11             cls.__instance = object.__new__(cls, *args)
12         return cls.__instance
13
14 class DBDriver(Singleton):
15     def __init__(self):
16         pass
17
18     def connect(self, url, user):
19         pass
20
21     def select_db(self, db_name):
22         pass
```

# Singleton

---

```
2 # exemplo de singleton
3 # a classe abaixo serve como classe base para outras
4 class Singleton(object):
5     __instance = None
6
7     # sobrescrita do magic method __new__ para
8     # ser nosso getInstance
9     def __new__(cls, *args):
10         if cls.__instance is None:
11             cls.__instance = object.__new__(cls, *args)
12         return cls.__instance
13
14 class DBDriver(Singleton):
15     def __init__(self):
16         pass
17
18     def connect(self, url, user):
19         pass
20
21     def select_db(self, db_name):
22         pass
```

# Singleton

---

## Prós

- Garantia de única instância
- Ponto de acesso global
- Permite Lazy initialization

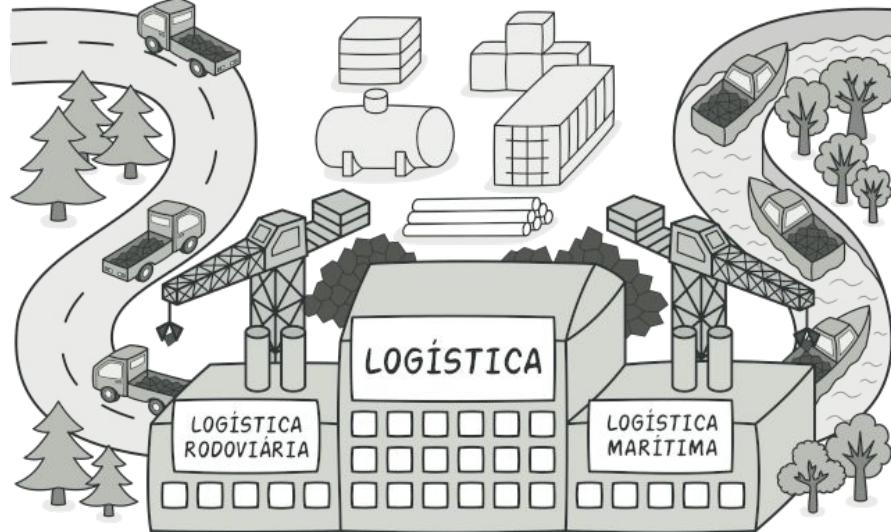
## Contras

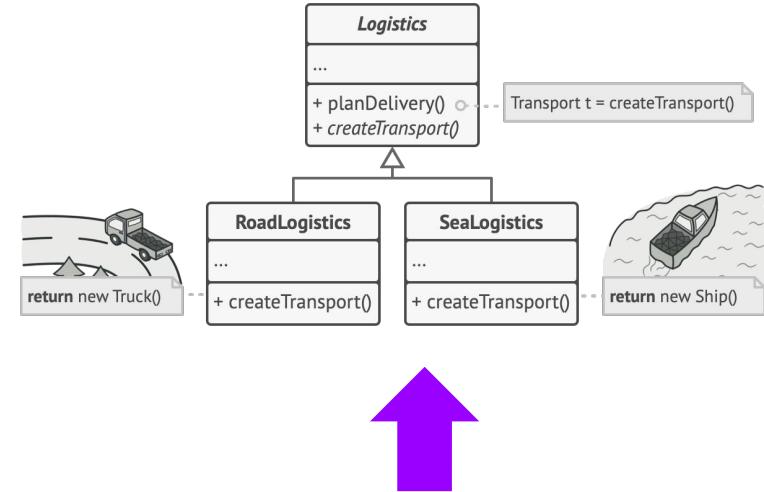
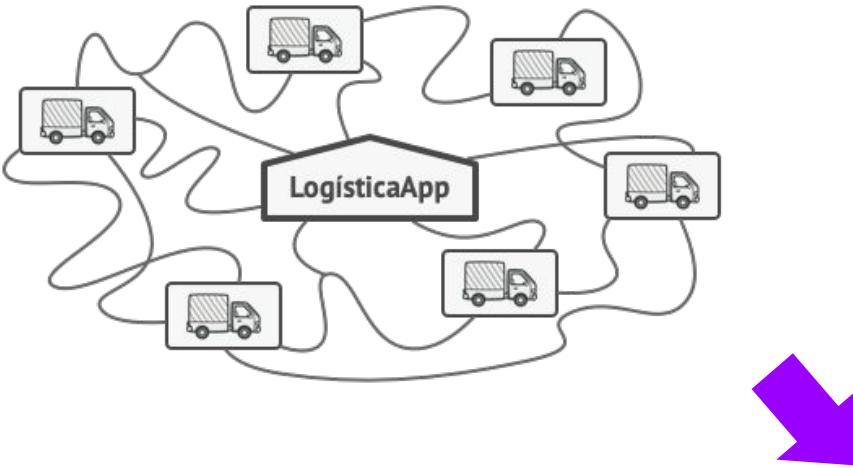
- Estado global normalmente não é desejado
- Alto acoplamento entre Singleton e clientes
- Dificultam teste unitário

# Factory

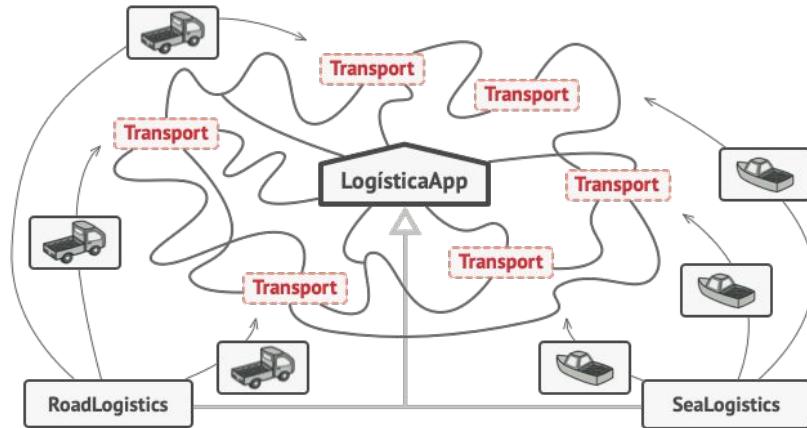
---

**Propósito** - Criar objetos na superclasse, mas alterar objetos nas subclasses





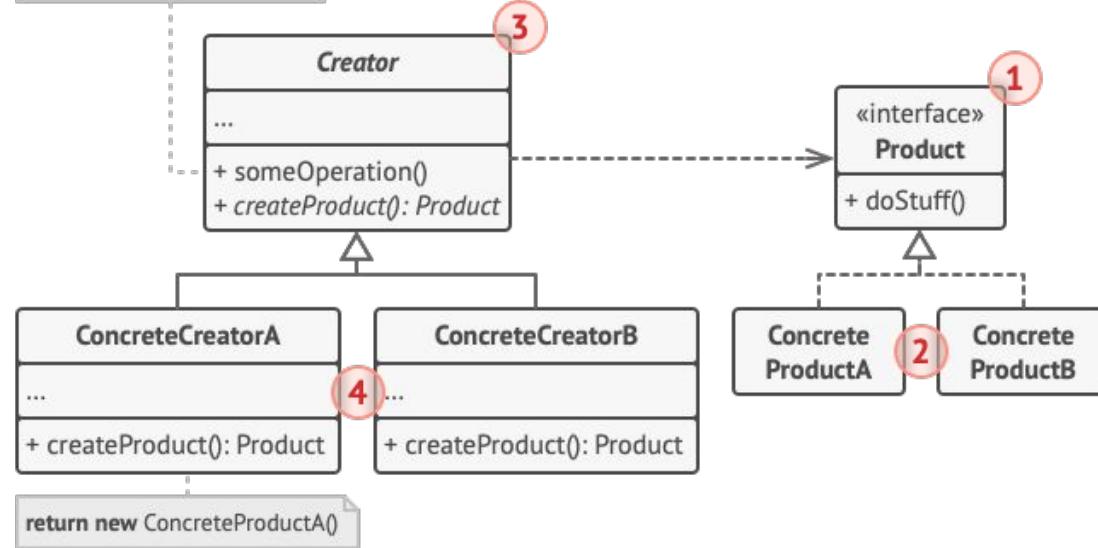
**Motivação** - Precisamos muitas vezes criar um novo componente que serve as mesmas funcionalidades de forma diferente



# Factory

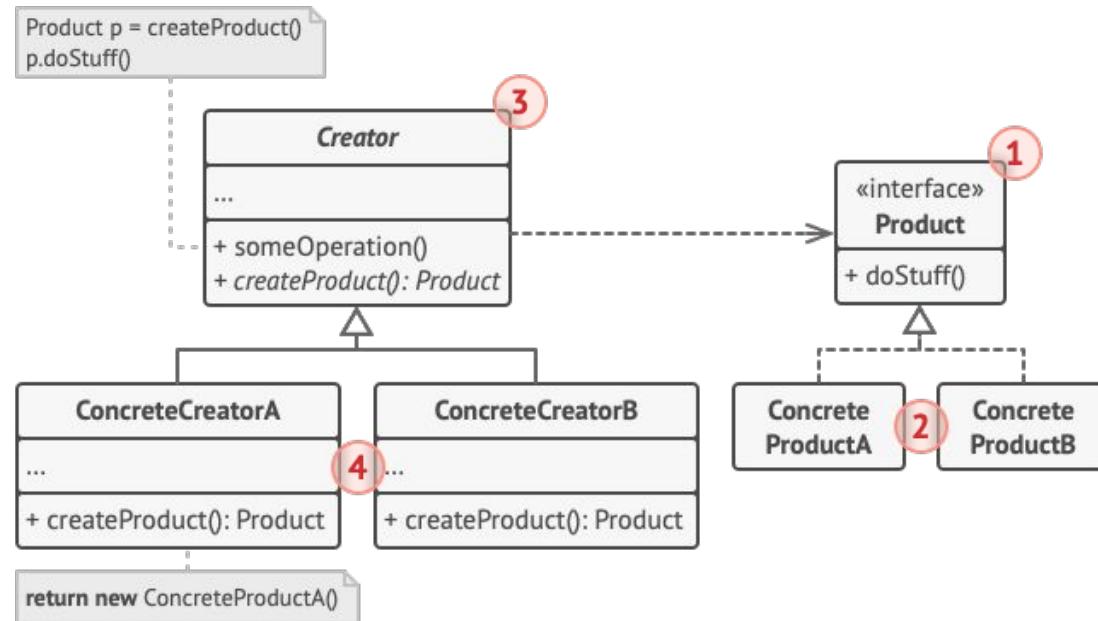
---

```
Product p = createProduct()  
p.doStuff()
```



# Factory

---



1 - Declaramos a interface **produto** que é comum a todos os objetos criados pelo Creator

2 - Produtos **concretos** implementam essa interface para cada tipo de situação

3 - A classe **Creator** possui o Factory Method (`createProduct`)

4 - Implementações **concretas** de Creator sobrescrevem o Factory Method para retornar o tipo adequado

```
# Factory MazeGame que criar jogos do tipo lab
class MazeGame(ABC):
    def __init__(self) -> None:
        self.rooms = []
        self._prepare_rooms()

    # Factory method make_room será especializada
    @abstractmethod
    def make_room(self):
        raise NotImplementedError("You should implement me!")


```

```
# versão abstrata da classe Room (sala)
class Room(ABC):
    def __init__(self) -> None:
        self.connected_rooms = []

    def connect(self, room) -> None:
        self.connected_rooms.append(room)


```

```
class MagicMazeGame(MazeGame):
    def make_room(self):
        return MagicRoom()

# classe derivada 2
class OrdinaryMazeGame(MazeGame):
    def make_room(self):
        return OrdinaryRoom()


```

```
# versão concreta de Room 1 - usada no exemplo
class MagicRoom(Room):
    def __str__(self):
        return "Magic room"

# versão concreta de Room 2 - usada no exemplo
class OrdinaryRoom(Room):
    def __str__(self):
        return "Ordinary room"


```

# Factory

---

## Prós

- Baixo acoplamento entre criador e produtos concretos
- **SRP**: criação do produto separada da lógica de negócio
- **OCP**: permite estender as funcionalidades de produtos sem quebrar aplicações clientes existentes

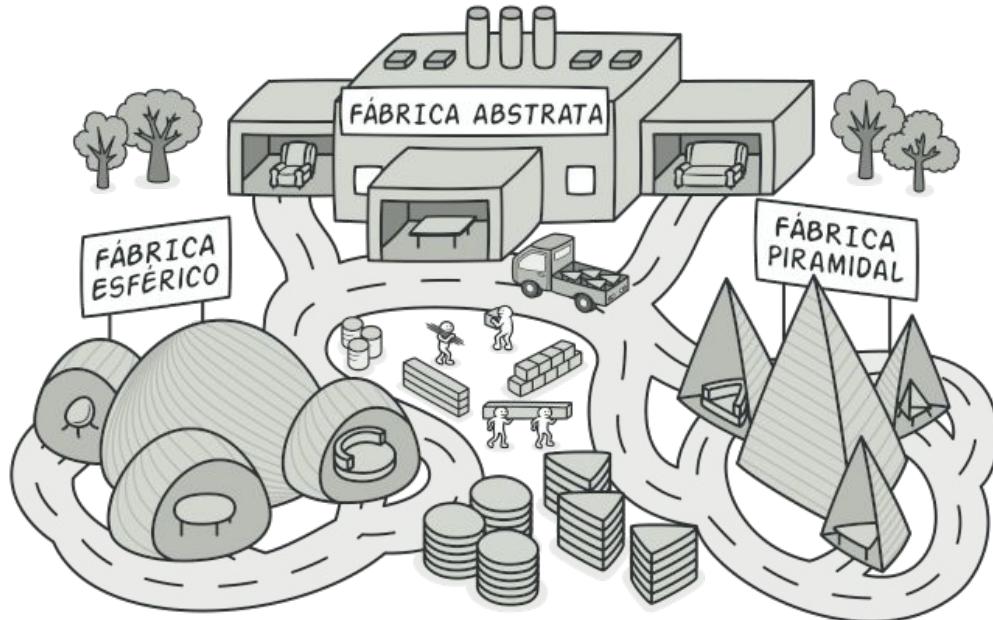
## Contras

- Aumenta a complexidade, pois exige criação de novas classes e interfaces

# Abstract Factory

---

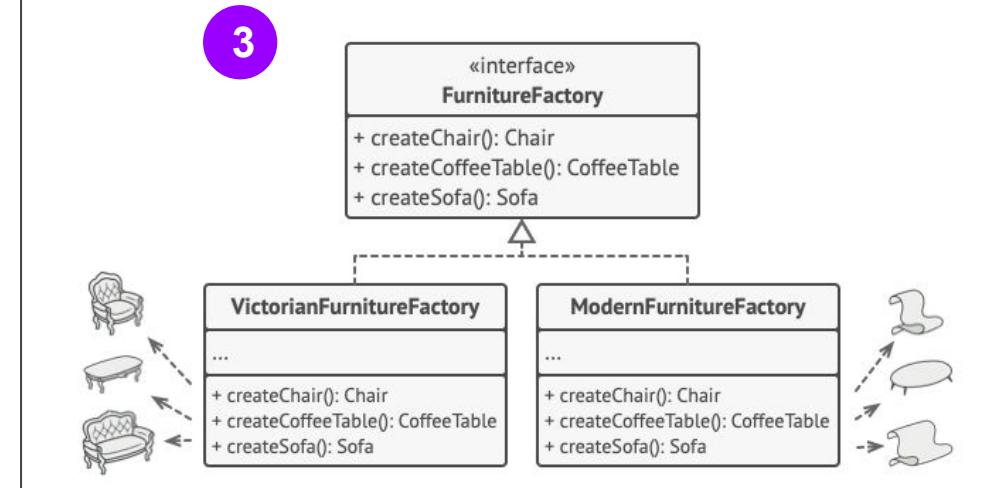
**Propósito** - Produzir **famílias** de objetos relacionados sem ter que especificar classes concretas



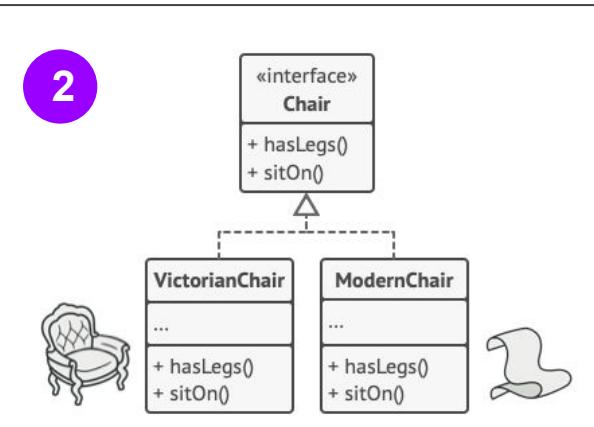
1



3

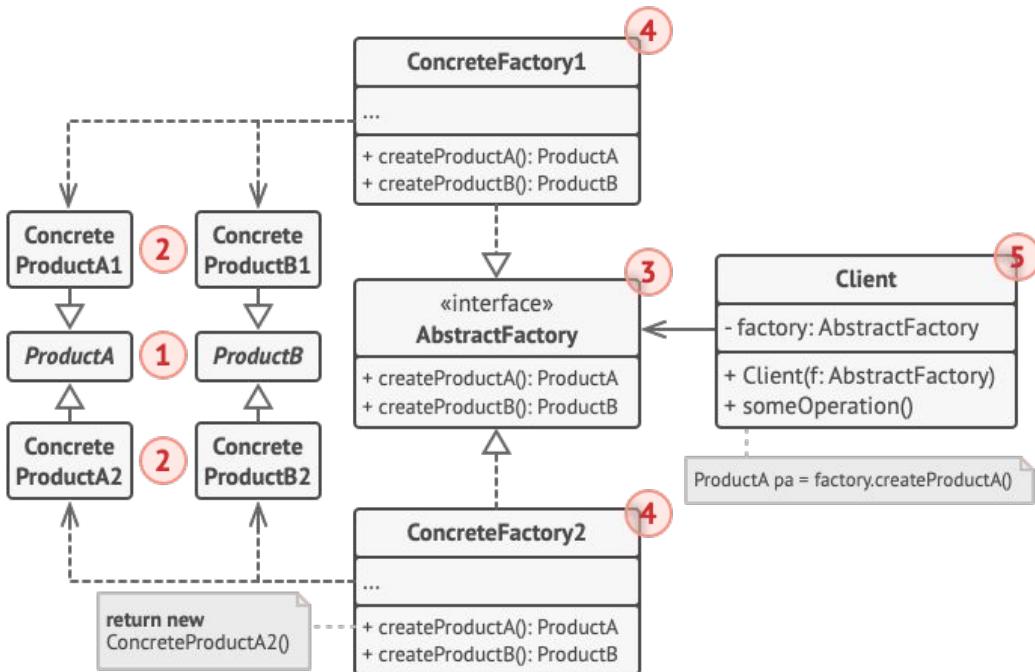


2



**Motivação** - Você possui diferentes fábricas (1), cada uma fabrica uma família de produtos comuns (2) e você precisa de uma interface com todos os fabricantes (3)

# Abstract Factory



**1 - Produtos Abstratos** declaram interfaces para um conjunto de produtos distintos mas que fazem parte de uma mesma família

**2 - Produtos Concretos** devem ser implementados em todas as variantes dadas (Vitoriano, Moderno, ...)

**3 - A interface Fábrica Abstrata** declara um conjunto de métodos para criação de cada um dos produtos abstratos

**4 - Fábricas Concretas** correspondem a uma variante específica de produtos e criam apenas aquelas variantes de produto

**5 - O Cliente** pode trabalhar com qualquer variante de produto/fábrica concreto, desde que ele se comunique com seus objetos via **interfaces abstratas**

```
4 from abc import ABC, abstractmethod
5 from sys import platform
6
7 # Abstract interfaces da família de widgets (produtos)
8 class Window(ABC):
9     @abstractmethod
10    def paint(self):
11        pass
12
13 class Button(ABC):
14     @abstractmethod
15    def paint(self):
16        pass
```

```
# implementações concreta para cada OS
class LinuxButton(Button):
    def paint(self):
        return "Render a button in a Linux style"

class LinuxWindow(Window):
    def paint(self):
        return "Render a window in a Linux style"

class WindowsButton(Button):
    def paint(self):
        return "Render a button in a Windows style"

class WindowsWindow(Window):
    def paint(self):
        return "Render a window in a Windows style"
```

```
45 # Abstract factory para GUIs
46 class GUIFactory(ABC):
47     @abstractmethod
48     def create_button(self):
49         pass
50     def create_window(self):
51         pass
52
53
54 # implementações concretas das GUIs
55 class LinuxFactory(GUIFactory):
56     def create_button(self):
57         return LinuxButton()
58
59     def create_window(self):
60         return LinuxWindow()
61
```

# Abstract Factory

---

## Prós

- Intercomunicação garantida graças às interfaces dos produtos
- Baixo acoplamento entre criador e produtos concretos
- **SRP**: criação do produto separada da lógica de negócio
- **OCP**: permite estender as funcionalidades de produtos sem quebrar aplicações clientes existentes

## Contras

- Aumenta a complexidade, pois exige criação de novas classes e interfaces

# Links

---

[Github com exemplos](#)

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

**Até a  
próxima!**

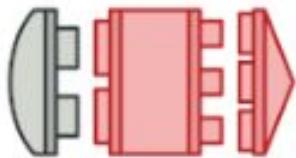
# Parte 3: Padrões Estruturais

Agradecimentos: [Refactoring Guru](#)

# Padrões Estruturais

---

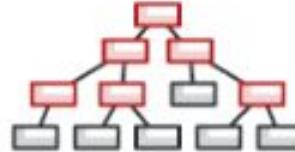
Facilitam a comunicação entre entidades através de estruturas bem definidas que mantêm o código flexível e eficiente



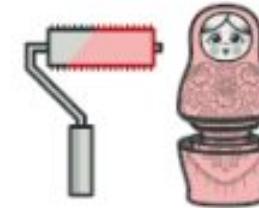
Adapter



Bridge



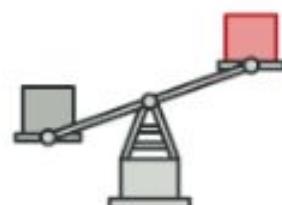
Composite



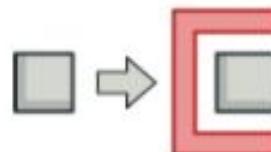
Decorator



Facade



Flyweight



Proxy

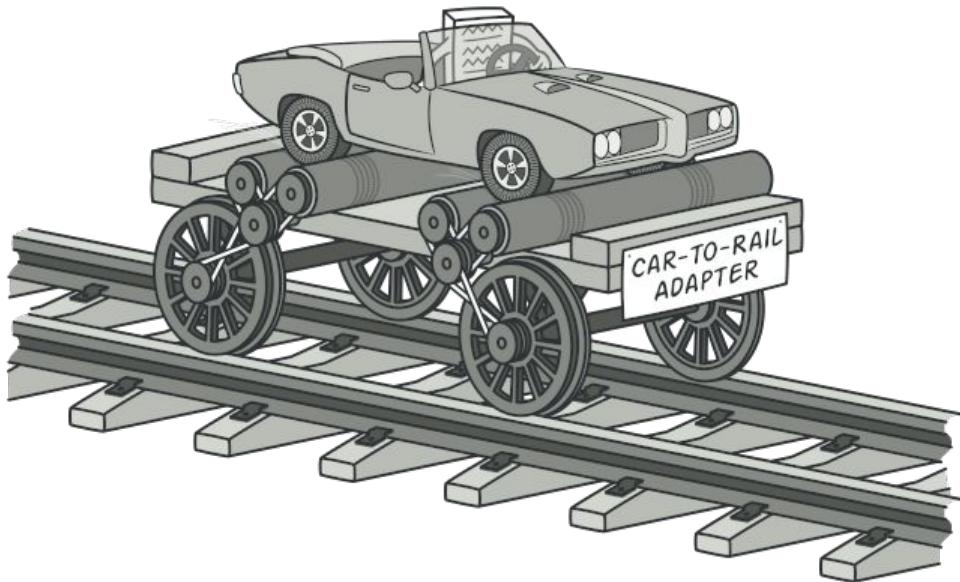
# Padrões Estruturais

[Fonte](#)

# Adapter

---

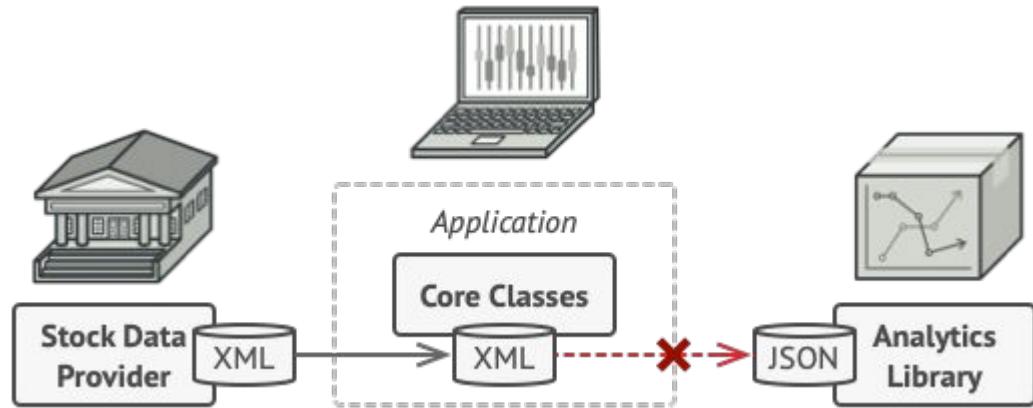
**Propósito** - adaptar interfaces incompatíveis para que duas entidades se comuniquem



# Adapter

---

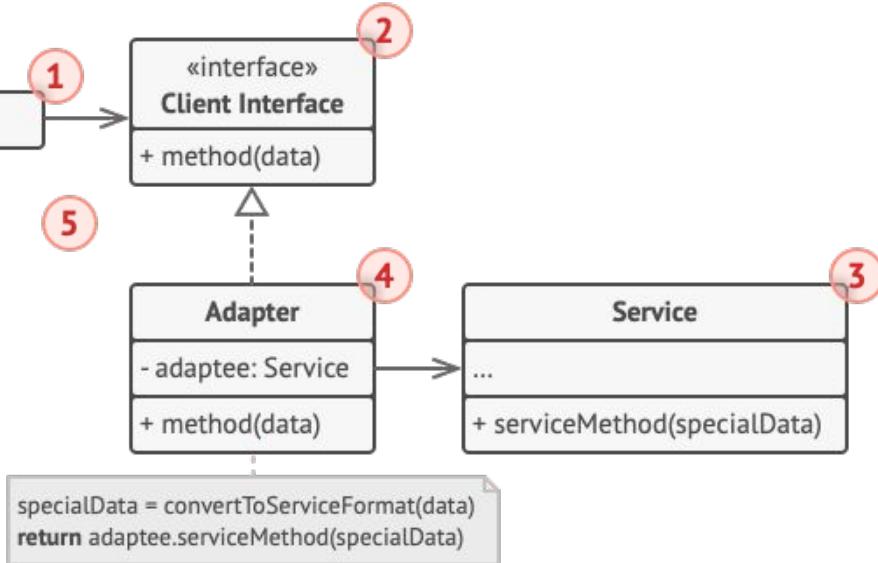
**Motivação** - muitas vezes temos aplicações que trabalham com dados em formatos distintos para atingir um propósito maior



Wrapper methods são exemplos clássicos de adaptadores

# Adapter

---



1 - A classe cliente contém a lógica de negócios do programa

2 - Client Interface descreve o protocolo que outras classes devem seguir

3 - Service é uma classe utilitária (API) com interface incompatível

4 - Adapter é a classe responsável por implementar a Client Interface e envelopar os métodos do serviço

```
def client_code(target: Target)
    # código cliente suporta
    print(target.request(),
```

```
class ITarget:
    #essa interface contém o método
    def request(self) -> str:
        pass
```

```
class Adapter(ITarget):
    __adaptee = Adaptee()
    # Adapter torna a interface do Adaptado (Adaptee) compatível
    def request(self) -> str:
        (id, name) = self.__adaptee.specific_request()
        return f'{name}\n'
```

```
class Adaptee:
    # contém métodos úteis, mas incompatíveis
    def specific_request(self) -> (int, str):
        return (1, "Mateus Grellert")
```

# Adapter

---

## Prós

- **SRP**: regras de negócios separadas da lógica de adaptação
- **OCP**: novos adaptadores podem ser criados sem quebrar o código existente

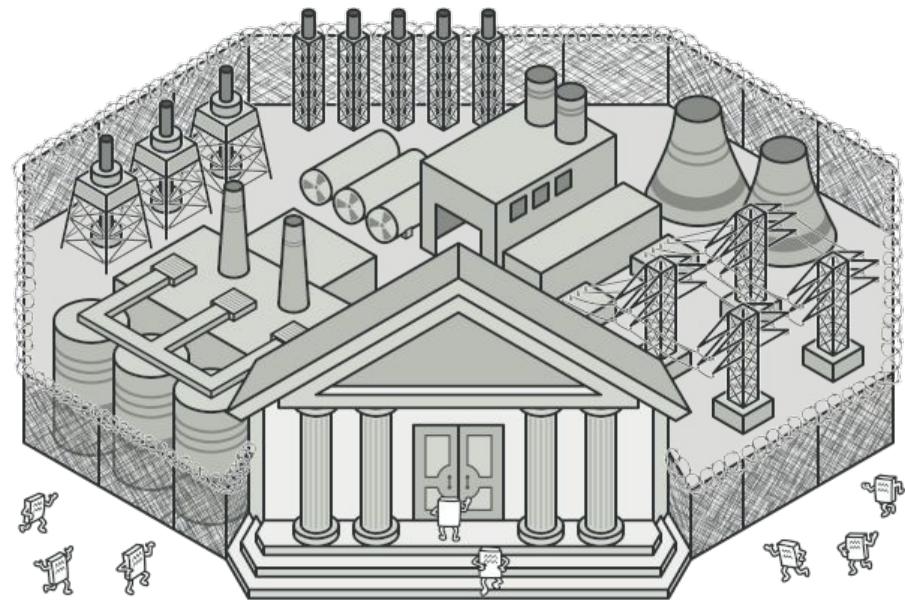
## Contras

- Aumenta a complexidade, pois exige criação de novas classes e interfaces

# Facade

---

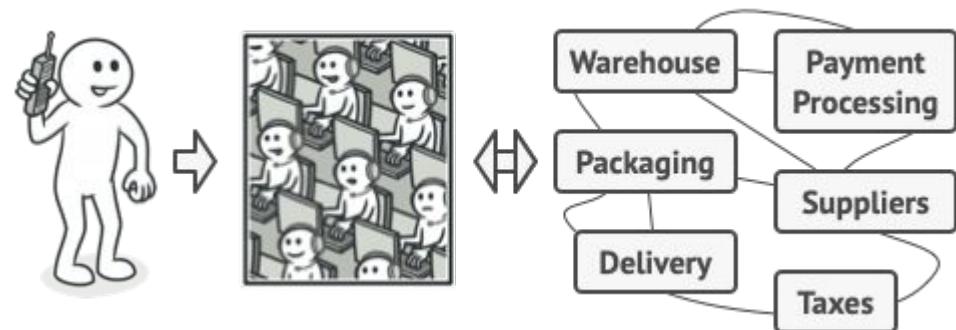
**Propósito** - prover uma interface simplificada para uma biblioteca, framework, ou qualquer conjunto mais complexo de classes



# Facade

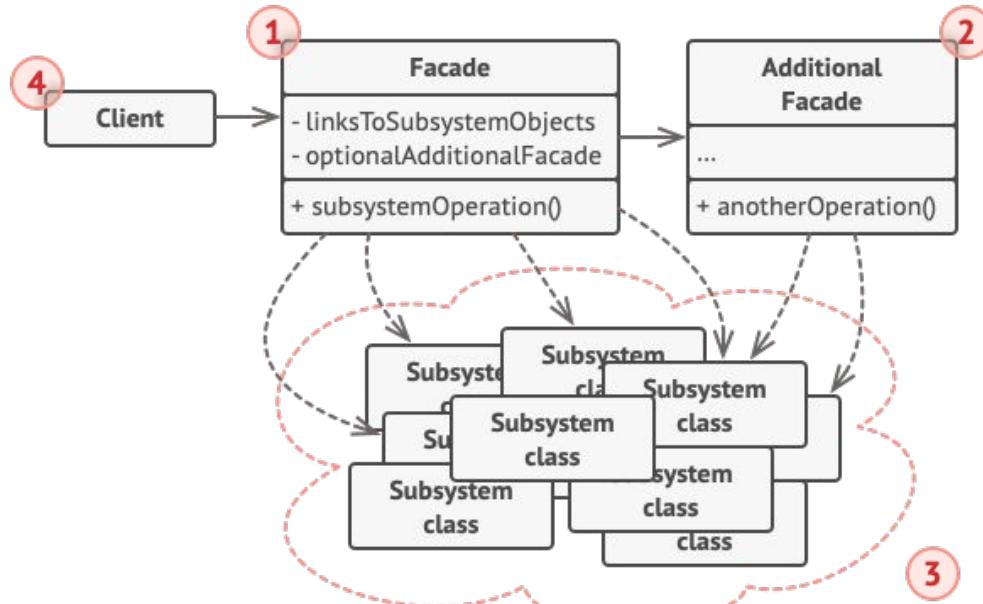
---

**Motivação** - Quando encomendamos um produto, os operadores de venda são uma fachada para uma multitudade de operações que acontecem para executar o pedido



# Facade

---



- 1 - A Facade provê acesso fácil a uma parte do subsistema.
- 2 - Uma Facade opcional pode ser criada para evitar poluir uma única Facade
- 3 - O subsistema é uma coleção complexa de objetos que exige muito conhecimento para fazer algo funcionar
- 4 - O cliente usa a Facade para usar as funcionalidades sem precisar trabalhar com os objetos do subsistema diretamente

# Nós criamos uma classe fachada para esconder a complexidade do framework atrás de uma interface simples.  
# É uma troca entre funcionalidade e simplicidade.

```
class VideoConverter:  
    def convert(filename, format):  
        file = VideoFile(filename)  
        sourceCodec = CodecFactory().extract(file)  
        if (format == "mp4"):  
            destinationCodec = MPEG4CompressionCodec()  
        else:  
            destinationCodec = VP9CompressionCodec()  
        buffer = BitrateReader.read(filename, sourceCodec)  
        result = BitrateReader.convert(buffer, destinationCodec)  
        result = AudioMixer().fix(result)  
        return result
```

# As classes da aplicação não dependem de um bilhão de classes # fornecidas por um framework complexo.  
# Também, se você decidir trocar de frameworks, você só precisa reescrever a classe Facade.

```
class Application:  
    def main():  
        conversor = VideoConverter()  
        mp4 = conversor.convert("funny-cats-video.mp4", "mp4")  
        mp4.save()
```

# Facade

---

## Prós

- Isola a implementação de subsistemas com implementações complexas

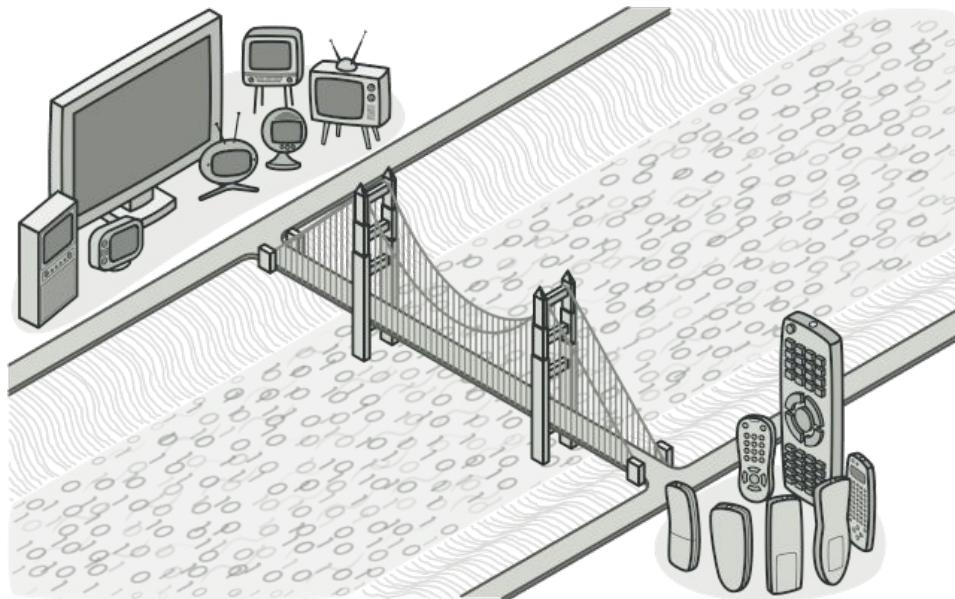
## Contras

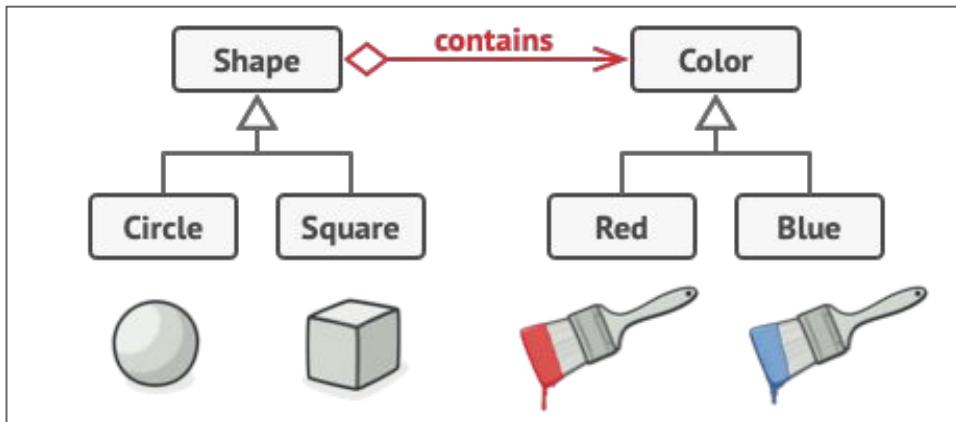
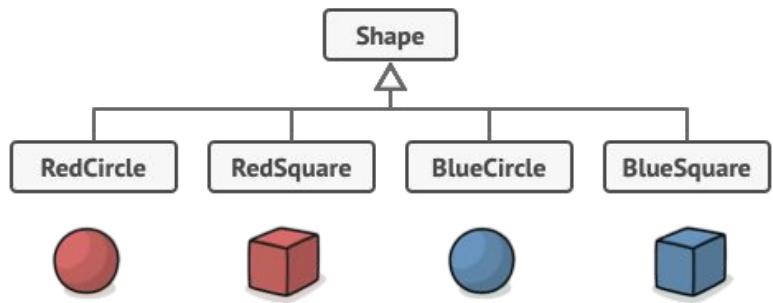
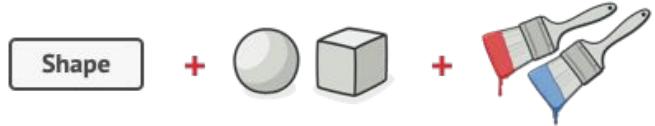
- Uma facade por acabar se tornando um god object com alto acoplamento a todas as classes da aplicação

# Bridge

---

**Propósito** - permite comunicar duas hierarquias de classes relacionadas (que podem se desenvolver de forma independente)

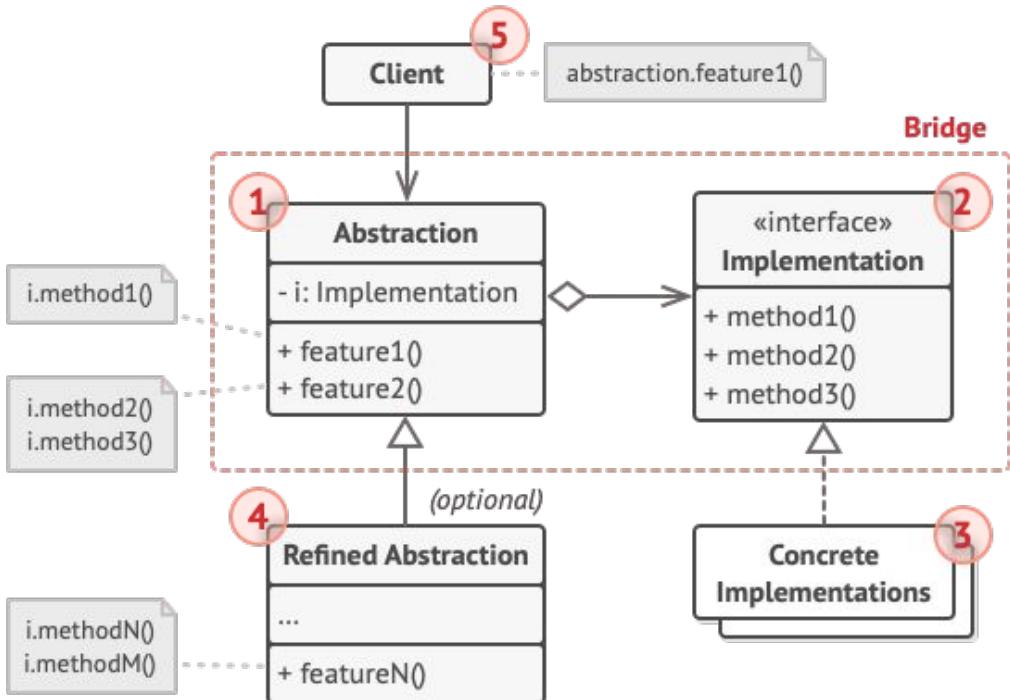




**Motivação** - Digamos que temos duas classes: forma e cor. Queremos adicionar cor a cada forma, mas não queremos ter que criar um objeto especializado para cada combinação

**Composition Over Inheritance**

# Bridge



**1 - Abstraction** contém somente lógica de alto nível e deixa o baixo nível para as implementações

**2 - A interface Implementation define todas as funções que devem ser implementadas nas versões concretas**

**3 - Versões concretas implementam código específico de cada caso**

**4 - Refined Abstractions** podem introduzir variantes de controle (opcional)

**5 - O Cliente só trabalha com a abstração, mas ele precisa ligar a abstração com uma das implementações**

```
for shape in shapes:  
    shape.resize_by_percentage(2.5)  
    print(shape.draw())
```

```
class Shape(ABC):  
    def __init__(self, api: DrawingAPI):  
        self.drawing_api = api  
  
    @abstractmethod  
    def draw(self):  
        raise NotImplementedError(NOT_IMPLEMENTED)  
  
    @abstractmethod  
    def resize_by_percentage(self, percent):  
        raise NotImplementedError(NOT_IMPLEMENTED)
```

# Interface define métodos que todas as APIs devem implementar

```
class DrawingAPI(ABC):  
    @abstractmethod  
    def draw_circle(self, x, y, radius):  
        raise NotImplementedError(NOT_IMPLEMENTED)
```

```
class CircleShape(Shape):  
    class CircleShape(Shape):  
        def __init__(self, x, y, radius, drawing_api):  
            self.x = x  
            self.y = y  
            self.radius = radius  
            super(CircleShape, self).__init__(drawing_api)  
  
        def draw(self):  
            return self.drawing_api.draw_circle(self.x, self.y, self.radius)
```

```
class DrawingAPI1(DrawingAPI):  
    def draw_circle(self, x, y, radius):  
        return "API1.circle at {0}:{1},{2},{3}"
```

# Bridge

---

## Prós

- Podemos criar classes independentes
- O código cliente trabalha com abstrações de alto nível
- **SRP:** lógica de alto nível na abstração e detalhes na implementação concreta
- **OCP:** Novas abstrações e implementações podem ser adicionadas de forma independente

## Contras

- Podemos complicar o código, especialmente aplicando bridge em uma classe com alta coesão

**Até a  
próxima!**

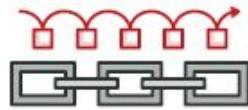
# Parte 4: Padrões Comportamentais

Agradecimentos: [Refactoring Guru](#)

# Padrões Comportamentais

---

Padrões que identificam formas de interação recorrentes entre objetos, aumentando a flexibilidade do sistema ao serem aplicados



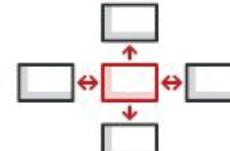
Chain of Responsibility



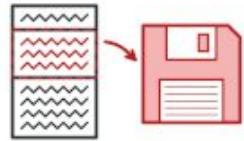
Command



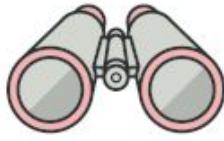
Iterator



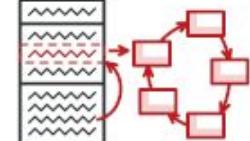
Mediator



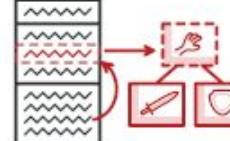
Memento



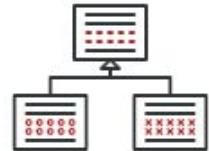
Observer



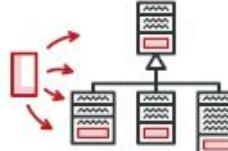
State



Strategy



Template Method



Visitor

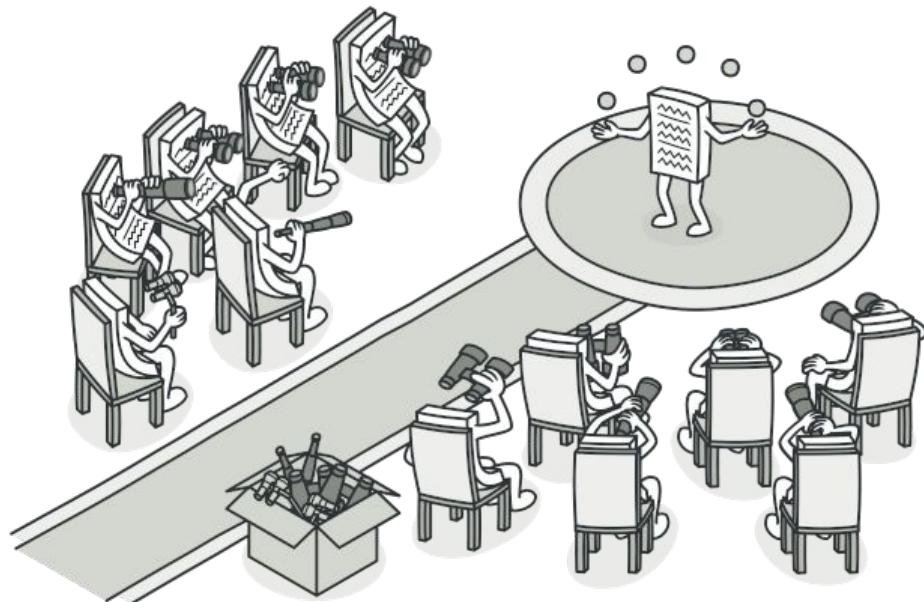
# Padrões Comportamentais

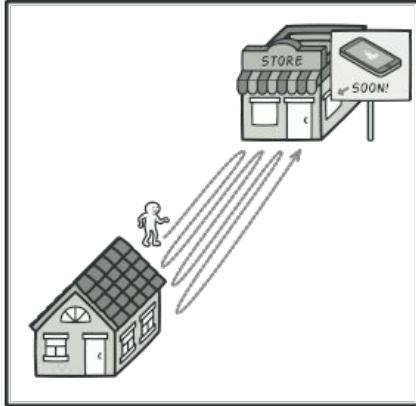
[Fonte](#)

# Observer

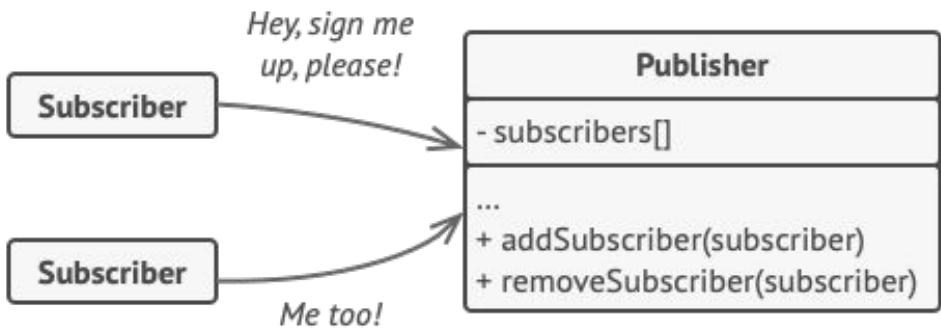
---

**Propósito** - permite definir um mecanismo de subscrição para notificar múltiplos objetos sobre qualquer evento que acontece no objeto sendo observado



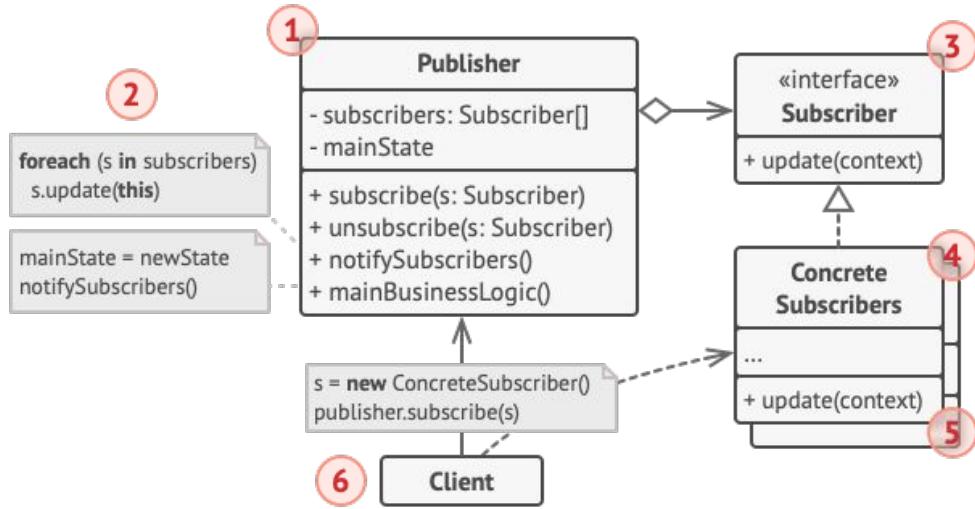


**Motivação** - queremos notificar vários objetos, mas não todos. Para isso, precisamos de um sistema de inscrição (subscribers)



# Observer

---



- 1 - **Publisher** manda eventos para outros objetos. Pubs contêm uma lógica de inscrição/desinscrição
- 2 - Quando queremos notificar algo, **percorremos** todos os objetos inscritos,
- 3 - chamando o método **update** definido na interface **Subscriber**
- 4 - Subscribers concretos especializam update com a lógica adequada
- 5 - Por vezes, Subs precisam de mais dados para tratar updates, daí o parâmetro **context** enviado pela classe Pub, que pode ser o próprio objeto

```
class Publisher:  
    def __init__(self):  
        self._observers = []  
  
    def register_observer(self, observer):  
        self._observers.append(observer)  
  
    def notify_observers(self, *args, **kwargs):  
        for observer in self._observers:  
            observer.notify(self, *args, **kwargs)
```

```
class Subscriber(ABC):  
    def __init__(self, pub: Publisher):  
        pub.register_observer(self)  
  
    @abstractmethod  
    def notify(self):  
        pass
```

```
pub = Publisher()  
sub1= Subscriber1(pub)  
sub2 = Subscriber2(pub)  
  
pub.notify_observers('test')
```

```
class Subscriber1(Subscriber):  
    def notify(self, observable, *args):  
        print('Recebido', args, 'de', observable)  
        print('Atualizando BD')  
  
class Subscriber2(Subscriber):  
    def notify(self, observable, *args):  
        print('Recebido', args, 'de', observable)  
        print('Notificando usuários do canal')
```

# Observer

---

## Prós

- OCP: podemos introduzir novas classes SUB sem alterar o código da classe PUB
- Podemos criar relações entre objetos em tempo de execução

## Contras

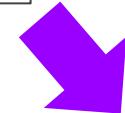
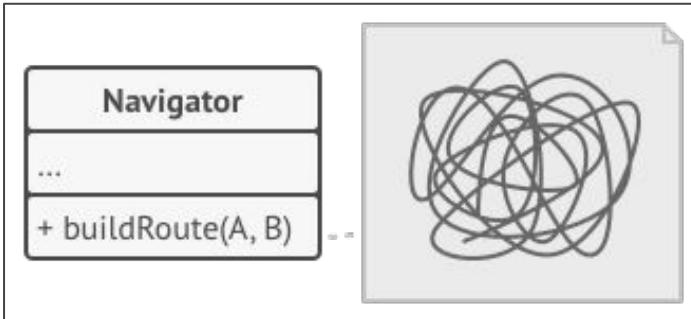
- Classes SUB são notificadas em ordem aleatória

# Strategy

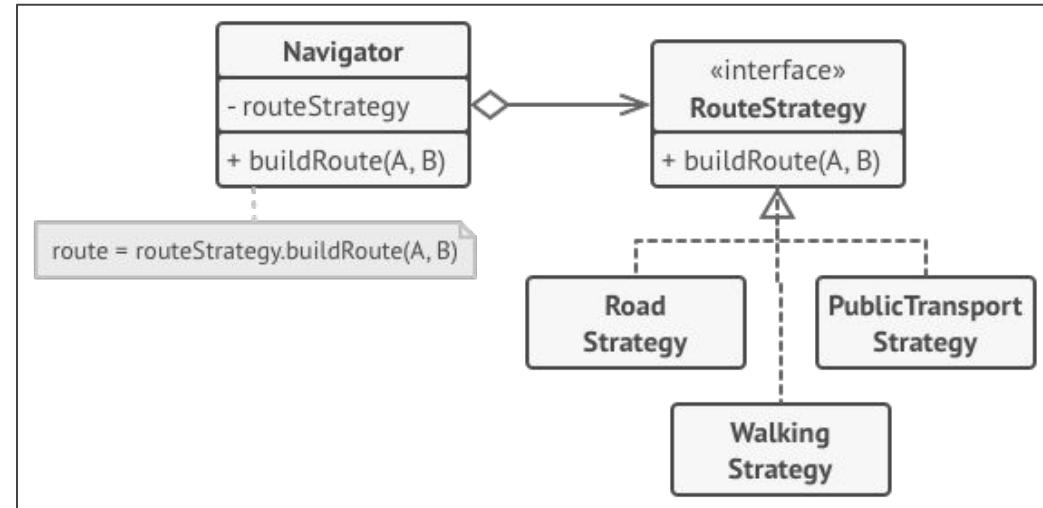
---

**Propósito** - permite definir uma família de algoritmos, colocá-los em uma classe separada e tornar seus objetos intercambiáveis



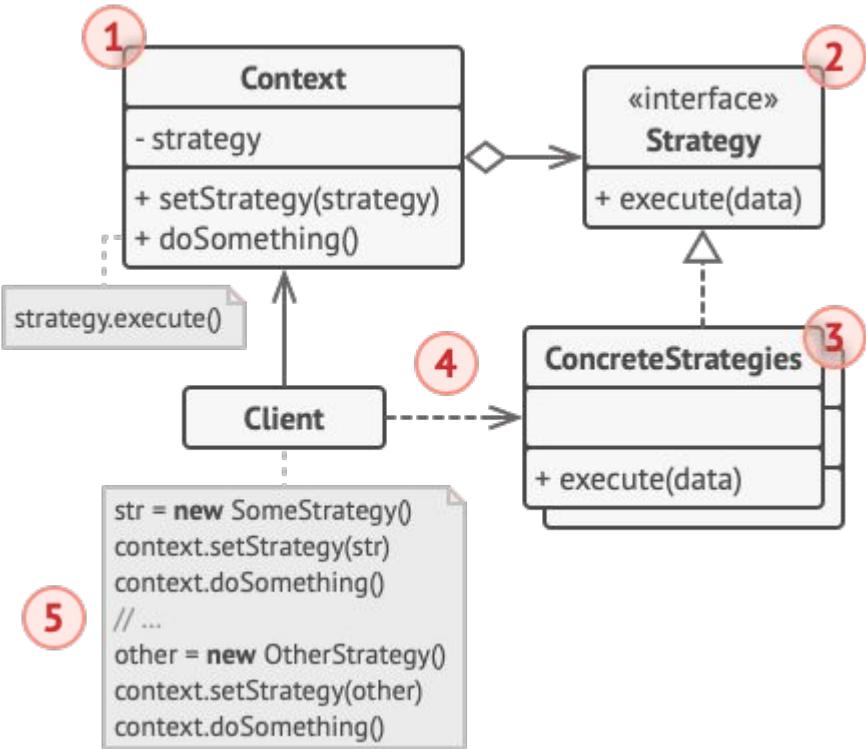


**Motivação** - Muitas vezes o mesmo conjunto de métodos é feito de forma diferente entre classes semelhantes



# Strategy

---



1 - O contexto mantém uma referência a uma das estratégias concretas e comunica-se através da interface

2 - A interface Strategy é comum a todas as estratégias concretas

3 - Cada versão concreta complementa uma versão de diferentes variações de um algoritmo utilizado por Context

4 - O contexto chama uma o método de execução em um objeto referenciado abstraindo como isso será feito

5 - O cliente instancia uma estratégia específica e passa ela para o contexto em tempo de execução

```
# define a interface de interesse a clientes
class Context():
    def __init__(self, strategy: Strategy) -> None:
        self._strategy = strategy

    @property
    def strategy(self) -> Strategy:
        return self._strategy

    @strategy.setter
    def strategy(self, strategy: Strategy) -> None:
        # Usualmente podemos mudar a estratégia em tmpo de execução.
        self._strategy = strategy

    def do_some_business_logic(self) -> None:
        print("Contexto: chamando algum tipo de sort (não importa qual)")
        result = self._strategy.do_algorithm(["a", "b", "c", "d", "e"])
        print(",".join(result))
```

```
# declara operações comuns a todas as versões suportadas dos algoritmos
class Strategy(ABC):
    @abstractmethod
    def do_algorithm(self, data: List):
        pass

# Estratégias concretas que seguem a interface
class ConcreteStrategyA(Strategy):
    def do_algorithm(self, data: List) -> List:
        return sorted(data)

class ConcreteStrategyB(Strategy):
    def do_algorithm(self, data: List) -> List:
        return reversed(sorted(data))
```

# Strategy

---

## Prós

- Podemos mudar o algoritmo executado em um objeto em tempo de execução
- Podemos isolar os detalhes de implementação do cliente
- Facilita Composition over Inheritance
- **OCP:** podemos adicionar novas estratégias sem ter que mudar o contexto

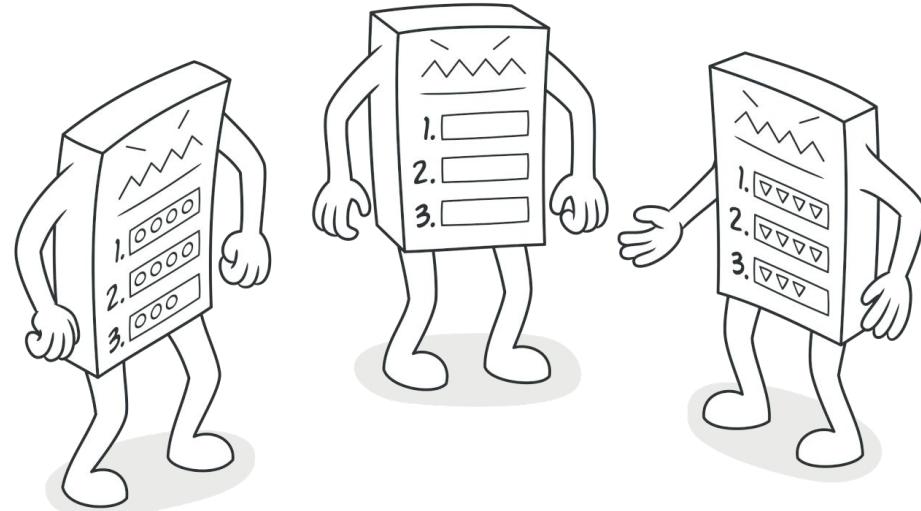
## Contras

- Clientes devem estar cientes das diferenças entre as estratégias para saber qual deve ser aplicada

# Template

---

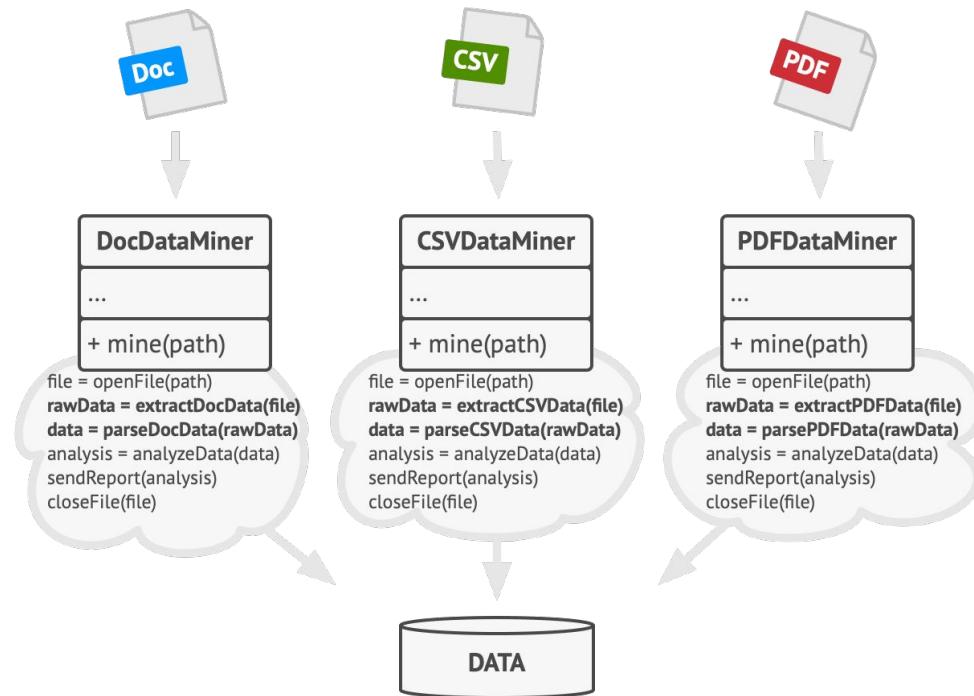
**Propósito** - definir esqueletos de implementações na superclasse, permitindo que as subclasses sobrescrevam passos específicos sem mudar a estrutura



# Template

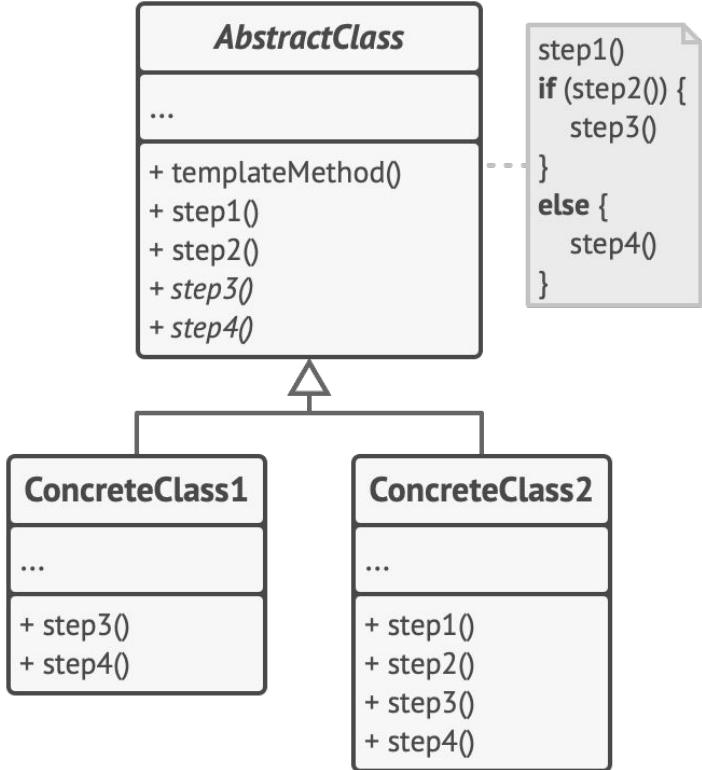
---

**Motivação** - muitas vezes  
temos rotinas que apresentam  
muita similaridade estrutural,  
mudando somente alguns  
passos



# Template

---



**1 -** A classe abstrata declara métodos que servem como **passos** de um algoritmo, bem como o **método template()** que chama esses passos em ordem específica

**2 -** Classes concretas podem sobrescrever todos os métodos **específicos** (menos o método template)

```
class GameAI(ABC):
    # Método template define o esqueleto do algoritmo.
    def turn(self):
        self.collectResources()
        self.buildStructures()
        self.buildUnits()
        self.attack()

    # Alguns passos podem ser implementados aqui
    def collectResources(self):
        for s in self.builtStructures:
            s.collect()

    # Outros definidos como abstract
    @abstractmethod
    def buildStructures(self):
        pass
```

```
# Classes concretas implementam todas as operações abstratas, mas não
# o método template
class OrcsAI(GameAI):
    def buildStructures(self):
        if self.resources:
            # Build farms, then barracks, then stronghold.
            pass

    def buildUnits(self):
        if self.resources > self.scout_cost:
            if not self.scouts:
                # Build peon, add it to scouts group.
                pass
            else:
```

```
class MonstersAI(GameAI):
    def collectResources():
        pass

    def buildStructures():
        pass

    def buildUnits():
        pass
```

# Template

---

## Prós

- Podemos deixar classes especializadas sobrescrever somente partes de um algoritmos maior, reduzindo acoplamento
- Podemos subir códigos duplicados para a superclasse

## Contras

- Possível violação do LSP se alguma subclasse não implementar um dos métodos esperados
- Não escala bem à medida que mais passos são adicionados ao template

**Até a  
próxima!**