

Introdução a Padrões de Projeto Orientado a Objetos (Parte 01)

Escrevendo código mais profissional com *Design Patterns*

Prof. Dirson S. de Campos

dirson_campos@ufg.br

Material elaborado em parceria com os professores **Nádia F. F. da Silva**, **Juliana P. Félix**, **Guilherme S. Marques** e **Reinaldo de S. Júnior**.

20/11/2023

INF

INSTITUTO DE
INFORMÁTICA



Sumário

1. Introdução a Padrões de Projeto Orientados a Objetos
2. Padrões de Criação
 - Factory
 - Singleton
3. Padrões Estruturais
 - Composite
4. Padrões Comportamentais
 - Observer
5. Padrões de Persistência de Dados Estruturados na Orientado a Objetos
 - Usando a Linguagem SQL
 - Instância de Banco de Dados nas nuvens (*Cloud SQL Instance*)
 - Database ORM (Object Relational Mapper)
6. Padrões de Persistência de Framework Web para Orientação a Objetos
 - Java
 - Python



Introdução a Padrões de Projeto

Padrões de Projeto

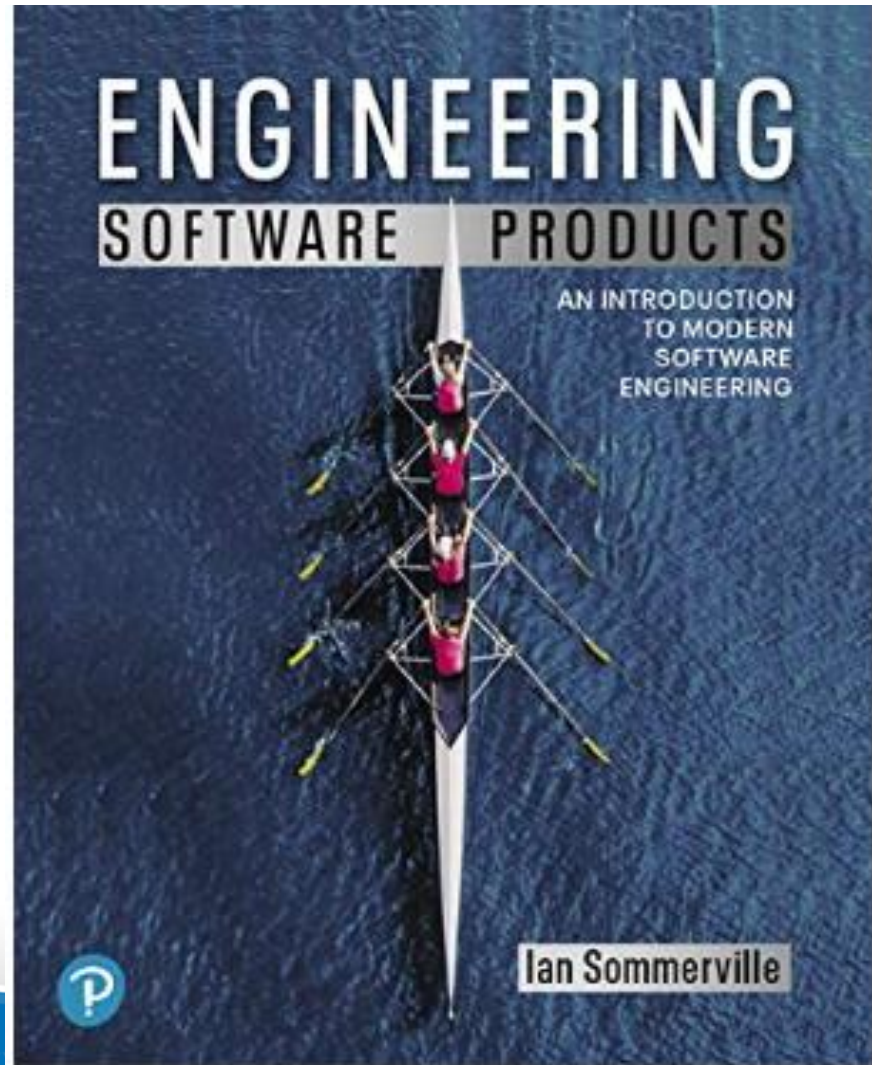
"O padrão é uma descrição do problema e essência de sua solução, onde pode ser reutilizada em diversos casos. O padrão não é uma especificação detalhada, pode-se pensar como uma descrição de conhecimento e experiência acumulados."


. An Introduction to Modern Software Engineering

IAN SOMMERVILLE, 2019

Engineering Software Products: An Introduction to Modern Software Engineering

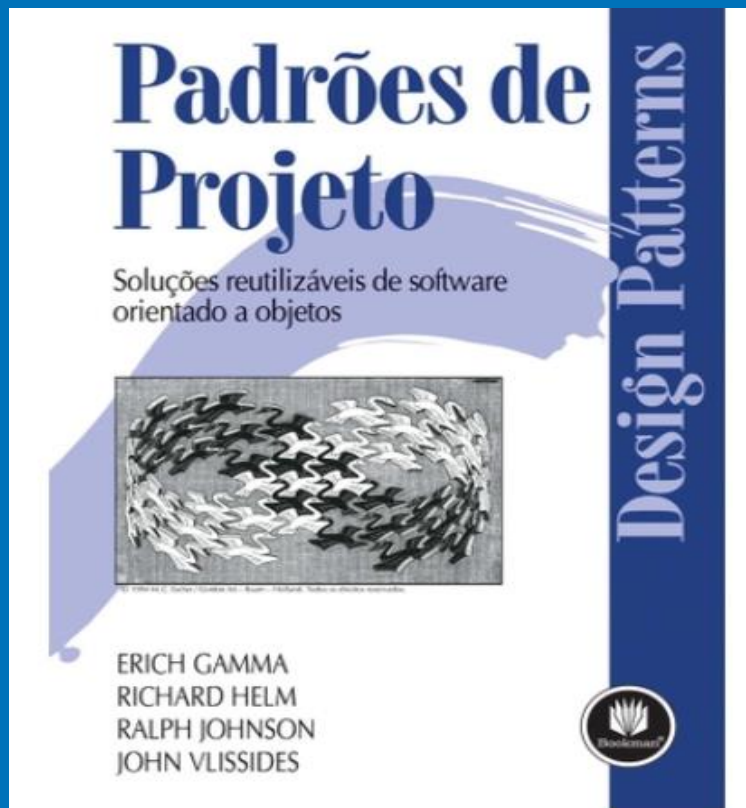
Ian Sommerville
9 maio 2019





Padrões de Projeto e a implementação em Linguagem Orientada a Objetos

Livro Design Patterns em português (Java)



Livro Design Patterns em português (Python)



Livro Design Patterns em português (C#)



Padrões de Projeto

Em 1995, os profissionais: Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides, escreveram e lançaram o livro chamado “**Design Patterns: Elements of Reusable Object-Oriented Software**”, que mostra os detalhes dos **23 Design Patterns**. Por essa realização, os profissionais foram batizados com o nome “*Gangue dos Quatro*” (Gang of Four ou GoF).

Por quê usar estes padrões?

- Projetar software OO reusável e de boa qualidade é difícil;
- Há uma grande mistura de conceitos do que deve ser específico ou genérico;
- Impossível acertar da primeira vez;
- Projetistas experientes usam soluções com as quais já trabalharam no passado, melhorando-as.

Assim, a padronização permite agilidade para as soluções de problemas recorrentes no desenvolvimento do sistema.

Padrões de Projeto

Existem outros "padrões" na área de software para se estudar além dos de projeto? Sim! São frequentemente estudados em Engenharia de Software:

Padrões de Análise:

Descrevem grupos de conceitos que representam construções comuns na modelagem do domínio. Estes padrões podem ser aplicáveis em um domínio ou em muitos.

Padrões de Arquitetura:

Descrevem a estrutura e o relacionamento dos componentes de um sistema de software.

Padrões de Processo

Descrevem modelos de processo, como a ordem e sequência de execução de tarefas.

Padrões de Projeto

Os Padrões da "*Gangue dos Quatro*" tem como objetivo, solucionar problemas comuns de softwares que tenham algum envolvimento com Orientação a Objetos.

São formados por três grupos:

- **Padrões de criação:** *Factory Method*, *Abstract Factory*, *Singleton*, *Builder*, *Prototype*;
- **Padrões estruturais:** *Adapter*, *Bridge*, *Composite*, *Decorator*, *Facade*, *Flyweight*, *Proxy*;
- **Padrões comportamentais:** *Chain of Responsibility*, *Command*, *Interpreter*, *Iterator*, *Mediator*, *Memento*, *Observer*, *State*, *Strategy*, *Template*, *Method*, *Visitor*.

Eles são desenvolvidos baseando-se nas melhores práticas, através de soluções refinadas por um longo processo de teste e reflexão sobre qual a melhor estratégia para resolução de um determinado problema.



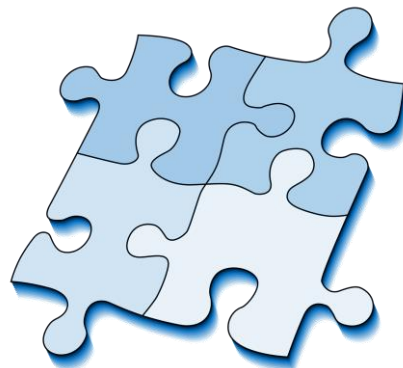
Padrões de Criação

Padrões de Criação

Os Padrões de Criação, ou Padrões Criacionais, exigem um tratamento de como os objetos são criados, para atenderem às diversas necessidades.

No Java, os objetos são instanciados através de seus **construtores**, porém a utilização deles fica limitada quando:

- O código que referencia a criação de um objeto precisa conhecer os construtores dele, isso aumenta o **acoplamento das classes**;
- O objeto não pode ser criado **parcialmente**;
- O construtor não consegue controlar o **número de instâncias** presentes na aplicação.



Padrões de Criação: *Singleton*

O primeiro padrão que veremos se chama *Singleton*

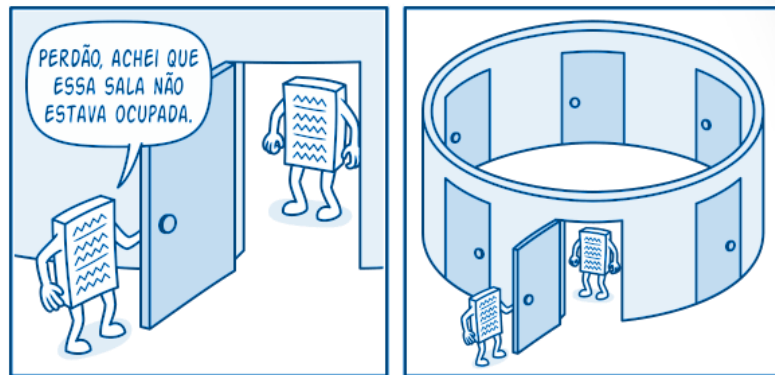
Problema que se quer resolver:

Como pode ser construída uma classe que só pode ter uma única instância e que pode ser acessada globalmente dentro da aplicação?

Contexto:

Em algumas aplicações, uma classe deve ter exatamente uma instância, como variáveis globais, que precisam ser acessadas em vários locais.

É o caso de objetos que costumam guardar configurações do sistema, ou dados de acesso de um usuário.



Padrões de Criação: *Singleton*

O primeiro padrão que veremos se chama *Singleton*

Solução:

Criar uma classe com um método de classe estático `getInstance()`, que:

- Quando é acessado pela primeira vez, a instância do objeto é criada e retornada.
- Nos acessos subsequentes ao método `getInstance()`, nenhuma instância adicional é criada, mas a identidade do objeto existente é retornada.

Este deve ser o único jeito de se criar/acessar a instância. Portanto o **construtor deve ser privado**. Além disso, **a classe é `final`**, pois não permite a criação de subclasses da própria classe.

Vantagens:

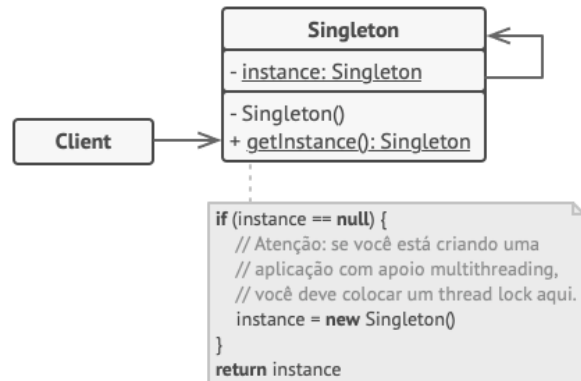
Oferece acesso controlado à única instância do objeto, pois a classe encapsula a instância. Uma variação pode ser usada para se criar um número específico de instâncias.

Padrões de Criação: *Singleton*

O primeiro padrão que veremos se chama *Singleton*.

Imagine uma classe de um sistema responsável por guardar as preferências de um usuário. Ela será acessada por diferentes componentes do sistema, para checar tais preferências, como tamanho de fonte, cor, itens favoritos, login... mas estes componentes não podem sair criando instâncias diferentes da classe que guarda tais informações!

```
public final class Preferências {  
    private static Preferências instância;  
  
    private Preferências() {}  
    public static Preferências getInstance() {  
        if (instância == null)  
            instância = new  
Preferências();  
        return instância;  
    }  
    // Acesso externo, por exemplo na função main:  
    // Preferências.getInstance().algumMétodo...  
}
```



Padrões de Criação: *Factory Method*

Outro padrão de criação importante é o **Factory Method** (Fábrica).

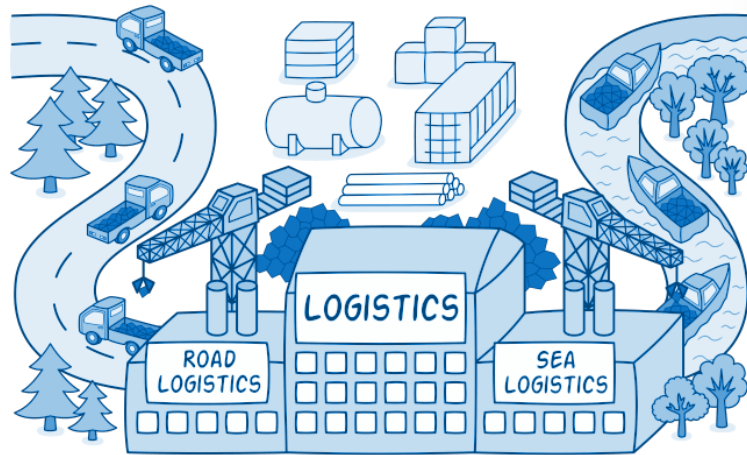
Problema que se quer resolver:

Como **adiar** a decisão de que tipo de classe instanciar?

Contexto:

Quando uma classe cria instâncias de outra classe, aumenta a **dependência** e o **acoplamento** entre elas, e isso é um grande problema para projetos.

O ideal é tornar o acoplamento o menor possível.



Padrões de Criação: *Factory Method*

Outro padrão de criação importante é o *Factory Method* (Fábrica).

Solução:

Uma classe de criador abstrata que chamamos de `Creator` que define um método fábrica abstrato que as subclasses implementam para criar um "produto", podendo possuir um ou mais métodos.

Cada `ConcreteCreator` implementará seu próprio método de criação.

Vantagens:

Encapsula-se a criação de objetos deixando as subclasses decidirem quais objetos criar – diminui o acoplamento.

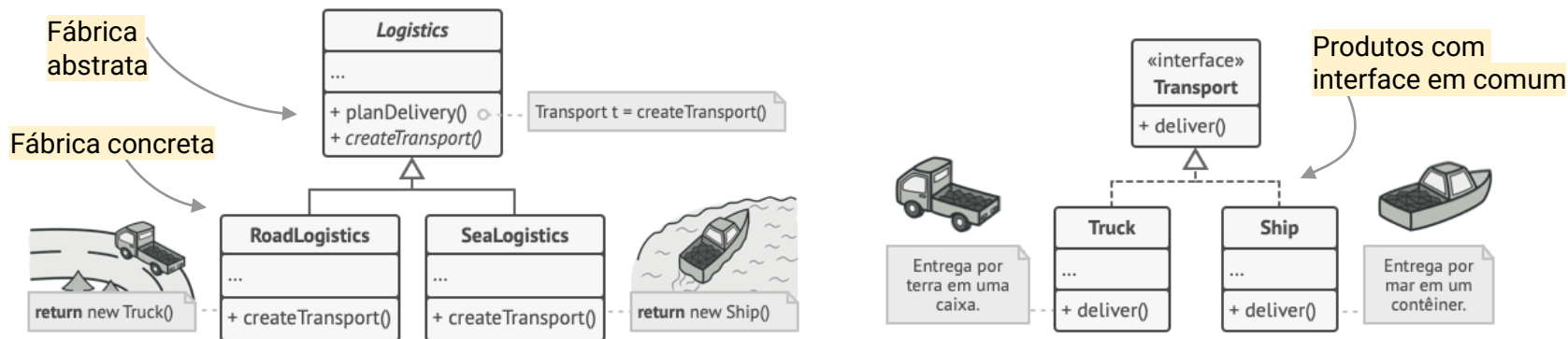
Permite a uma classe decidir a instanciação para suas subclasses

Padrões de Criação: *Factory Method*

Outro padrão de criação importante é o **Factory Method (Fábrica)**.

Suponha um cenário de um aplicativo de logísticas. De início, o código pode ter sido pensado para logística rodoviária. Com o tempo, porém, a demanda faz com que seu aplicativo cresça e você precisa adaptar seu código para logística marítima. Vale a pena ter um bando de IFs pra decidir quando criar um Caminhão e quando criar um Navio?

Além da classe abstrata que atua como Fábrica, temos uma interface em comum entre os produtos, para que assim a classe abstrata possa oferecer métodos em comum.

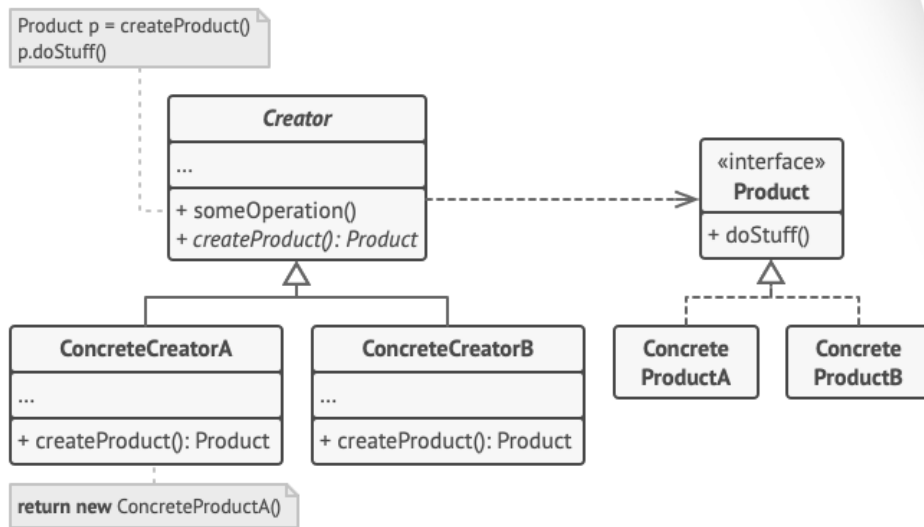


Padrões de Criação: *Factory Method*

Outro padrão de criação importante é o **Factory Method (Fábrica)**.

Em nossa classe `Principal`, instanciamos a `ConcreteCreator` apropriada e usamos seu método `createProduct` para fabricar o objeto do tipo adequado. Dentro dele é que é chamado o construtor daquele tipo.

A partir daí, podemos usar uma lógica que é comum a todos os produtos, graças à interface `Produto` sem nos preocupar com qual tipo foi escolhido.



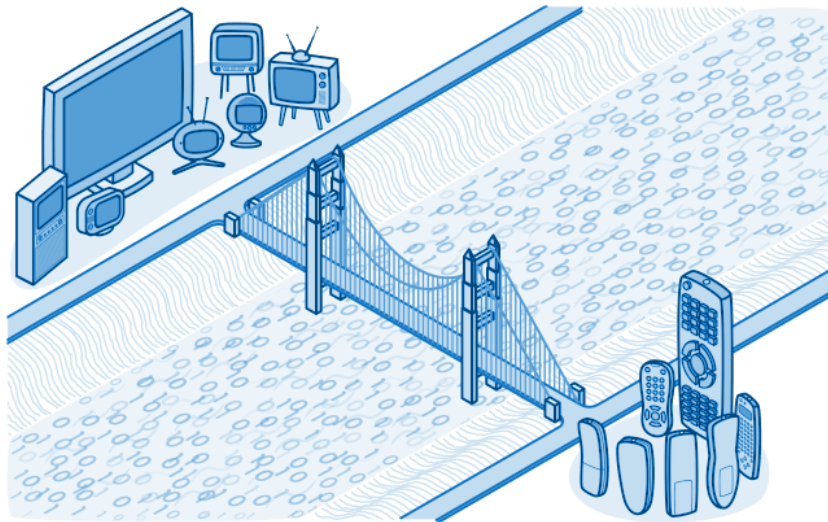


Padrões Estruturais

Padrões Estruturais

Os Padrões Estruturais descrevem aspectos como a *elaboração*, *associação* e a *organização* entre objetos e classes/interfaces.

Permitem combinar objetos em estruturas mais complexas, ou descrever como as classes são herdadas ou compostas a partir de outras.



Padrões Estruturais: Composite

Um interessante padrão estrutural é o **Composite (Árvore de Objetos)**.

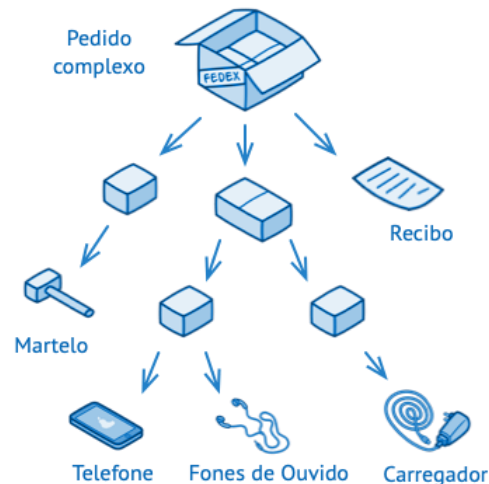
Problema que se quer resolver:

Se existe um requisito para representar hierarquias complexas todo-parte, então ambos objetos (todo e parte) oferecem a mesma interface para objetos cliente.

Contexto:

Numa aplicação tanto o objeto que contém quanto o que é componente devem oferecer o mesmo comportamento.

Objetos cliente devem ser capazes de tratar objetos compostos ou componentes do mesmo jeito.



Padrões Estruturais: Composite

Um interessante padrão estrutural é o *Composite (Árvore de Objetos)*.

Imagine um cenário de e-commerce, onde existem `Produtos` e `Caixas`. As `Caixas` podem conter `Produtos` dentro de si, ou mesmo `Caixas` menores, que por sua vez podem conter mais `Caixas` ou `Produtos`.

Para se implementar um sistema de pedidos que verifica o preço total destes componentes, qual a melhor maneira de se checar o *preço* total?

O requisito que os objetos, tanto composto (`Caixa`) como componente (`Produto`), ofereçam a mesma interface, sugere que eles pertençam à mesma hierarquia de herança (uma que informa o *preço*). Isto permite que operações sejam herdadas e redefinidas com a mesma assinatura (polimorfismo).

A necessidade de representar hierarquias *todo-parte* indica a necessidade de uma estrutura de agregação, que chamamos de classe `Composite` para nossas `Caixas`.

Padrões Estruturais: Composite

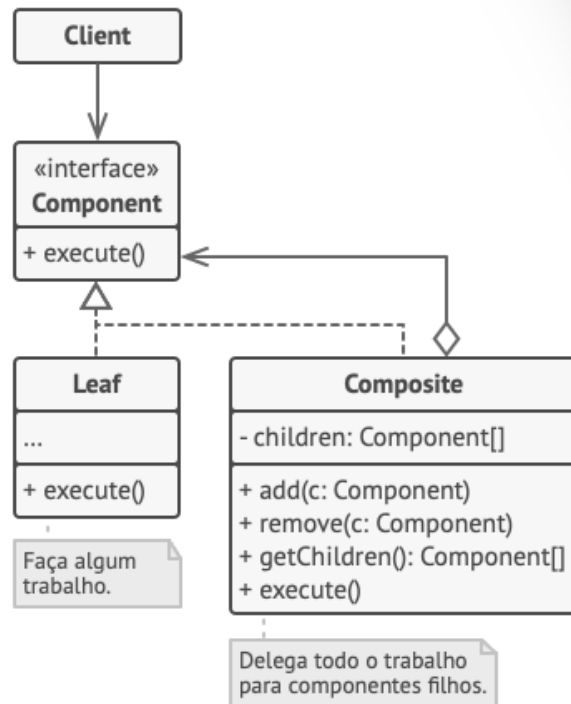
Um interessante padrão estrutural é o *Composite (Árvore de Objetos)*.

Solução

Ambas subclasses, `Leaf (Produto)` e `Composite (Caixa)`, têm uma operação redefinida (implementada ou sobrescrita) usando polimorfismo `execute (verPreço)`.

Na classe `Composite` esta operação redefinida invoca a operação relevante a partir de seus componentes (*children*) usando um loop, ou seja, delega para seus itens internos esta operação.

A subclasse `Composite` também tem operações adicionais para gerenciar a hierarquia de agregação, como a gestão de como os componentes podem ser adicionados ou removidos.



Padrões Estruturais: Composite

Um interessante padrão estrutural é o *Composite (Árvore de Objetos)*.

Outros Exemplos:

- Em um *Sistema de Arquivos*, tanto *Pastas* (Composite), quanto *Documentos* (Leaf), implementam operações em comum de um *Item* (interface Component), para operações como *abrir*, *mover*, *renomear*, *remover*, *copiar*...
- Em uma aplicação de edição de gráficos, *Elementos* diversos como *pontos*, *linhas* e *formas geométricas* (Leaf) devem implementar métodos de uma interface em comum (Component), como *redimensionar*, *mover*, *rotacionar*. Mas estas ações também precisam poder ser feitas por *Grupos de elementos* (Composite), como quando selecionamos mais de um elemento e os movemos de uma vez.

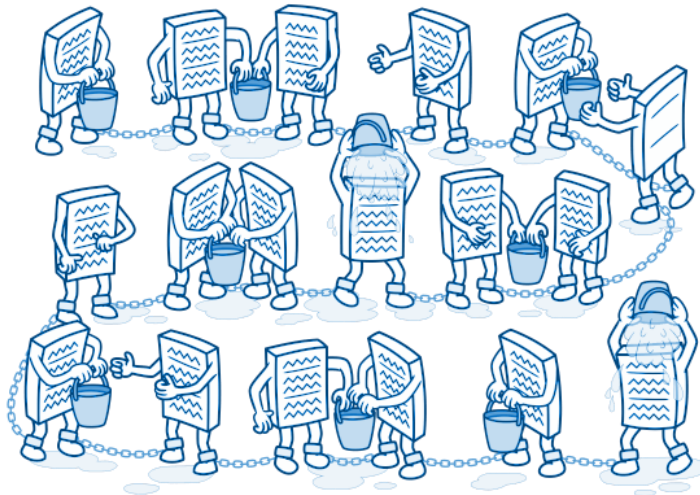


Padrões Comportamentais

Padrões Comportamentais

Os Padrões Comportamentais mostram o processo de como os objetos ou classes se comunicam.

Em geral, buscam um **baixo acoplamento** entre os objetos, apesar da existências de uma necessidade de comunicação entre eles.



Padrões Comportamentais: Observer

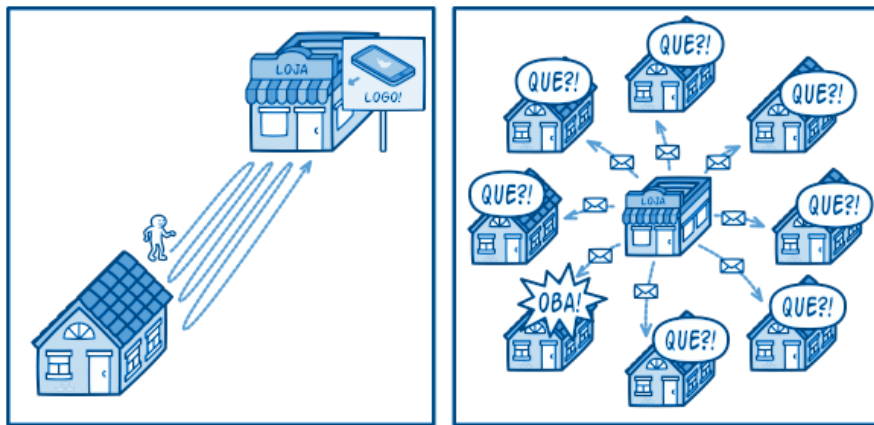
Um padrão comportamental muito usado é o **Observer** (*Escutador, Assinante*).

Problema que se quer resolver:

Os dados do componente sujeito modificam-se constantemente e precisam ser atualizados nos outros componentes. O número de componentes pode variar.

Contexto:

Situações nas quais vários componentes dependem de dados que são modificados em outro componente (*sujeito*). A constante consulta pela informação atualizada não é viável.



Padrões Comportamentais: Observer

Um padrão comportamental muito usado é o *Observer (Escutador, Assinante)*.

Solução:

Utilizar um mecanismo de registro que permite ao componente *sujeito* notificar aos interessados sobre mudanças. O *sujeito* passa então a ser "Publicador/Observable". Todos os outros objetos que querem saber das mudanças do estado do Publicador são chamados de "Assinantes/Observers".

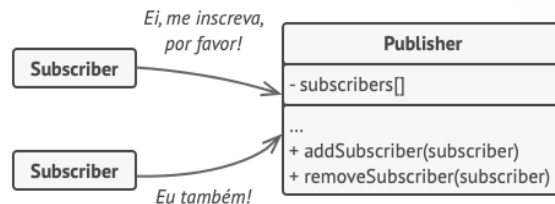
O padrão Observer sugere que você adicione um mecanismo de assinatura para a classe publicadora para que objetos individuais possam assinar ou desassinar uma corrente de eventos vindo daquela publicadora. Isso se dá da seguinte forma:

- 1) Um vetor para armazenar uma lista de referências aos objetos Assinantes/Observers;
- 2) Alguns métodos públicos que permitem adicionar assinantes e removê-los da lista.

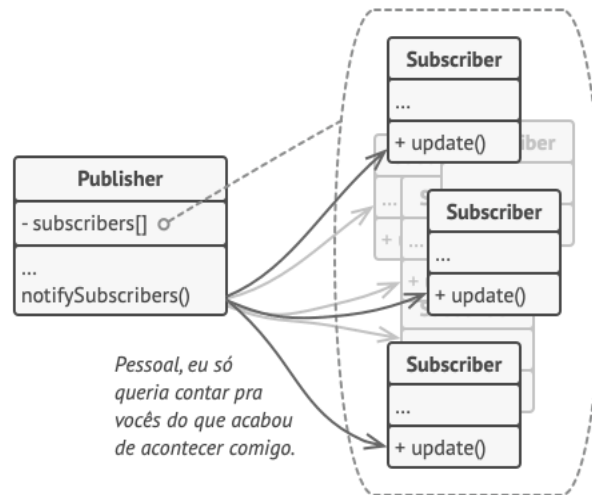
Padrões Comportamentais: Observer

Um padrão comportamental muito usado é o **Observer** (*Escutador, Assinante*).

Em um primeiro momento, o Assinante se registra para entrar na lista do Publicador
(`publisher.addSubscriber(subscriber)`).



Mais adiante no código, quando o Publicador deseja informar mudanças, ele chama, em seu método `notifySubscribers()` os métodos `update()` de cada `Subscriber`, através de um loop.



Padrões Comportamentais: Observer

Um padrão comportamental muito usado é o **Observer** (*Escutador, Assinante*).

Felizmente, o Java já possui uma solução padrão para isso:

Interface pública `Observer`: As classes que implementarem essa interface (os Assinantes), que por sua vez conterá um método de atualização (`update`), deverão ser notificadas quando da atualização ou mudança do estado do objeto observado.

Classe `Observable`: Define a classe que será observada (o Publicador). Essa classe geralmente guarda como atributo uma lista dos `Observers` associados à mesma, para que os mesmos possam ser notificados quando a mudança acontecer na classe.

«interface»
java.util.Observer
+ update(o: Observable, arg: Object)

java.util.Observable
+ Observable()
+ addObserver(o: Observer)
+ countObservers(): int
+ deleteObserver(o: Observer)
+ deleteObservers()
+ hasChanged(): boolean
+ notifyObservers(arg: Object)
+ notifyObservers()

Padrões Comportamentais: Observer

Um padrão comportamental muito usado é o **Observer** (*Escutador, Assinante*).

Felizmente, o Java já possui uma solução padrão para isso:

Interface pública Observer: As classes que implementam essa interface (os Assinantes) que por sua vez conterá um método para ser notificadas quando ocorrer uma mudança do estado do objeto observado.

Classe Observable: Define a classe que será observada (o Publicador). Essa classe geralmente guarda como atributo uma lista dos Observers associados à mesma, para que os mesmos possam ser notificados quando a mudança acontecer na classe.

Depreciado na Versão 9 do Java



```
interface Observer
+ update(): Observable, Object)
```



```
java.util.Observable
+ Observable()
+ addObserver(o: Observer)
+ countObservers(): int
+ deleteObserver(o: Observer)
+ deleteObservers()
+ hasChanged(): boolean
+ notifyObservers(arg: Object)
+ notifyObservers()
```

Padrões Comportamentais: Observer

Um padrão comportamental muito usado é o *Observer* (*Escutador, Assinante*).

Felizmente, o Java já possui uma solução padrão para isso:

Novo Observer: um objeto que deseja "observar" estas mudanças deve ser de uma classe que implementa [PropertyChangeListener](#).

Novo Observable: Um objeto que deseja ser "observável", precisa manter uma referência (geralmente um atributo) que seja uma instância da classe [PropertyChangeSupport](#). Nele, você pode adicionar listeners e disparar notificações.

```
<<interface>>  
PropertyChangeListener
```

```
+ propertyChange(event: PropertyChangeEvent)
```

```
PropertyChangeSupport
```

```
+addPropertyChangeListener(p: PropertyChangeListener)  
+removePropertyChangeListener(p: PropertyChangeListener)  
+firePropertyChange(pName: String, oldValue, newValue)
```

Padrões Comportamentais: Observer

Um padrão comportamental muito usado é o *Observer* (*Escutador, Assinante*).

Outros Exemplos:

- Em qualquer sistema de interface gráfica, há uma série de soluções orientadas a eventos. Estes eventos são justamente notificações de mudanças que precisam ser passadas para diferentes componentes. Por exemplo, uma classe `Botão` deve registrar-se para escutar os eventos de Clique, passar o mouse por cima, etc, presentes na classe que recebe interações do usuário;
- Requisições de dados online costumam ser feitas em *threads* separadas. Se um usuário faz uma pesquisa em um campo de busca, o software não fica parado sem fazer nada até a resposta chegar do servidor. Ele precisa instanciar um estado de "carregando" e permitir que outras coisas possam ser feitas enquanto a busca não termina. O controle de como a informação "avisa" que está pronta para ser lida é geralmente feito usando-se o padrão *publisher-subscriber*, que vimos neste padrão comportamental.

Questões sobre Padrões

Ao se escolher um padrão, é importante considerar:

- Existe algum padrão que trata um problema similar?
- O contexto do padrão é consistente com o problema real?
- O padrão tem uma solução alternativa que pode ser mais aceitável?
- Existe uma solução mais simples?
- Existem algumas restrições que sejam impostas pelo ambiente de software que está sendo usado?



Padrões de Projeto para Persistência de Banco de Dados

Padrões de Persistência de Dados Estruturados na Orientado a Objetos

1. Padrões de Persistência p/ Banco de Dados para Orientação a Objeto
 - Código SQL embutido em uma Linguagem Orientada a Objetos
 - Instância de Banco de Dados nas nuvens (*Cloud SQL Instance*)
 - Database ORM (Object Relational Mapper)

Código SQL embutido em uma Linguagem Orientada a Objetos

- Criar um Objeto de Conexão com o Sistema Gerenciador de Banco de Dados (SGBD) importada através de uma biblioteca da linguagem.
- Manipulação da linguagem SQL embutida numa aplicação feita em uma Linguagem Orientada a Objetos (Java, Python, C#, etc) que é de máxima utilidade e facilidade na operação dos comandos SQL dentro do código da linguagem Orientada a Objetos
 - *Structured Query Language* (SQL), literalmente “Linguagem de Consulta Estruturada”, é uma linguagem de domínio específico desenvolvida para gerenciar dados relacionais em um sistema de gerenciamento de banco de dados, ou para processamento de fluxo de dados em um sistema de gerenciamento de fluxo de dados.

Instância de Banco de Dados nas nuvens (*Cloud SQL Instance*)

- São serviço, geralmente oferecidos por Big Techs que tenha DataCenters com um servidor Cloud (em nuvens) com suporte a SGBDs, por exemplo, MySQL, PostgreSQL e SQL Server, etc. com coleções avançadas de extensões, além de sinalizações de configuração e ecossistemas de desenvolvedores.
- Exemplos:
 - Servidores Cloud da Under

<https://under.com.br/servidor-cloud/>

- Google Cloud

<https://cloud.google.com/?hl=pt-br>

Instância de Banco de Dados nas nuvens (Exemplos de serviços da Google Cloud)

cloud.google.com/?hl=pt-br

Plataforma Turing d... Chapter 15staff.em... BancoDados (M/D)... Meet - bgf-pzsf-dom Gmail YouTube Maps FIE 2020 Moodle - INF-UFG

Google Cloud Informações Gerais Soluções Produtos Preços Recurs > Documentos Suporte Português -... Console

Entre em contato conosco Comece gratuitamente

Escolha entre mais de 150 produtos de ponta

Conheça e avalie o Google Cloud com o uso gratuito de mais de 20 produtos. Novos clientes recebem US\$ 300 em créditos ao fazer a inscrição.

Comece agora

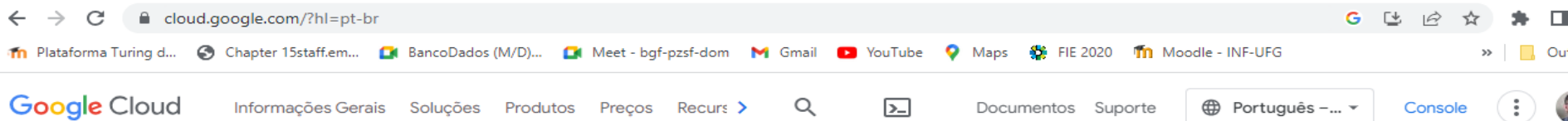
[Veja todos os produtos](#)

Armazenamento

- ✓ Armazene qualquer tipo e quantidade de dados e recupere-os quando quiser usando o [armazenamento de objetos](#)
- ✓ Mova dados com soluções de transferência on-line e off-line, incluindo o [Serviço de transferência do Cloud Storage](#) e o [Transfer Appliance](#)
- ✓ O [armazenamento em blocos do Persistent Disk](#) é totalmente integrado aos produtos do Google Cloud, como o Compute Engine e o GKE

Instância de Banco de Dados nas nuvens

(Exemplos de serviços da Google Cloud)



Escolha entre mais de 150 produtos de ponta

Conheça e avalie o Google Cloud com o uso gratuito de mais de 20 produtos. Novos clientes recebem US\$ 300 em créditos ao fazer a inscrição.

[Comece agora](#)[Veja todos os produtos](#)

IA e machine learning

Bancos de dados

- ✓ Reduza os custos de manutenção com bancos de dados [MySQL](#), [PostgreSQL](#), e SQL Server totalmente gerenciados
- ✓ Simplifique migrações para o [Cloud SQL](#) a partir do MySQL e PostgreSQL usando o [Database Migration Service](#)
- ✓ [Desenvolva aplicativos complexos](#) usando um banco de dados de documentos totalmente gerenciado, escalonável e sem servidor

Padrões de Persistência de Dados

Database ORM (Object Relational Mapper)

- Object-Relational Mapping ou ORM é uma técnica para converter dados entre objetos de uma Linguagem de Programação a Objetos (ex. Java, Python e C#, etc.) e bancos de dados relacionais.
- ORM converte dados entre dois sistemas de tipos de dados incompatíveis (ex. tipo de dados de Java e tipo de dados SQL), de modo que cada classe de modelo se torne uma tabela em nosso banco de dados e cada instância uma linha da tabela.

Exemplo de ORM para Java

```
public class Person {
```

```
    private Long id;  
    private String firstName;  
    private String lastName;  
    private ContactAddress address;  
}
```

```
public class ContactAddress {
```

```
    private String contactNo;  
    private String streetName;  
    private String city;  
    private String houseNumber;  
    private String country;  
    private String zipcode;  
}
```



JavaByDeveloper™

Exemplo de ORM para Python

fullstackpython.com/object-relational-mappers-orms.html

Plataforma Turing d... Chapter 15staff.em... BancoDados (M/D)... Meet - bgf-pzsf-dom Gmail YouTube Maps FIE 2020 Moodle - INF-UFG

Object-relational Mappers (ORMs)

An object-relational mapper (ORM) is a code library that automates the transfer of data stored in relational database tables into objects that are more commonly used in application code.

Relational database (such as PostgreSQL or MySQL)

ID	FIRST_NAME	LAST_NAME	PHONE
1	John	Connor	+16105551234
2	Matt	Makai	+12025555689
3	Sarah	Smith	+19735554512
...

Python objects

```
class Person:
    first_name = "John"
    last_name = "Connor"
    phone_number = "+16105551234"

class Person:
    first_name = "Matt"
    last_name = "Makai"
    phone_number = "+12025555689"

class Person:
    first_name = "Sarah"
    last_name = "Smith"
    phone_number = "+19735554512"
```

ORMs provide a bridge between
**relational database tables, relationships
and fields and Python objects**

Why are ORM's useful?

ORMs provide a high-level abstraction upon a relational database that allows a developer to write Python code instead of SQL to create, read, update and delete data and schemas




Adobe Creative Cloud for Teams.
Put creativity to work.

ADS VIA CARBON

Table of Contents

1. Introduction
2. Development Environments
3. Data
 - Relational Databases
 - PostgreSQL
 - MySQL
 - SQLite



Padrões de Projeto para Framework para Desenvolvimento WEB Orientado a Objetos

Framework para Web

- Um conjunto de ferramentas pronto para uso que fornece componentes para resolver problemas e dificuldades de desenvolvimento com velocidade rápida e alta funcionalidade é conhecido como framework.
- Na programação, um framework geralmente consiste em bibliotecas de código, APIs, documentos de referência, compilador e outras ferramentas de suporte.
- Em resumo, um framework funciona como um modelo para fornecer possível suporte de infraestrutura tecnológica e conceitual a um determinado Paradigma de Programação.

Exemplo de Framework Web para Java 1

Jakarta Faces, formerly **Jakarta Server Faces** and **JavaServer Faces (JSF)** is a [Java](#) specification for building [component](#)-based [user interfaces](#) for [web applications](#)^[2] and was formalized as a standard through the [Java Community Process](#) being part of the [Java Platform, Enterprise Edition](#). It is also an [MVC web framework](#) that simplifies the construction of [user interfaces](#) (UI) for server-based applications by using reusable UI components in a page.^[3]

JSF 2.x uses [Facelets](#) as its default templating system. Users of the software may also choose to employ technologies such as [XUL](#), or [Java](#).^[4] JSF 1.x uses [JavaServer Pages](#) (JSP) as its default templating system.

History [\[edit\]](#)

Jakarta Faces

Original author(s)	Sun Microsystems
Developer(s)	Eclipse Foundation
Stable release	4.0.0 / May 15, 2022; 13 months ago ^[1]
Repository	github.com/eclipse-ee4j/faces-api 
Written in	Java
Type	Web application framework
Website	jakarta.ee/specifications/faces/ 

Exemplo de Framework Web para Java 1

specification for building [component-based user interfaces](#) for [web applications](#)^[2] and was

standardized as a standard through the [Java Community Process](#) being part of the [Java](#) [Platform, Enterprise Edition](#). It is also an [MVC web framework](#) that simplifies the

construction of [user interfaces](#) (UI) for [web applications](#) by providing a set of [components](#) in a page.^[3]

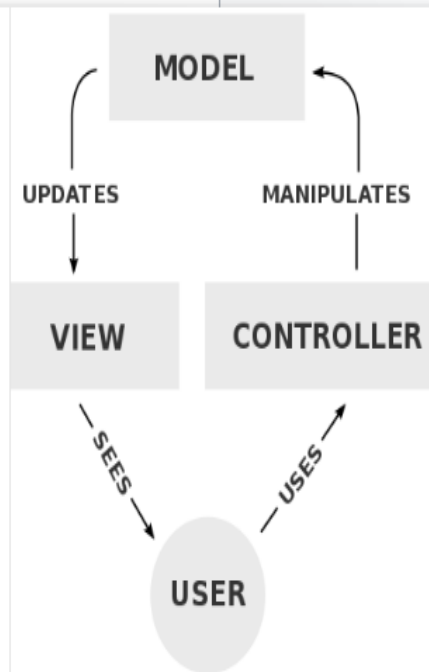
JSF 2.x uses [Facelets](#) as its default templating system and allows developers to choose to employ technologies such as [XML](#) or [HTML](#) (JSP) as its default templating system.

History [\[edit\]](#)



This section **needs expansion**.
You can help by [adding to it](#).
(August 2013)

Model–view–controller (MVC) is a software design pattern commonly used for developing user interfaces that divides the related program logic into three interconnected elements. This is done to separate internal representations of information from the ways information is



Original author(s) [Sun Microsystems](#)

Developer(s) [Eclipse Foundation](#)

Stable release 4.0.0 / May 15, 2022; 13 months ago^[1]

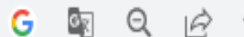
github.com/eclipse-ee4j/faces-api [↗](#) [✎](#)

[Java](#)

[Web application framework](#)
jakarta.ee/specifications/faces/ [↗](#) [✎](#)

Exemplo de Framework Web para Java 2

https://en.wikipedia.org/wiki/Spring_Framework



Chapter 15staff.em... BancoDados (M/D)... Meet - bgf-pzsf-dom Gmail YouTube Maps FIE 2020 Moodle - INF-UFG

WIKIPEDIA
The Free Encyclopedia

Search Wikipedia

Search

Create

Spring Framework

32 languages

Article Talk

Read Edit View history Tools

From Wikipedia, the free encyclopedia

The **Spring Framework** is an [application framework](#) and [inversion of control container](#) for the [Java platform](#).^[2] The framework's core features can be used by any Java application, but there are extensions for building [web applications](#) on top of the [Java EE](#) (Enterprise Edition) platform. The framework does not impose any specific [programming model](#).^[citation needed] The framework has become popular in the Java community as an addition to the [Enterprise JavaBeans](#) (EJB) model.^[3] The Spring Framework is [free and open source software](#).^{[4]: 121–122[5]}

Spring Framework



Developer(s) VMware

Initial release 1 October 2002; 21 years ago

Stable release 6.0.10^[1] / 15 June 2023; 4 months ago

51

Exemplo de Framework Web para Python



Conteúdo [ocultar]

Início

✓ Principais características

Mapeamento Objeto-Relacional (ORM)

Interface Administrativa

Formulários

URLs Amigáveis

Sistema de Templates

Sistema de Cache

Internacionalização

Ver também

Notas e Referências

Ligações externas

Django é um **framework** para desenvolvimento rápido para web, escrito em **Python**, que utiliza o padrão model-template-view (MTV). Foi criado originalmente como sistema para gerenciar um site jornalístico na cidade de Lawrence, no Kansas. Tornou-se um projeto de código aberto e foi publicado sob a **licença BSD** em 2005. O nome *Django* foi inspirado no músico de jazz **Django Reinhardt**.^[1]

Django utiliza o princípio **DRY** (Don't Repeat Yourself), onde faz com que o desenvolvedor aproveite ao máximo o código já feito, evitando a repetição.

Principais características

Mapeamento Objeto-Relacional (ORM)

Com o **ORM** do Django você define a modelagem de dados através de classes em Python. Com isso é possível gerar suas tabelas no **banco de dados** e manipulá-las sem necessidade de utilizar **SQL** (o que também é possível).

Interface Administrativa

No Django é possível gerar automaticamente uma interface para administração dos modelos criados através do ORM.

Formulários

É possível gerar formulários automaticamente através dos modelos de dados.

URLs Amigáveis

No Django não há limitações para criação de **URLs** amigáveis e de maneira simples.

Django

django

django

The install worked successfully! Congratulations!

You are seeing this page because Django+Free is in your settings file and you have not configured any URLs.

Django Documentation
Topics, references, & how-to

Tutorial: A Polling App
Get started with Django

Django Community
Connect, get help, or contribute

Desenvolvedor	Django Software Foundation
Plataforma	Multiplataforma
Modelo do desenvolvimento	Software Livre
Lançamento	21 de Julho de 2005
Versão estável	4.1 (13 de agosto de 2022; há 10 meses)
Escrito em	Python

Exemplo de Framework Web para Python

Features [edit]

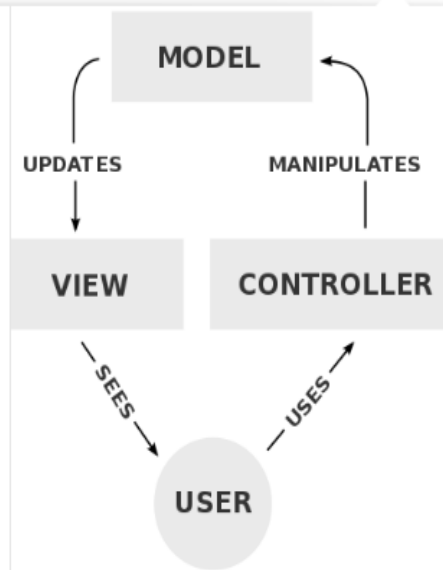
Website

www.djangoproject.com

Components [edit]

Despite having its own nomenclature, such as naming the **callable objects** generating the HTTP responses "views",^[7] the core Django framework can be seen as an **MVC** architecture.^[8] It

Model-view-controller (MVC) is a software design pattern commonly used for developing user interfaces that divides the related program logic into three interconnected elements. This is done to separate internal representations of information from the ways information is



a models (defined as
using HTTP requests
URL dispatcher

HTML forms and values

om object-oriented

The screenshot shows the Django administration interface for changing a user. The page title is 'Django administration' and the breadcrumb is 'Home > Authentication and Authorization > Users > Change user'. The form includes fields for 'Username' (set to 'Whisper'), 'Password' (with a strength indicator), 'First name', 'Last name', and 'Email address'. Below these are sections for 'Permissions' (with 'Active' checked), 'Groups' (with 'Available groups' and 'Chosen groups' lists), and 'User permissions' (with 'Available user permissions' and 'Chosen user permissions' lists). The page number '53' is visible in the bottom right corner.

Material de Apoio

- Um excelente material de apoio, de onde as imagens deste slide foram retiradas: <https://refactoring.guru/pt-br>