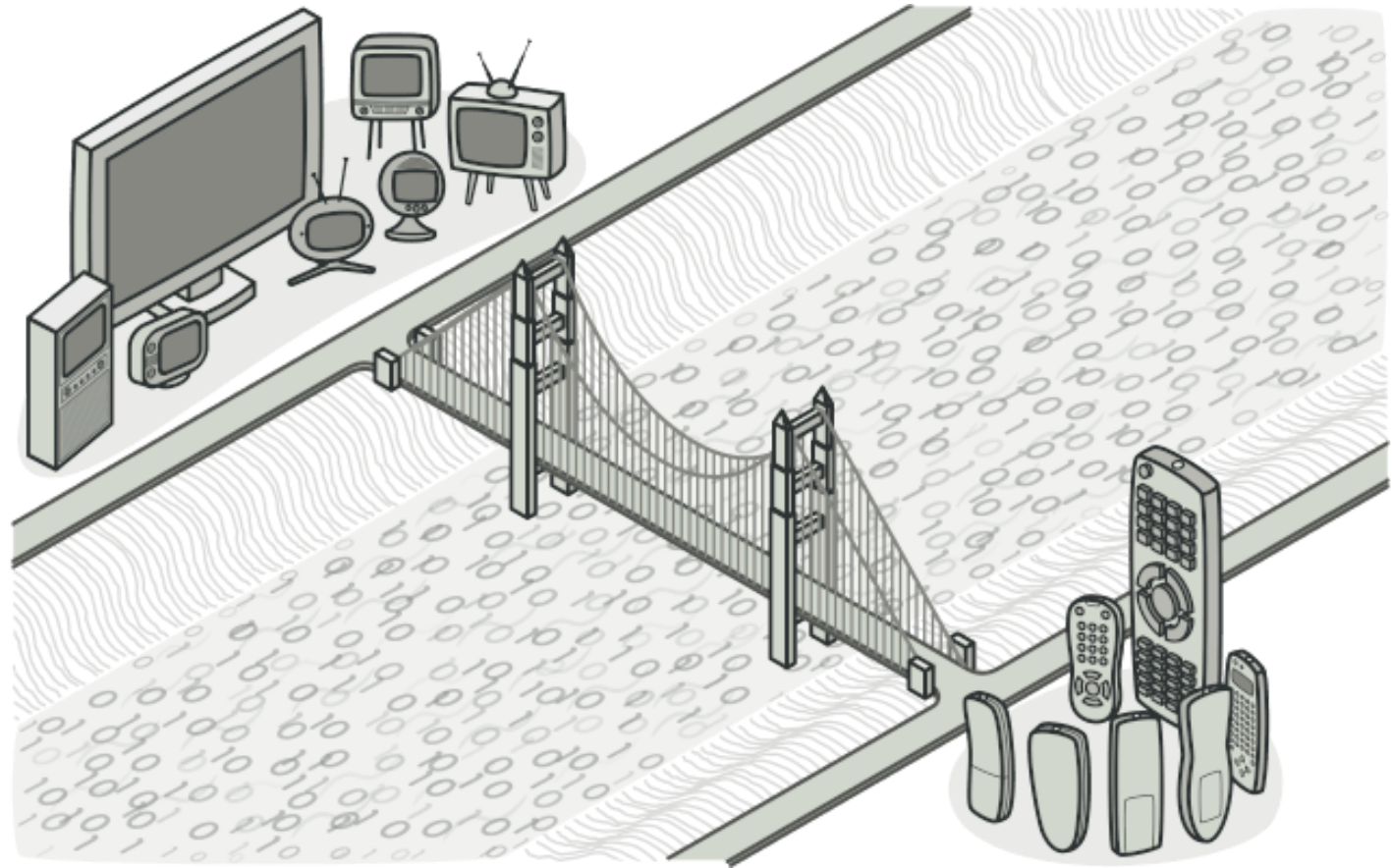


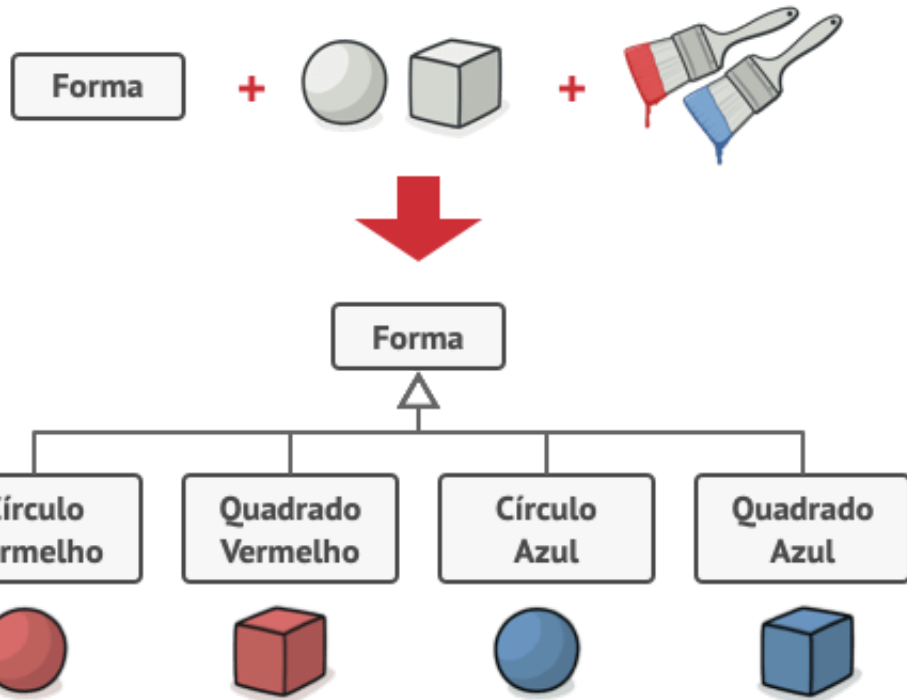
TÓPICO 09: BRIDGE

Design Patterns - Professor Ramon Venson - SATC 2026.2

Definição

Bridge é um padrão de projeto **estrutural** que divide uma classe (ou conjunto de classes) em **duas hierarquias independentes**: a **abstração** e a **implementação**.





Problema

Quando estendemos uma hierarquia em **duas dimensões ao mesmo tempo** (ex.: tipos × variantes), o número de subclasses **explode**:

- `Forma` (Círculo, Quadrado) × `Cor` (Vermelho, Azul)
- cada combinação vira uma **classe nova**: `CírculoAzul`, `QuadradoVermelho` ...

Neste exemplo, cada combinação de forma + cor é uma **classe separada**:

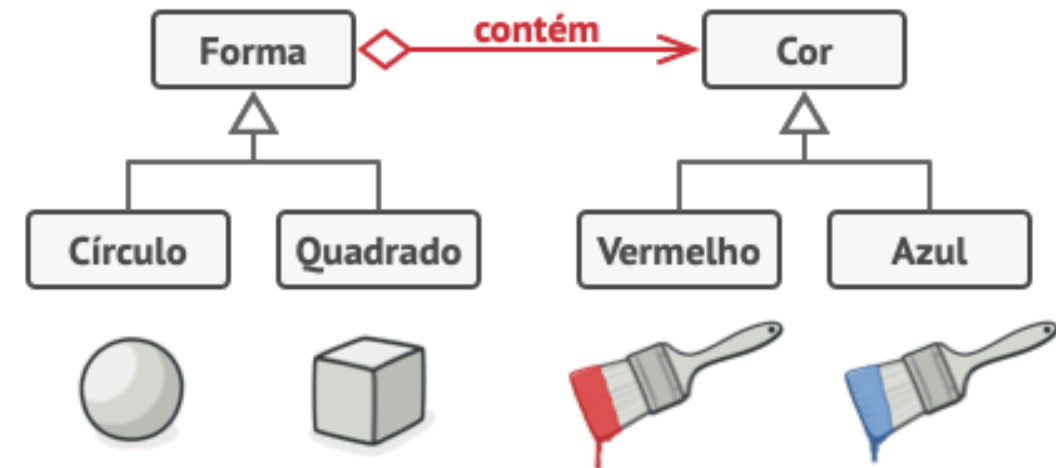
```
public class CirculoVermelho extends Circulo { /* ... */ }  
public class CirculoAzul      extends Circulo { /* ... */ }  
public class QuadradoVermelho extends Quadrado { /* ... */ }  
public class QuadradoAzul     extends Quadrado { /* ... */ }
```

- **2 formas × 2 cores = 4 classes**
- adicionar 1 forma nova (Triângulo) → +2 classes
- adicionar 1 cor nova (Verde) → +3 classes
- com **N** formas e **M** cores são **N × M** subclasses

A raiz do problema: herança tenta variar a classe em **duas dimensões** de uma vez.

Quando isso acontece com frequência, o código:

- **duplica** a lógica de desenho/estado em cada combinação
- fica **difícil de manter** e de **estender**
- mistura responsabilidades que deveriam evoluir **separadas**



Solução

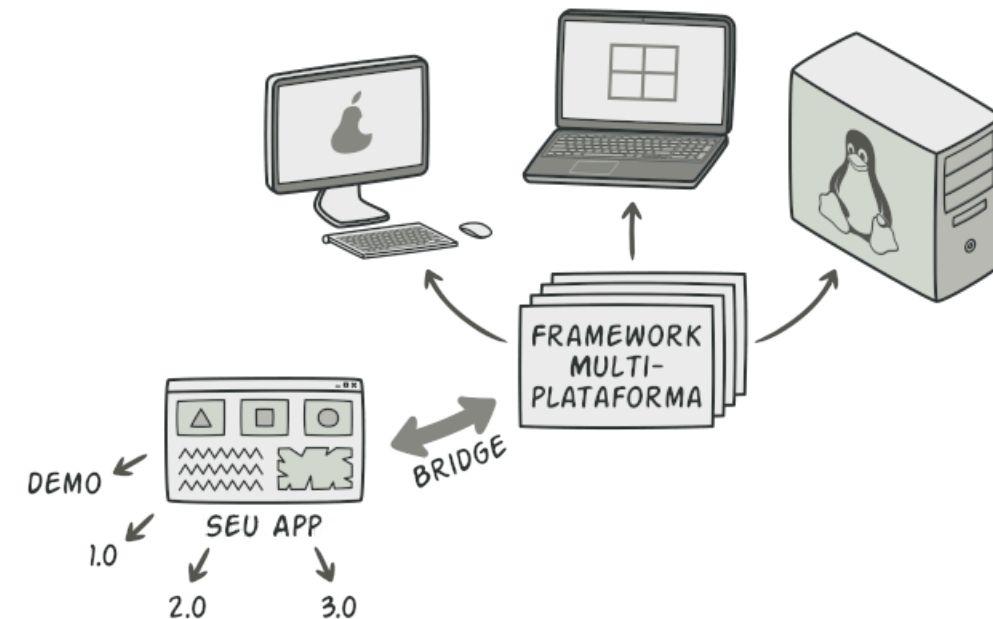
O Bridge **troca herança por composição**: extraímos **uma das dimensões** para uma hierarquia própria, e a classe original passa a **referenciar** um objeto dessa nova hierarquia.

- **Abstração**: a lógica de alto nível (ex.: a "forma", o "controle")
- **Implementação**: a parte variável (ex.: a "cor", o "dispositivo")
- A abstração guarda uma **referência** à implementação e **delega** o trabalho
- Cada hierarquia pode **evoluir independente** da outra

Abstração × Implementação

- **Abstração** = camada de alto nível, que **não faz o trabalho pesado**
- **Implementação** = a parte concreta/variável (plataforma, driver, canal)

Exemplo real: uma **GUI** (abstração) que roda sobre **APIs de SO** diferentes (implementação: Windows, Linux, macOS). A mesma GUI funciona em qualquer plataforma.



Implementação

1. Definimos a interface de **implementação** — as operações primitivas comuns a todos os dispositivos:

```
public interface Dispositivo {  
    boolean estaLigado();  
    void ligar();  
    void desligar();  
    int getVolume();  
    void setVolume(int volume);  
}
```

2. Criamos as **implementações concretas**, uma para cada aparelho:

```
public class Tv implements Dispositivo {  
    private boolean ligado = false;  
    private int volume = 30;  
  
    public boolean estaLigado() { return ligado; }  
    public void ligar() { ligado = true; }  
    public void desligar() { ligado = false; }  
    public int getVolume() { return volume; }  
    public void setVolume(int volume) { this.volume = volume; }  
}  
// Radio segue o mesmo contrato
```

3. A **abstração** guarda a referência ao `Dispositivo` e **delega** todo o trabalho:

```
public abstract class ControleRemoto {  
    protected Dispositivo dispositivo;  
  
    public ControleRemoto(Dispositivo dispositivo) {  
        this.dispositivo = dispositivo;  
    }  
  
    public void ligarOuDesligar() {  
        dispositivo.estaLigado() ? dispositivo.desligar() : dispositivo.ligar();  
    }  
  
    public void aumentarVolume() {  
        dispositivo.setVolume(dispositivo.getVolume() + 10);  
    }  
}
```

4. **Abstrações refinadas** adicionam comportamento sem depender do aparelho:

```
public class ControleAvancado extends ControleRemoto {  
    public ControleAvancado(Dispositivo dispositivo) {  
        super(dispositivo);  
    }  
  
    public void silenciar() {  
        dispositivo.setVolume(0);  
    }  
}
```

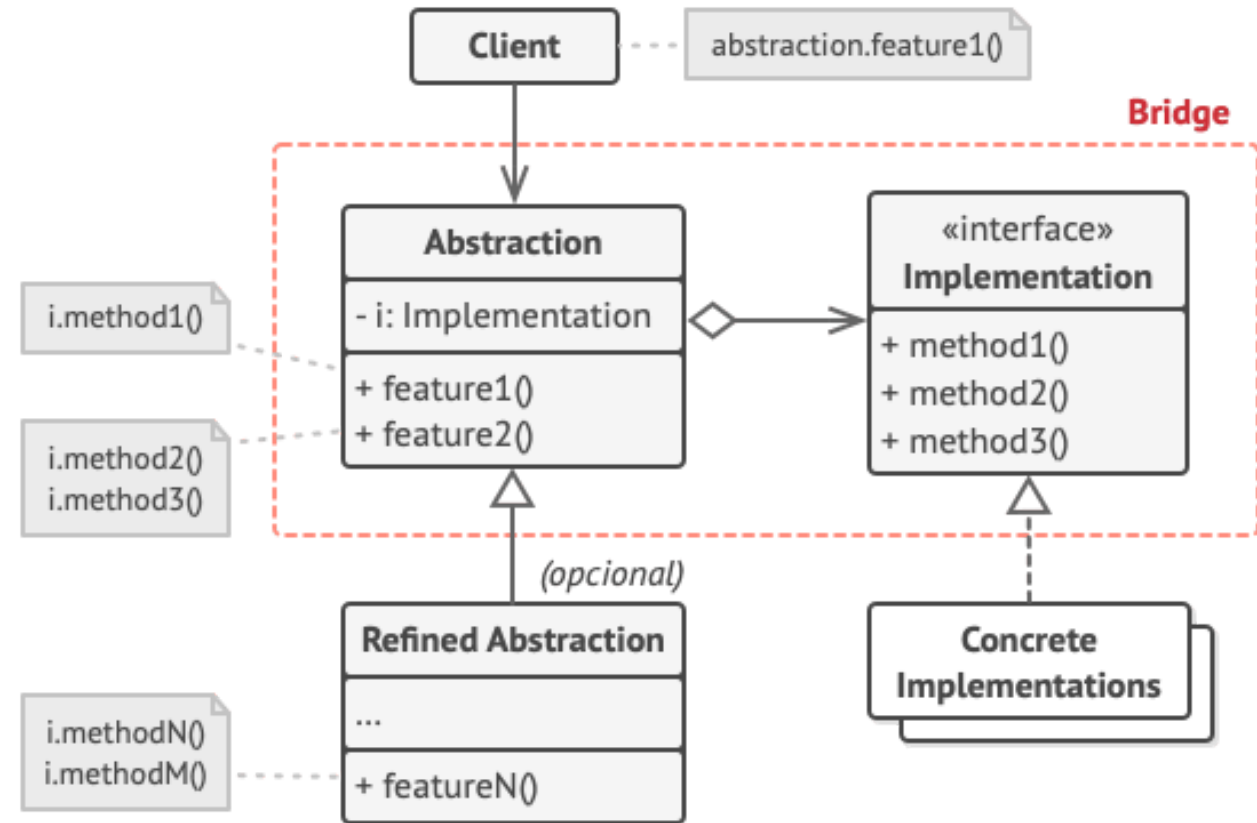
Utilização

O cliente **liga** uma abstração a uma implementação no construtor:

```
Dispositivo tv = new Tv();  
ControleRemoto controle = new ControleRemoto(tv);  
controle.ligarOuDesligar();  
controle.aumentarVolume();  
  
Dispositivo radio = new Radio();  
ControleAvancado avancado = new ControleAvancado(radio);  
avancado.silenciar();
```

- o mesmo controle funciona com **qualquer dispositivo**
- adicionar `Projetor` → nova implementação, controles **não mudam**
- adicionar `ControleComVoz` → nova abstração, dispositivos **não mudam**

Diagramas



- **Abstraction:** lógica de alto nível, com referência à Implementação
- **RefinedAbstraction:** variantes da abstração
- **Implementor:** interface das operações primitivas
- **ConcreteImplementor:** implementações
- **Client:** liga abstração + implementação

Analogias

- **Carro e motor:**
 - dirigir um carro não depende do motor (combustão ou elétrico)
 - o carro **delega** a força ao motor
 - carros (sedã, esportivo) e motores evoluem **independentes**
- Trocar o motor não muda a direção; trocar o carro não muda o motor

Aplicações

- Evitar a **explosão de subclasses** quando há duas (ou mais) dimensões independentes
- Dividir uma **classe monolítica** com várias variantes de plataforma
- **Trocar a implementação em tempo de execução** (ex.: driver de banco)

Exemplos típicos:

- GUI multiplataforma (Windows/Linux/macOS)
- acesso a bancos (MySQL/PostgreSQL) com a mesma lógica de repositório
- envio de mensagens por canais diferentes (SMS/Email/Push)

Prós

- **Independência de plataforma:** classes desacopladas do detalhe concreto
- **OCP:** novas abstrações e implementações entram sem se tocarem
- **SRP:** alto nível (abstração) e detalhe (implementação) separados
- Permite **trocar a implementação em runtime**

Contras

- Pode **complicar** o código se aplicado sem necessidade (classe coesa simples)
- Exige pensar em **duas hierarquias** desde o início
- Não confundir com Strategy: o Bridge separa **abstração de plataforma**, não só um algoritmo

Regra: use quando a variação acontece em **mais de uma dimensão**.

Resumo

- Bridge: divide uma classe em **abstração** e **implementação** (composição)
- Cada hierarquia **evolui independente**: sem explosão de subclasses
- Ideal para **duas dimensões** de variação (tipo × plataforma)
- Diferente do Adapter: Bridge é **projetado antes**; Adapter adapta **código existente**