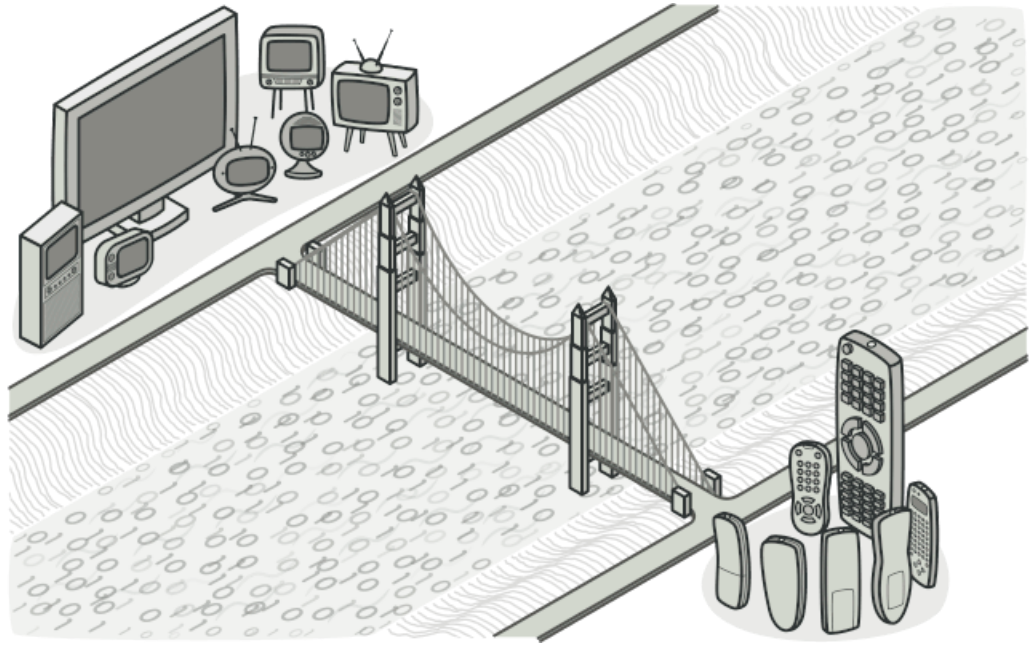
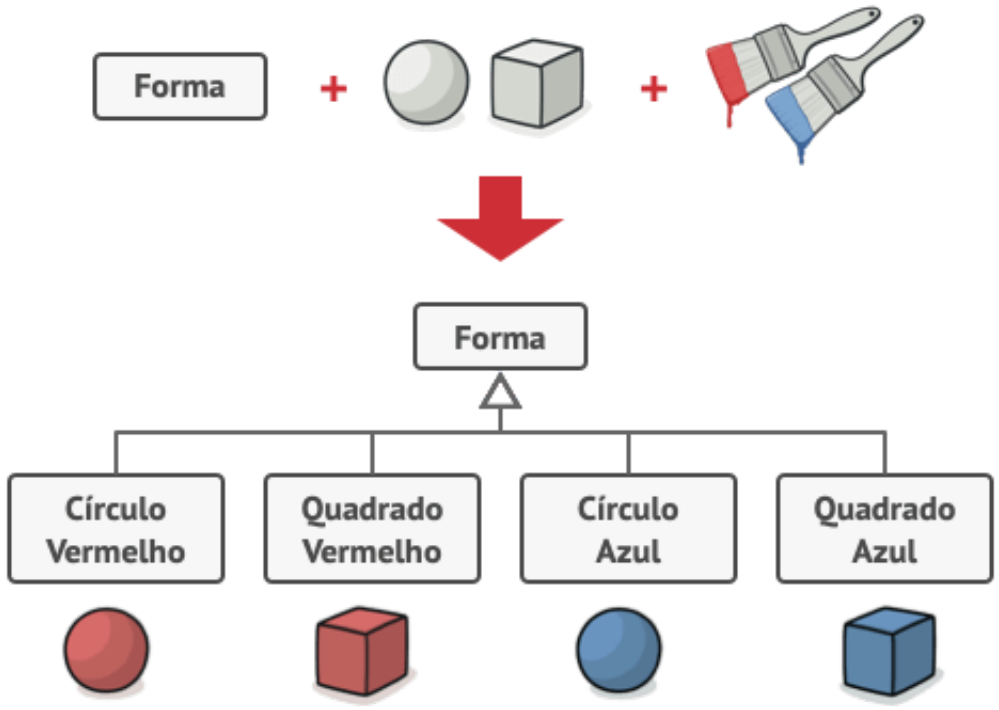
TÓPICO 10 - BRIDGE

Design Patterns - Professor Ramon Venson - SATC 2026.1

Motivação

- Alguns problemas variam em duas dimensões
- Herança pura multiplica subclasses
- O código cresce mais que o necessário
- Precisamos separar responsabilidades
- Bridge reduz combinações desnecessárias





Problema

- Notificação simples ou urgente
- Envio por email, SMS ou push
- Herança gera muitas combinações
- Cada novo eixo multiplica classes
- Manutenção vira custo alto

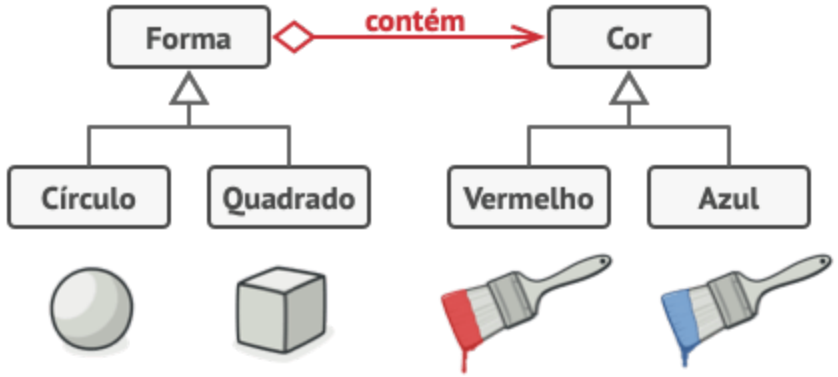
Conceito principal

- **Bridge** é um padrão estrutural
- Ele separa duas hierarquias relacionadas
- Uma hierarquia representa a abstração
- A outra representa a implementação
- Ambas evoluem de forma independente

Como funciona

- A abstração mantém uma referência
- Essa referência aponta para a implementação
- Parte do trabalho é delegada
- O cliente usa a abstração principal
- A implementação pode ser trocada depois

Estrutura / Componentes



Papéis no padrão

- **Abstraction** define alto nível
- **RefinedAbstraction** especializa comportamentos
- **Implementor** define operações básicas
- **ConcreteImplementor** realiza detalhes concretos
- O cliente combina os dois lados

Exemplo conceitual

- Sistema de avisos acadêmicos
- Tipos: simples e urgente
- Canais: email e push
- Não queremos uma classe por combinação
- Bridge conecta tipo e canal

Código: implementação

```
interface CanalEnvio {
    void enviarMensagem(String titulo, String conteudo);
}

class EmailCanal implements CanalEnvio {
    public void enviarMensagem(String titulo, String conteudo) {
        System.out.println("Email: " + titulo);
    }
}
```

Código: abstração

```
abstract class Notificacao {
    protected CanalEnvio canal;

    public Notificacao(CanalEnvio canal) {
        this.canal = canal;
    }

    public abstract void enviar(String titulo, String conteudo);
}
```

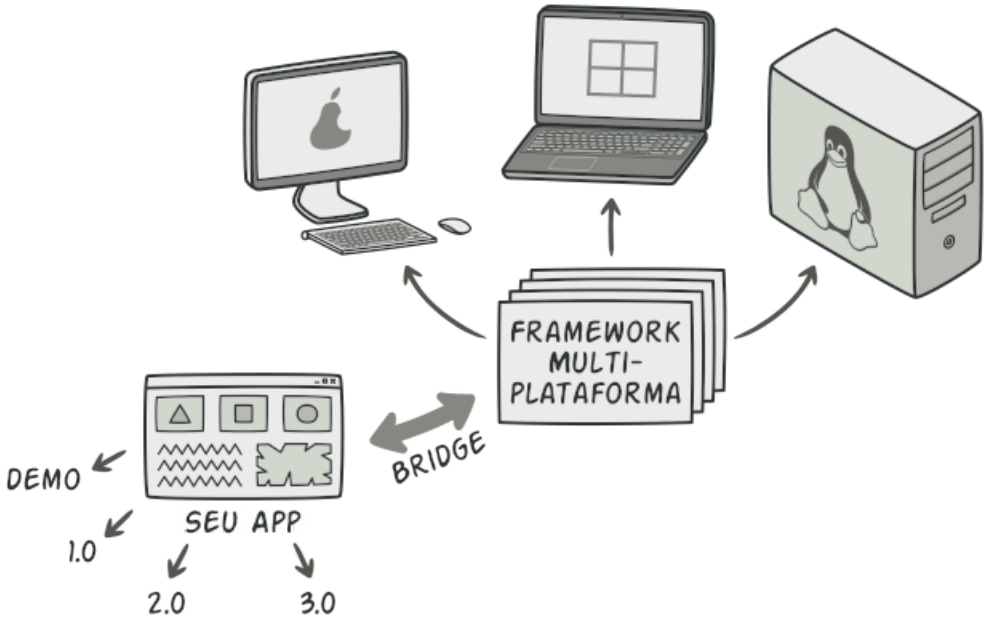
Código: abstração refinada

```
class NotificacaoSimples extends Notificacao {
    public NotificacaoSimples(CanalEnvio canal) {
        super(canal);
    }

    public void enviar(String titulo, String conteudo) {
        canal.enviarMensagem(titulo, conteudo);
    }
}
```

Analogia

- Controle remoto e dispositivo
- O controle nao precisa conhecer detalhes internos
- A TV ou radio seguem contrato comum
- Podemos trocar um lado sem quebrar o outro



Quando usar

- Ha duas dimensoes ortogonais
- A hierarquia cresce por combinacoes
- Implementacoes mudam com frequencia
- Queremos trocar plataforma em runtime
- Dominio e infraestrutura devem evoluir separados

Quando nao usar

- O problema tem uma so variacao real
- A modelagem ficaria mais complexa
- Nao ha necessidade de independencia
- O codigo atual ainda e simples
- Composicao adicional nao traz ganho real

Vantagens

- Evita explosao de subclasses
- Diminui acoplamento entre eixos
- Facilita extensao independente
- Melhora modularidade
- Permite trocar implementacoes facilmente

Desvantagens

- Aumenta numero de classes
- Exige modelagem mais cuidadosa
- Pode confundir iniciantes
- Nao vale para casos triviais
- Estrutura fica mais abstrata

Relações com outros padrões

- **Adapter** integra interfaces incompatíveis
- **Strategy** troca algoritmos
- **Decorator** adiciona comportamento
- **Abstract Factory** pode criar implementações
- **Bridge** separa eixos de evolução

Exemplo prático

```
AvisoAcademico aviso = new AvisoUrgente(new PushNotificacao());
aviso.disparar();
```

- `AvisoUrgente` varia o tipo
- `PushNotificacao` varia o canal
- A combinacao acontece por composicao

Resumo final

- Bridge separa abstração e implementação
- Resolve explosão de subclasses
- Usa composição em vez de herança excessiva
- Funciona bem com duas dimensões independentes
- Ajuda a manter o sistema flexível

Referência

- Refactoring.Guru - Bridge
- https://refactoring.guru/pt-br/design-patterns/bridge