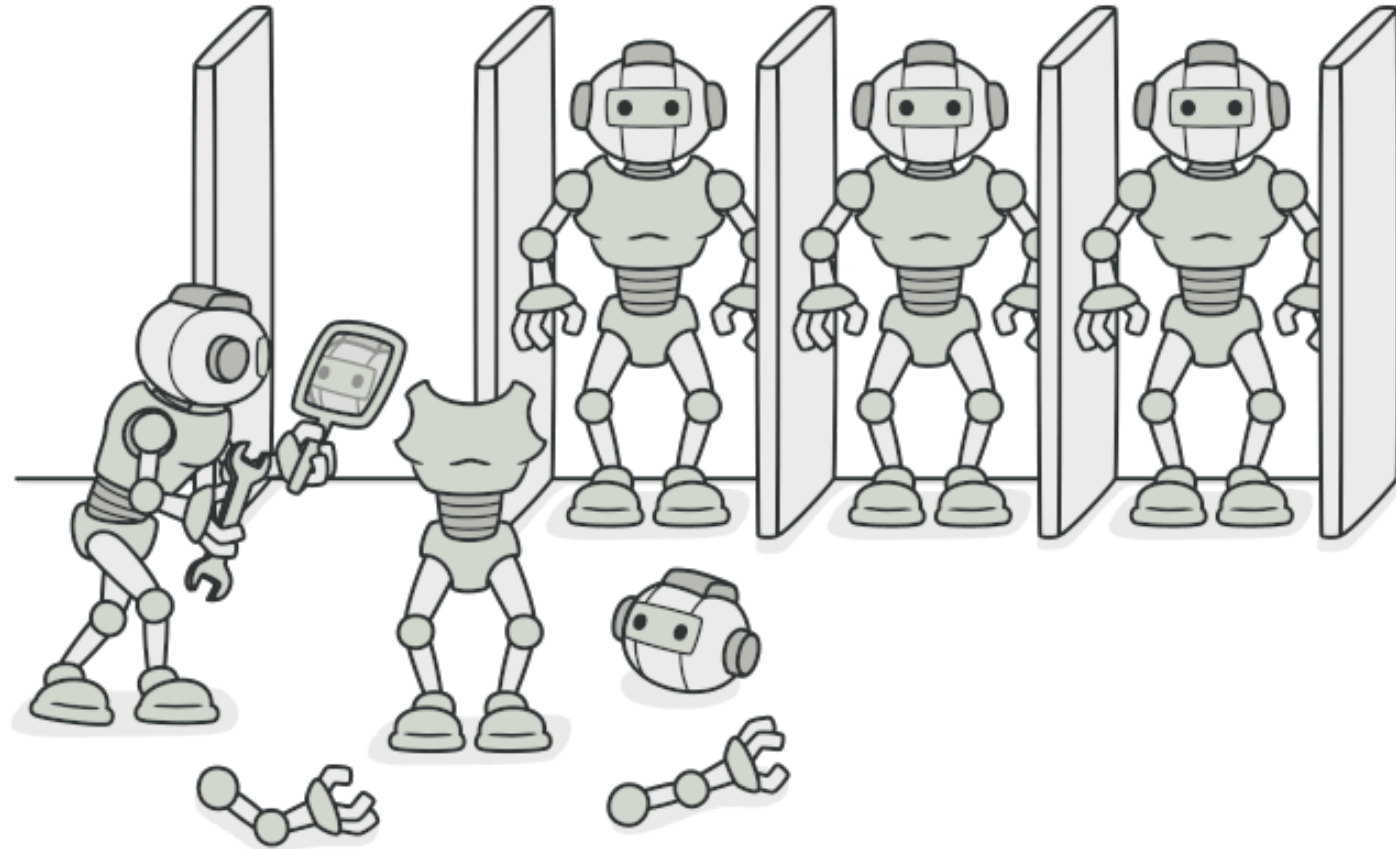


TÓPICO 05: PROTOTYPE

Design Patterns - Professor Ramon Venson - SATC 2026.2

Definição

Prototype é um padrão de projeto **criacional** que permite criar novos objetos **copiando** um **objeto existente** em vez de instanciá-lo do zero.



Problema

Alguns objetos exigem **muita configuração** e nascem de uma **base parecida**. Recriar cada variação do zero tende a levar a:

- repetição de código de construção
- erros ao esquecer um parâmetro
- impossibilidade de copiar atributos privados



Neste exemplo, cada fase precisa de um inimigo com configuração parecida, mas o código repete a construção:

```
public class Fase {  
    private void criarInimigos() {  
        Inimigo guerreiroA = new Inimigo("Guerreiro", 100, new Espada());  
        Inimigo guerreiroB = new Inimigo("Guerreiro", 100, new Espada());  
        Inimigo guerreiroC = new Inimigo("Guerreiro", 100, new Espada());  
        // ...  
    }  
}
```

Uma das soluções sugere **copiar campo a campo** um objeto existente:

```
public class Fase {  
    private Inimigo criarCopia(Inimigo antigo) {  
        Inimigo copia = new Inimigo();  
        copia.setTipo(antigo.getTipo());  
        copia.setVida(antigo.getVida());  
        copia.setArma(antigo.getArma());  
        return copia;  
    }  
}
```

Porém isso não é o ideal, pois o cliente precisa conhecer a classe concreta, acessar **getters e setters** e expor o **estado interno** do objeto. Qualquer novo campo exige alterar a cópia manualmente.

Solução

Vamos fazer com que **o próprio objeto saiba como se clonar**, separando a criação da cópia do código do cliente.

- **Contrato de clonagem:** o objeto expõe um método `clone()`
- **O objeto cria a nova instância:** copia seu próprio estado
- **O cliente ajusta pequenas variações:** muda só o que precisa
- **Um modelo pronto vira protótipo:** serve de base para novas cópias

Implementação

1. Vamos declarar um **construtor** que recebe outro objeto da mesma classe e copia seus atributos:

```
class Inimigo {  
    private String tipo;  
    private Double vida;  
  
    // Construtor padrão  
    public Inimigo() {}  
  
    // Construtor de cópia  
    public Inimigo(Inimigo base) {  
        this.tipo = base.tipo;  
        this.vida = base.vida;  
    }  
}
```

2. Em seguida, vamos declarar um **método de clonagem** que cria uma nova instância usando o construtor de cópia:

```
public class Inimigo {  
    // ...  
    public Inimigo clonar() {  
        return new Inimigo(this);  
    }  
}
```

Utilização

O cliente pede uma **cópia** de um protótipo pronto e ajusta apenas o necessário:

```
Inimigo base = new Inimigo("Guerreiro", 100);  
Inimigo copia = base.clonar();  
copia.setVida(150);
```

- `base` funciona como **modelo** (protótipo)
- `copia` nasce rápido e consistente
- O cliente muda só o que precisa

Contrato de Clonagem

Uma interface pode declarar o contrato de clonagem, permitindo que diferentes classes implementem a cópia:

```
public interface InimigoPrototype {  
    InimigoPrototype clonar();  
}  
  
public class Inimigo implements InimigoPrototype {  
    // ...  
    @Override  
    public InimigoPrototype clonar() {  
        return new Inimigo(this);  
    }  
}
```

Registro de protótipos

O **Registro** é um mecanismo opcional que **guarda protótipos frequentes** em um mapa, evitando que o cliente conheça as classes concretas:

```
Map<String, InimigoPrototype> prototipos = new HashMap<>();  
prototipos.put("guerreiro", base);  
  
Inimigo elite = (Inimigo) prototipos.get("guerreiro").clonar();
```

- Bom para modelos frequentes
- Muito útil em sistemas de jogos e acadêmicos
- O cliente pede a cópia **pelo nome**, não pela classe

Cópia profunda

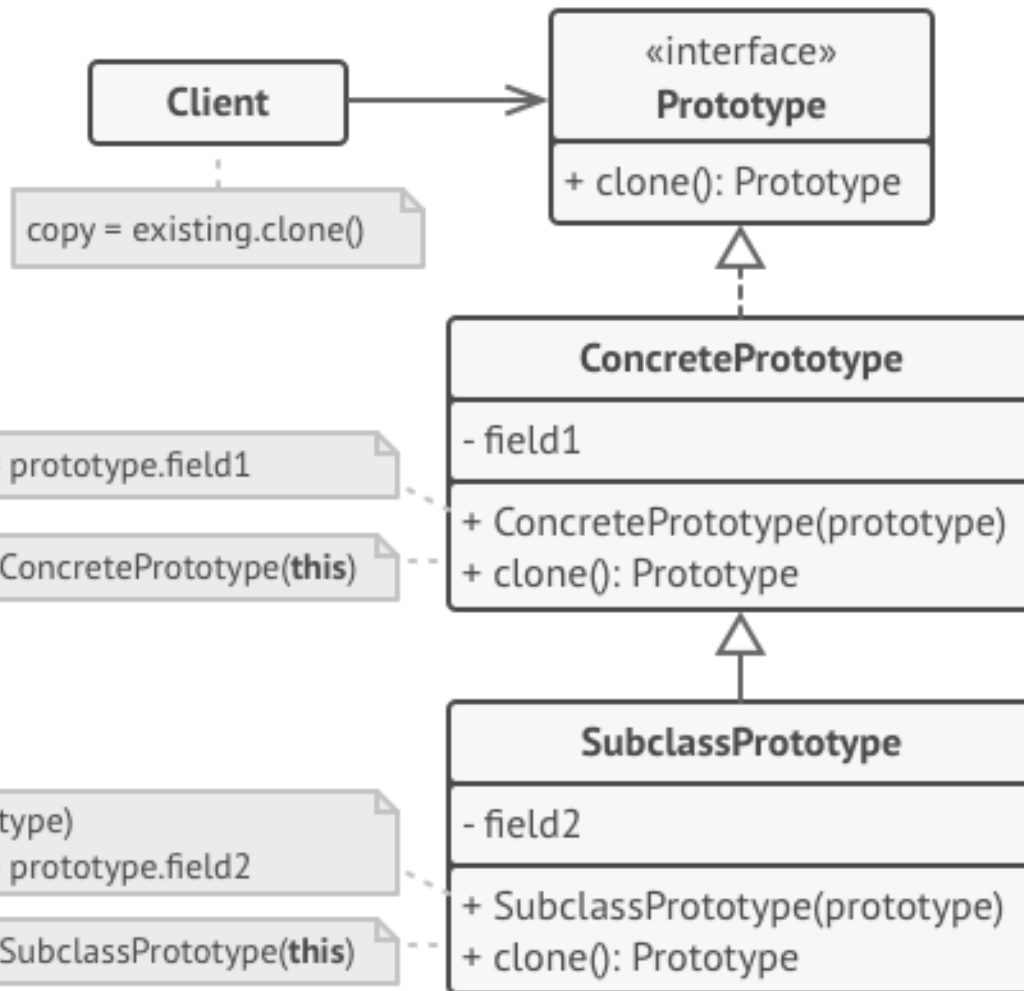
O construtor de cópia faz uma **cópia rasa**: o clone **compartilha** os objetos internos (como a `Arma`). Para estados mutáveis, copiamos também os **objetos internos**, evitando efeitos colaterais:

```
public Inimigo(Inimigo outro) {  
    this.tipo = outro.tipo;  
    this.vida = outro.vida;  
    this.arma = arma.clone(); // cópia profunda  
}
```

Diagramas

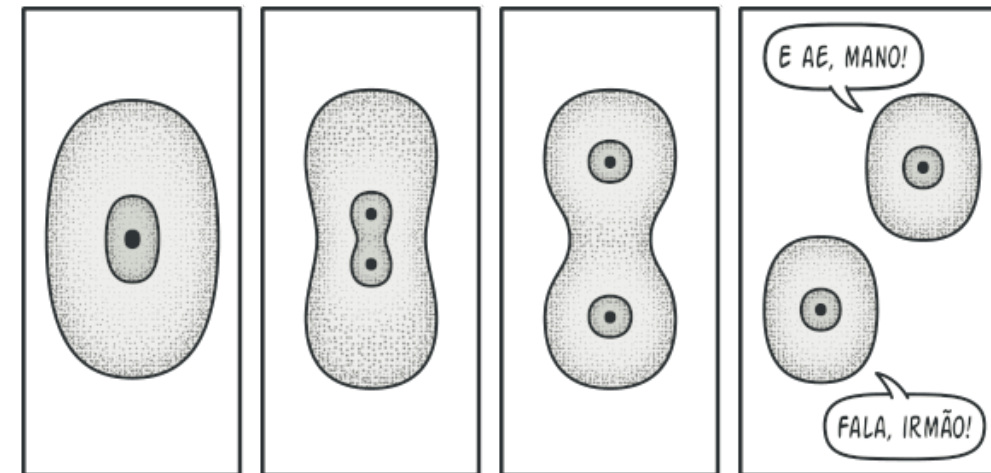
A estrutura envolve:

- **Prototype**: contrato de clonagem
- **Concrete Prototype**: implementa a cópia
- **Client**: usa clones via abstração
- **Registro** (opcional): guarda protótipos frequentes



Analogias

- **Mitose:**
 - Uma célula origina outra **semelhante**
 - O modelo participa da criação da cópia
 - A nova versão nasce **pronta para uso**
 - A ideia é copiar, não reconstruir



Aplicações

- Muitos objetos **parecidos entre si**
- Criação **longa ou custosa**
- Muitas **configurações pré-definidas**
- Cliente não deve conhecer classes concretas
- Variações partem de uma **base comum**

Exemplos típicos:

- Personagens/armas em jogos
- Documentos a partir de um template
- Configurações de objetos prontas

Prós

- **Reduz repetição** de configuração
- **Diminui acoplamento** com classes concretas
- Facilita criar **variações** rapidamente
- **Reaproveita** objetos já prontos
- Pode melhorar o **desempenho**

Contras

- Clonagem correta pode ser **difícil**
- **Referências circulares** complicam muito
- **Cópia rasa x profunda** exige cuidado
- Mais responsabilidade nas **classes**
- Pode virar **overengineering**

Regra: evite para objetos simples, quando construtores já resolvem bem.

Resumo

- Prototype cria por **cópia**, não por reconstrução
- O objeto conhece a própria **clonagem**
- Ideal para **variações** sobre uma base comum
- O **Registro** evita conhecer classes concretas
- Exige cuidado com **estado mutável** e cópia profunda