# Padrões de Criação (Creational Patterns)

## Introdução

Os **padrões de criação** focam em **como objetos são instanciados**, oferecendo mecanismos flexíveis e reutilizáveis para a construção de sistemas complexos. Abaixo está uma explicação detalhada dos principais padrões, com exemplos e aplicações:

## 1 Padrões de Criação

### 1.1 Factory Method (Método de Fábrica)

- **Intenção**: Delegar a criação de objetos para subclasses, permitindo que elas decidam qual classe instanciar.
- **Estrutura**:
  - Uma classe abstrata define um método de fábrica (`factoryMethod()`), que subclasses concretas implementam para criar objetos específicos.
- **Exemplo**:

```
interface Arvore { void regar(); }
class Figueira implements Arvore { ... }
class Viveiro {
    public static Arvore factory(String tipo) {
        if (tipo.equals("Figueira")) return new Figueira();
        // ...
    }
}
```

- **Quando usar**:
  - Quando a classe não pode antecipar o tipo de objeto a ser criado.
  - Para promover baixo acoplamento entre classes (princípio GRASP *Low Coupling*).
- **Vantagens**: Flexibilidade, extensibilidade e separação de responsabilidades.
- **Armadilha**: Pode introduzir muitas subclasses se mal planejado.

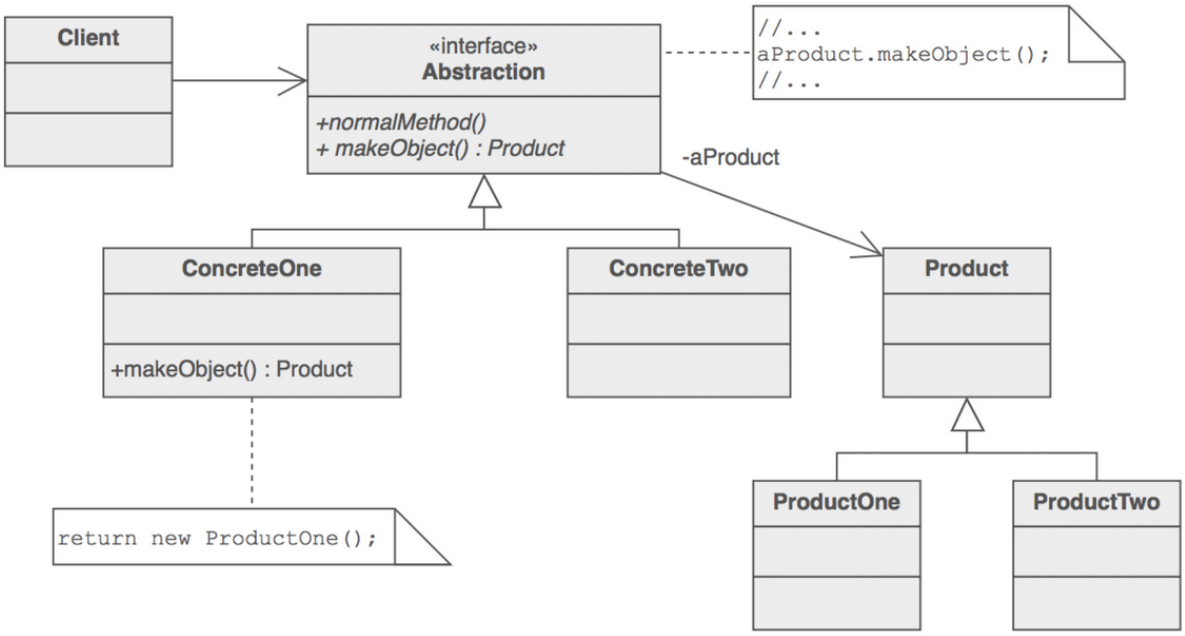


Figura 1: Diagrama UML do padrão Factory Method

## 1.2 Abstract Factory (Fábrica Abstrata)

- **Intenção**: Criar famílias de objetos relacionados sem especificar suas classes concretas.
- **Estrutura**:
  - Uma interface (`AbstractFactory`) define métodos para criar múltiplos produtos (ex.: `createWindow()`, `createButton()`).
  - Fábricas concretas (`MacOSXWidgetFactory`, `MsWindowsWidgetFactory`) implementam esses métodos.
- **Exemplo**:

```
interface AbstractWidgetFactory { Window createWindow(); }
class MacOSXWidgetFactory implements AbstractWidgetFactory {
    public Window createWindow() { return new MacOSXWindow(); }
}
```

- **Quando usar**:
  - Sistemas que precisam ser independentes de plataforma (ex.: GUI, conexões de banco de dados).
  - Para isolar a criação de objetos complexos.
- **Vantagens**: Troca fácil de famílias de objetos; código cliente desacoplado.
- **Armadilha**: Complexidade aumentada com muitas interfaces.

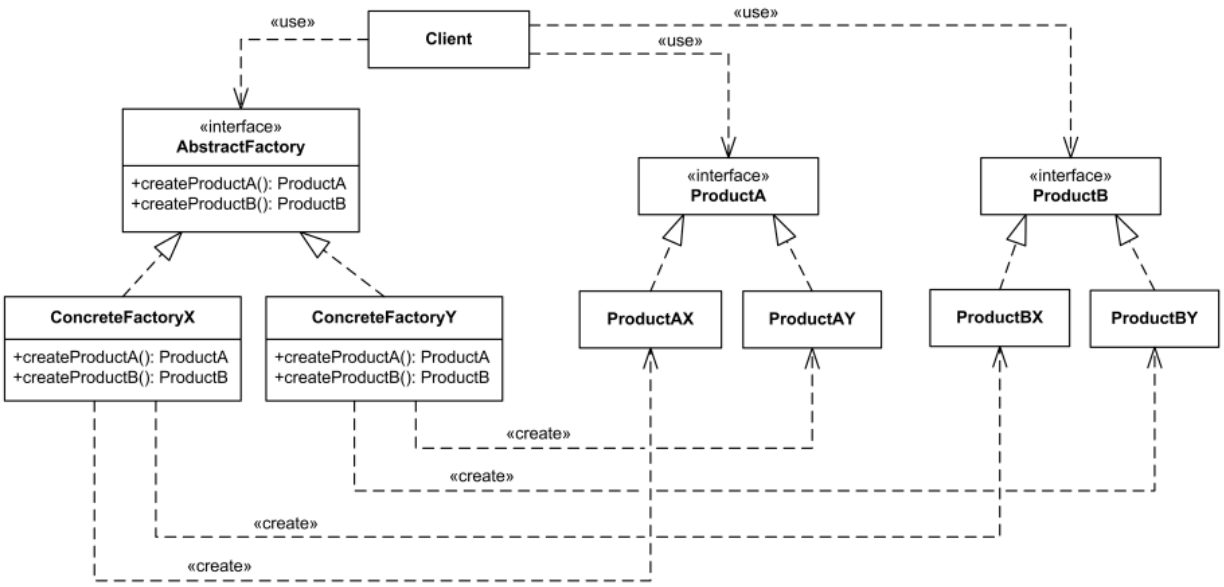


Figura 2: Diagrama UML do padrão Abstract Factory

## 1.3 Builder (Construtor)

- **Intenção**: Separar a construção de um objeto complexo de sua representação, permitindo o mesmo processo de construção criar diferentes representações.
- **Estrutura**:
  - `Director` (ex.: `Waiter`) coordena a construção usando um `Builder` (ex.: `PizzaBuilder`).
  - `ConcreteBuilder` (ex.: `HawaiianPizzaBuilder`) implementa etapas específicas.
- **Exemplo**:

```
class PizzaBuilder {
    protected Pizza pizza = new Pizza();
    public void buildDough() { /* ... */ }
    public Pizza getPizza() { return pizza; }
}
class Waiter {
    public Pizza constructPizza(PizzaBuilder builder) {
        builder.buildDough();
        builder.buildSauce();
        return builder.getPizza();
    }
}
```

- **Quando usar**:
  - Quando um objeto requer muitos parâmetros opcionais (ex.: `NutritionFacts` com Builder interno).
  - Para construir representações diferentes do mesmo objeto.
- **Vantagens**: Controle granular sobre o processo de construção; evita construtores telescópicos.
- **Armadilha**: Sobrecarga de código para objetos simples.

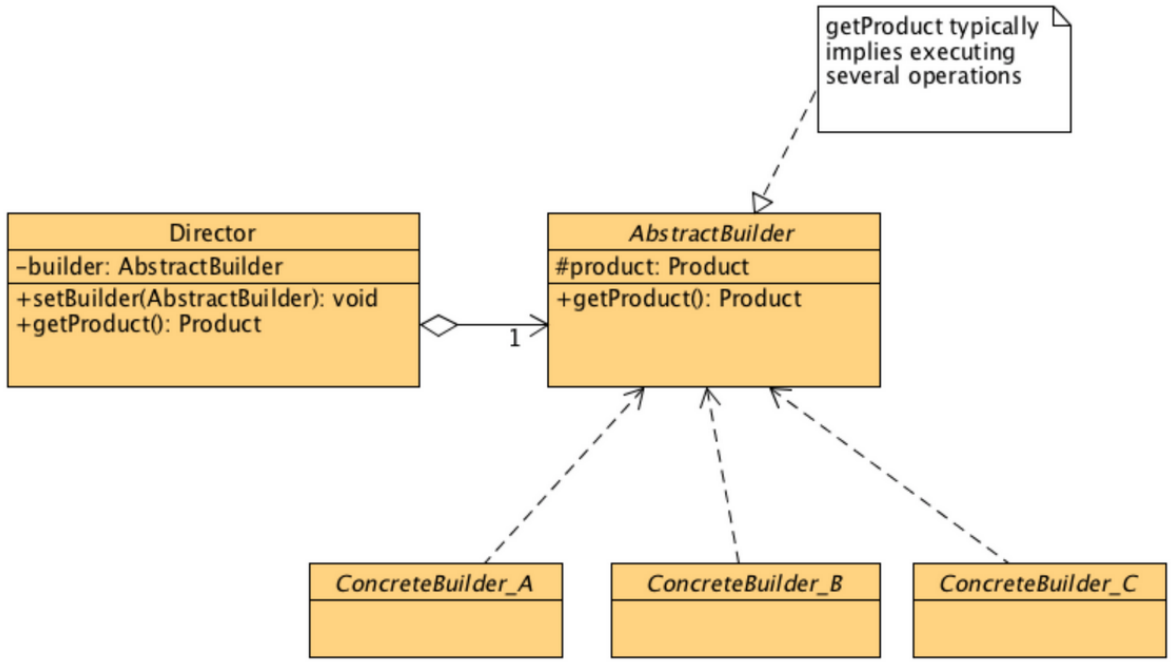


Figura 3: Diagrama UML do padrão Builder

## 1.4 Singleton (Única Instância)

- **Intenção**: Garantir que uma classe tenha apenas uma instância e fornecer um ponto global de acesso a ela.
- **Estrutura**:
  - Construtor privado e método estático `getInstance()` que retorna a instância única.
- **Exemplo**:

```
class Singleton {
    private static Singleton instance;
    private Singleton() { ... }
    public static Singleton getInstance() {
        if (instance == null) instance = new Singleton();
        return instance;
    }
}
```

- **Quando usar**:
  - Recursos compartilhados (ex.: conexão de banco de dados, configurações globais).
- **Vantagens**: Acesso controlado a recursos críticos.
- **Armadilha**: Dificulta testes unitários; pode violar o princípio *Single Responsibility*.

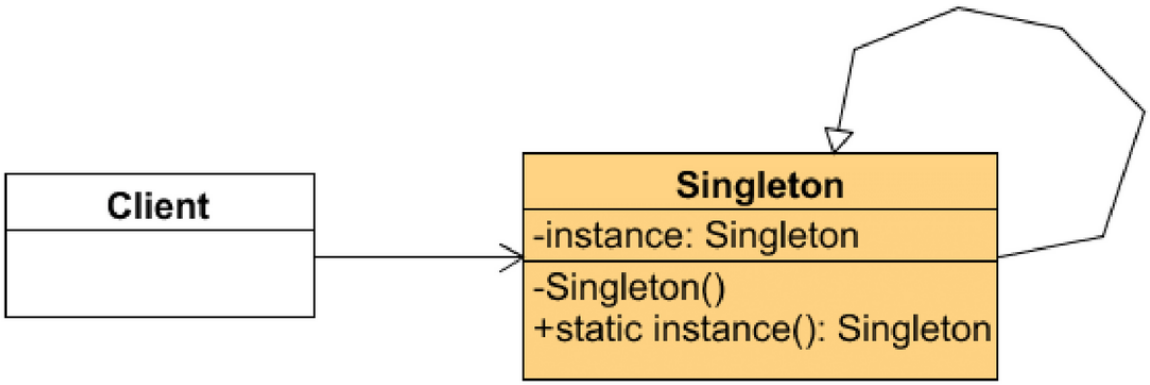


Figura 4: Diagrama UML do padrão Singleton

## 1.5 Object Pool (Pool de Objetos)

- **Intenção**: Reutilizar objetos caros para melhorar desempenho (ex.: conexões de rede, threads).
- **Estrutura**:
  - Um `Pool` gerencia uma coleção de objetos pré-inicializados (`acquire()`, `release()`).
- **Exemplo**:

```
public class AbstractPool {
    private PooledObject[] freeObjects;
    public PooledObject newObject() {
        if (freeObjects.isEmpty()) return factory.create();
        else return freeObjects.remove();
    }
}
```

- **Quando usar**:
  - Quando a criação de objetos é custosa e o uso é frequente.
- **Vantagens**: Redução de overhead de criação/coleta de lixo.
- **Armadilha**: Gerenciamento complexo de estados; não recomendado para objetos leves.

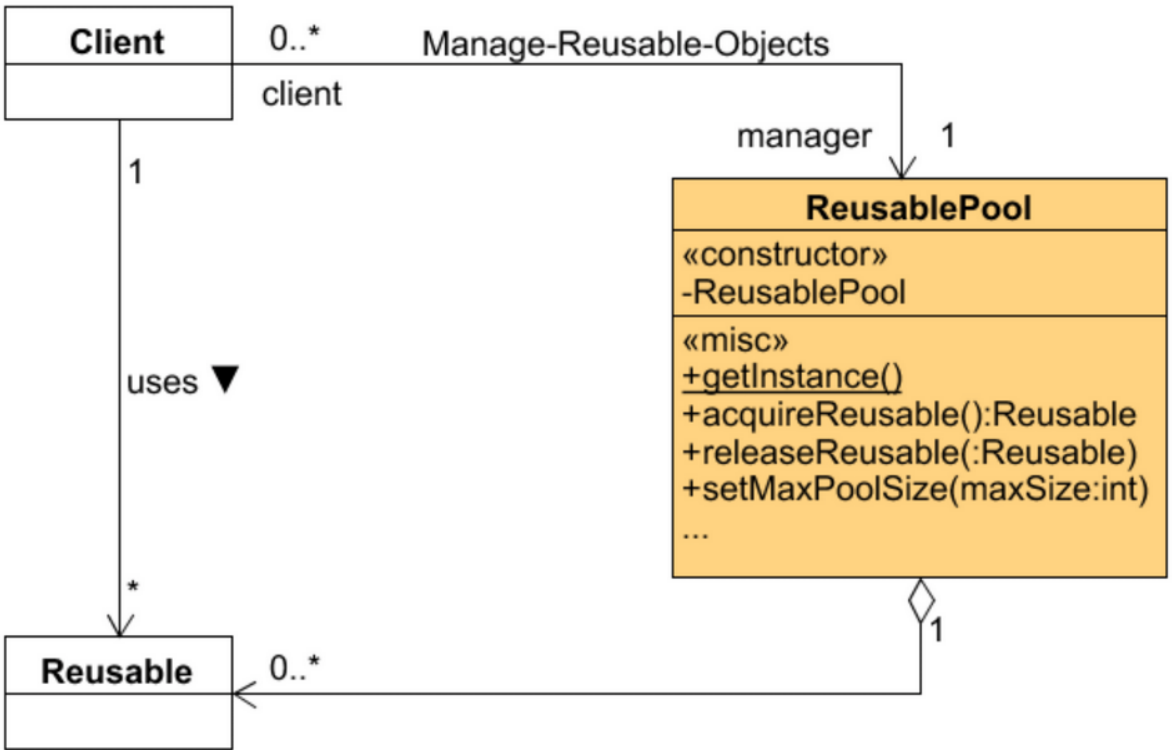


Figura 5: Diagrama UML do padrão Object Pool

## 1.6 Prototype (Protótipo)

- **Intenção**: Criar novos objetos copiando um protótipo existente.
- **Estrutura**:
  - Classes implementam `clone()` para permitir cópia de objetos.
  - Um registro (`PrototypeFactory`) armazena protótipos pré-definidos.
- **Exemplo**:

```
interface PrototypeCapable extends Cloneable { PrototypeCapable clone(); }
class Album implements PrototypeCapable {
    public Album clone() { return (Album) super.clone(); }
}
```

- **Quando usar**:
  - Quando a criação de um objeto é mais eficiente por cópia do que por inicialização do zero.
- **Vantagens**: Evita repetição de inicialização complexa.
- **Armadilha**: Dificuldades com objetos que possuem referências complexas.

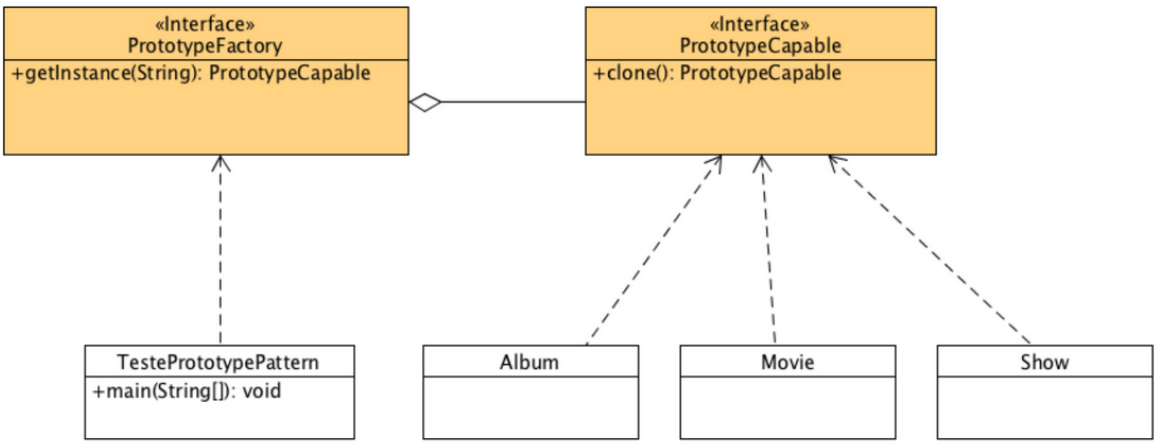


Figura 6: Diagrama UML do padrão Prototype

## 2 Conclusão

Os padrões de criação resolvem problemas comuns de instanciação de objetos, promovendo **flexibilidade**, **reuso** e **desacoplamento**. Escolha o padrão conforme o contexto:

- Use `Factory Method` para delegação simples.
- Opte por `Abstract Factory` para famílias de objetos.
- Prefira `Builder` para objetos complexos com múltiplas configurações.
- Aplique `Singleton` com cautela para recursos globais.
- Utilize `Object Pool` para otimização de desempenho.
- Recorra a `Prototype` para evitar inicializações custosas.