Padrões de Criação

Douglas Iracet
Jefferson Frigo

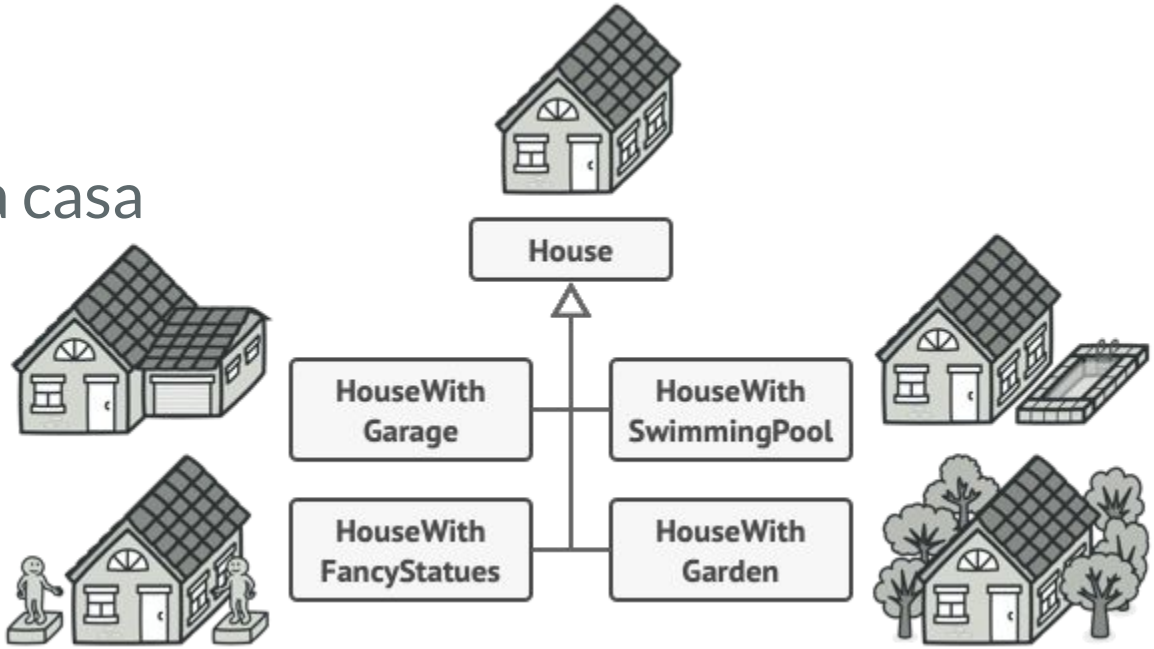
Introdução

- O que são Padrões
- Para que servem
- Padrões de Criação

Creational Patterns

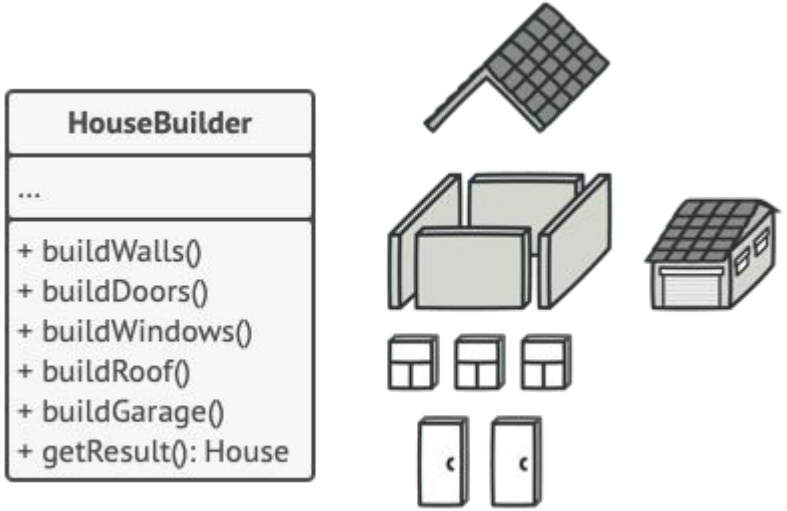
Builder: Introdução e Problema

- Propósito
- Objeto Complexo
- Como construir uma casa
- Analogia



Builder: Solução

- Descentralizar o construtor
- Parametrizar
- Componentes:
  - Builders
  - Diretor



Builder: Exemplo

- Construção
- Etapas da construção

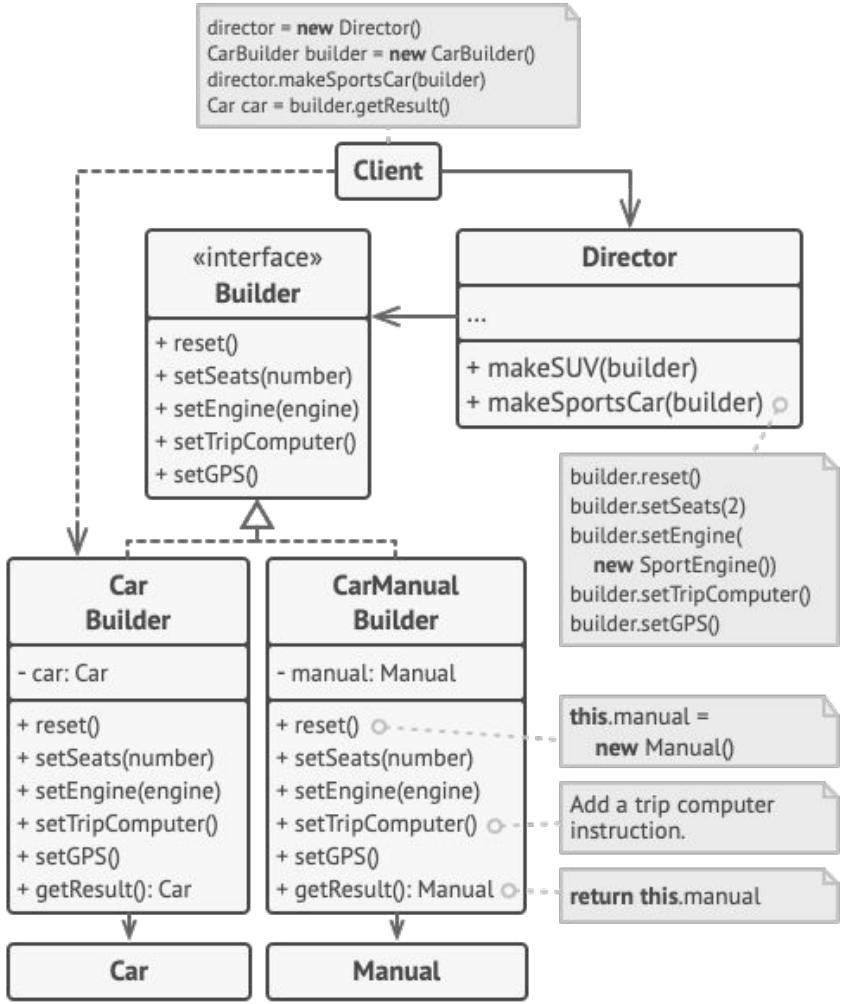
```
class MazeBuilder {
public:
    virtual void BuildMaze() { }
    virtual void BuildRoom(int room) { }
    virtual void BuildDoor(int roomFrom, int roomTo) { }

    virtual Maze* GetMaze() { return 0; }
protected:
    MazeBuilder();
};

Maze* MazeGame::CreateMaze (MazeBuilder& builder) {
    builder.BuildMaze();

    builder.BuildRoom(1);
    builder.BuildRoom(2);
    builder.BuildDoor(1, 2);

    return builder.GetMaze();
}
```



Builder: Vantagens e Desvantagens

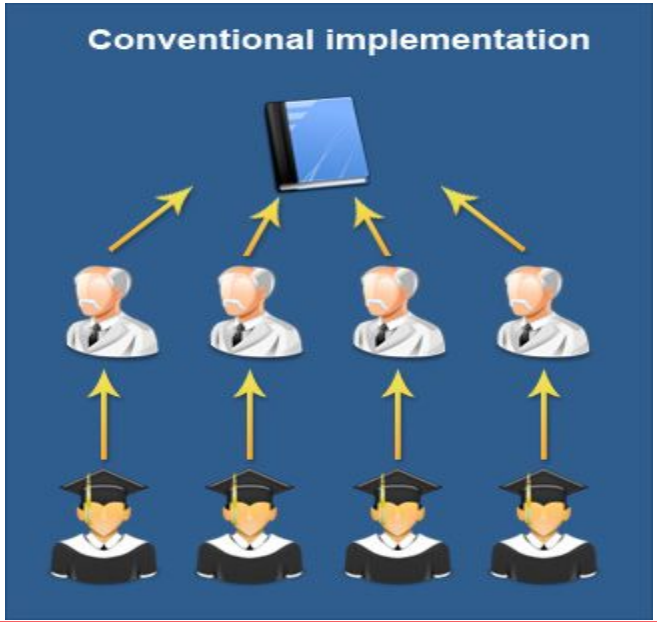
- Vantagens:
  - É possível construir objetos passo a passo
  - Reutilizar o Construtor
  - Princípio da responsabilidade única
- Desvantagens:
  - Aumento de complexidade

Builder: Quando usar e Conclusão

- Usado para objetos complexos
- Diferentes representações do mesmo tipo
- Recapitulando:
  - Criação em etapas
  - Aumento de complexidade, em prol da organização e flexibilidade

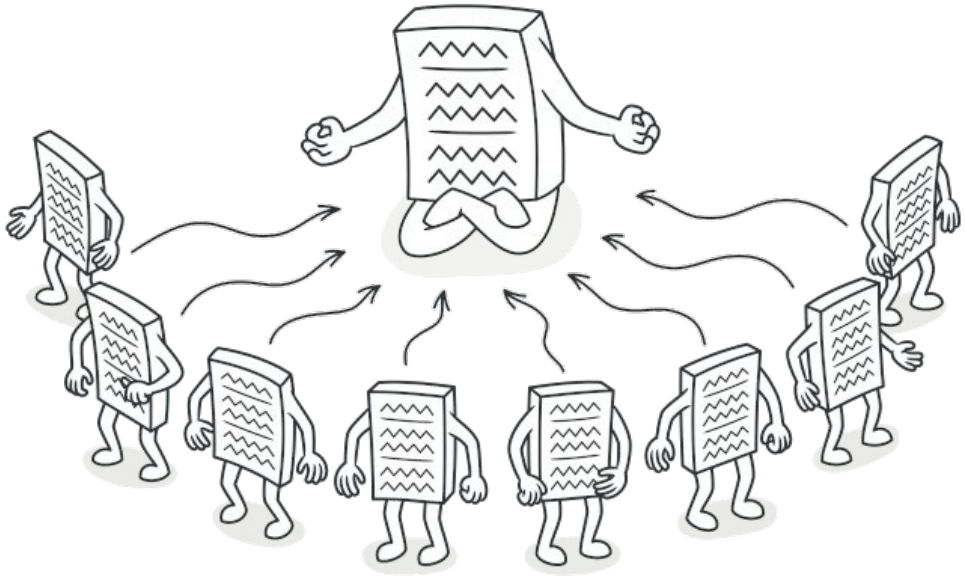
Singleton: Introdução e Problema

- Propósito
- Problema: As instâncias de um objeto



Singleton: Solução

- Apenas uma instância
- Método estático
- Construtor Privado



Singleton: Exemplo

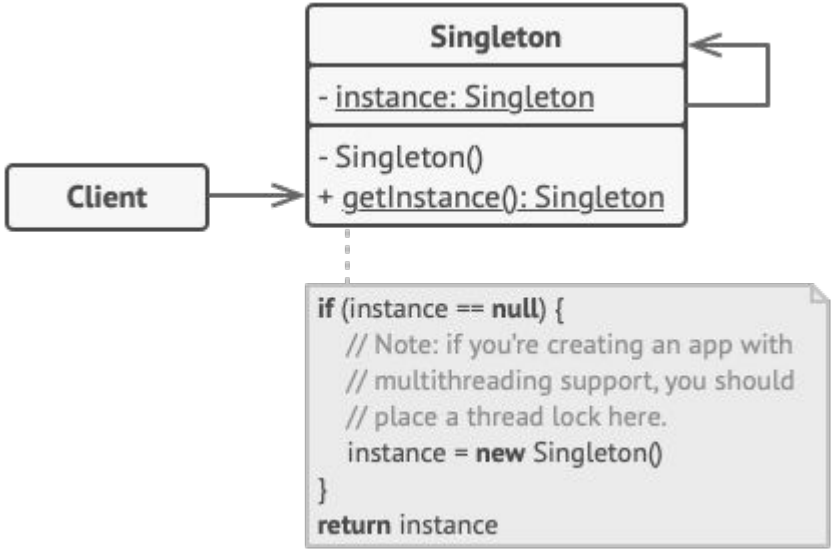
java Copy code

```java
public class Logger {
    private static Logger instance;

    private Logger() {}

    public static Logger getInstance() {
        if (instance == null) {
            instance = new Logger();
        }
        return instance;
    }

    public void log(String message) {
        System.out.println("Log: " + message);
    }
}
```



Singleton: Vantagens e Desvantagens

- Vantagens
  - Garante que uma classe tenha apenas uma instância
    Ponto global de acesso
  - Evita a criação desnecessária de instâncias
- Desvantagens
  - Acoplamento Global
  - Pode violar o princípio de responsabilidade única
  - Pode ser difícil de estender e testar

Singleton: Quando usar e Conclusão

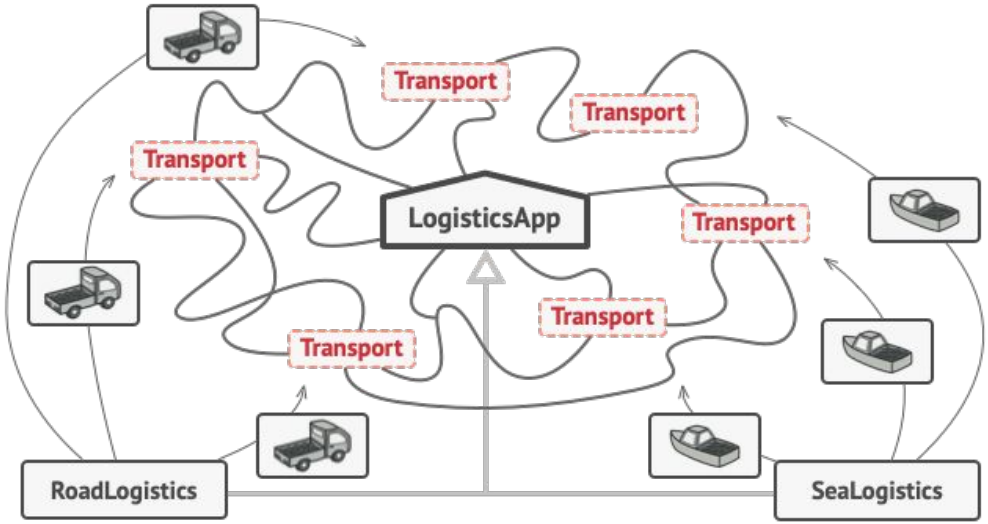
- Onde se faz necessário garantir uma única instância
- Compartilhamento de recursos
- Recapitulando:
    - Uma única instância
    - Construtor privado
    - Acesso via método público

Factory: Introdução e Problema

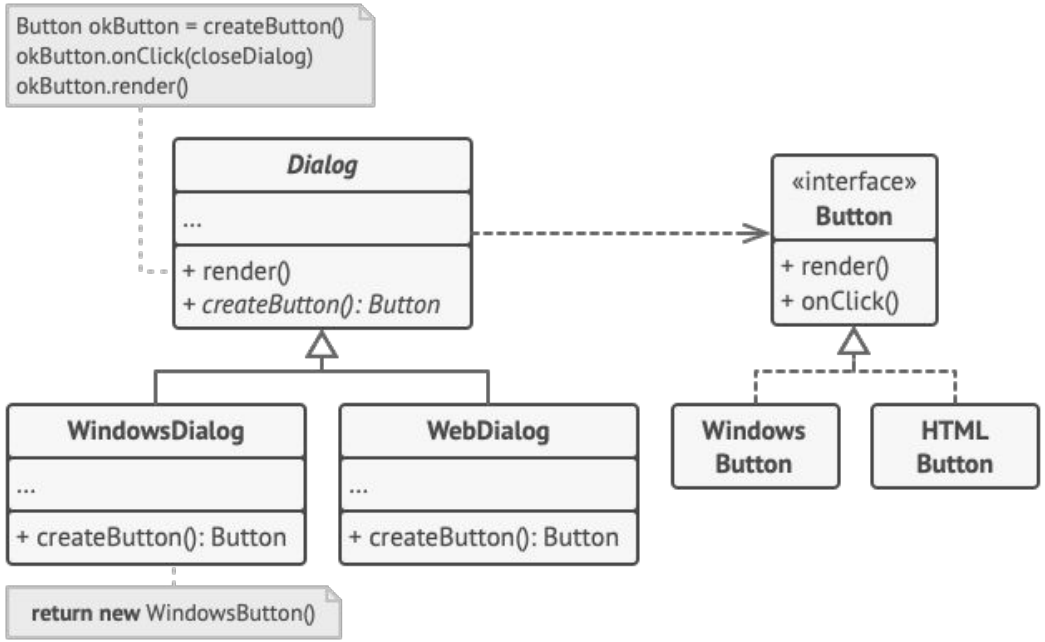
- Propósito
- Problema: Herança de criação

Factory: Solução

- Interface de criação
- Liberdade de criação das subclasses
- Componentes:
    - Criador
    - Criador Concreto
    - Produto
    - Produto Concreto



Factory: Exemplo



```
// The creator class declares the factory method that must
// return an object of a product class. The creator's subclasses
// usually provide the implementation of this method.
class Dialog is
    // The creator may also provide some default implementation
    // of the factory method.
    abstract method createButton():Button

    // Note that, despite its name, the creator's primary
    // responsibility isn't creating products. It usually
    // contains some core business logic that relies on product
    // objects returned by the factory method. Subclasses can
    // indirectly change that business logic by overriding the
    // factory method and returning a different type of product
    // from it.
    method render() is
        // Call the factory method to create a product object.
        Button okButton = createButton()
        // Now use the product.
        okButton.onClick(closeDialog)
        okButton.render()


// Concrete creators override the factory method to change the
// resulting product's type.
class WindowsDialog extends Dialog is
    method createButton():Button is
        return new WindowsButton()

class WebDialog extends Dialog is
    method createButton():Button is
        return new HTMLButton()


// The product interface declares the operations that all
// concrete products must implement.
interface Button is
    method render()
    method onClick(f)
```

```
// Concrete products provide various implementations of the
// product interface.
class WindowsButton implements Button is
    method render(a, b) is
        // Render a button in Windows style.
    method onClick(f) is
        // Bind a native OS click event.

class HTMLButton implements Button is
    method render(a, b) is
        // Return an HTML representation of a button.
    method onClick(f) is
        // Bind a web browser click event.


class Application is
    field dialog: Dialog

    // The application picks a creator's type depending on the
    // current configuration or environment settings.
    method initialize() is
        config = readApplicationConfigFile()

        if (config.OS == "Windows") then
            dialog = new WindowsDialog()
        else if (config.OS == "Web") then
            dialog = new WebDialog()
        else
            throw new Exception("Error! Unknown operating system.")

    // The client code works with an instance of a concrete
    // creator, albeit through its base interface. As long as
    // the client keeps working with the creator via the base
    // interface, you can pass it any creator's subclass.
    method main() is
        this.initialize()
        dialog.render()
```

Factory: Vantagens e Desvantagens

- Vantagens
  - Evita um acoplamento forte entre o criador e os produtos concretos
  - Princípio da responsabilidade única
  - Princípio Aberto/Fechado
- Desvantagens
  - Aumenta a complexidade e complicação do código ao adicionar novas subclasses

Factory: Quando usar e Conclusão

- Quando não se sabe exatamente as dependências e tipos
- Frameworks e compartilhamento de código
- Reutilização de objetos
- Recapitulando:
  - Herança e Abstração
  - Criador e Produto

Referências

- https://refactoring.guru/design-patterns/factory-method
- https://refactoring.guru/design-patterns/builder
- https://refactoring.guru/design-patterns/singleton
- Design Patterns, Booch Grady et al., 2009