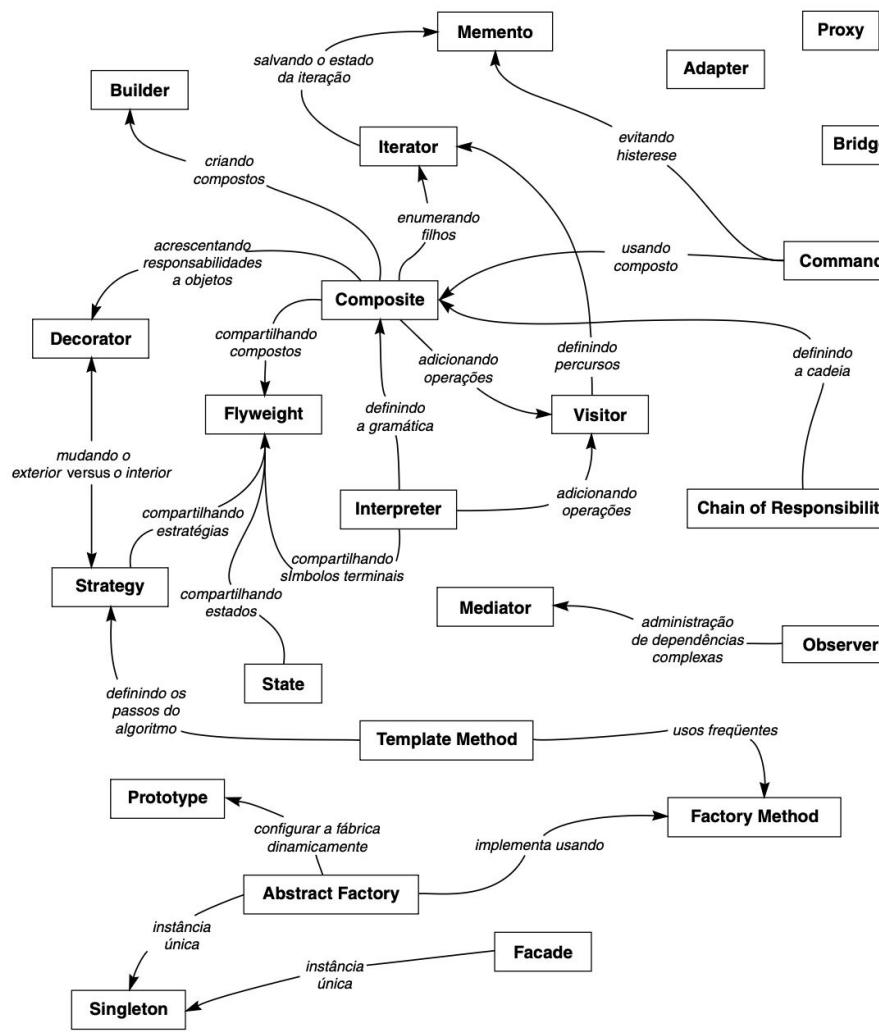


Padrões de projeto criacionais

Matheus Barbosa
matheus.barbosa@dcx.ufpb.br



O espaço dos padrões de projeto

		Propósito		
Escopo	Classe	De criação	Estrutural	Comportamental
		Factory Method (112)	Adapter (class) (140)	Interpreter (231) Template Method (301)
	Objeto	Abstract Factory (95) Builder (104) Prototype (121) Singleton (130)	Adapter (object) (140) Bridge (151) Composite (160) Decorator (170) Façade (179) Flyweight (187) Proxy (198)	Chain of Responsibility (212) Command (222) Iterator (244) Mediator (257) Memento (266) Observer (274) State (284) Strategy (292) Visitor (305)

Fornecem vários mecanismos de **criação** de **objetos**, que aumentam a flexibilidade e reutilização de código já existente.

Além dos construtores

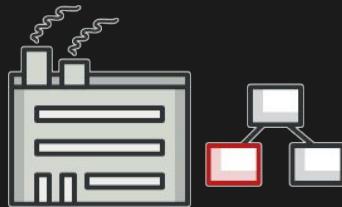
Construtores em Java definem maneiras padrão de construir objetos.

- Cliente pode não ter todos os dados necessários para instanciar um objeto
- Cliente fica acoplado a uma implementação concreta (precisa saber a classe concreta para usar new com o construtor)
- Cliente de herança pode criar construtor que chama métodos que dependem de valores ainda não inicializados (vide processo de construção)
- Objeto complexo pode necessitar da criação de objetos menores previamente, com certo controle difícil de implementar com construtores
- Não há como limitar o número de instâncias criadas



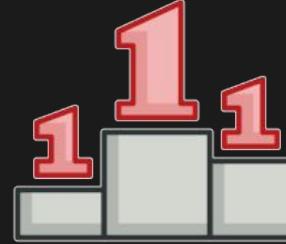
Builder

Permite construir objetos complexos passo a passo. O padrão permite produzir diferentes tipos e representações de um objeto usando o mesmo código de construção.



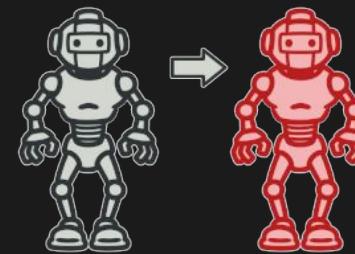
Factory Method

Fornece uma interface para criar objetos em uma superclasse, mas permite que as subclasses alterem o tipo de objetos que serão criados.



Singleton

Permite a você garantir que uma classe tem apenas uma instância, enquanto provê um ponto de acesso global para esta instância.



Prototype

Permite que você copie objetos existentes sem fazer seu código ficar dependente de suas classes.

Builder

Separar a construção de um objeto complexo de sua representação para que o mesmo processo de construção possa criar representações diferentes.

Problema

Imagina que queremos representar uma **Casa** no código.
Ela pode ter vários detalhes: número de quartos, se tem piscina, se tem garagem, etc.

Construtor Direto

House

...

+ House(windows, doors, rooms,
hasGarage, hasSwimPool,
hasStatues, hasGarden, ...)

`new House(4, 2, 4, true, null, null, null, ...)`



`new House(4, 2, 4, true, true, true, true, ...)`



Você consegue adivinhar
só olhando esse **true/null** o
que cada um significa?

A maioria dos
parâmetros não será
usada

E se você quiser uma casa maior e mais iluminada, com um jardim e outras miudezas (como um sistema de aquecimento, encanamento, e fiação elétrica)?

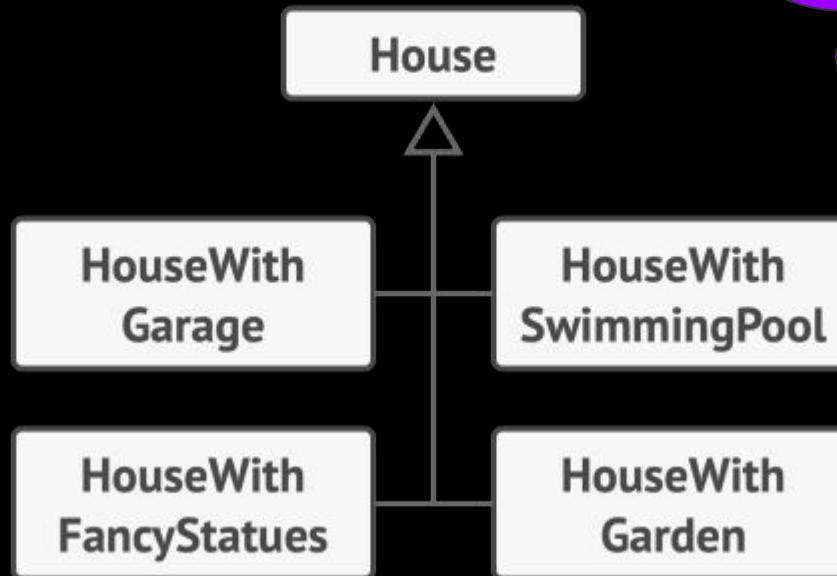
Como você faria pra deixar essa criação mais legível, sem precisar decorar a ordem dos parâmetros?



Usando subclasses



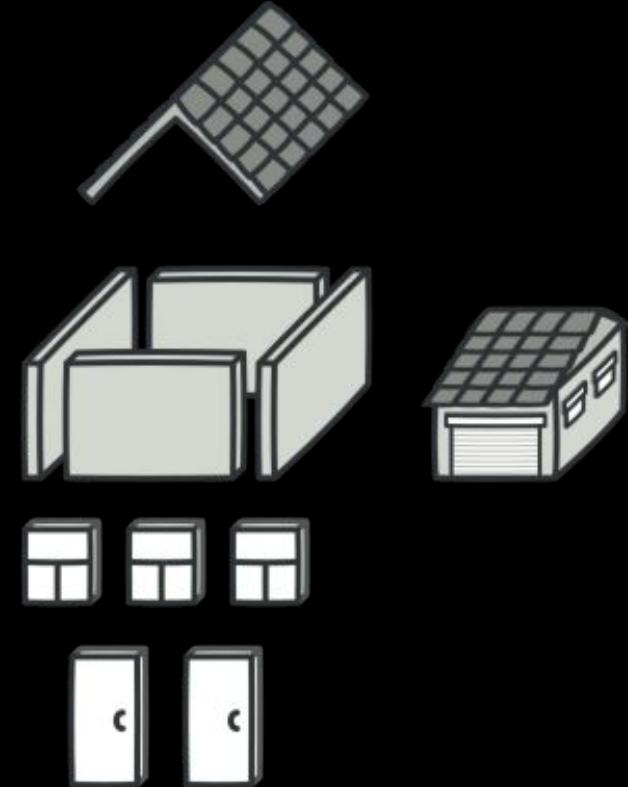
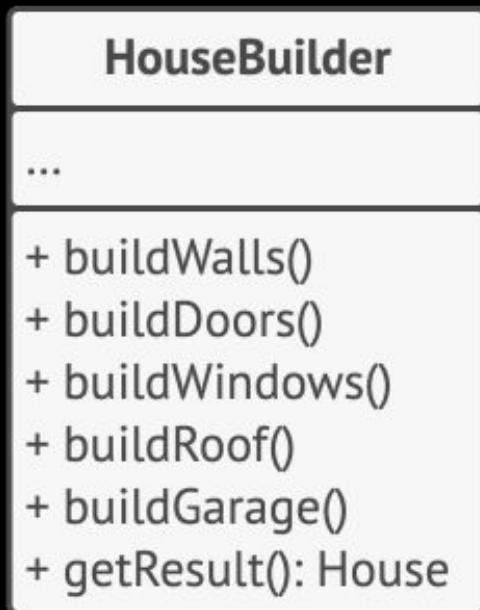
Aumenta
consideravelmente a
hierarquia cada vez
mais



Solução

O padrão organiza a construção de objetos em uma série de etapas

Você não precisa chamar todas as etapas



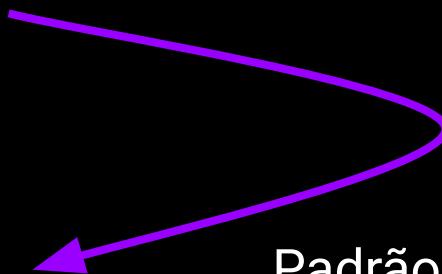
Implementação

• • •

```
1 Casa casa = new Casa(3, true, ...);
```

• • •

```
1 Casa casa = new Casa.CasaBuilder() // Depois  
2             .comQuartos(3)  
3             .comPiscina(true)  
4             .comGaragem(false)  
5             .build();
```



Padrão Builder



Casa.java

```
1 public class Casa {  
2     private int quartos;  
3     private boolean piscina;  
4     private boolean garagem;  
5  
6     private Casa() {} // só Builder pode criar  
7  
8     public int getQuartos() { return quartos; }  
9     public boolean temPiscina() { return piscina; }  
10    public boolean temGaragem() { return garagem; }  
11 }
```



CasaBuilder.java

```
1 public static class CasaBuilder {
2     private int quartos;
3     private boolean piscina;
4     private boolean garagem;
5
6     public CasaBuilder comQuartos(int q){ this.quartos = q; return this; }
7     public CasaBuilder comPiscina(boolean p){ this.piscina = p; return this; }
8     public CasaBuilder comGaragem(boolean g){ this.garagem = g; return this; }
9
10    public Casa build() {
11        Casa casa = new Casa();
12        casa.quartos = this.quartos;
13        casa.piscina = this.piscina;
14        casa.garagem = this.garagem;
15        return casa;
16    }
17 }
```

Você pode ir além e extrair uma série de chamadas para as etapas do *builder* que você usa para construir um produto em uma classe separada chamada *diretor*.



```
1 Casa.CasaBuilder builder = new Casa.CasaBuilder();
2 Diretor diretor = new Diretor(builder);
3
4 Casa simples = diretor.construirCasaSimples();
5 Casa luxo = diretor.construirCasaLuxo();
6
```



Diretor.java

```
1 public class Diretor {
2     private Casa.CasaBuilder builder;
3
4     public Diretor(Casa.CasaBuilder builder) {
5         this.builder = builder;
6     }
7
8     public Casa construirCasaSimples() {
9         return builder
10            .comQuartos(2)
11            .comGaragem(false)
12            .comPiscina(false)
13            .build();
14     }
15
16     public Casa construirCasaLuxo() {
17         return builder
18            .comQuartos(5)
19            .comGaragem(true)
20            .comPiscina(true)
21            .build();
22     }
23 }
```

Quando usar Diretor?

Quando você tem **modelos padronizados de objetos** (ex.: “pizza vegetariana”, “pizza calabresa”).

Quando quer **separar ainda mais responsabilidades**:

- Builder → sabe construir os pedaços.
- Diretor → sabe a ordem dos passos pra criar diferentes produtos.

A complexidade geral do código aumenta
uma vez que o padrão exige criar
múltiplas classes novas.

Factory Method

Definir uma interface para criar um objeto mas deixar que subclasses decidam que classe instanciar. Factory Method permite que uma classe delegue a responsabilidade de instanciação às subclasses.

Problema

Imagine um software de logística que inicialmente só entrega com caminhões. O código que solicita a entrega faz isso diretamente:

...

```
Transporte transporte = new Caminhao();
transporte.entregar();
```

O que acontece quando a empresa cresce? Se adicionarmos barcos, precisamos mudar o código existente:

...

```
Transporte transporte = new Navio(); // Tivemos que alterar essa linha!
transporte.entregar();
```

Toda vez que um novo tipo de transporte surge (um Aviao, um Trem), precisamos ir lá no código que usa o objeto e alterá-lo. Isso deixa o nosso código fortemente acoplado às classes concretas como Caminhao e Navio.



Logistica.java

```
// Classe "mãe" (Creator)
abstract class Logistica {
    // Lógica principal que não muda
    public void planejarEntrega() {
        // Pede para o método fábrica criar o objeto
        Transporte t = criarTransporte();
        t.entregar();
    }

    // O "Factory Method". Cada filha vai ter que implementar
    public abstract Transporte criarTransporte();
}
```

```
● ● ●
```

```
// Filha que sabe criar Caminhão (Concrete Creator)
class LogisticaTerrestre extends Logistica {
    public Transporte criarTransporte() {
        return new Caminhao(); // A responsabilidade está aqui!
    }
}

// Filha que sabe criar Navio (Concrete Creator)
class LogisticaMaritima extends Logistica {
    public Transporte criarTransporte() {
        return new Navio(); // A responsabilidade está aqui!
    }
}
```

Pense no seu aplicativo como se ele tivesse duas partes:

A Parte de Configuração (o "início" de tudo):

```
...  
  
// Ponto de entrada da aplicação (só roda uma vez)  
Logistica logistica;  
if (configuracao.getTipo() == "TERRESTRE") {  
    logistica = new LogisticaTerrestre();  
} else if (configuracao.getTipo() == "MARITIMA") {  
    logistica = new LogisticaMaritima();  
}
```

O Resto do Aplicativo:

```
...  
  
// Em qualquer outro lugar do código...  
// Ele não sabe qual é a logística, só a utiliza.  
logistica.planejarEntrega();
```

Sem o padrão, cada um desses cem lugares no código teria um `if/else` para decidir entre `new Caminhao()` e `new Navio()`

O código pode se tornar mais complicado, pois você precisa introduzir muitas subclasses novas para implementar o padrão.

Abstract Factory

Prover uma interface para criar famílias de objetos relacionados ou dependentes sem especificar suas classes concretas.

Problema

Imagina que você vai desenvolver uma interface gráfica que pode ter tema claro e tema escuro.

Cada tema precisa de:

- Botão
- Janela

Se usar Factory Method, você só cria um objeto de cada vez.

Mas com Abstract Factory, conseguimos garantir que os objetos sejam coerentes entre si (tudo claro ou tudo escuro).

Solução

A primeira coisa é declarar explicitamente **interfaces** para cada produto distinto.

Então você pode fazer todas as **variantes** dos produtos seguirem essas interfaces.

```
...
public interface Botao {
    void render();
}

public class BotaoClaro implements Botao {
    public void render() {
        System.out.println("Renderizando botão CLARO");
    }
}

public class BotaoEscuro implements Botao {
    public void render() {
        System.out.println("Renderizando botão ESCURO");
    }
}
```

Mesmo para Janela

Solução

O próximo passo é declarar uma **interface (Fabrica Abstrata)** com uma lista de métodos de criação para todos os produtos

...

```
public interface GUIFactory {  
    Botao criarBotao();  
    Janela criarJanela();  
}
```

Esses métodos devem retornar tipos abstratos de produtos representados pelas interfaces que extraímos previamente

Solução

Agora, para cada variante de uma família de produtos nós criamos uma **classe fábrica** separada baseada na interface Fábrica Abstrata.

```
public class FactoryTemaClaro implements  
GUIFactory {  
    public Botao criarBotao() {  
        return new BotaoClaro();  
    }  
  
    public Janela criarJanela() {  
        return new JanelaClara();  
    }  
}  
  
public class FactoryTemaEscuro implements  
GUIFactory {  
    public Botao criarBotao() {  
        return new BotaoEscuro();  
    }  
  
    public Janela criarJanela() {  
        return new JanelaEscura();  
    }  
}
```



Main.java

```
public class Main {  
    public static void main(String[] args) {  
        // podemos decidir em tempo de execução qual tema usar  
        GUIFactory factory = new FactoryTemaEscuro();  
  
        Botao botao = factory.criarBotao();  
        Janela janela = factory.criarJanela();  
  
        botao.render(); //Renderizando botão ESCURO  
        janela.abrir(); //Abrindo janela ESCURA  
    }  
}
```

O código pode tornar-se mais complicado do que deveria ser, uma vez que muitas novas interfaces e classes são introduzidas junto com o padrão.

Prototype

Especificar os tipos de objetos a serem criados usando uma instância como protótipo e criar novos objetos ao copiar este protótipo

Problema

Digamos que você tenha um objeto, e você quer criar uma cópia exata dele. Como você o faria?



Main.java

```
public class Main {  
    public static void main(String[] args) {  
        // Criando documento original  
        Document doc1 = new Document("Relatório", "Conteúdo importante", "Alice", 1);  
  
        // Criando cópia manualmente  
        Document doc2 = new Document(doc1.getTitle(), doc1.getContent(), ...);  
        doc2.setTitle("Relatório - Cópia");  
        doc2.setVersion(2);  
  
        System.out.println(doc1); // Relatório - Alice - v1  
        System.out.println(doc2); // Relatório - Cópia - Alice - v2  
    }  
}
```



Document.java

```
public class Document {  
    private String title;  
    private String content;  
    private String author;  
    private int version;  
  
    public Document(String title, String content, String author, int version) {  
        ...  
    }  
  
    // getters e setters  
  
    public String toString() {  
        return title + " - " + author + " - v" + version;  
    }  
}
```

Cada cópia exige repetir todos os campos manualmente

Difícil de manter se a classe tiver muitos atributos

Se mudarmos a classe, precisamos atualizar todas as cópias manuais



```
public interface Prototype {
    Prototype clone();
}

public class Document implements Prototype {
    private String title;
    private String content;
    private String author;
    private int version;

    public Document(String title, String content, String author, int version) {
        ...
    }

    // getters e setters

    @Override
    public Prototype clone() {
        return new Document(this.title, this.content, this.author, this.version);
    }

    public String toString() {
        return title + " - " + author + " - v" + version;
    }
}
```

Main.java

```
public class Main {  
    public static void main(String[] args) {  
        Document doc1 = new Document("Relatório", "Conteúdo importante", "Alice", 1);  
  
        // Clonando documento  
        Document doc2 = (Document) doc1.clone();  
        doc2.setTitle("Relatório - Cópia");  
        doc2.setVersion(2);  
  
        System.out.println(doc1); // Relatório - Alice - v1  
        System.out.println(doc2); // Relatório - Cópia - Alice - v2  
    }  
}
```

O padrão *Prototype* está disponível e pronto para uso em Java com a interface Cloneable.

Qualquer classe pode implementar essa interface para se tornar clonável.

Clonar objetos complexos que têm referências circulares pode ser bem complicado.

Singleton

Garantir que uma classe só tenha uma única instância, e prover um ponto de acesso global a ela

Problema

O padrão *Singleton* resolve dois problemas de uma só vez.

1. Garantir que uma classe tenha apenas uma única instância
2. Fornece um ponto de acesso global para aquela instância

Por que usar Singleton?

Imagine que você tem algo que não faz sentido criar mais de uma vez, como:

1. Conexão com banco de dados
2. Logger (registro de logs)
3. Configurações globais do sistema

```
1  public class Singleton {  
2      // 1. Guarda a única instância  
3      private static Singleton instance;  
4  
5      // 2. Construtor privado impede new fora da classe  
6      private Singleton() {}  
7  
8      // 3. Método público para acessar a instância  
9      public static Singleton getInstance() {  
10          if (instance == null) {  
11              instance = new Singleton();  
12          }  
13          return instance;  
14      }  
15  
16      // Exemplo de método  
17      public void showMessage() {  
18          System.out.println("Usando a instância Singleton!");  
19      }  
20  }
```

O campo para armazenar a instância *singleton* deve ser declarado como privado estático.

Construtor privativo (nem subclasses têm acesso)

Ponto de acesso simples, estático e global



Main.java

```
1 public class Main {  
2     public static void main(String[] args) {  
3         Singleton s1 = Singleton.getInstance();  
4         Singleton s2 = Singleton.getInstance();  
5  
6         s1.showMessage();  
7  
8         System.out.println(s1 == s2); // true, é a mesma instância  
9     }  
10 }  
11
```

O padrão requer tratamento especial em um ambiente multithreaded para que múltiplas threads não possam criar um objeto singleton várias vezes.

Todo acesso a
getInstance() fica
mais lento, porque
cada chamada precisa
do *synchronized*.

Bloco deve ser *synchronized* para evitar
que dois objetos tentem criar o
objeto ao mesmo tempo



Singleton.java

```
1 public static synchronized Singleton getInstance() {  
2     if (instance == null) {  
3         instance = new Singleton();  
4     }  
5     return instance;  
6 }
```

Viola o princípio de responsabilidade única. O padrão resolve dois problemas de uma só vez.

Não é possível trabalhar com interfaces.
(Fere o S.O.L.I.D)

Monostate

Garante que todas as instâncias de uma classe compartilhem o mesmo estado, mas não impede que você crie várias instâncias.

```
1 public class Monostate {
2     // estado compartilhado
3     private static String sharedState;
4
5     public void setState(String state) {
6         sharedState = state;
7     }
8
9     public String getState() {
10        return sharedState;
11    }
12 }
13
14 public class Main {
15     public static void main(String[] args) {
16         Monostate m1 = new Monostate();
17         Monostate m2 = new Monostate();
18
19         m1.setState("Olá do Monostate!");
20
21         System.out.println(m2.getState()); // imprime: "Olá do Monostate!"
22         System.out.println(m1 == m2);      // false → são instâncias diferentes
23     }
24 }
25
```