

# Padrões de Projeto

# 4

## Padrões de Construção

# Introdução

- *A maneira padrão de construir objetos em Java é através de construtores*
  - *Toda classe tem um construtor: operação declarada com o mesmo nome da classe, que não retorna valor e só pode ser usada na inicialização*
  - *O construtor é uma tarefa de classe (estática)*
  - *Se um construtor não é explicitamente declarado em uma classe, o sistema cria um construtor default para a classe*
  - *Todo construtor inicializa a hierarquia de classes do objeto antes de executar*
  - *Todo construtor sempre contém ou uma referência à superclasse (implícita ou explícita) ou uma referência a outro construtor da classe como primeira ou única instrução*

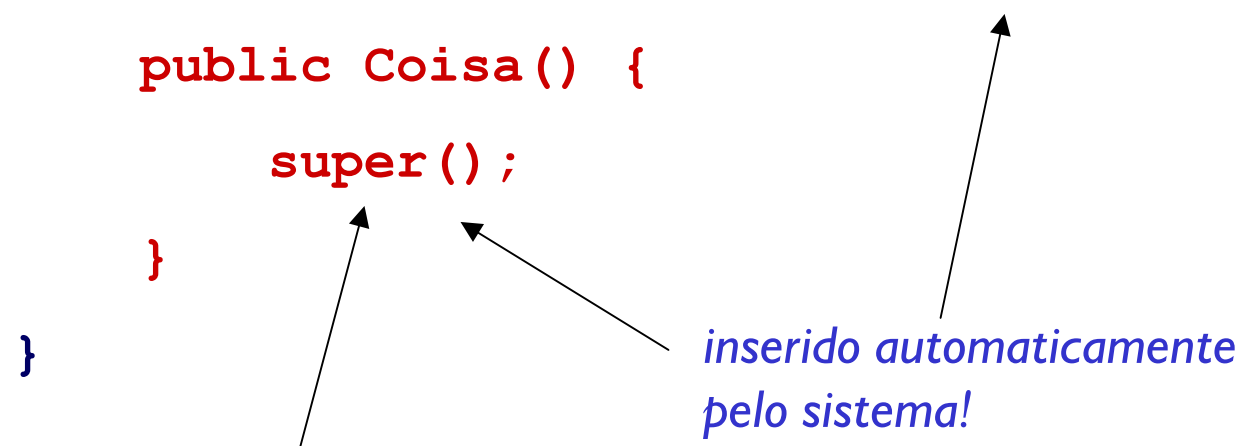
# Construtores em Java

- Escrever

```
public class Coisa {}
```

é o mesmo que

```
public class Coisa extends java.lang.Object {  
    public Coisa() {  
        super();  
    }  
}
```



*chama construtor default (com assinatura sem argumentos) da superclasse*

*inserido automaticamente pelo sistema!*

# Construtores invisíveis

- *Implicações:*

```
public class OutraCoisa extends Coisa {}
```

*funciona enquanto Coisa não tiver construtor explícito:*

```
public class Coisa {}
```

- *Se um construtor for adicionado explicitamente a Coisa:*

```
public class Coisa extends java.lang.Object {  
    private String nome;  
    public Coisa(String nome) {  
        this.nome = nome;  
    }  
}
```

- *A subclasse OutraCoisa não compila mais. Por que?*

## OutraCoisa não compila!

- *OutraCoisa* tem um construtor implícito que chama o construtor default da superclasse (identificado pela assinatura)

```
public class OutraCoisa extends Coisa {  
    public OutraCoisa() {  
        super();  
    }  
}
```

- Na nova versão de *Coisa*, o construtor default não existe mais, pois foi substituído pelo construtor que requer um argumento: ***Coisa(String)***
- Indique duas soluções para fazer a classe *OutraCoisa* compilar novamente.

```
java.lang.Object
```

# Sobrecarga de construtores

```
public class Coisa {  
    private String nome = "Ainda Sem Nome";  
    public Coisa() {  
        this("Coisa Sem Sentido");  
    }  
    public Coisa(String nome) {  
        this.nome = nome;  
    }  
}
```

*Duas maneiras de construir uma coisa:*

```
Coisa c = new Coisa();
```

*cria uma Coisa Sem Sentido, e*

```
Coisa d = new Coisa("Algo Útil");
```

*cria uma Coisa que guarda Algo Útil*

*Apenas o segundo construtor faz chamada via `super()` à superclasse.*

```
public class OutraCoisa  
    extends Coisa {  
    private int estado = 10;  
    public OutraCoisa() {  
        super("Outra Coisa");  
        estado = 5;  
    }  
}
```

*Processo de criação de uma OutraCoisa:*

*1. Cliente chama `new OutraCoisa()`*

*2. Sistema inicializa `estado = 0;`*

*3. Sistema chama `Coisa("Outra Coisa")`*

*3.1 `Coisa.nome = null;`*

*3.2 Sistema chama `Object()` ;*

*3.2.1 Variáveis de Object inicializadas*

*3.2.2 Construtor Object executado()*

*3.3 Inicialização explícita: `Coisa.nome="Ainda Sem Nome"`*

*3.4 Execução de `Coisa()`: `Coisa.nome="Outra Coisa"`*

*4. Sistema inicializa `estado = 10;`*

*5. Sistema executa construtor: `estado = 5;`*

*6. Objeto inicializado e pronto para ser usado*

Quais as  
implicações disto?



# Além dos construtores

- *Construtores em Java definem maneiras padrão de construir objetos. Sobrecarga permite ampla flexibilidade*
- *Alguns problemas em depender de construtores*
  - *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*

# Além dos construtores

- *Padrões que oferecem alternativas à construção de objetos*
  - **Builder**: obtém informação necessária em passos antes de requisitar a construção de um objeto
  - **Factory Method**: adia a decisão sobre qual classe concreta instanciar
  - **Abstract Factory**: construir uma família de objetos que compartilham um "tema" em comum
  - **Prototype**: especificar a criação de um objeto a partir de um exemplo fornecido
  - **Memento**: reconstruir um objeto a partir de uma versão que contém apenas seu estado interno



# Builder

*"Separar a construção de um objeto complexo de sua representação para que o mesmo processo de construção possa criar representações diferentes." [GoF]*

# Problema

Cliente

*Cliente precisa de uma casa. Passa as informações necessárias para seu diretor*

Diretor

*Utilizando as informações passadas pelo cliente, ordena a criação da casa pelo construtor usando uma interface uniforme*

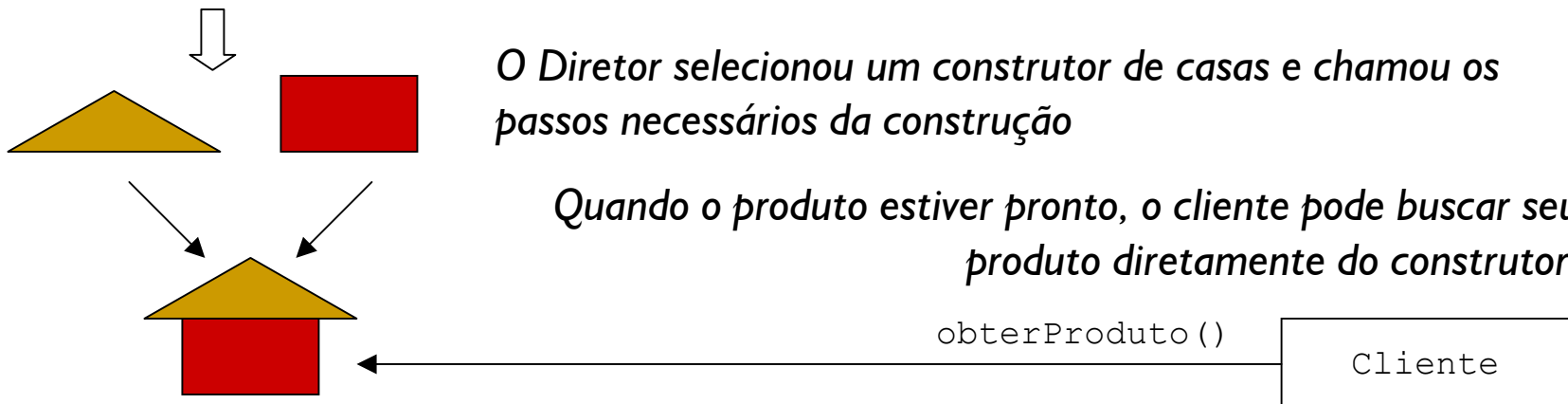
Construtor  
passoUm()  
passoDois()  
obterProduto()

*O construtor é habilitado para construir qualquer objeto complexo (poderia, por exemplo, construir um prédio em vez de uma casa, caso o cliente tivesse indicado esse desejo)*

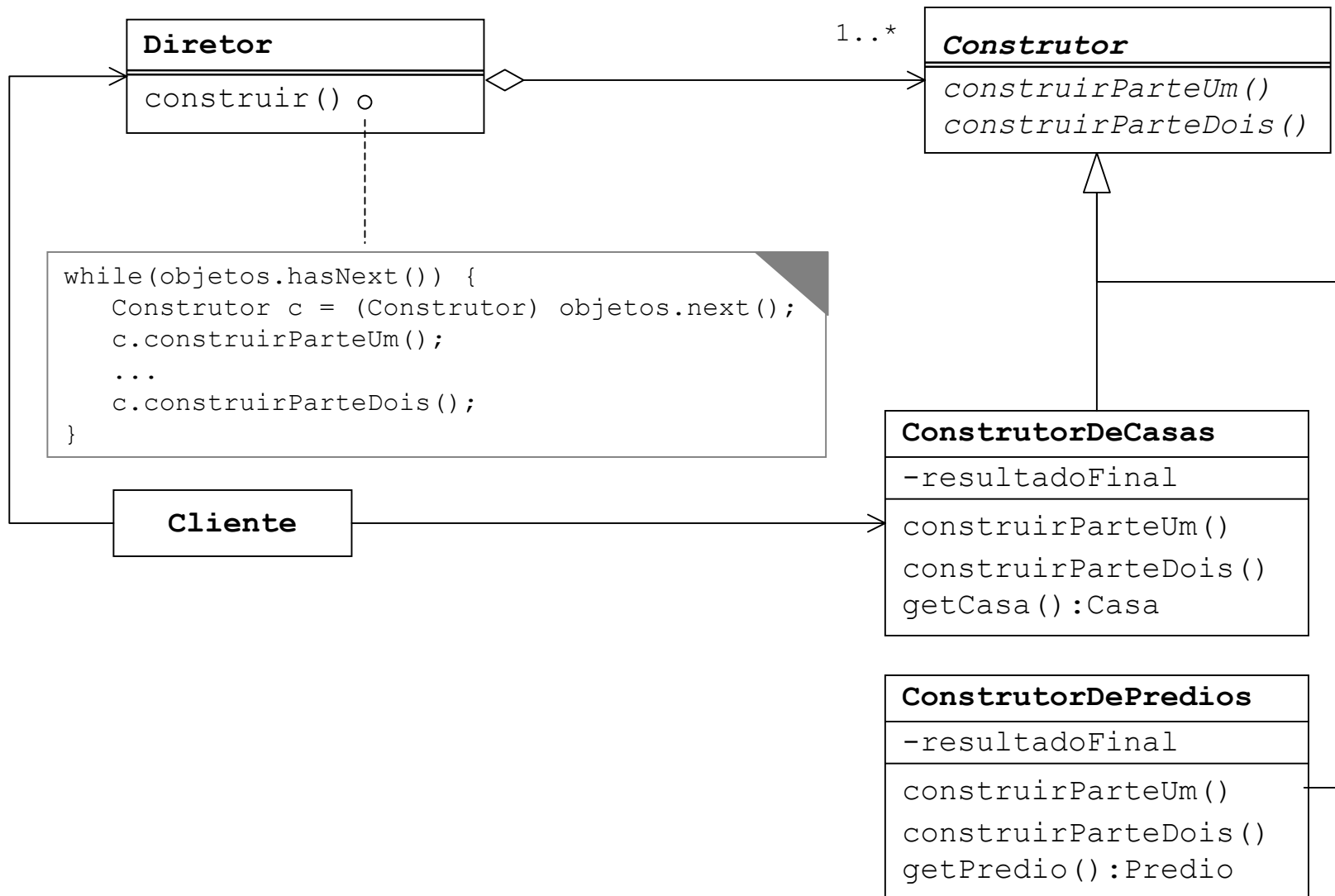


*O Diretor selecionou um construtor de casas e chamou os passos necessários da construção*

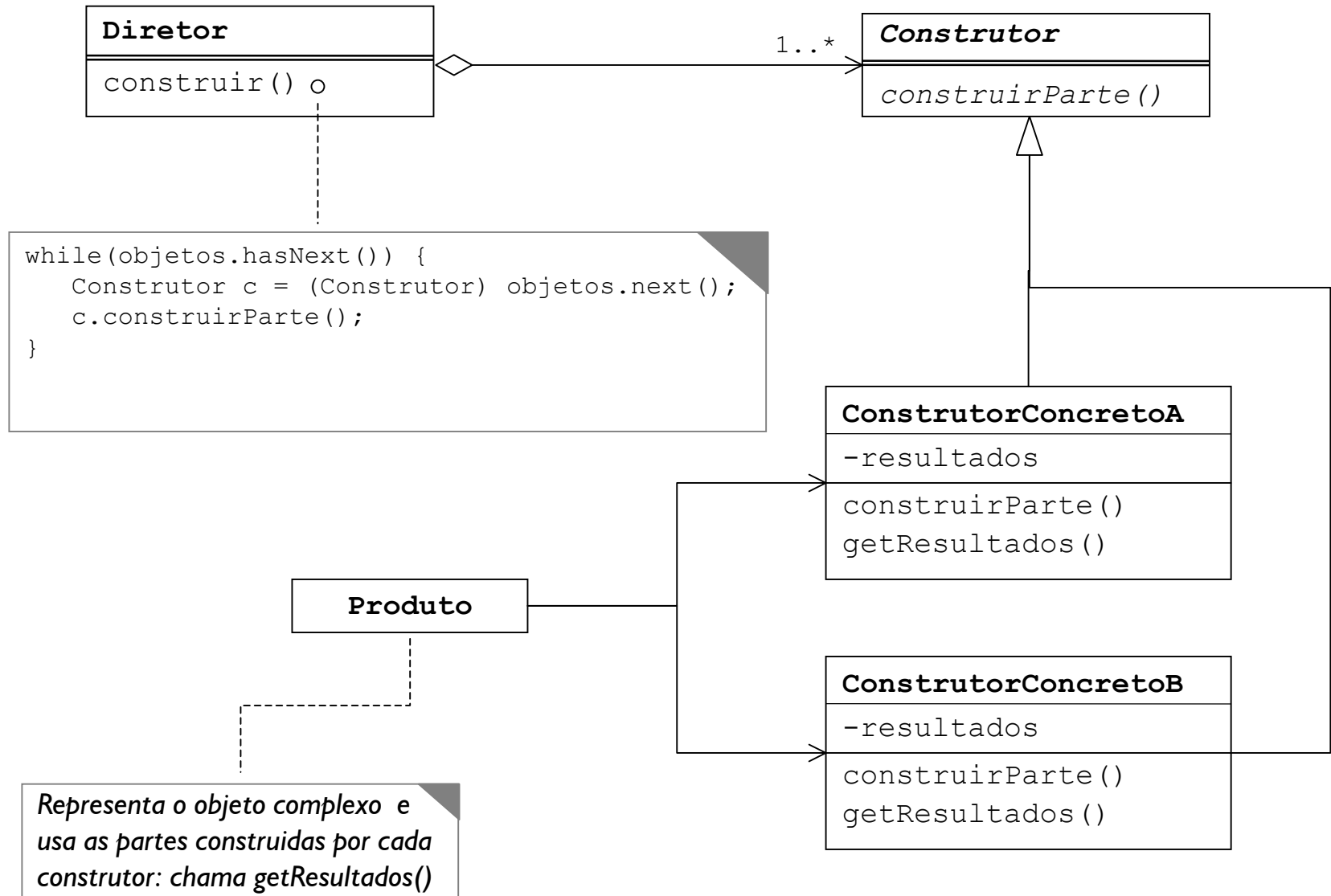
*Quando o produto estiver pronto, o cliente pode buscar seu produto diretamente do construtor.*



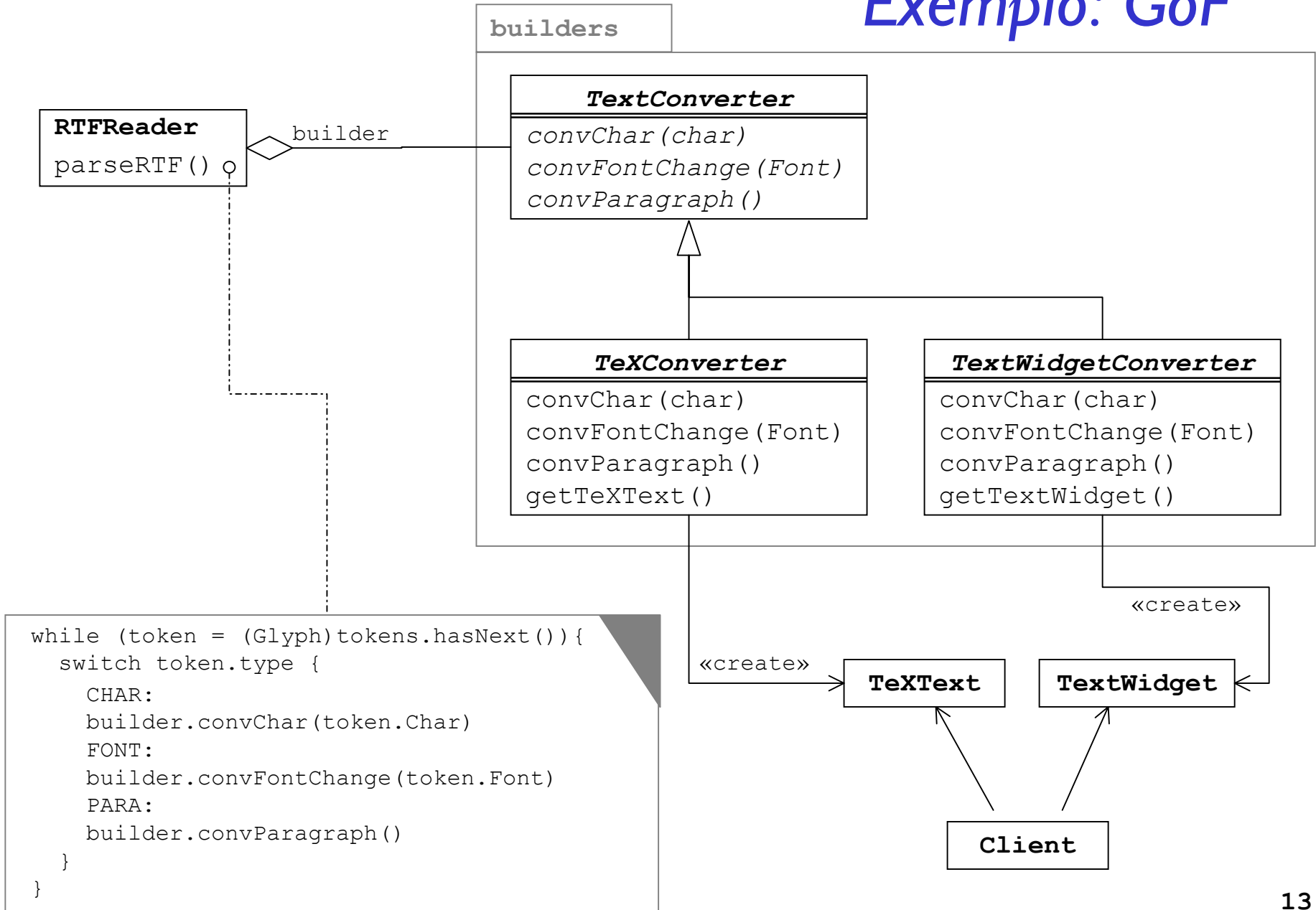
# Exemplo



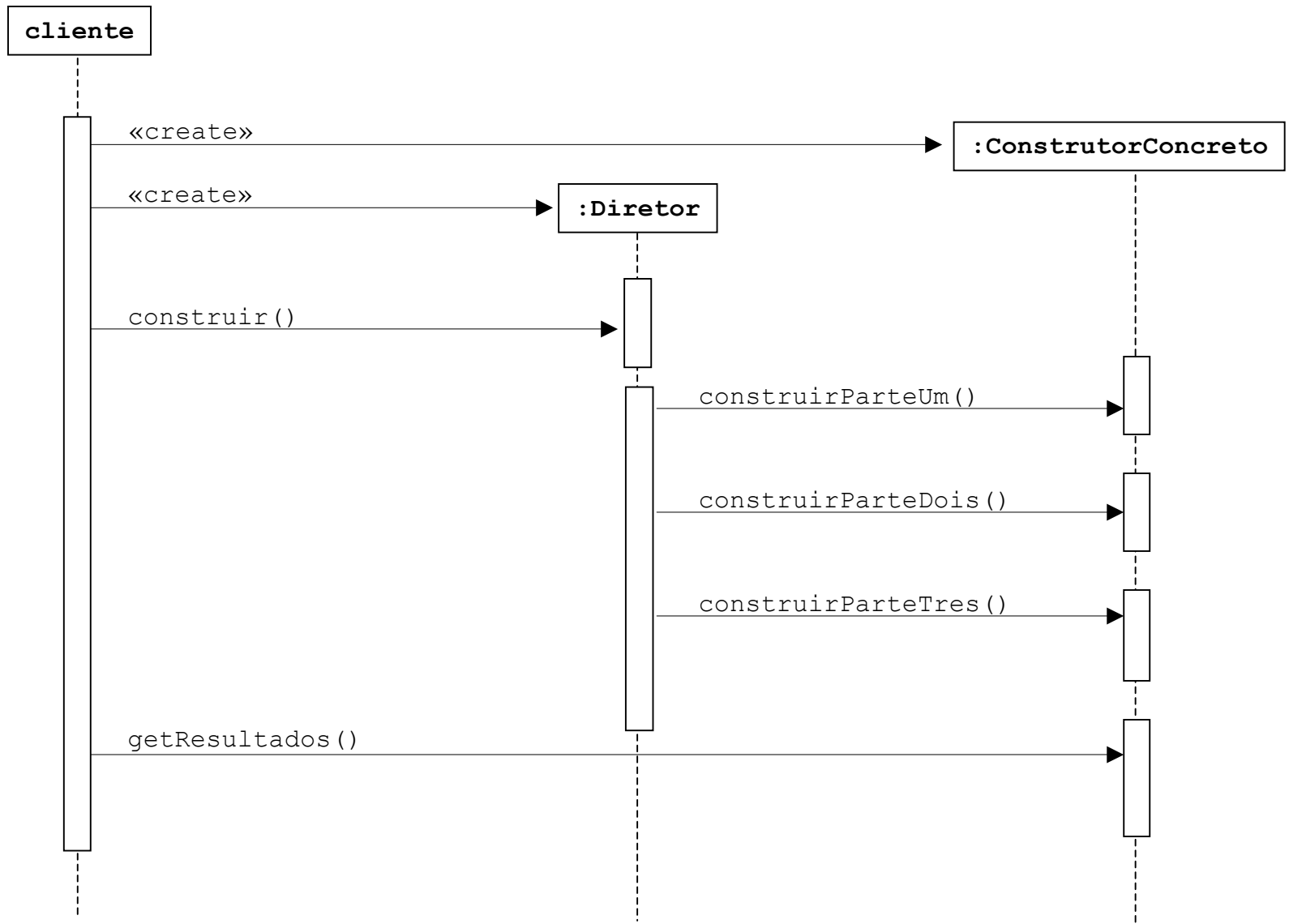
# Estrutura de Builder



# Exemplo: GoF



# Seqüência de Builder



## Quando usar?

- *Builder permite que uma classe se preocupe com apenas uma parte da construção de um objeto. É útil em algoritmos de construção complexos*
  - *Use-o quando o algoritmo para criar um objeto complexo precisar ser independente das partes que compõem o objeto e da forma como o objeto é construído*
- *Builder também suporta substituição dos construtores, permitindo que a mesma interface seja usada para construir representações diferentes dos mesmos dados*
  - *Use quando o processo de construção precisar suportar representações diferentes do objeto que está sendo construído*

# Exercícios

- *11.1 Uma aplicação precisa construir objetos Pessoa, e Empresa. Para isto, precisa ler dados de um banco para cada produto.*
  - *Para construir uma Pessoa é preciso obter nome e identidade. Apenas se os dois forem lidos a pessoa pode ser criada*
  - *Para construir uma empresa é preciso ler o nome e identidade do responsável e depois construir a pessoa do responsável.*
- *Mostre como poderia ser implementada uma aplicação que realizasse as tarefas acima. Se possível, implemente (simule os dados com Strings)*

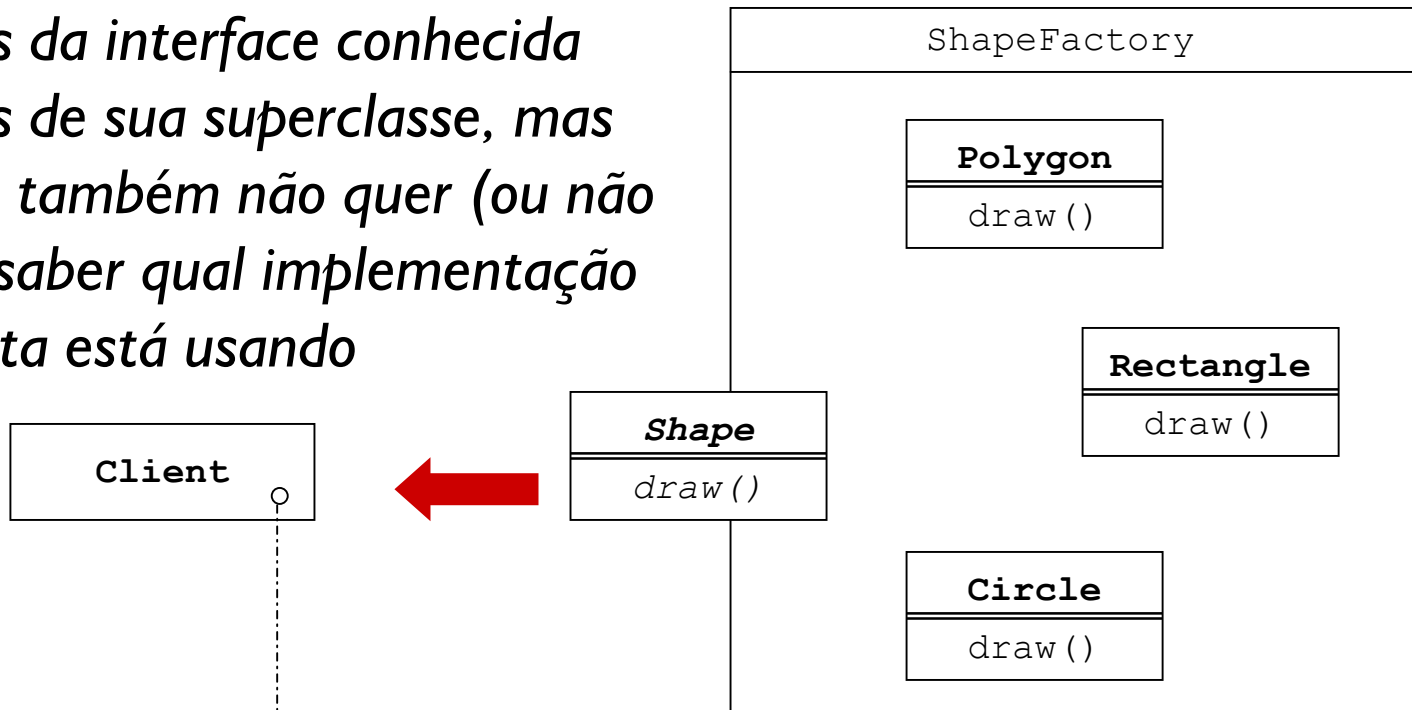


# Factory Method

*"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 instanciamento às subclasses." [GoF]*

# Problema

*O acesso a um objeto concreto será através da interface conhecida através de sua superclasse, mas cliente também não quer (ou não pode) saber qual implementação concreta está usando*



~~Shape shape = new Rectangle();~~

Shape shape = ShapeFactory.getShape("rect");  
shape.draw();

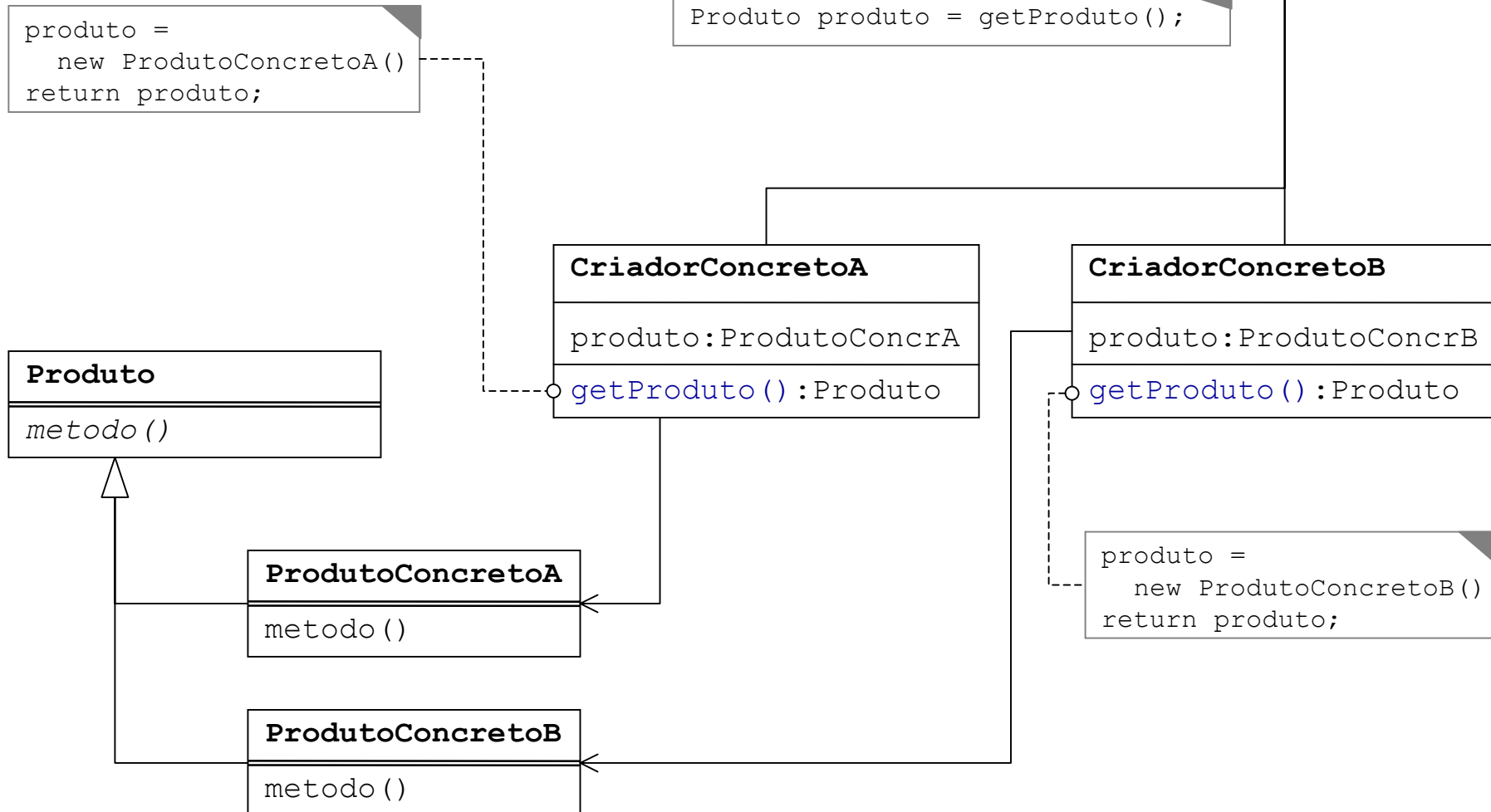
return new Rectangle()  
**nesto contexto**

```
public static Shape getShape(String type) {
    ShapeFactory factory = (ShapeFactory)typeMap.get(type);
    return factory.getShape(); // non-static Factory Method
}
```

## Como implementar?

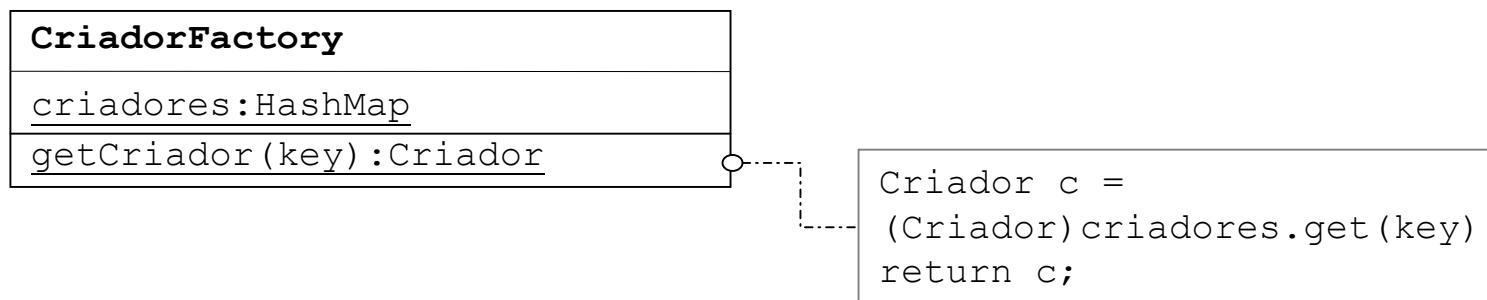
- *É possível criar um objeto sem ter conhecimento algum de sua classe concreta?*
  - *Esse conhecimento deve estar em alguma parte do sistema, mas não precisa estar no cliente*
  - *FactoryMethod define uma interface comum para criar objetos*
  - *O objeto específico é determinado nas diferentes implementações dessa interface*
  - *O cliente do FactoryMethod precisa saber sobre implementações concretas do objeto criador do produto desejado*

# Estrutura de Factory Method



## Como selecionar o criador

- Para criar objetos não é mais preciso saber a classe concreta do objeto a ser criado, mas ainda é preciso saber a classe do criador.
- Para escolher qual criador usar sem que seja preciso instanciá-lo com um construtor, crie uma classe *Factory* com um método estático que decida qual criador usar com base em um parâmetro



- O objeto criador pode ser selecionado com base em outros critérios que não requeiram parâmetros

# Prós e contras

- **Vantagens**

- *Criação de objetos é desacoplada do conhecimento do tipo concreto do objeto*
- *Conecta hierarquias de classe paralelas*
- *Facilita a extensibilidade*

- **Desvantagens**

- *Ainda é preciso saber a classe concreta do criador de instâncias (pode-se usar uma classe *Factory*, com método estático e parametrizado que chame diretamente o *Factory Method*):*

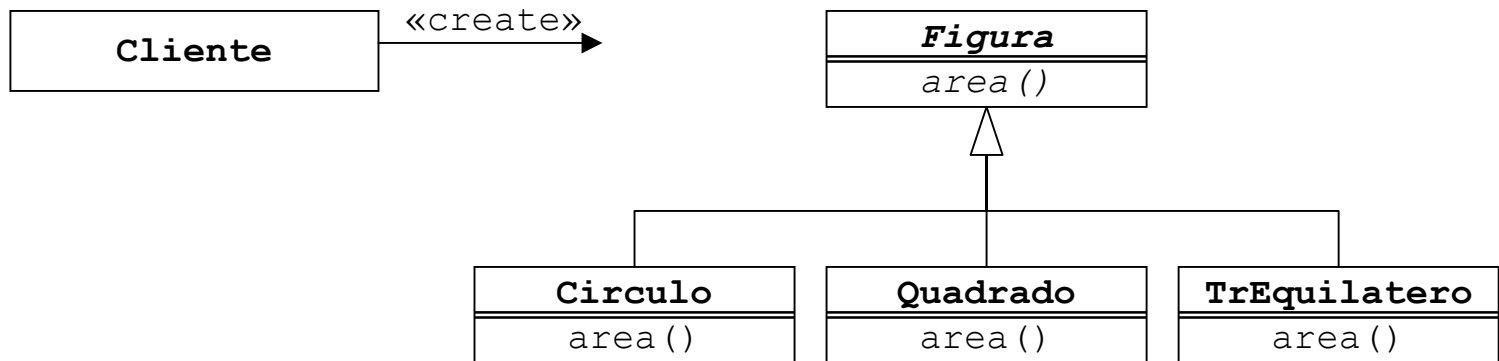
```
public static Thing createThing(int type) {  
    if (type == 1) {  
        creator = new ConcreteThingCreator();  
        return creator.createThing();  
    } ...  
}
```

Factory

Factory Method

# Exercícios

- *12.1 Implemente a aplicação abaixo usando Factory Method para criar os objetos*



- *Crie um objeto construtor para cada tipo de objeto*
- *Crie uma fachada contendo um HashMap, onde os construtores são guardados, e um método estático que permita selecionar o construtor desejado (ou embute o Factory Method neste método) com base em uma chave*

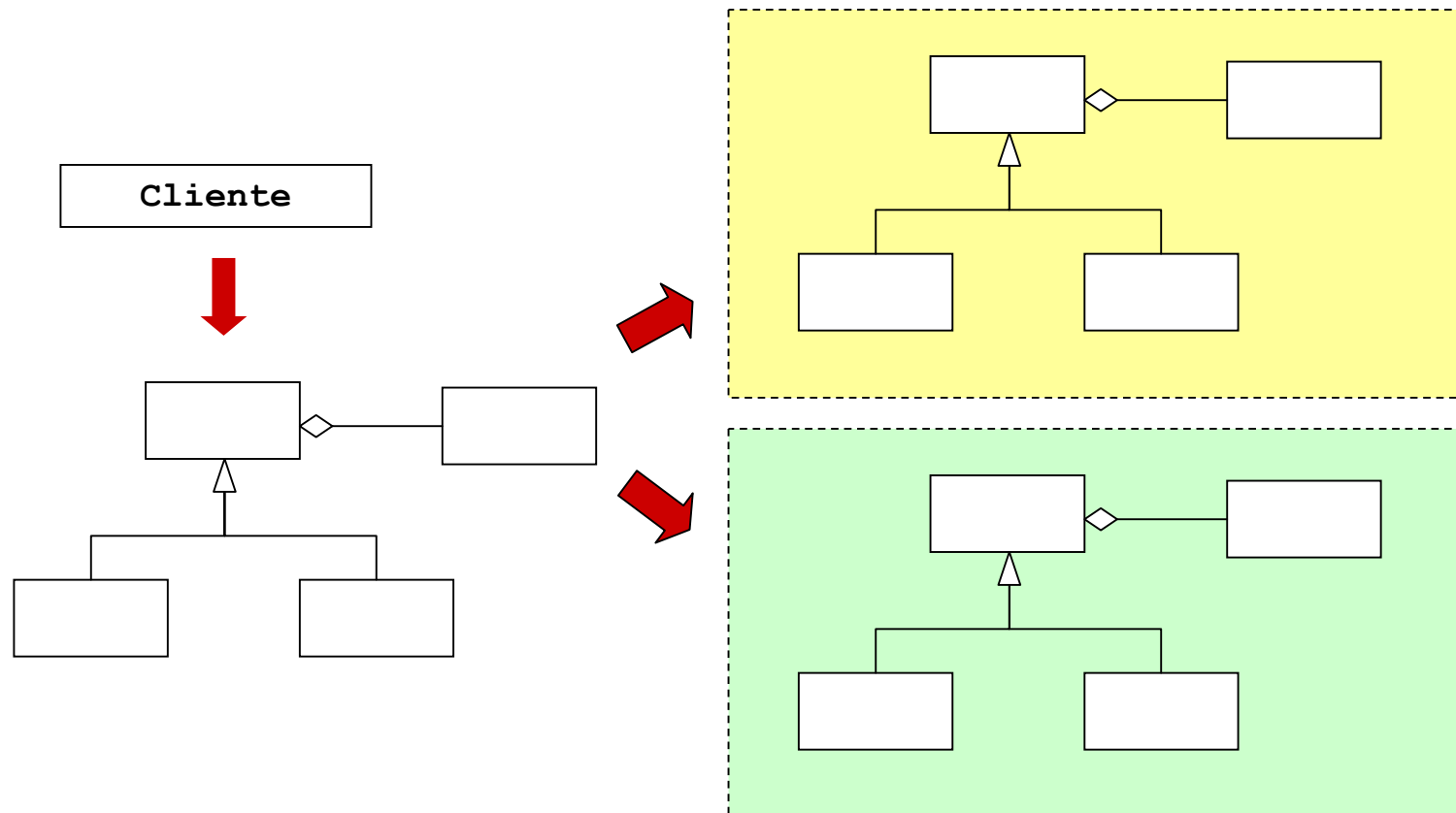
# Abstract Factory

*"Prover uma interface para criar famílias de objetos relacionados ou dependentes sem especificar suas classes concretas." [GoF]*

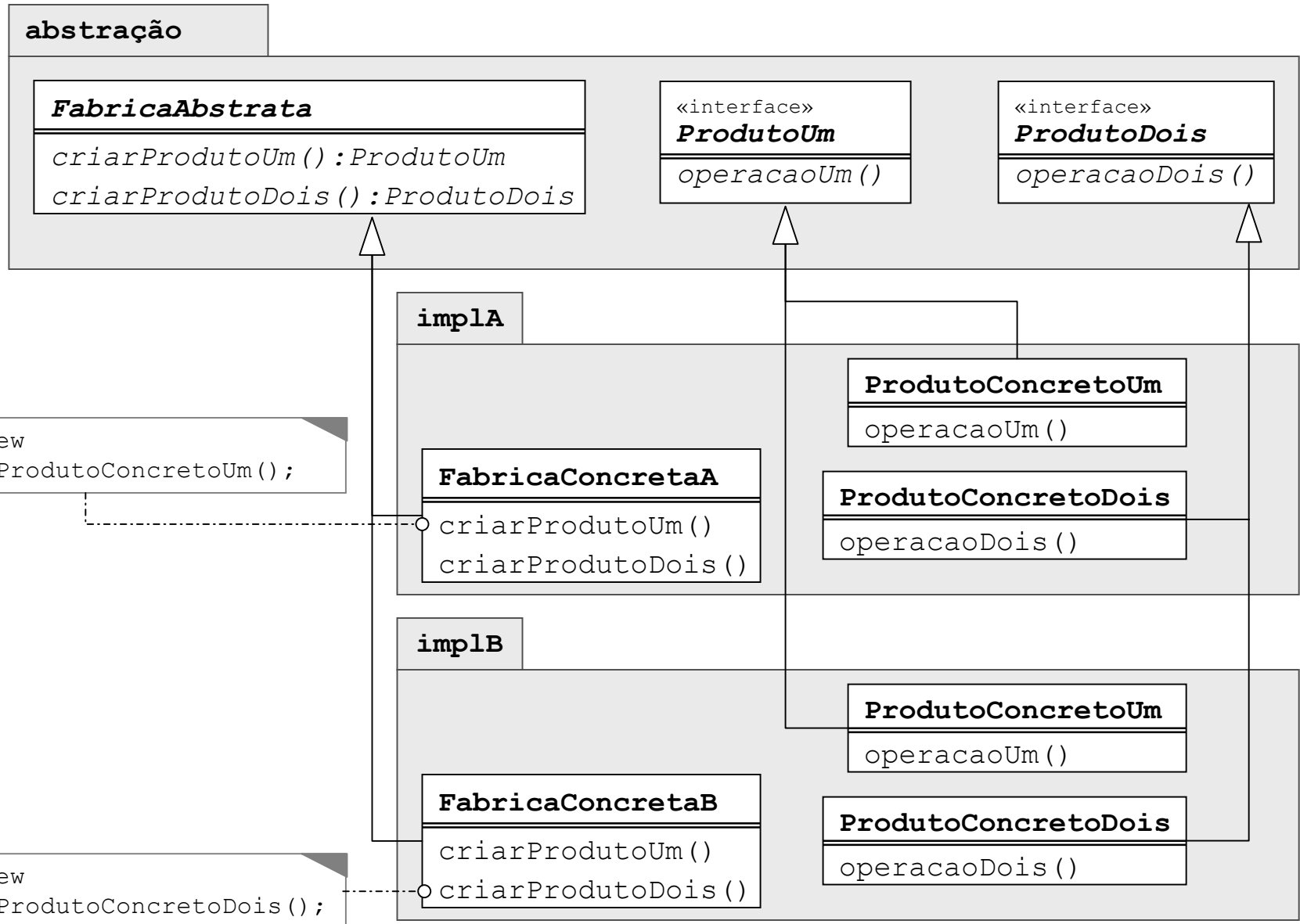


# Problema

- *Criar uma família de objetos relacionados sem conhecer suas classes concretas*



# Estrutura de Abstract Factory



## Exercícios

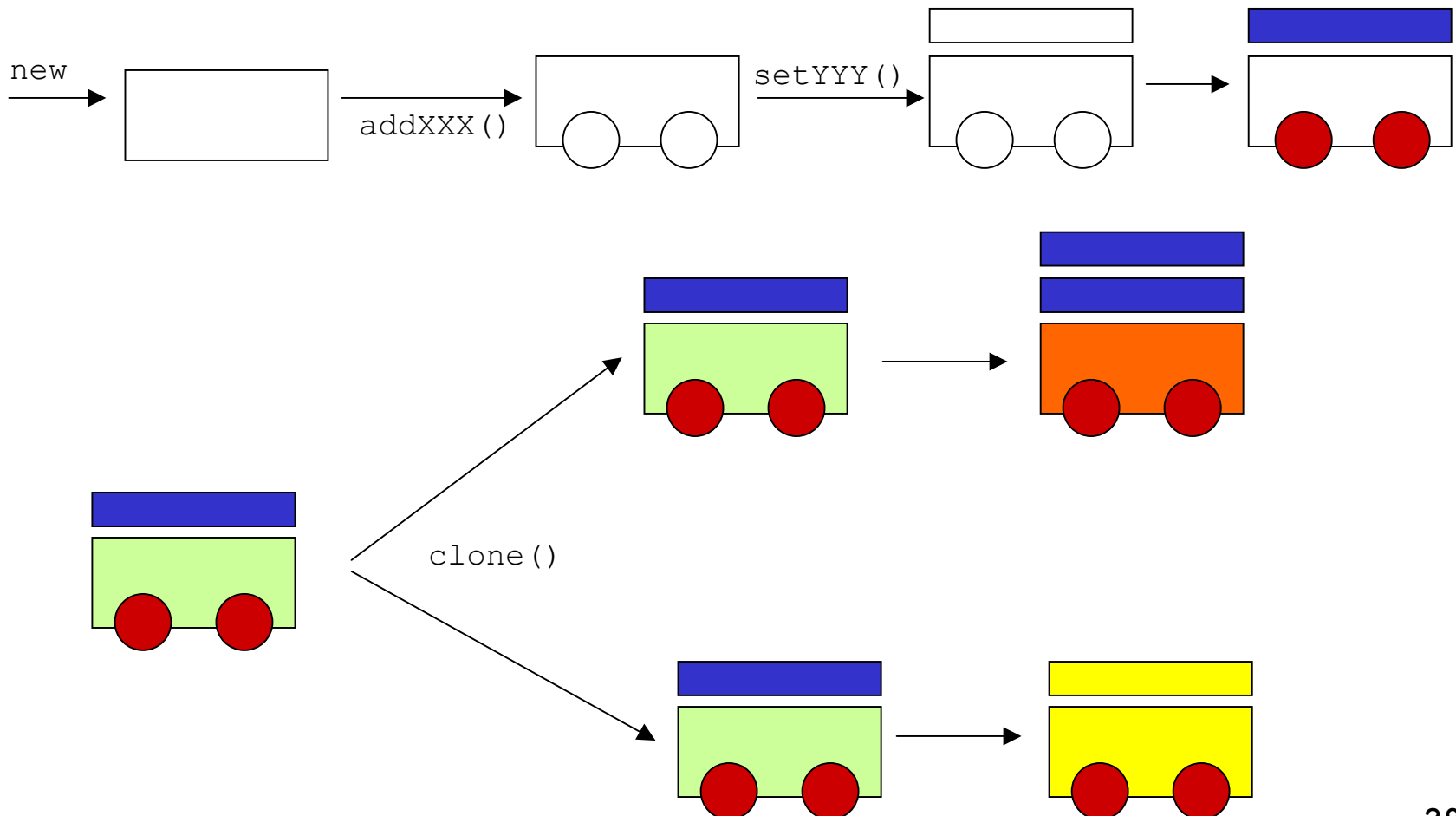
- *13.1 Implemente uma aplicação que constrói Pontos (x, y), Circulos (Ponto, raio), Retangulos (Ponto, Ponto), Triangulos (Ponto, Ponto, Ponto) e use uma fábrica abstrata para controlar a criação de todos os objetos*
- *13.2 Cite exemplos de Abstract Factory no J2SDK*

# Prototype

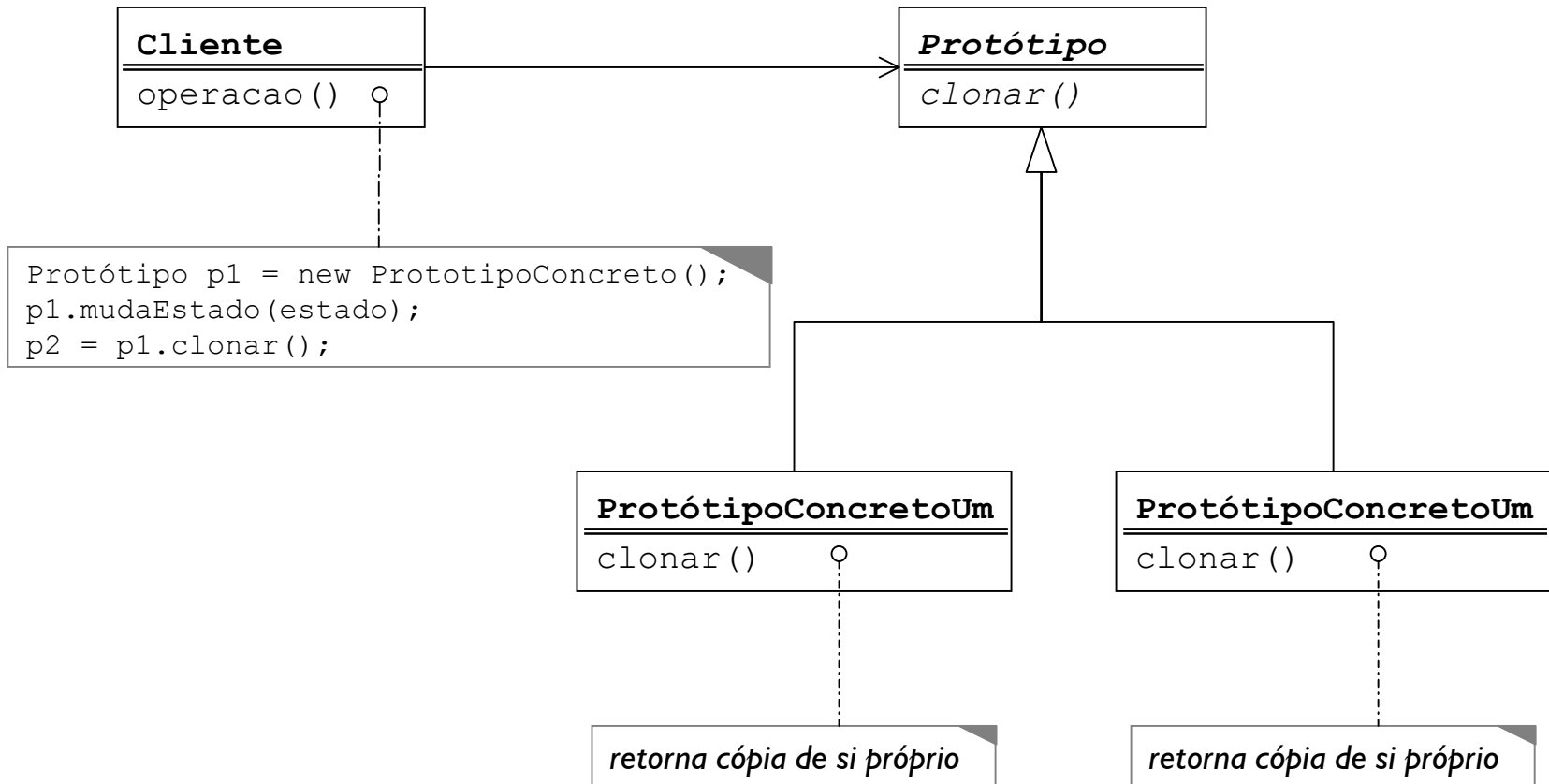
*"Especificar os tipos de objetos a serem criados usando uma instância como protótipo e criar novos objetos ao copiar este protótipo." [GoF]*

# Problema

- Criar um objeto novo, mas aproveitar o estado previamente existente em outro objeto



# Estrutura de Prototype



# Prototype em Java

- *Object.clone()* é um ótimo exemplo de Prototype em Java

```
Circulo c = new Circulo(4, 5, 6);
```

```
Circulo copia = (Circulo) c.clone();
```

- Se o objeto apenas contiver tipos primitivos em seus campos de dados, é preciso

- *declarar que a classe implementa Cloneable*
- *sobrepôr clone() da seguinte forma:*

*é preciso sobrepôr  
clone() porque ele  
é definido como  
protected*

```
public Object clone() {  
    try {  
        return super.clone();  
    } catch (CloneNotSupportedException e) {  
        return null;  
    }  
}
```

# Prototype em Java: Clone

- Se o objeto contiver campos de dados que são referências a objetos, é preciso fazer cópias desses objetos também

```
public class Circulo {  
    private Point origem;  
    private double raio;  
    public Object clone() {  
        try {  
            Circulo c = (Circulo)super.clone();  
            c.origem = origem.clone(); // Point deve ser clonável!  
            return c;  
        } catch (CloneNotSupportedException e) {return null;}  
    }  
}
```



- *O padrão Prototype permite que um cliente crie novos objetos ao copiar objetos existentes*
- *Uma vantagem de criar objetos deste modo é poder aproveitar o estado existente de um objeto*
- *Object.clone() pode ser usado como implementação do Prototype pattern em Java mas é preciso lembrar que ele só faz cópias rasas: é preciso copiar também cada objeto membro e seus campos recursivamente.*

## Exercício

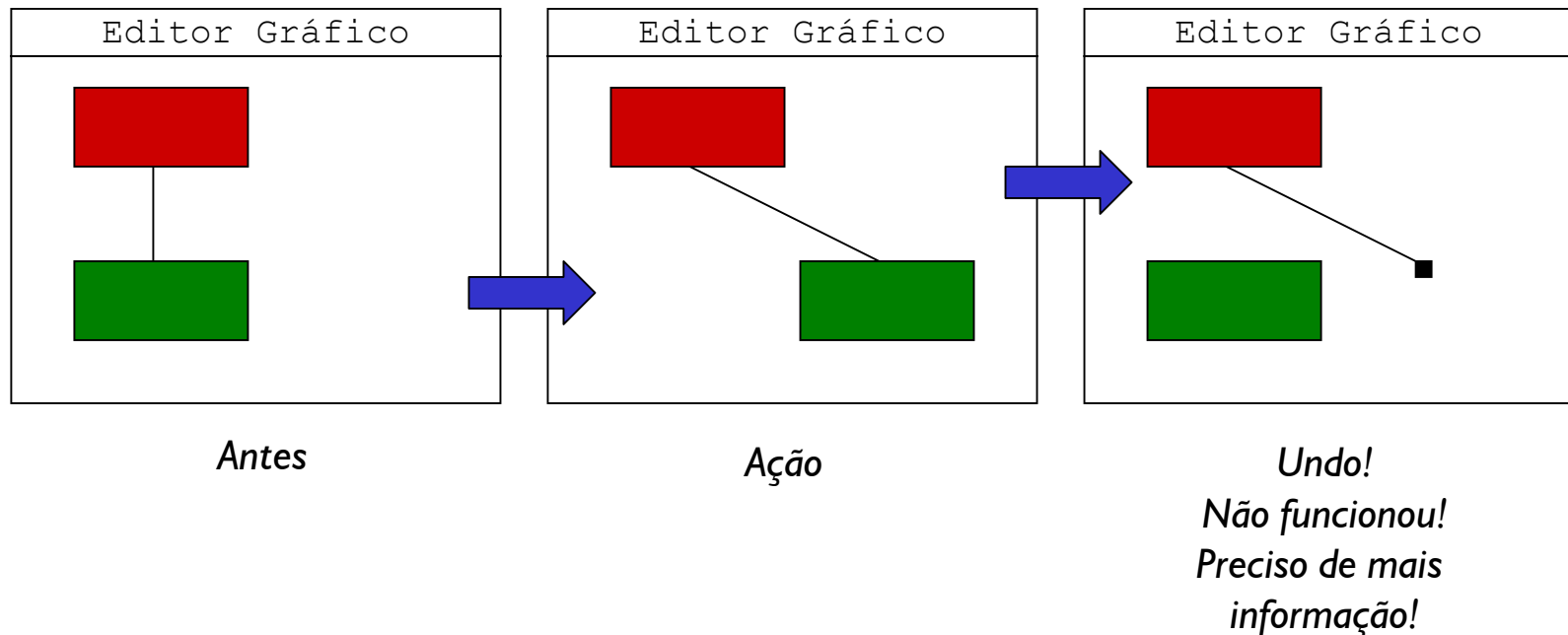
- *14.1 Implemente, na fábrica de Figuras (capítulos anteriores), métodos createXXX() que aceitem um objeto como argumento e retornem um clone*
  - *Implemente o método clone() em todos os objetos*
  - *Garanta que a cópia realmente copia todo o estado do objeto.*

# Memento

*"Sem violar o encapsulamento, capturar e externalizar o estado interno de um objeto para que o objeto possa ter esse estado restaurado posteriormente." [GoF]*

# Problema

- *É preciso guardar informações sobre um objeto suficientes para desfazer uma operação, mas essas informações não devem ser públicas*



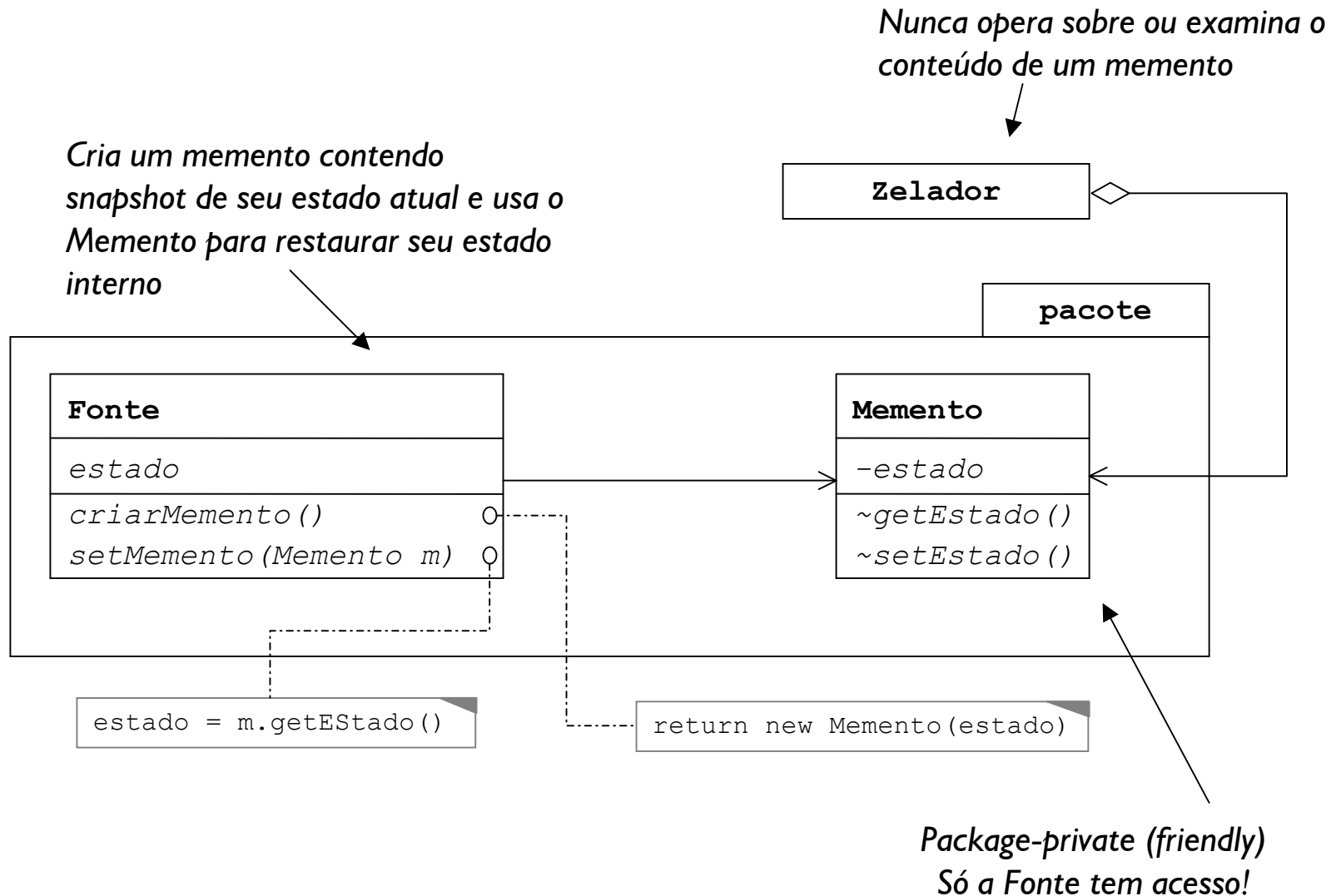
# Solução: Memento

- *Um memento é um pequeno repositório para guardar estado dos objetos*
  - *Pode-se usar outro objeto, um string, um arquivo*
- *Memento guarda um snapshot no estado interno de outro objeto - a Fonte*
  - *Um mecanismo de Undo irá requisitar um memento da fonte quando ele necessitar verificar o estado desse objeto*
  - *A fonte reinicializa o memento com informações que caracterizam seu estado atual*
  - *Só a fonte tem permissão para recuperar informações do memento (o memento é "opaco" aos outros objetos)*

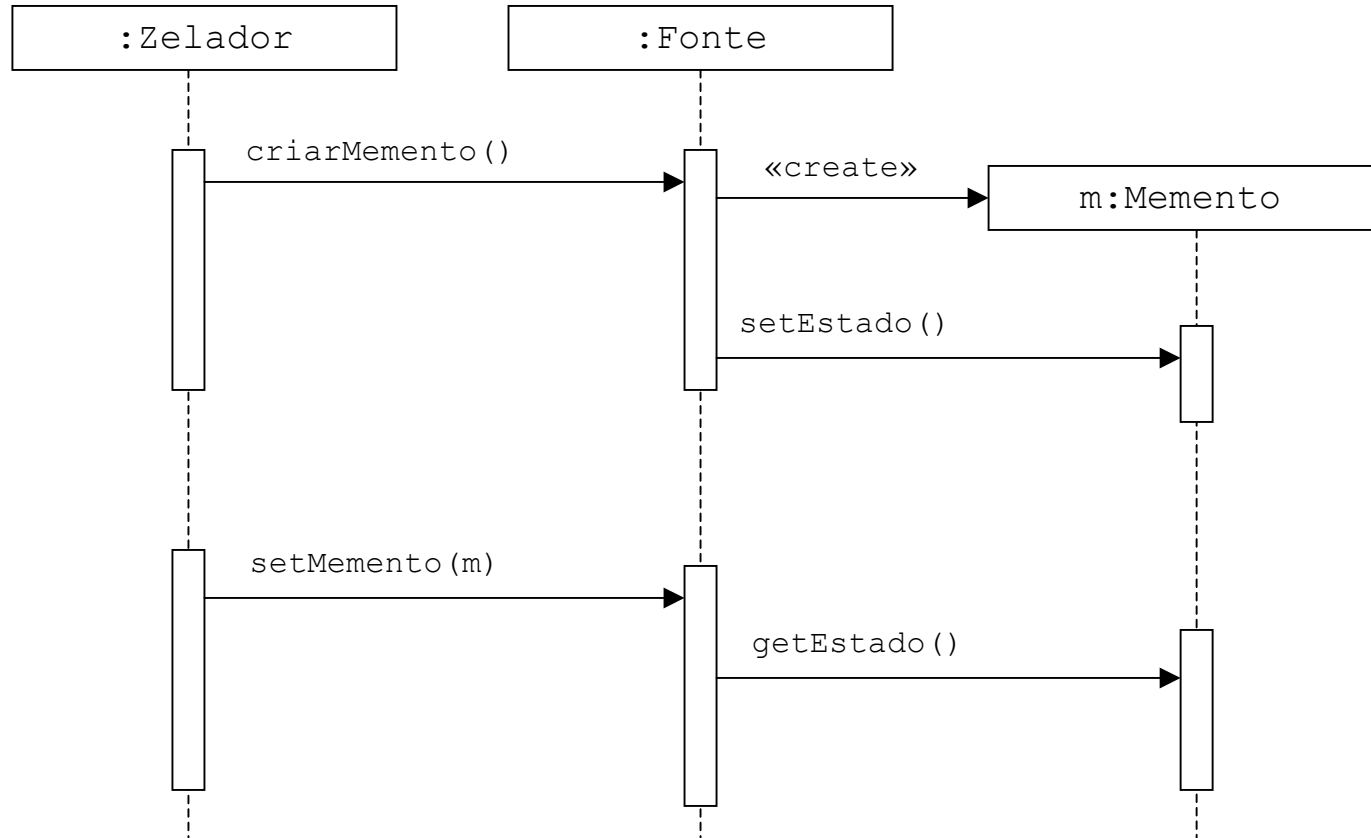
# Quando usar?

- *Use Memento quando*
  - *Um snapshot do (parte do) estado de um objeto precisa ser armazenada para que ele possa ser restaurado ao seu estado original posteriormente*
  - *Uma interface direta para se obter esse estado iria expor detalhes de implementação e quebrar o encapsulamento do objeto*

# Estrutura de Memento



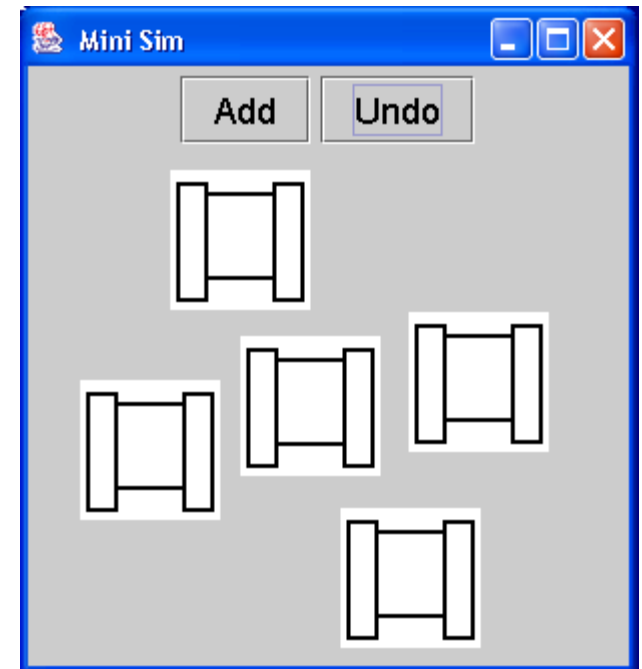
# Seqüência





## Exemplo:

- Rode o simulador
  - `exemplos/oozinoz/visualization.bat`
- Cada vez que um objeto for criado ou movido, o sistema criará um memento do objeto e o guardará em uma pilha
- Cada vez que o usuário clicar no botão Undo, o código irá recuperar o memento mais recente e restaurar a simulação ao estado armazenado no topo da pilha



## Exemplo genérico

```
package memento;

public class Fonte {
    private Memento memento;
    private Object estado;
    public Memento criarMemento() {
        return new Memento();
    }
    public void setMemento(Memento m) {
        memento = m;
    }
}
```

```
package memento;

public class Memento {
    private Object estado;
    Memento() { }
    void setEstado(Object estado) {
        this.estado = estado;
    }
    Object getEstado() {
        return estado;
    }
}
```

## Memento em Java

- *Veja exemplos do simulador no pacote com.oozinoz.visualization*
  - *Visualization*
  - *Visualization2 (usa armazenamento persistente)*

- *Memento permite capturar o estado de um objeto para que seja possível recuperá-lo posteriormente*
- *O meio de armazenamento utilizado depende de quando o objeto terá que ser recuperado e dos riscos envolvidos na não recuperação*
- *A aplicação mais comum de memento é o suporte a operações de Undo.*

# Exercícios

- *15.1 Escreva uma aplicação gráfica simples que permite digitar texto em um TextField que é copiado para um TextArea (um objeto em cada linha) quando o usuário aperta o botão gravar. Em seguida o TextField é esvaziado*
  - *Crie um botão "Desfazer"*
  - *Implemente uma operação de Undo que permita desfazer todas as operações (recuperar o texto anterior no TextField e mostrar o TextArea sem o texto)*
- *15.2 Grave as alterações em disco de forma que, se a aplicação fechar, quando ela reiniciar, ela "lembre" do estado em que estava antes de fechar.*

# Resumo: Quando usar?

- **Builder**
  - Para construir objetos complexos em várias etapas e/ou que possuem representações diferentes
- **Factory Method**
  - Para isolar a classe concreta do produto criado da interface usada pelo cliente
- **Abstract Factory**
  - Para criar famílias inteiras de objetos que têm algo em comum sem especificar suas interfaces.
- **Prototype**
  - Para criar objetos usando outro como base
- **Memento**
  - Para armazenar o estado de um objeto sem quebrar o encapsulamento. O uso típico deste padrão é na implementação de operações de Undo.

## *1. Descreva a diferença entre*

- *Factory Method e Abstract Factory*
- *Builder e Chain of Responsibility*
- *Factory Method e Façade*

# Fontes

- [1] Steven John Metsker, *Design Patterns Java Workbook*. Addison-Wesley, 2002, Caps. 14 a 19. *Exemplos em Java, diagramas em UML e exercícios sobre Builder, Abstract Factory, Factory Method, Prototype e Memento.*
- [2] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley, 1995. *Builder, Abstract Factory, Factory Method, Memento, Prototype. Referência com exemplos em C++ e Smalltalk.*

# *Curso J930: Design Patterns*

*Versão 1.0*

*[www.argonavis.com.br](http://www.argonavis.com.br)*

© 2003, *Helder da Rocha*  
(*helder@acm.org*)