

Padrões de projeto de software orientado a objetos

Template, Factory, Memento e Prototype

Profa. Thienne Johnson

Conteúdo

- ▶ E. Gamma and R. Helm and R. Johnson and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- ▶ GoF Design Patterns – with examples using Java and UML2. Christiansson et al
 - Licença CreativeCommons
- ▶ Software Architecture Design Patterns in Java .
 - Partha Kuchana. AUERBACH PUBLICATIONS, 2004

Design Patterns

Elements of Reusable
Object-Oriented Software

Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides



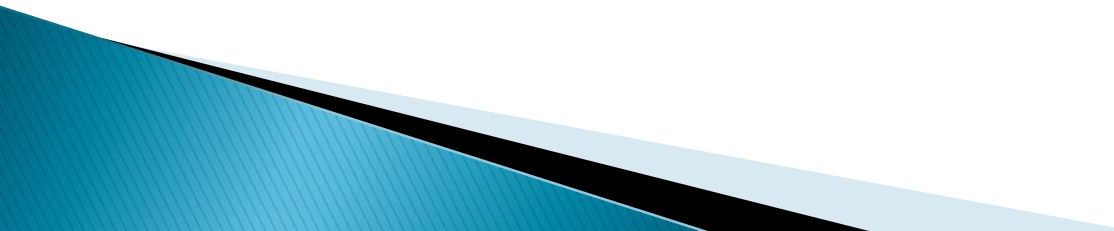
Foreword by Grady Booch



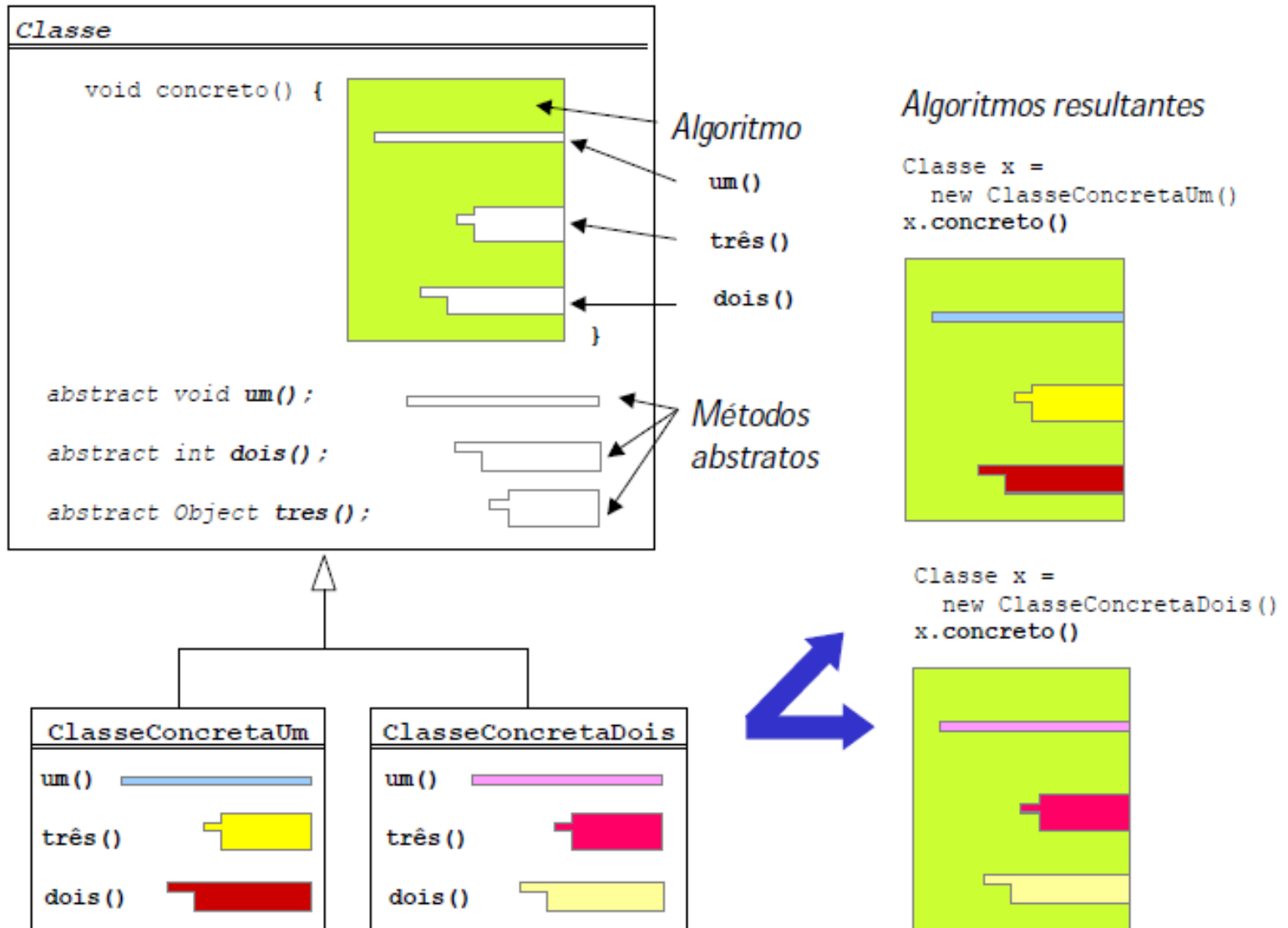
Template

Padrão comportamental

Intenção

- ▶ Define o ‘esqueleto’ de um algoritmo em operação, transferindo alguns sub-passos para as sub-classes.
 - ▶ Permite que as subclasses redefinam certos passos de um algoritmo sem mudar a estrutura do algoritmo.
- 

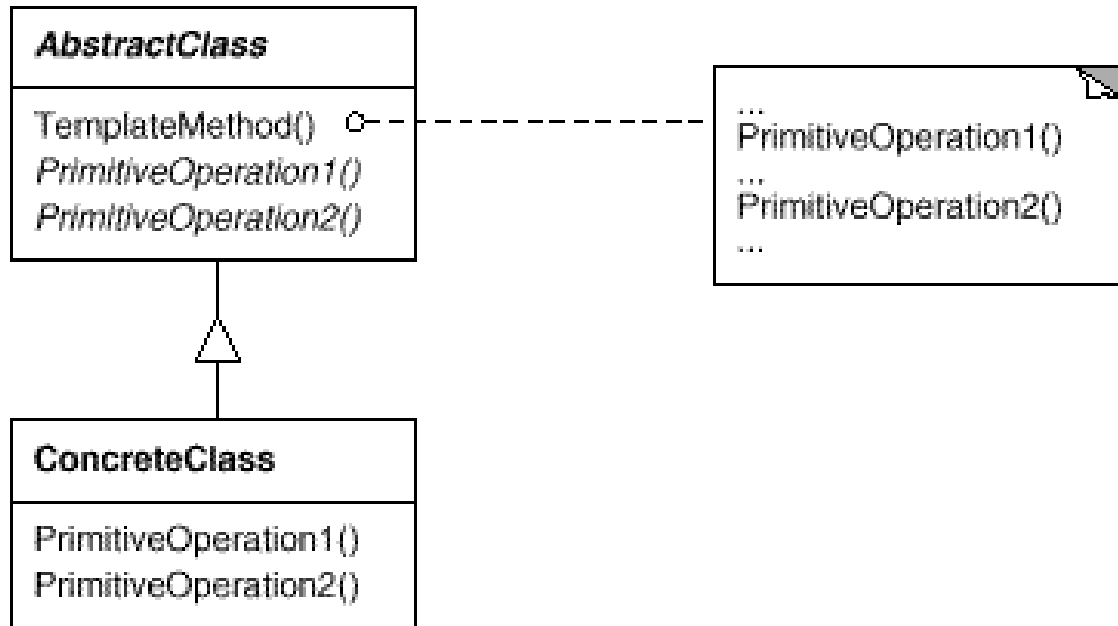
Motivação



Aplicabilidade

- Quando a estrutura fixa de um algoritmo puder ser definida pela superclasse deixando certas partes para serem preenchidos por implementações que podem variar

Estrutura



Participantes

▶ **AbstractClass**

- Define operações abstratas primitivas que as subclasses concretas definem para implementar passos de um algoritmo.
- Implementa um método template definindo o esqueleto do algoritmo.
 - O método template chama as operações primitivas e as operações definidas na Classe Abstrata ou operações de outros objetos.

▶ **ConcreteClass**

- Implementa as operações primitivas para executar os passos específicos da subclasse no algoritmo.

Exemplo 1

```
public abstract class Template {  
    protected abstract String link(String texto, String url);  
    protected String transform(String texto) { return texto; }  
    public final String templateMethod() {  
        String msg = "Endereço: " + link("Empresa", "http://www.empresa.com");  
        return transform(msg);  
    }  
}
```

```
public class XMLData extends Template {  
    protected String link(String texto, String url) {  
        return "<endereco xlink:href=\"" + url + "\">" + texto + "</endereco>";  
    }  
}
```

```
public class HTMLData extends Template {  
    protected String link(String texto, String url) {  
        return "<a href=\"" + url + "\">" + texto + "</a>";  
    }  
    protected String transform(String texto) {  
        return texto.toLowerCase();  
    }  
}
```

Exemplo 2

- ▶ O método `Arrays.sort (java.util)` é um bom exemplo de Template Method. Ele recebe como parâmetro um objeto do tipo `Comparator` que implementa um método `compare(a, b)` e utiliza-o para definir as regras de ordenação

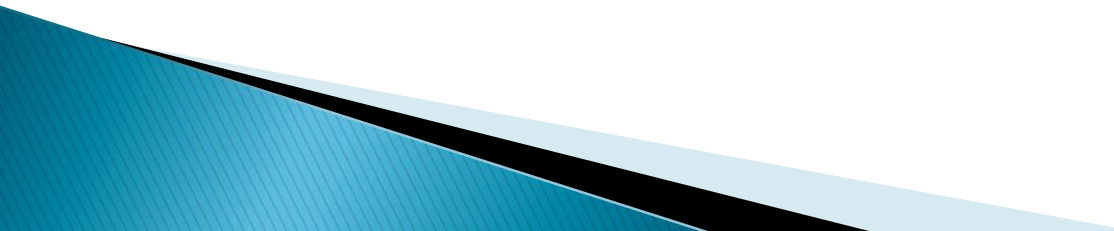
```
public class MedeCoisas implements Comparator {
    public int compare(Object o1, Object o2) {
        Coisa c1 = (Coisa) o1;
        Coisa c2 = (Coisa) o2;
        if (c1.getID() > c2.getID()) return 1;
        if (c1.getID() < c2.getID()) return -1;
        if (c1.getID() == c2.getID()) return 0;
    }
}

...
Coisa coisas[] = new Coisa[10];
coisas[0] = new Coisa("A");
coisas[1] = new Coisa("B");
...
Arrays.sort(coisas, new MedeCoisas());
...
```

Factory

Padrão de criação

Intenção

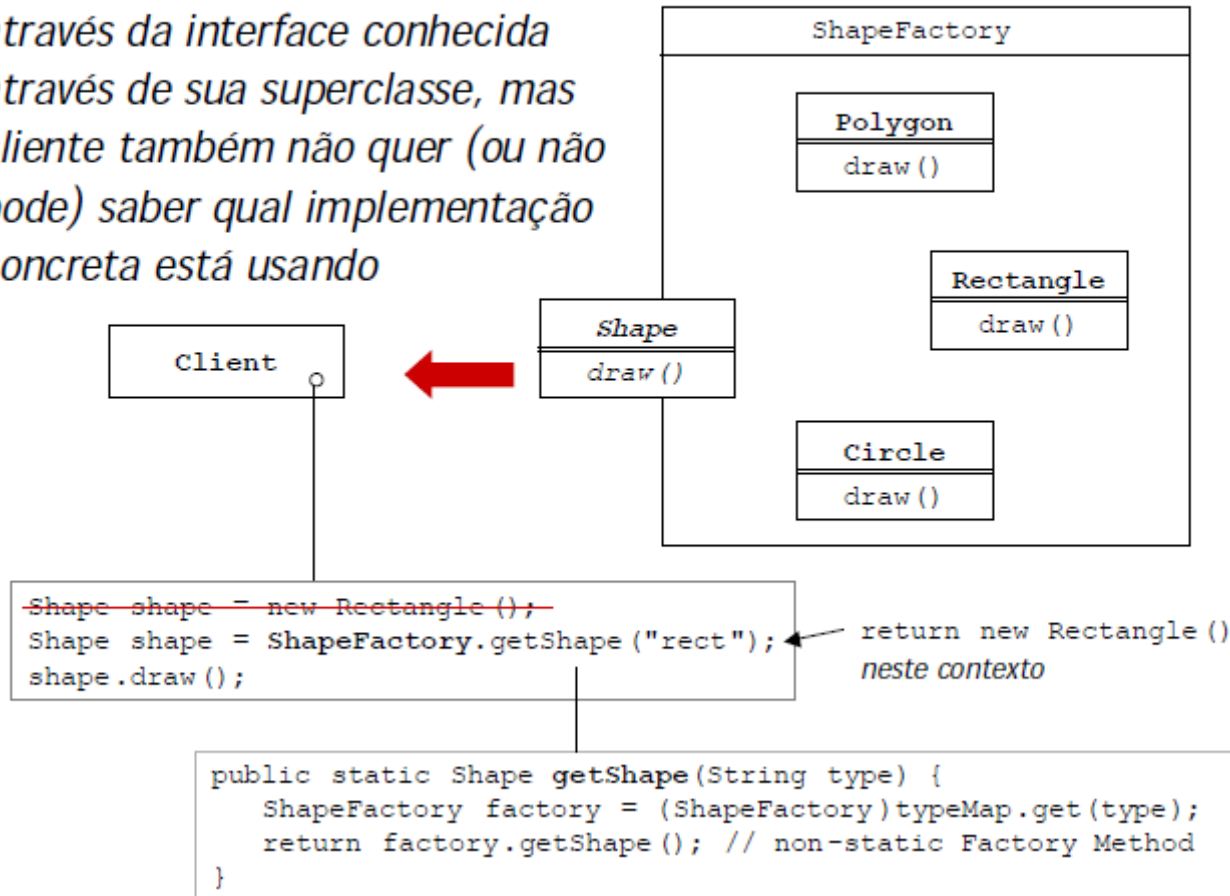
- ▶ Define uma interface de criação de um objeto mas permite que subclasses decidam qual classe instanciar.
 - ▶ Transfere a instanciação para subclasses.
- 

Definição

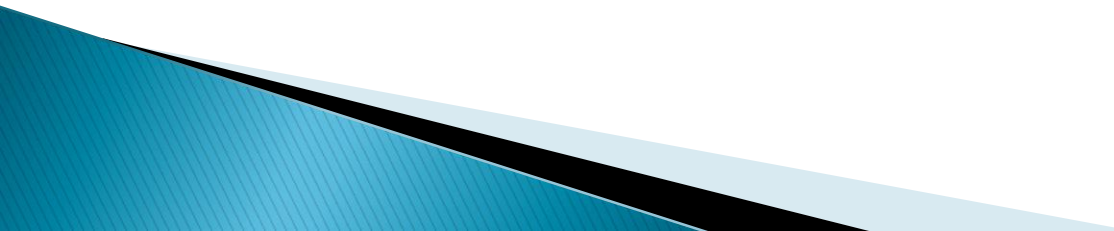
- ▶ Fornece uma forma de usar uma instância como fábrica de objetos.
- ▶ A fábrica pode retornar uma instância de uma de várias classes possíveis (em uma hierarquia de subclasses), dependendo dos dados fornecidos à fábrica.
- ▶ **Também conhecido como**
 - Virtual Constructor

Motivação

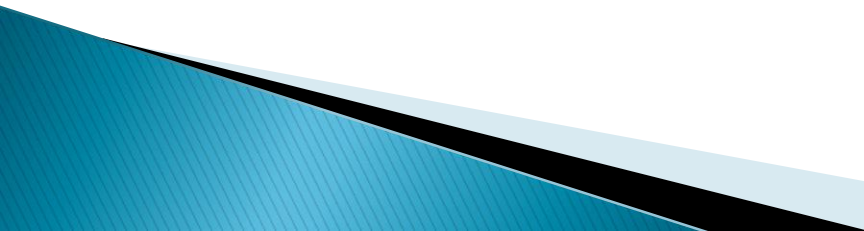
O acesso a um objeto concreto será através da interface conhecida através de sua superclasse, mas cliente também não quer (ou não pode) saber qual implementação concreta está usando



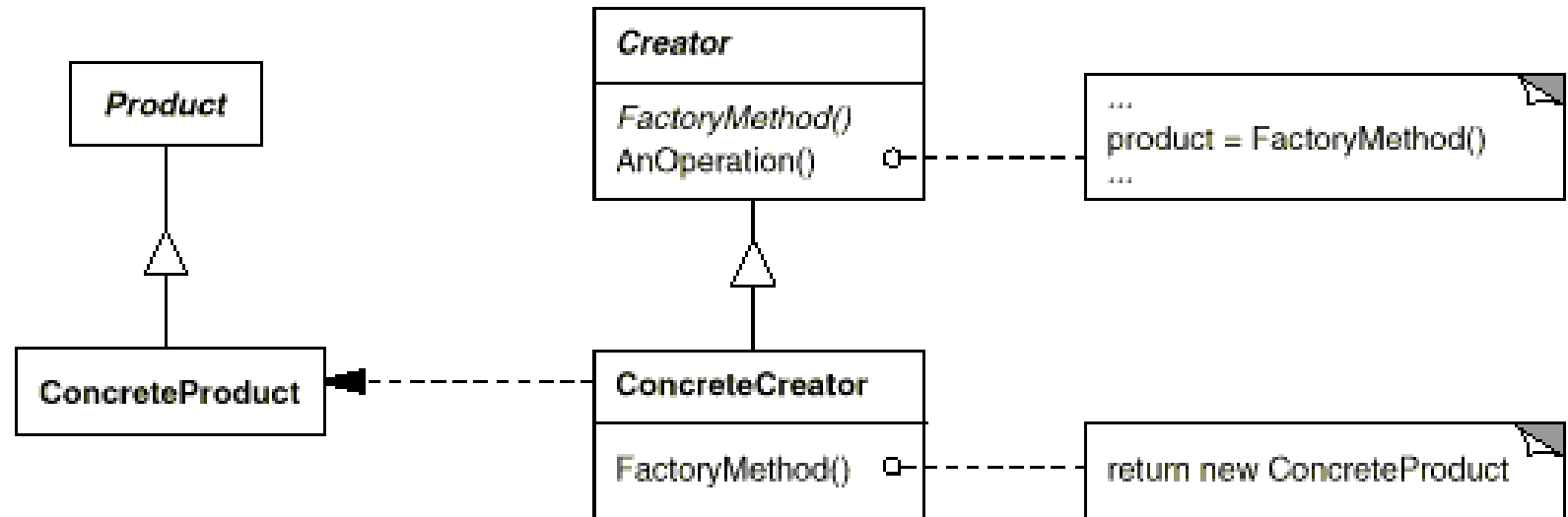
Aplicabilidade

- Quando uma classe não pode antecipar qual tipo de classe que um objeto deve criar.
 - Quando temos classes que são derivadas das mesmas subclasses, ou elas podem ser de fato não relacionadas mas compartilham a mesma interface.
 - De qualquer forma, os métodos das instâncias são os mesmos e podem ser usados de forma intercambiável.
 - Quando queremos isolar o cliente do tipo atual que está sendo instanciado.
- 

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



Participantes

▶ **Product**

- Define a interface de objetos que a fábrica cria.

▶ **ConcreteProduct**

- Implementa a interface Produto.

▶ **Creator**

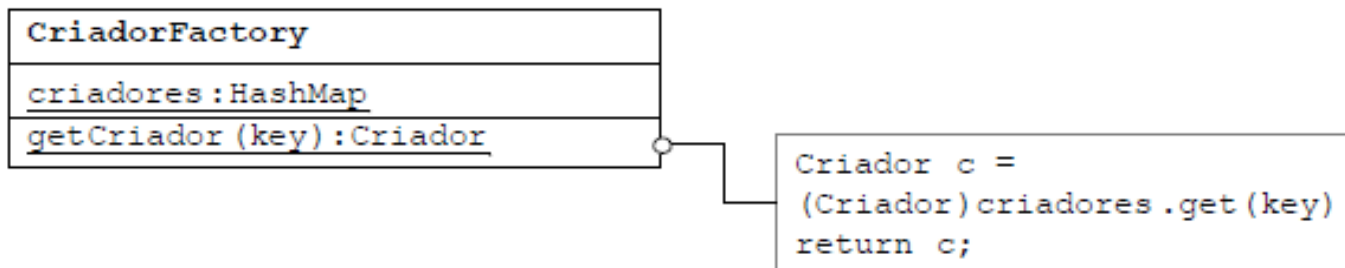
- Declara o método de fábrica, que retorna um objeto do tipo Produto.
- Pode também definir uma implementação padrão do método de fábrica que retorna um objeto padrão ConcreteProduct
- Pode chamar o método de fábrica para criar o objeto Produto.

▶ **ConcreteCreator**

- Sobrescreve o método de fábrica para retornar uma instância de um ConcreteProduct.

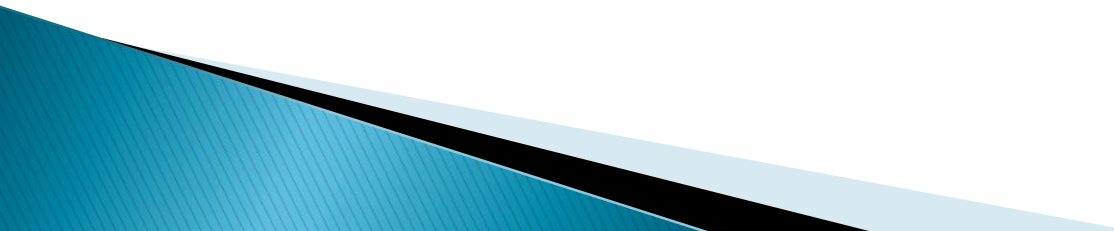
Como selecionar o criador

- ▶ 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

Benefícios

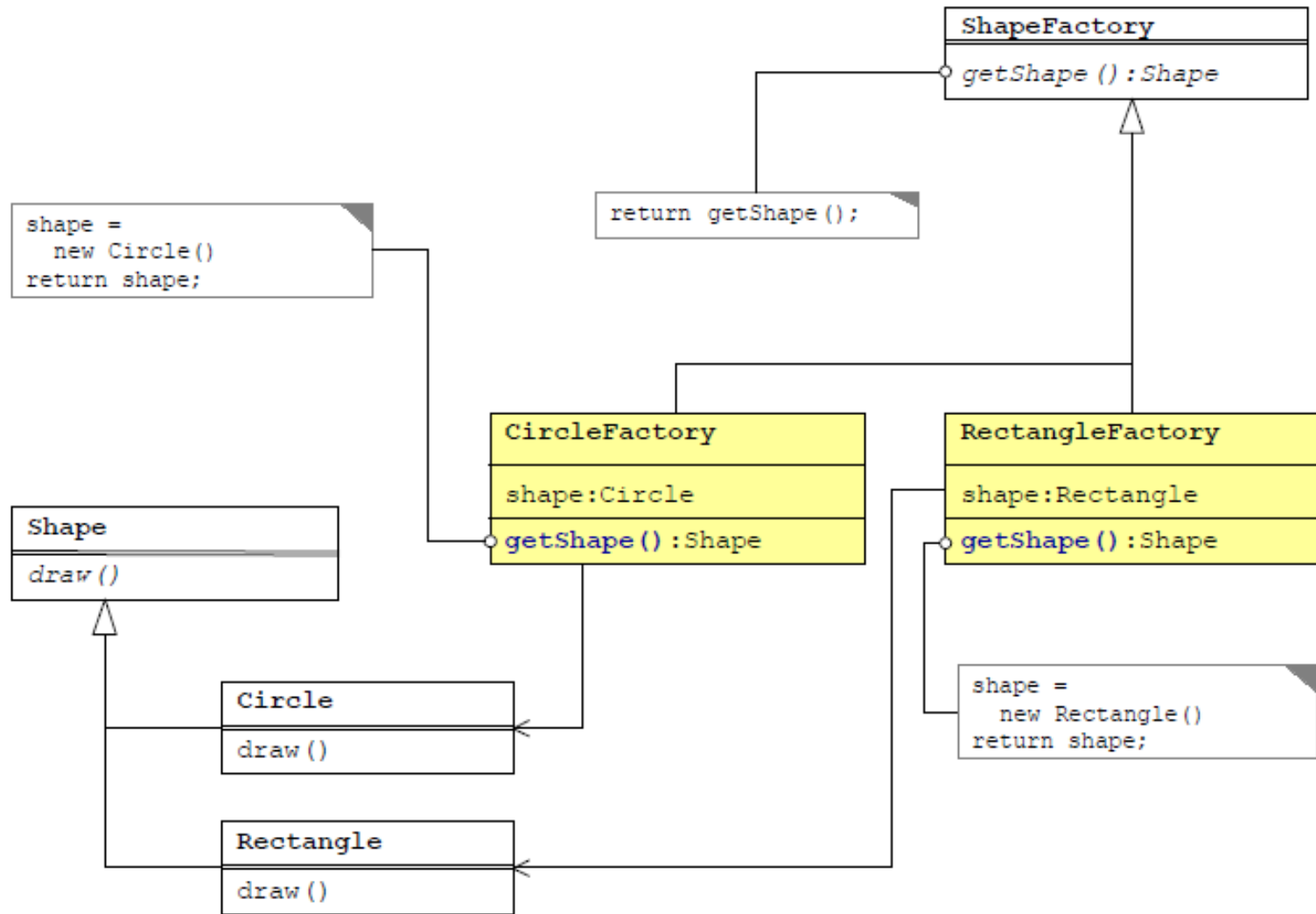
- ▶ O cliente não precisa conhecer todas as subclasses de um objeto que deve criar
 - Só necessita de uma referência a classe/interface e o objeto fábrica.
 - ▶ A fábrica encapsula a criação de objetos. Pode ser útil se o processo de criação é complexo.
- 

Conseqüências

- ▶ Não existe forma de mudar a implementação de uma classe sem recompilação.
- ▶ 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();  
    } ...  
}
```

Exemplo 1



Exemplo 2

```
public abstract class Product {  
    public void writeName(String name) {  
        System.out.println("My name is "+name);  
    }  
}
```

```
public class ProductA extends Product { }
```

```
public class ProductB extends Product {  
    public void writeName(String name) {  
        StringBuilder tempName = new StringBuilder().append(name);  
        System.out.println("My reversed name is"+tempName.reverse());  
    }  
}
```

Exemplo 2

```
public class ProductFactory {
    Product createProduct(String type) {
        if(type.equals("B"))
            return new ProductB();
        else
            return new ProductA();
    }
}
```

```
public class TestClientFactory {
    public static void main(String[] args) {
        ProductFactory pf = new ProductFactory();
        Product prod;
        prod = pf.createProduct("A");
        prod.writeName("John Doe");
        prod = pf.createProduct("B");
        prod.writeName("John Doe");
    }
}
```


Memento

Padrão comportamental

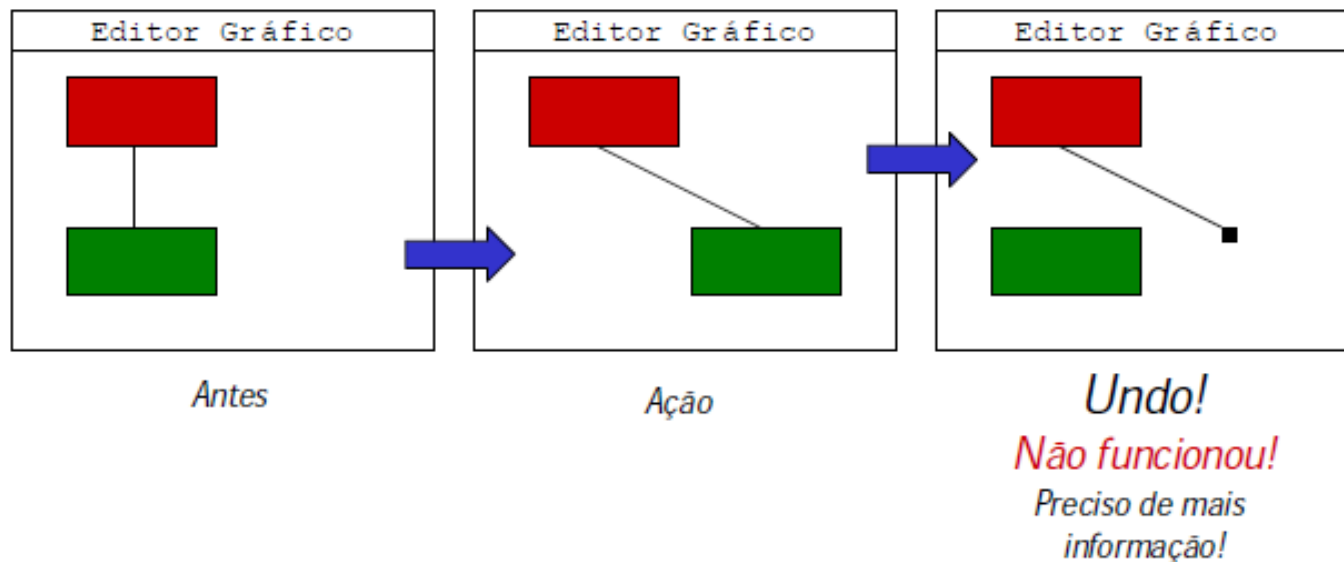
Intenção

- ▶ Para gravar o estado interno de um objeto sem violar encapsulamento e recuperar depois sem conhecer o objeto original.
- ▶ Um memento é um objeto que armazena um *instantâneo* do estado interno de outro objeto.
- ▶ Pode ser usado para fazer o objeto retornar a um estado anterior

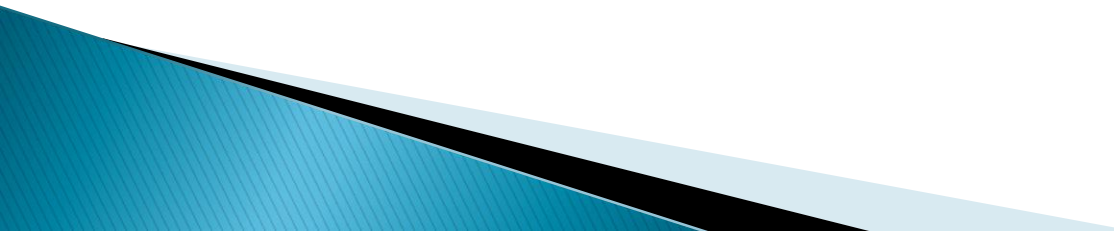
- ▶ **Também conhecido como**
 - Token

Motivação

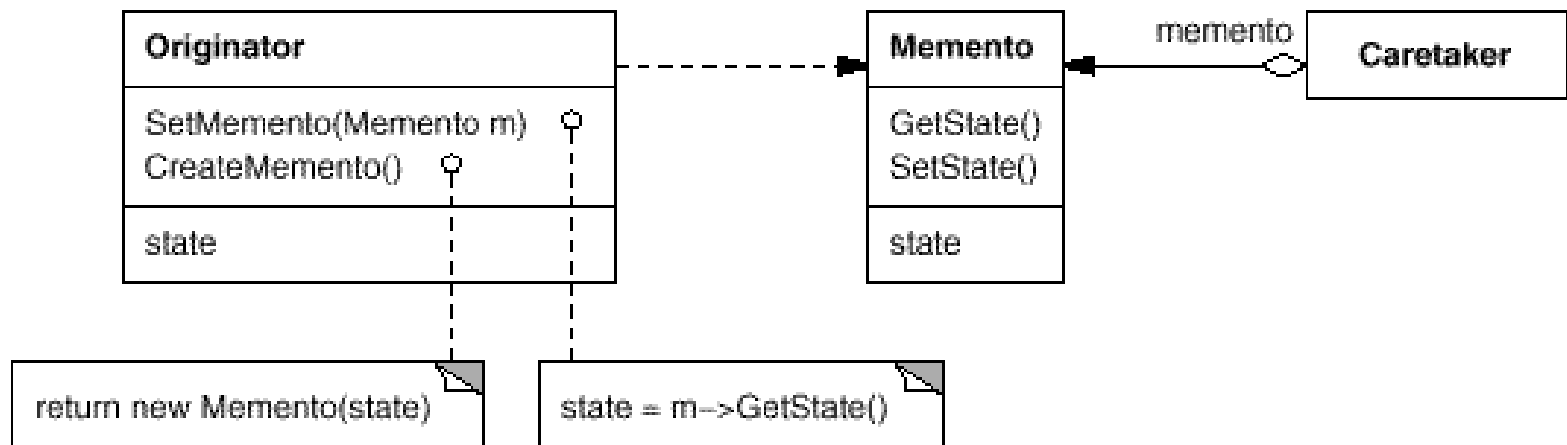
- *É preciso guardar informações sobre um objeto suficientes para desfazer uma operação, mas essas informações não devem ser públicas*



Aplicabilidade

- ▶ Quando deixamos algumas informações de o objeto disponível para outro objeto.
 - ▶ Quando queremos criar instantâneos de um estado para um objeto.
 - ▶ Quando precisamos de operações *undo/redo*.
- 

Estrutura



Participantes

▶ Memento

- Armazena o estado de um objeto Originador.
- Pode armazenar muitas ou poucas informações do estado interno do Originador – quanto for necessários para a descrição do Originador.
- Protege contra acesso de objetos a não ser o Originador

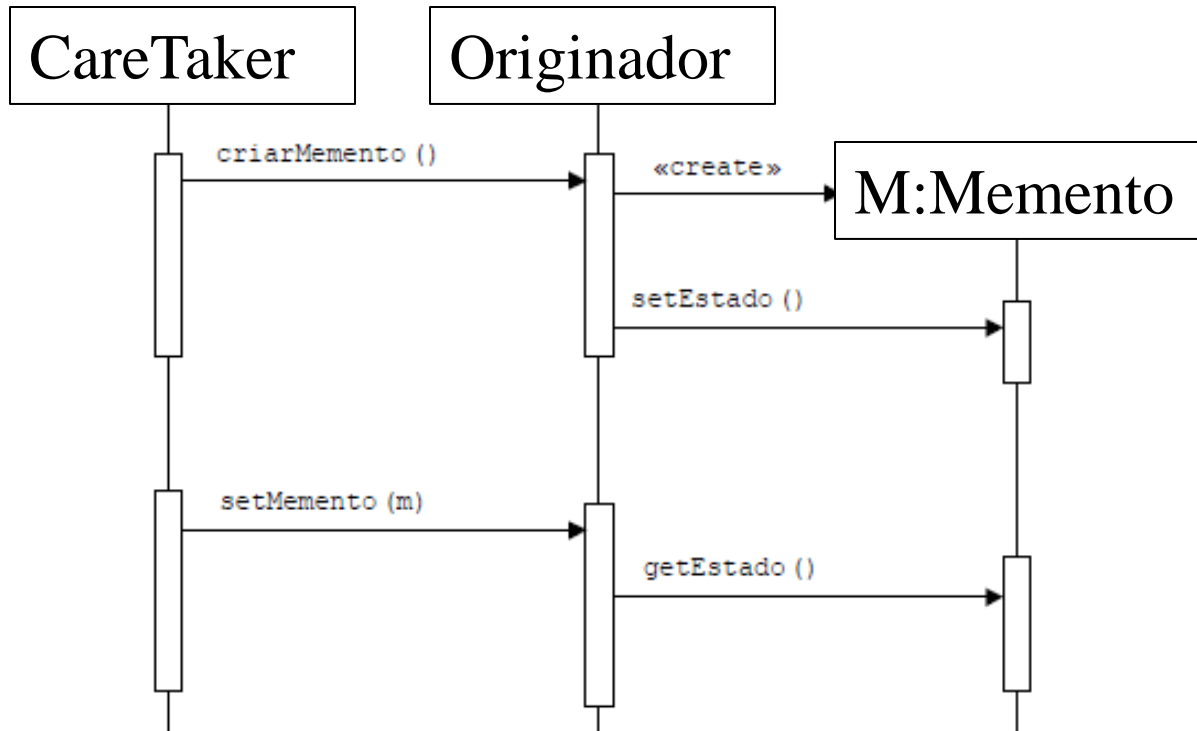
▶ Originator

- Cria um memento contendo um instantâneo de seu estado interno atual.
- Usa o memento para restaurar o seu estado interno.

▶ Caretaker

- É responsável pela segurança do memento.
- Nunca executa ou examina o conteúdo de um memento.

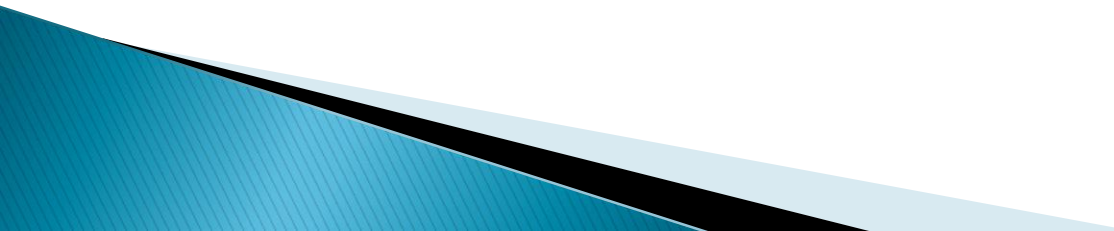
Seqüência



Benefícios

- ▶ Habilidade de restaurar um objeto a seu estado anterior.

Conseqüências

- ▶ Deve ser tomando cuidado se o Originator pode mudar outros objetos ou recursos – o padrão Memento opera em um único objeto.
 - ▶ Usar memento para armazenar grandes quantidades de dados do Originator pode ser caro se os clientes criam e retornam mementos freqüentemente.
- 

Exemplo 1

```
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;  
    }  
}
```

Exemplo 2

```
public class Originator {
    private String state;
    /* lots of memory using private data that does not have to be
    saved. Instead we use a small memento object. */

    public void set(String state) {
        System.out.println("Originator: Setting state to " + state);
        this.state = state;
    }

    public Object saveToMemento() {
        System.out.println("Originator: Saving to Memento.");
        return new Memento(state);
    }

    public void restoreFromMemento(Object m) {
        if (m instanceof Memento) {
            Memento memento = (Memento) m;
            state = memento.getSavedState();
            System.out.println("Originator: State after restoring
            from Memento: " + state);
        }
    }
}
```

Exemplo 2 (cont.)

```
import java.util.ArrayList;
import java.util.List;
public class Caretaker {
    private List<Object> savedStates = new ArrayList<Object>();
    public void addMemento(Object m) {
        savedStates.add(m);
    }
    public Object getMemento(int index) {
        return savedStates.get(index);
    }
}

public class Memento {
    private String state;
    public Memento(String stateToSave) {
        state = stateToSave;
    }
    public String getSavedState() {
        return state;
    }
}
```

Exemplo 2 (cont.)

```
public class MementoExample {  
    public static void main(String[] args) {  
        Caretaker caretaker = new Caretaker();  
        Originator originator = new Originator();  
        originator.set("State1");  
        originator.set("State2");  
        caretaker.addMemento(originator.saveToMemento());  
        originator.set("State3");  
        caretaker.addMemento(originator.saveToMemento());  
        originator.set("State4");  
        originator.restoreFromMemento(caretaker.getMemento(1));  
    }  
}
```

Prototype

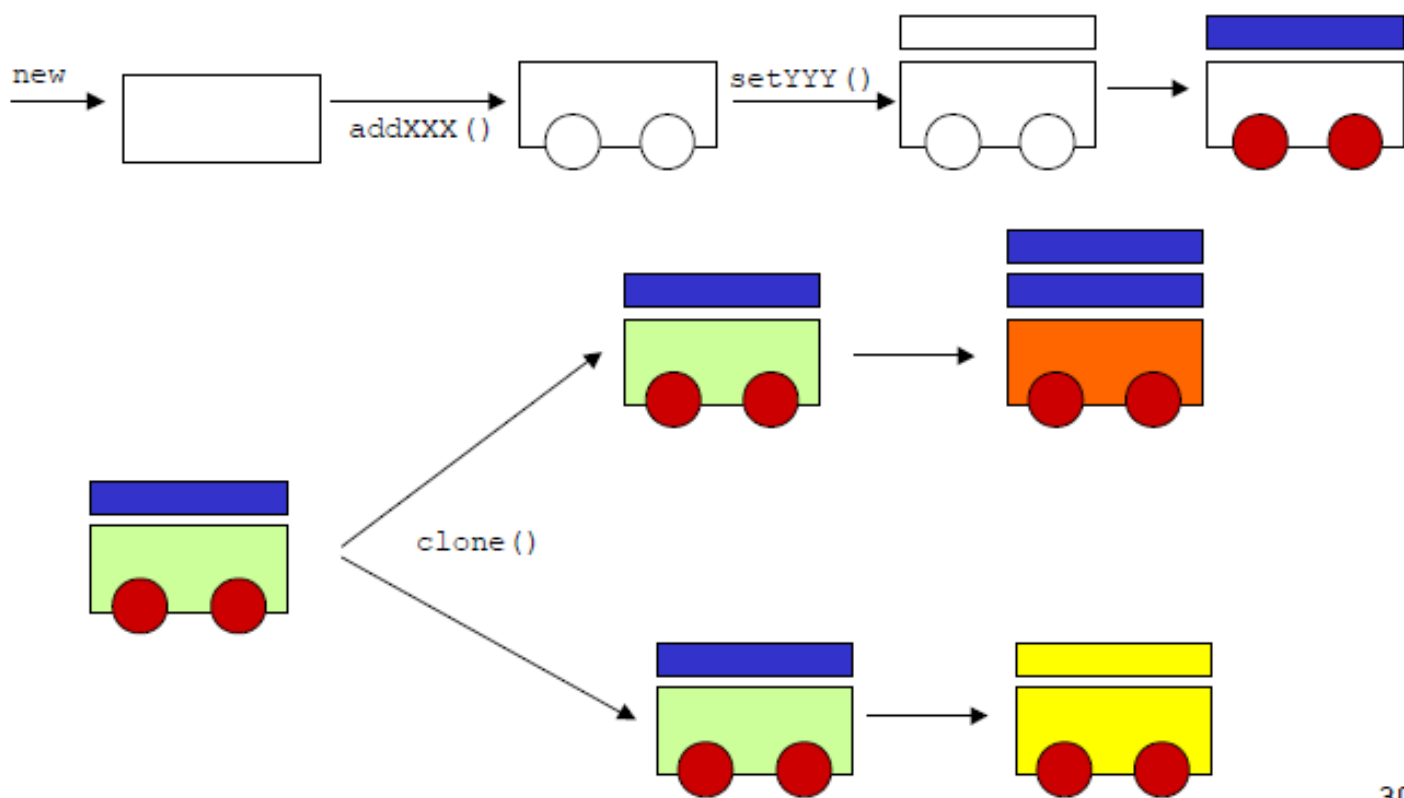
Padrão de criação

Intenção

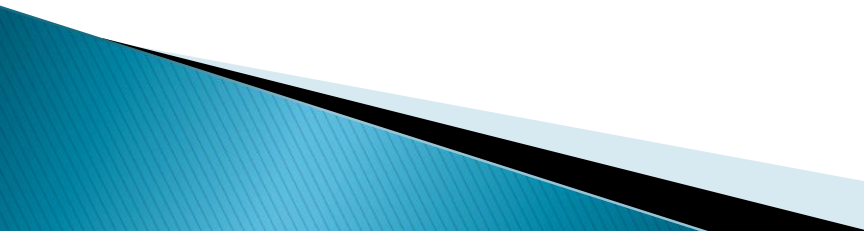
- ▶ Especifica os tipos de objetos a serem criados usando instâncias protótipos e novos objetos são criados ao copiar o protótipo.

Motivação

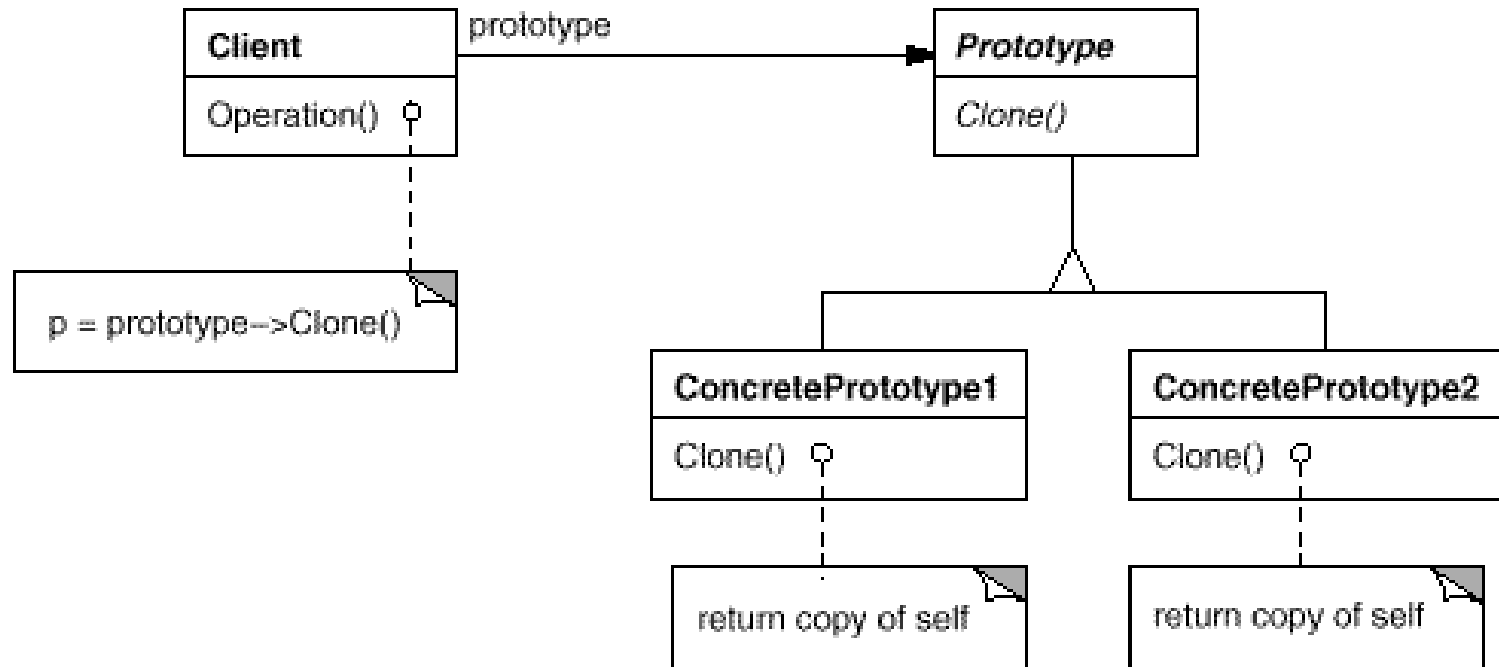
- *Criar um objeto novo, mas aproveitar o estado previamente existente em outro objeto*



Aplicabilidade

- ▶ Quando um sistema precisa ser independente de como seus objetos são criados, compostos e representados.
 - ▶ Quando queremos adicionar e remover objetos em tempo de execução.
 - ▶ Quando queremos especificar novos objetos ao mudar uma estrutura de objetos existentes.
 - ▶ Quando queremos configurar uma aplicação com classes dinamicamente.
 - ▶ Quando tentamos manter o número de classes de um sistema a um mínimo.
- 

Estrutura



Participantes

▶ **Prototype**

- Declara uma interface para se auto-clonar.

▶ **ConcretePrototype**

- Implementa uma operação de auto-clonagem.

▶ **Cliente**

- Cria um novo objeto ao pedir que um protótipo se clone.

Benefícios

- ▶ Acelera a instanciação de classes grandes e dinamicamente carregadas.
- ▶ Reduz subclassing

Conseqüências

- ▶ Cada subclasse de um Prototype deve implementar a operação Clone.
- ▶ Pode ser difícil com classes existentes com objetos internos e referências circulares ou que não dão suporte para cópia.

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`
 - sobrepor `clone()` da seguinte forma:

```
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;
        }
    }
}
```

Exemplo 1

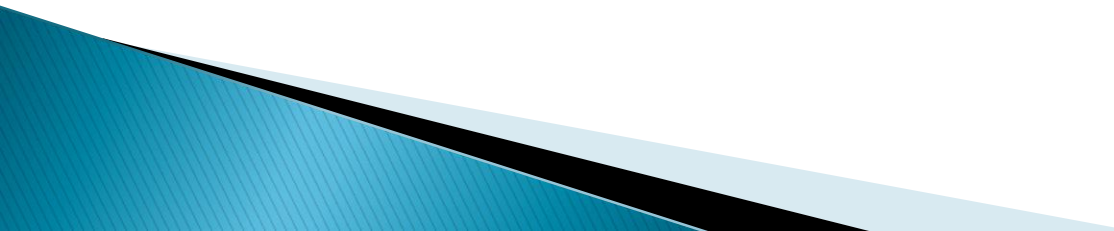
```
import java.util.Hashtable;
public class PrototypeExample {
    Hashtable<String, Product> productMap = new Hashtable<String,Product>();

    public Product getProduct(String productCode) {
        Product cachedProduct =(Product)productMap.get(productCode);
        return (Product)cachedProduct.clone();
    }
    public void loadCache() {
        //for each product run expensive query and instantiate
        // product productMap.put(productKey, product);
        // for exemplification, we add only two products
        Book b1 = new Book();
        b1.setDescription("Oliver Twist");
        b1.setSKU("B1");
        b1.setNumberOfPages(100);
        productMap.put(b1.getSKU(), b1);
        DVD d1 = new DVD();
        d1.setDescription("Superman");
        d1.setSKU("D1");
        d1.setDuration(180);
        productMap.put(d1.getSKU(), d1);
    }
}
```


Exemplo 1 (cont.)

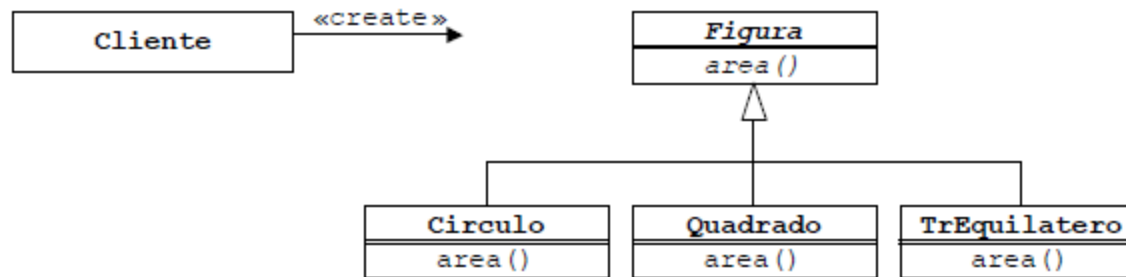
```
public static void main(String[] args) {  
    PrototypeExample pe = new PrototypeExample();  
    pe.loadCache();  
    Book clonedBook = (Book)pe.getProduct("B1");  
    System.out.println("SKU=" + clonedBook.getSKU());  
    System.out.println("SKU=" + clonedBook.getDescription());  
    System.out.println("SKU= " + clonedBook.getNumberOfPages());  
    DVD clonedDVD = (DVD)pe.getProduct("D1");  
    System.out.println("SKU = " + clonedDVD.getSKU());  
    System.out.println("SKU = " + clonedDVD.getDescription());  
    System.out.println("SKU = " + clonedDVD.getDuration());  
}  
}
```

Exercício 1

- ▶ Qual a diferença entre:
 - Factory Method e Façade
 - Template e Strategy
 - Memento e State
- 

Exercício 2

- ▶ Implemente a aplicação abaixo usando FactoryMethod para criar os objetos



- ▶ Crie um objeto construtor para cada tipo de objeto (XXXFactory para criar figuras do tipo XXX)
- ▶ Utilize a fachada Figuras que contém um HashMap, onde os construtores são guardados, e um método estático que seleciona o construtor desejado com uma chave