Análise e Projeto Orientados por Objetos

Aula 11 – Padrões GoF (Bridge e Decorator)

Edirlei Soares de Lima
<edirlei@iprj.uerj.br>

Padrões GoF

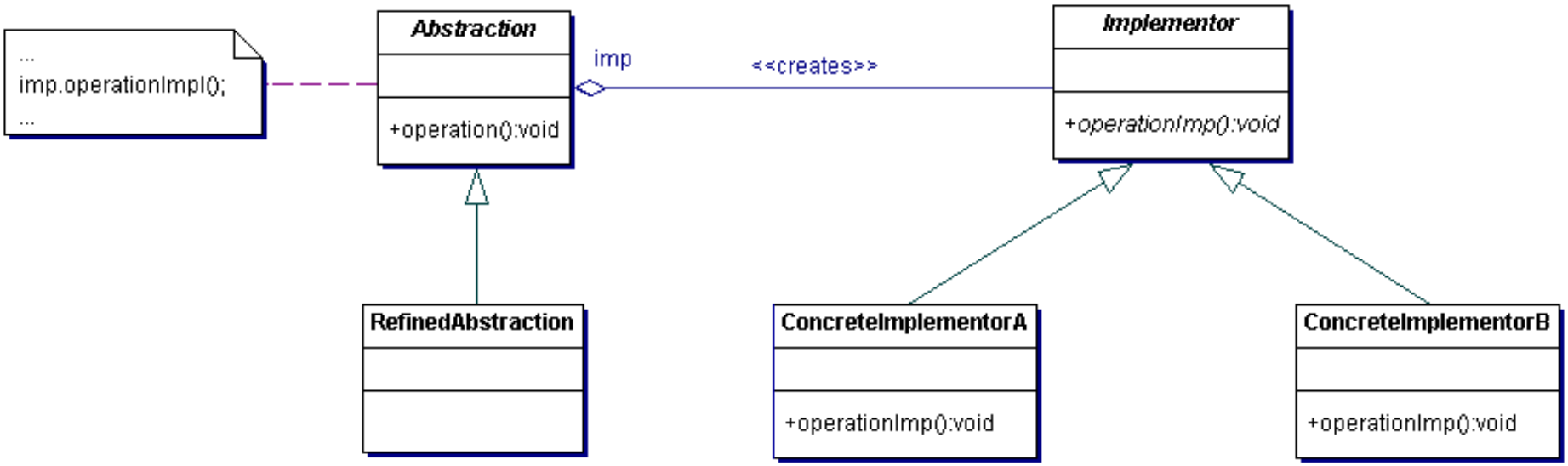
- Criação:
  - Abstract Factory
  - Builder
  - Factory Method
  - Prototype
  - Singleton
- Estruturais:
  - Adapter
  - **Bridge**
  - Composite
  - **Decorator**
  - Façade
  - Flyweight
  - Proxy
- Comportamentais:
  - Chain of Responsibility
  - Command
  - Interpreter
  - Iterator
  - Mediator
  - Memento
  - Observer
  - State
  - Strategy
  - Template Method
  - Visitor

Bridge

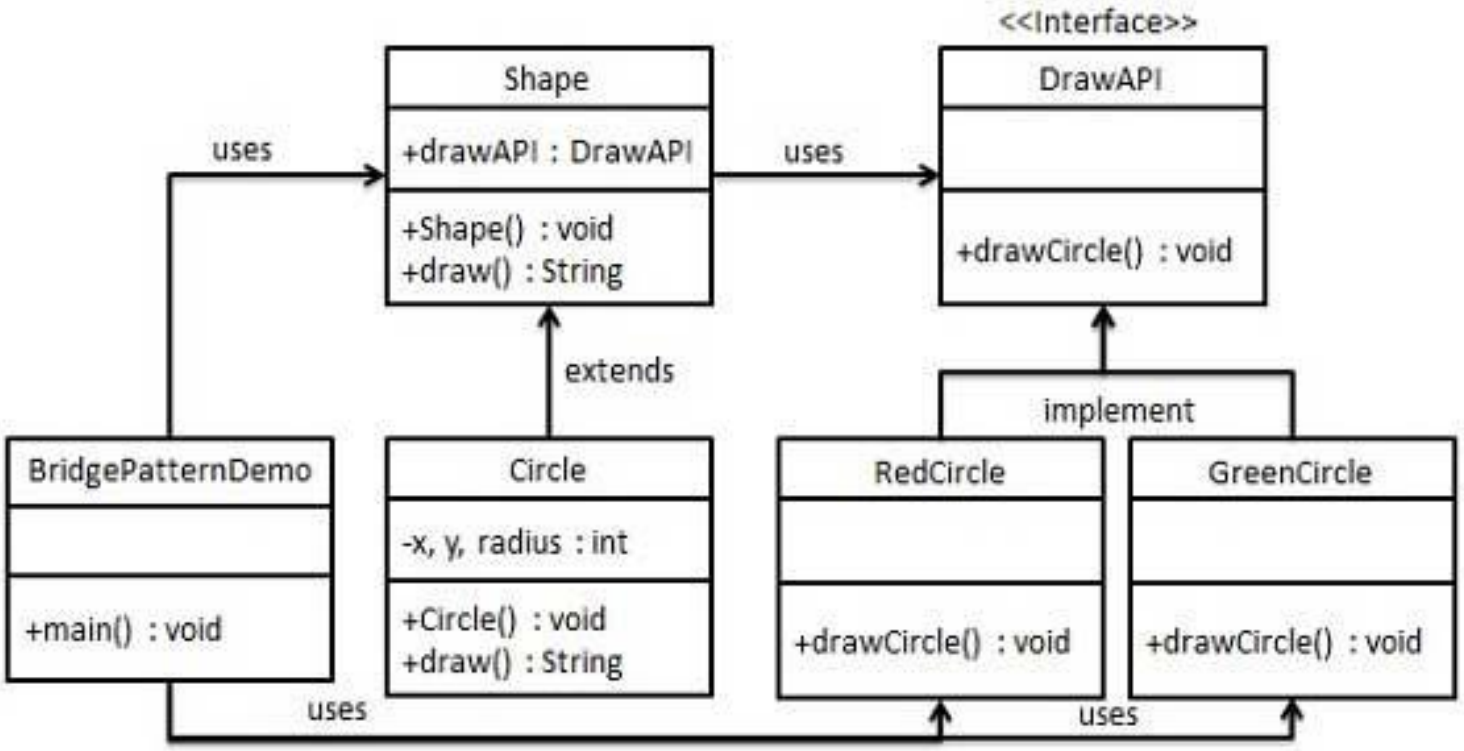
- **Intenção:** desacoplar uma **abstração** de sua **implementação** de tal forma que a implementação possa ser facilmente trocada.

- **Solução:** encapsular os detalhes de implementação em um objeto que é um componente da abstração.

Bridge


...
imp.operationImpl();
...
Abstraction
+operation():void
imp
<<creates>>
Implementor
+operationImp():void
RefinedAbstraction
ConcreteImplementorA
+operationImp():void
ConcreteImplementorB
+operationImp():void

Bridge – Exemplo



Bridge – Implementação

```
public interface DrawAPI {
   public void drawCircle(int radius, int x, int y);
}
```

```
public class RedCircle implements DrawAPI {
   @Override
   public void drawCircle(int radius, int x, int y) {
      System.out.println("Drawing Circle[ color: red, radius: " +
                         radius + ", x: " + x + ", " + y + "]");
   }
}
```

```
public class GreenCircle implements DrawAPI {
   @Override
   public void drawCircle(int radius, int x, int y) {
      System.out.println("Drawing Circle[ color: green, radius: "
                        + radius + ", x: " + x + ", " + y + "]");
   }
}
```

Bridge – Implementação

```
public abstract class Shape {
   protected DrawAPI drawAPI;
   protected Shape(DrawAPI drawAPI){
      this.drawAPI = drawAPI;
   }
   public abstract void draw();
}
```

```
public class Circle extends Shape {
   private int x, y, radius;
   public Circle(int x, int y, int radius, DrawAPI drawAPI) {
      super(drawAPI);
      this.x = x;
      this.y = y;
      this.radius = radius;
   }
   public void draw() {
      drawAPI.drawCircle(radius,x,y);
   }
}
```

Bridge – Implementação

```
public static void main(String[] args)
{
   Shape redCircle = new Circle(100,100, 10, new RedCircle());
   Shape greenCircle = new Circle(100,100, 10, new GreenCircle());

   redCircle.draw();
   greenCircle.draw();
}
```

Bridge – Aplicabilidade

- O padrão Bridge é útil quando se tem uma **abstração** que tem **diferentes implementações**.

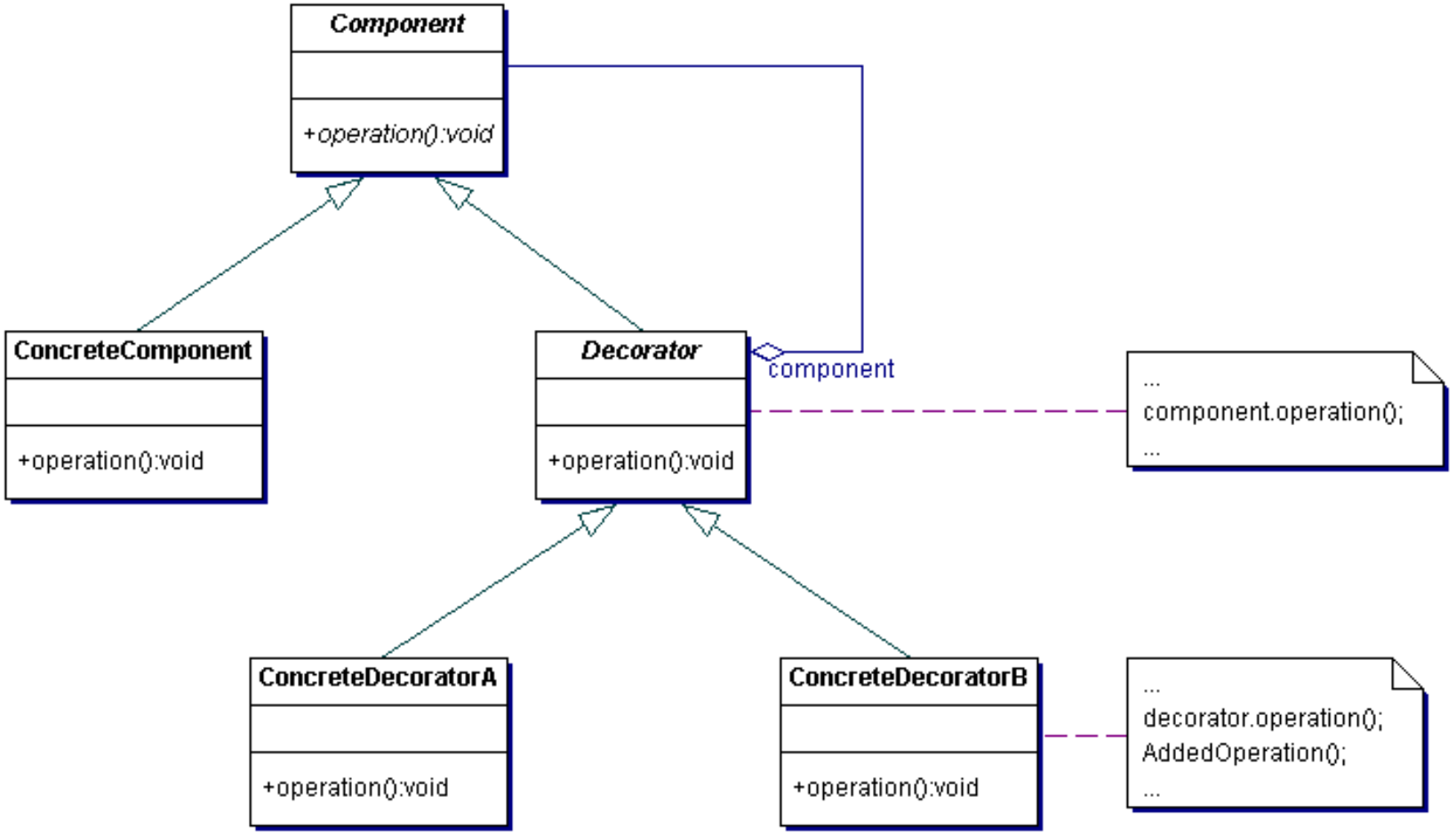
- Este padrão permite que a abstração e a sua implementação variem independentemente.

Decorator

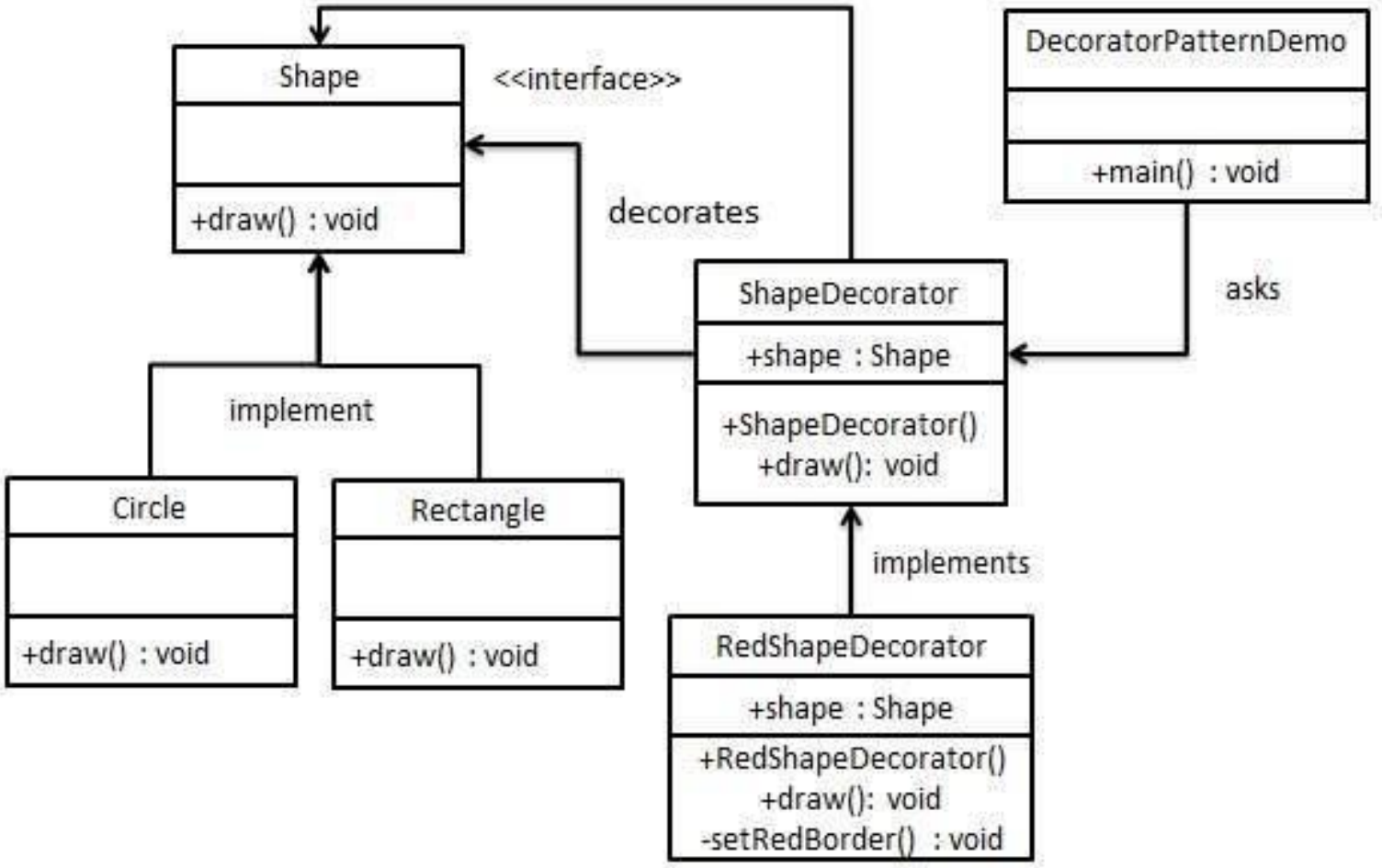
- **Intenção:** permitir que seja possível adicionar uma nova funcionalidade a um objeto existente sem alterar sua estrutura.
  - Funcionalidades devem ser adicionadas em instancias individuais e não na classe.

- **Solução:** criar uma classe que envolva a classe original, mantendo intacta a assinatura dos métodos e fornecendo funcionalidades adicionais.

Decorator


Component
+operation():void
ConcreteComponent
+operation():void
Decorator
+operation():void
component
...
component.operation();
...
ConcreteDecoratorA
+operation():void
ConcreteDecoratorB
+operation():void
...
decorator.operation();
AddedOperation();
...

Decorator – Exemplo



Decorator – Implementação

```
public interface Shape
{
   void draw();
}
```

```
public class Rectangle implements Shape
{
   @Override
   public void draw()
   {
      System.out.println("Shape: Rectangle");
   }
}
```

Decorator – Implementação

```
public class Circle implements Shape
{
   @Override
   public void draw()
   {
      System.out.println("Shape: Circle");
   }
}
```

```
public abstract class ShapeDecorator implements Shape {
   protected Shape decoratedShape;

   public ShapeDecorator(Shape decoratedShape){
      this.decoratedShape = decoratedShape;
   }

   public void draw(){
      decoratedShape.draw();
   }
}
```

Decorator – Implementação

```
public class RedShapeDecorator extends ShapeDecorator
{
   public RedShapeDecorator(Shape decoratedShape)
   {
      super(decoratedShape);
   }

   @Override
   public void draw()
   {
      decoratedShape.draw();
      setRedBorder(decoratedShape);
   }

   private void setRedBorder(Shape decoratedShape)
   {
      System.out.println("Border Color: Red");
   }
}
```

Decorator – Implementação

```
public static void main(String[] args)
{
   Shape circle = new Circle();

   Shape redCircle = new RedShapeDecorator(new Circle());

   Shape redRectangle = new RedShapeDecorator(new Rectangle());
   System.out.println("Circle with normal border");
   circle.draw();

   System.out.println("\nCircle of red border");
   redCircle.draw();

   System.out.println("\nRectangle of red border");
   redRectangle.draw();
}
```

Decorator – Aplicabilidade

- Aplicável quando é necessário acrescentar ou remover responsabilidades a objetos individuais dinamicamente, de forma transparente.

- Ajuda a evitar a explosão de subclasses para prover todas as combinações de responsabilidades.