Análise e Projeto Orientados por Objetos

Aula 13 – Padrões GoF (Proxy e Flyweight)

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

Padrões GoF

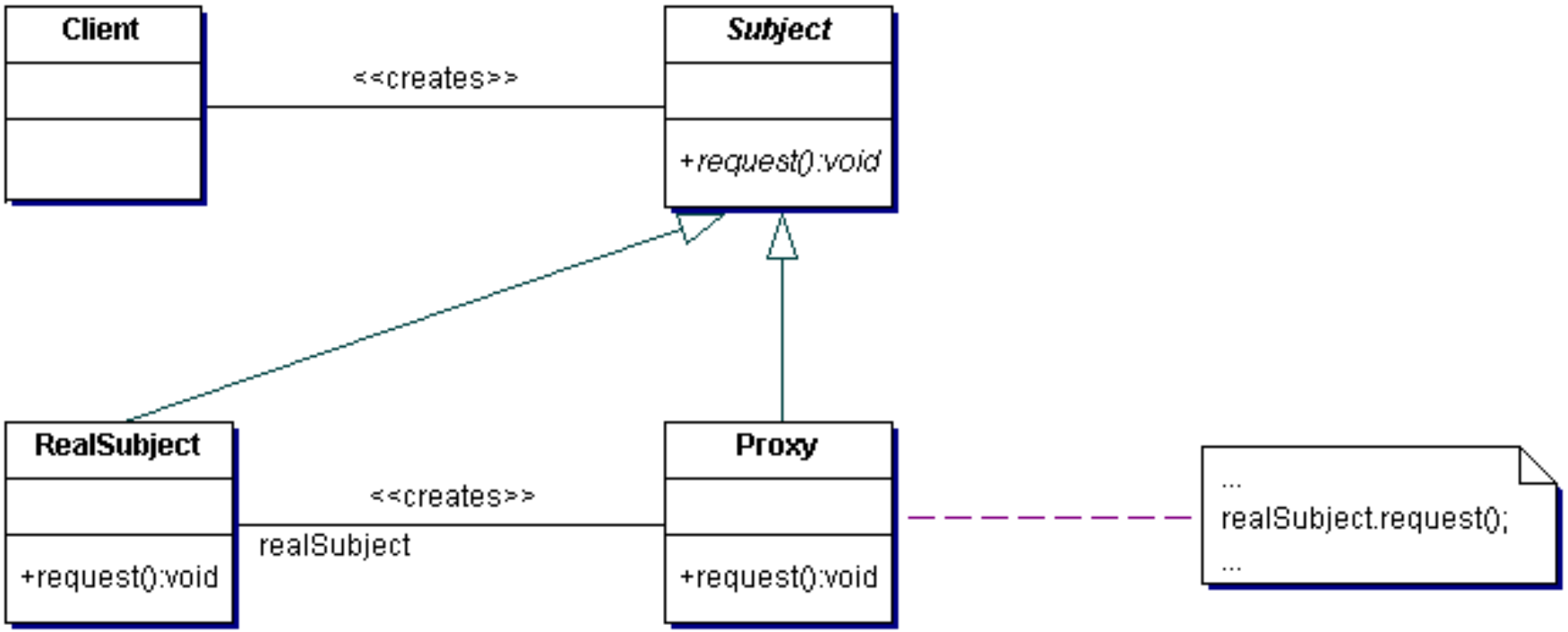
- Criação:
  - Abstract Factory
  - Builder
  - Factory Method
  - Prototype
  - Singleton
- Estruturais:
  - Adapter
  - Bridge
  - Composite
  - Decorator
  - Façade
  - **Flyweight**
  - **Proxy**
- Comportamentais:
  - Chain of Responsibility
  - Command
  - Interpreter
  - Iterator
  - Mediator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

Proxy

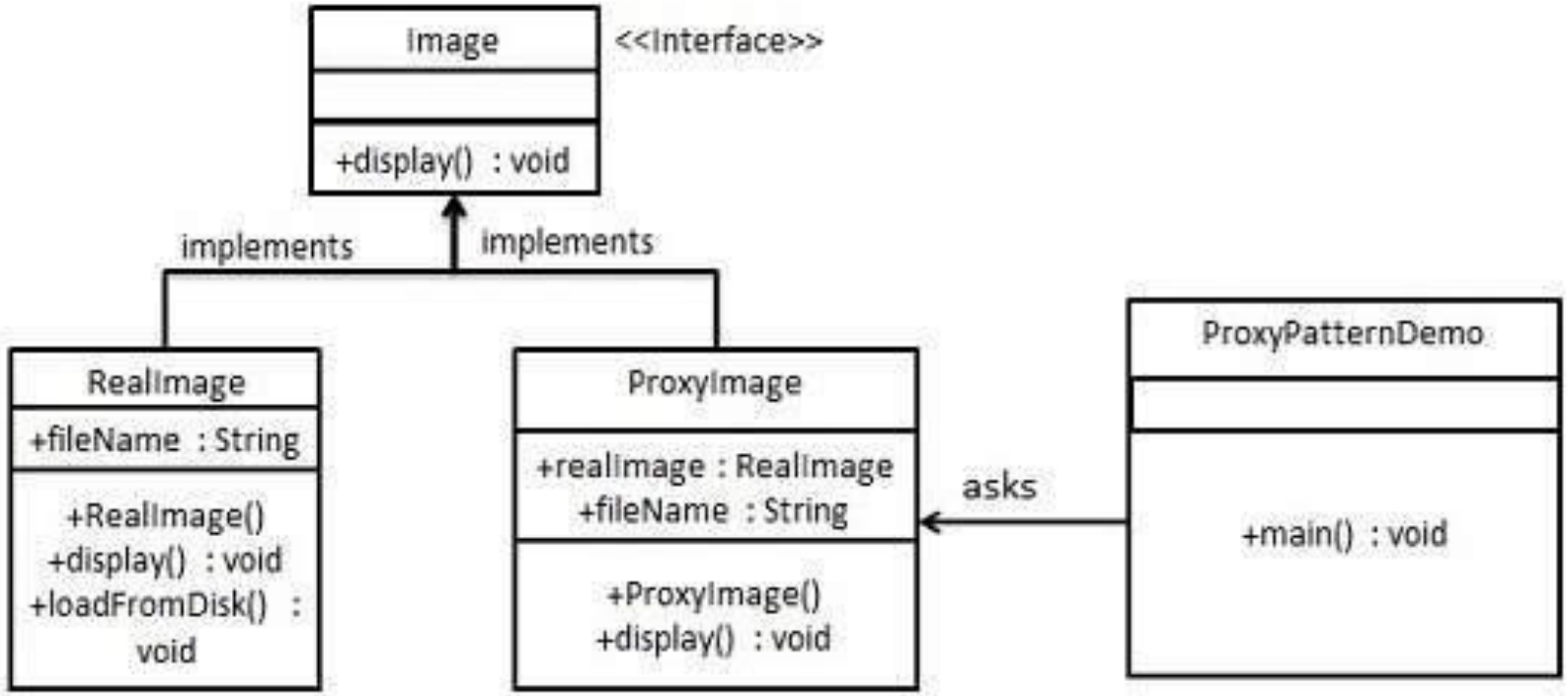
- **Intenção:** fornecer um substituto de outro objeto para controlar o acesso a esse objeto.

- **Solução:** criar uma classe para representar as funcionalidades de outra classe.

Proxy


Client
Subject
<<creates>>
+request():void
RealSubject
+request():void
<<creates>>
realSubject
Proxy
+request():void
...
realSubject.request();
...

Proxy – Exemplo



Proxy – Implementação

```
public interface Image {
   void display();
}
```

```
public class RealImage implements Image
{
   private String fileName;

   public RealImage(String fileName){
      this.fileName = fileName;
      loadFromDisk(fileName);
   }
   @Override
   public void display() {
      System.out.println("Displaying " + fileName);
   }
   private void loadFromDisk(String fileName){
      System.out.println("Loading " + fileName);
   }
}
```

Proxy – Implementação

```
public class ProxyImage implements Image
{
   private RealImage realImage;
   private String fileName;

   public ProxyImage(String fileName){
      this.fileName = fileName;
   }

   @Override
   public void display() {
      if(realImage == null){
         realImage = new RealImage(fileName);
      }
      realImage.display();
   }
}
```

Proxy – Implementação

```
public static void main(String[] args)
{
    Image image = new ProxyImage("test.jpg");

    image.display();

    System.out.println("");

    image.display();
}
```

Proxy vs Decorator

- Embora o Proxy possa ter uma implementação semelhante a um padrão Decorator, suas finalidades são diferentes.

- Um objeto decorador acrescenta uma ou mais responsabilidades a um objeto, enquanto que um proxy controla o acesso a um objeto.

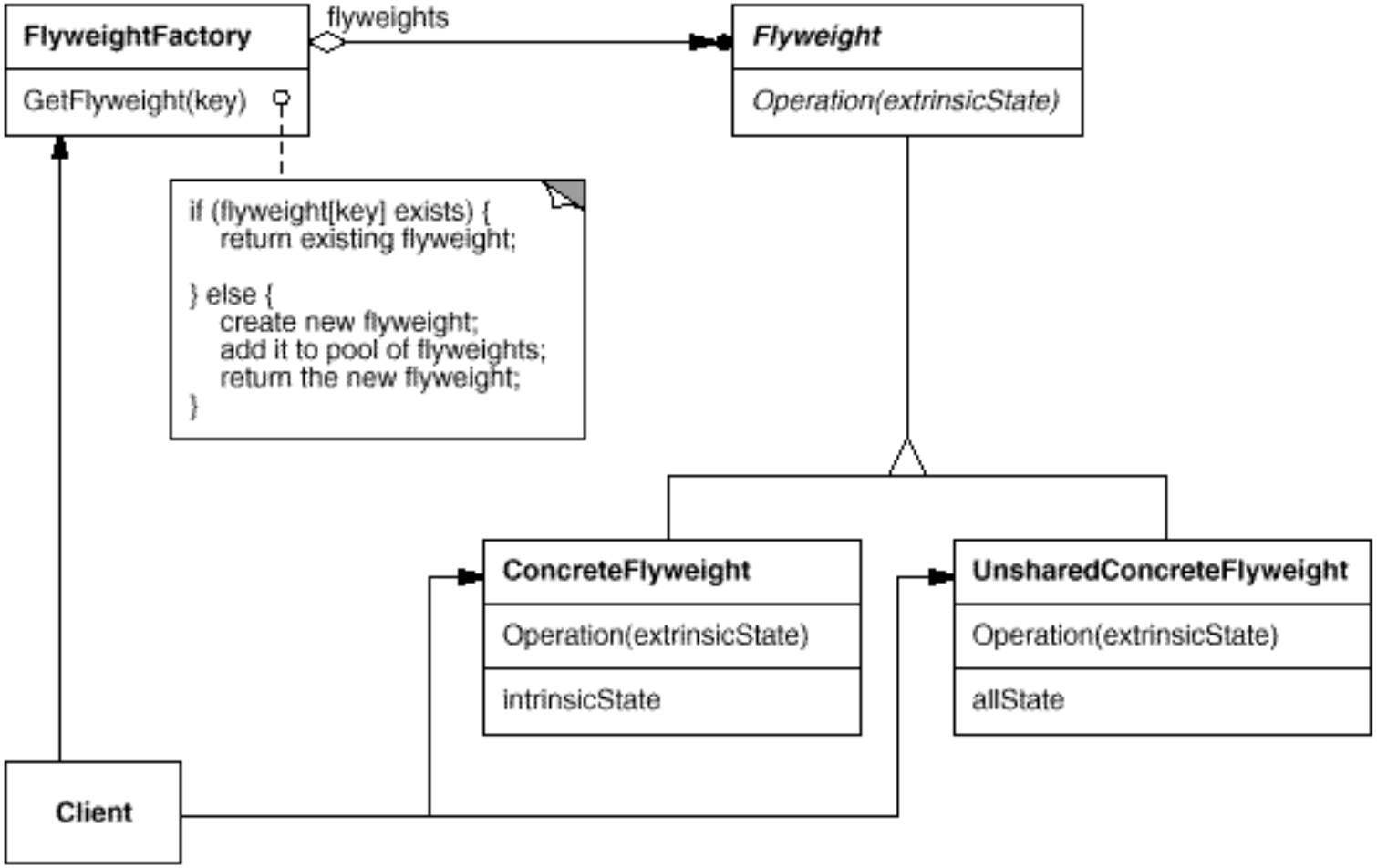
- No padrão Proxy, o objeto fornece as funcionalidades-chave e o proxy fornece (ou recusa) acesso ao objeto.

Flyweight

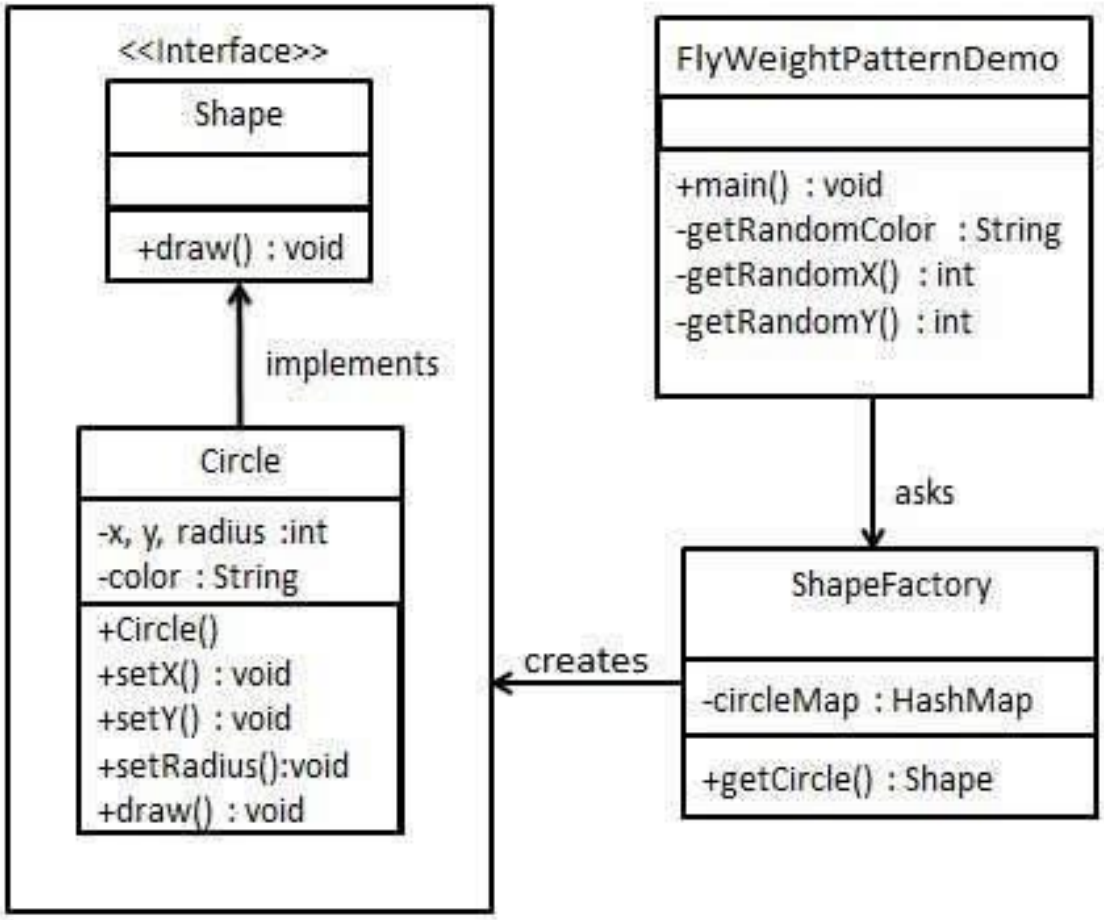
- **Intenção:** reduzir o número de objetos criados, visando a redução da sobrecarga de memória e a melhoria da performance.

- **Solução:** reutilizar objetos similares já existentes ou criar novos objetos quando nenhum objeto similar for encontrado para ser reutilizado.

Flyweight


FlyweightFactory
GetFlyweight(key)
flyweights
Flyweight
Operation(extrinsicState)
if (flyweight[key] exists) {
return existing flyweight;
} else {
create new flyweight;
add it to pool of flyweights;
return the new flyweight;
}
ConcreteFlyweight
Operation(extrinsicState)
intrinsicState
UnsharedConcreteFlyweight
Operation(extrinsicState)
allState
Client

Flyweight – Exemplo



Flyweight – Implementação

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

Flyweight – Implementação

```
public class Circle implements Shape {
   private String color;
   private int x;
   private int y;
   private int radius;
   public Circle(String color){
      this.color = color;
   }
   public void setX(int x){
      this.x = x;
   }
   public void setY(int y){
      this.y = y;
   }
   public void setRadius(int radius){
      this.radius = radius;
   }
   @Override
   public void draw() {
      System.out.println("Circle: Draw() [Color : " + color +
            ", x : " + x + ", y :" + y + ", radius :" + radius);
   }
}
```

Flyweight – Implementação

```
public class ShapeFactory
{
   private static HashMap<String, Shape> circleMap = new HashMap();

   public static Shape getCircle(String color)
   {
      Circle circle = (Circle)circleMap.get(color);

      if(circle == null)
      {
         circle = new Circle(color);
         circleMap.put(color, circle);
         System.out.println("Creating circle of color : " + color);
      }
      return circle;
   }
}
```

Flyweight – Implementação

```
public class FlyweightPatternDemo{
  private static final String colors[] = { "Red", "Green", "Blue",
                                           "White", "Black" };
  private static String getRandomColor(){
    return colors[(int)(Math.random()*colors.length)];
  }
  private static int getRandomX(){
    return (int)(Math.random()*100 );
  }
  private static int getRandomY(){
    return (int)(Math.random()*100);
  }
  public static void main(String[] args){
    for(int i=0; i < 20; i++) {
      Circle circle = (Circle)ShapeFactory.getCircle(getRandomColor());
      circle.setX(getRandomX());
      circle.setY(getRandomY());
      circle.setRadius(100);
      circle.draw();
    }
  }
}
```

Flyweight – Aplicabilidade

- Aplicável em situações onde muitos objetos semelhantes são criados e mantidos em memória.

- Em algumas aplicações pode reduzir consideravelmente o uso de memória e aumentar performance do sistema.