Prototype pattern

Vlad-Alexandru Russu
Group 30432

Brief description

- Creational design pattern
- Hides the complexity of making new instances from the client
- Concept:
  - copy an existing object rather than creating a new instance from scratch
  - the existing object acts as a prototype and contains the state of the object

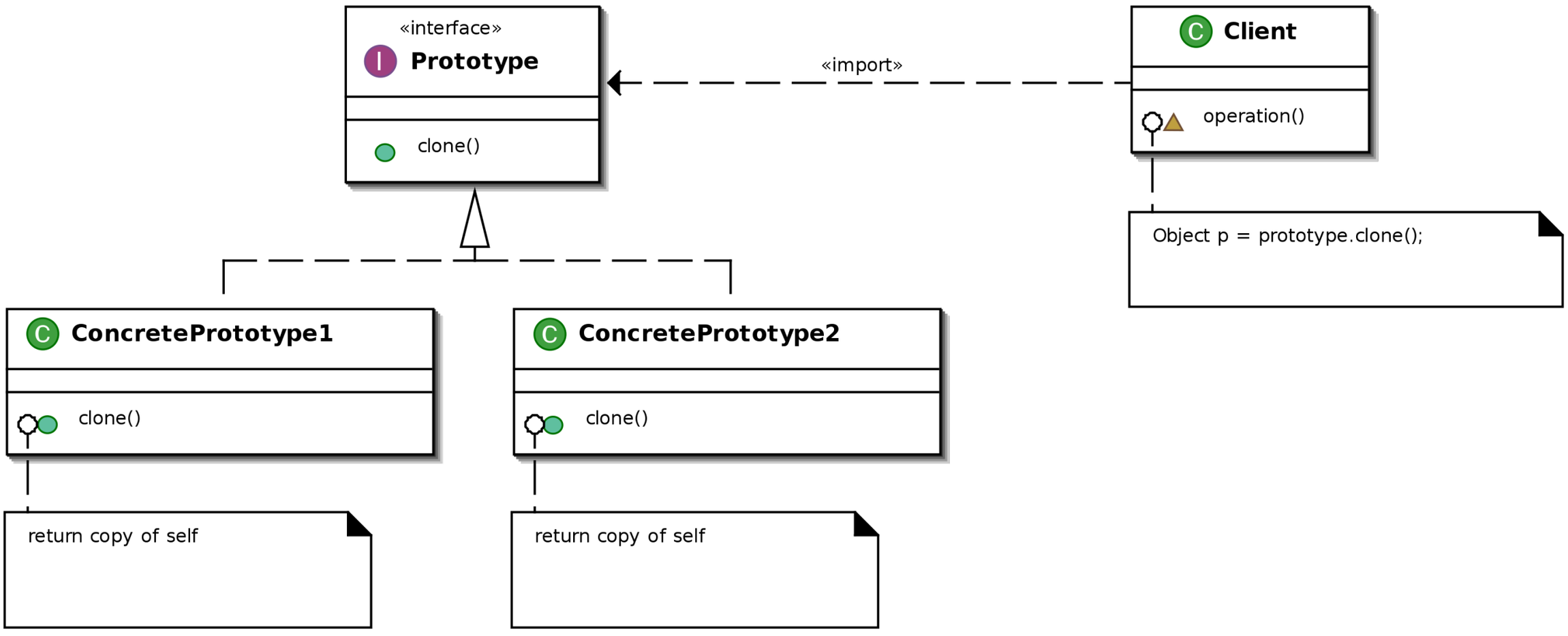
When to use?

- When a system should be independent of how its products are created, composed, and represented
- When the classes to instantiate are specified at run-time

Participants

1. **Prototype:** the prototype of the actual object
2. **Prototype registry:** registry service to have all prototypes accessible using string parameters
3. **Client:** uses registry service to access prototype instances

UML Diagram


«interface»
Prototype
clone()
«import»
Client
operation()
Object p = prototype.clone();
ConcretePrototype1
clone()
return copy of self
ConcretePrototype2
clone()
return copy of self

Example

```java
public abstract class Shape implements Cloneable {
    private String id;
    protected String type;
    abstract void draw();

    public String getType(){
        return type;
    }
    public String getId() {
        return id;
    }
    public void setId(String id) {
        this.id = id;
    }
    public Object clone() {
        Object clone = null;
        try {
            clone = super.clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
        }
        return clone;
    }
}

public class Square extends Shape{
    public Square(){
        type = "Square";
    }

    @Override
    public void draw() {
        System.out.println("Inside Square::draw() method.");
    }
}

public class Rectangle extends Shape {
    public Rectangle(){
        type = "Rectangle";
    }

    @Override
    public void draw() {
        System.out.println("Inside Rectangle::draw() method.");
    }
}
```

```java
public class ShapeRegistry {
    private static Hashtable<String, Shape> shapeMap  = new Hashtable<String, Shape>();

    public static Shape getShape(String shapeId) {
        Shape cachedShape = shapeMap.get(shapeId);
        return (Shape) cachedShape.clone();
    }

    public static void loadCache() {
        Square square = new Square();
        square.setId("1");
        shapeMap.put(square.getId(), square);

        Rectangle rectangle = new Rectangle();
        rectangle.setId("2");
        shapeMap.put(rectangle.getId(), rectangle);
    }
}

public class Prototype {
    public static void main(String[] args) {
        ShapeRegistry.loadCache();

        Shape clonedShape = (Shape) ShapeRegistry.getShape("1");
        System.out.println("Shape : " + clonedShape.getType());

        Shape clonedShape2 = (Shape) ShapeRegistry.getShape("2");
        System.out.println("Shape : " + clonedShape2.getType());
    }
}
```

Output:

```
Shape : Square
Shape : Rectangle
```

Advantages & Disadvantages

Pros:

- You can clone objects without coupling to their concrete classes
- You can get rid of repeated initialization code in favor of cloning pre-built prototypes
- You can produce complex objects more conveniently
- You get an alternative to inheritance when dealing with configuration presets for complex objects

Cons:

- Constructor separate from prototype definition
- Cloning complex objects that have circular references might be very tricky
- Hides concrete product classes from the client
- Overkill for a project that uses very few objects