Design Patterns

SoftUni Team

Technical Trainers

Software University

SoftUni

Software University

https://softuni.bg

Have a Question?

Software University

sli.do

#java-advanced

Table of Contents

1. **Definition** of **Design** Patterns
2. **Benefits** and **Drawbacks**
3. **Types** of Design Patterns
   - Creational
   - Structural
   - Behavioral

Design Patterns

What Are Design Patterns?

Software University

- **General** and **reusable solutions** to common problems in software design
- A **template** for solving given problems
- Add additional layers of **abstraction** in order to reach flexibility

What Do Design Patterns Solve?

- Patterns solve **software structural problems** like:
  - Abstraction
  - Encapsulation
  - Separation of concerns
  - Coupling and cohesion
  - Separation of interface and implementation
  - Divide and conquer

Elements of a Design Pattern

- Pattern name - Increases **vocabulary** of designers
- Problem - **Intent**, context, and when to apply
- Solution - **Abstract** code
- Consequences - **Results** and trade-offs

Why Design Patterns?

Benefits

- Names form a common vocabulary
- Enable large-scale **reuse** of software architectures
- Help improve developer **communication**
- Help ease the **transition** to Object-Oriented technology
- Can **speed-up** the development

Drawbacks

- Do not lead to a direct code reuse
- Deceptively simple
- Developers may suffer from **pattern overload** and **overdesign**
- Validated by **experience** and discussion, not by automated testing
- Should be used only if **understood well**

Types of Design Patterns

Main Types

- Creational patterns
  - Deal with **initialization and configuration** of classes and objects
- Structural patterns
  - Describe ways to **assemble** objects to implement **new functionality**
  - **Composition** of classes and objects
- Behavioral patterns
  - Deal with dynamic **interactions** among societies of classes
  - Distribute **responsibility**

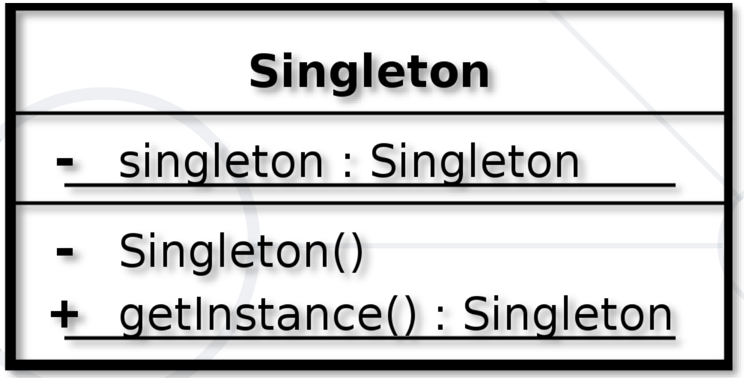
Creational Patterns

Purposes

- Deal with **object creation** mechanisms
- Trying to create objects in a **manner suitable** to the **situation**
- Two main ideas
  - **Encapsulating** knowledge about which classes the system uses
  - **Hiding** how instances of these classes are created

Singleton Pattern

- The most often used creational design pattern
- A Singleton class is supposed to have **only one instance**
- It is **not a global variable**
- Possible problems
  - Lazy loading
  - Thread-safe



Singleton

- singleton : Singleton

- Singleton()

+ getInstance() : Singleton

Double-Check Singleton Example

```java
public static Singleton getInstanceDC() {
    if(instance == null) { // Single checked
        synchronized (Singleton.class) {
            if (instance == null) { // Double checked
                instance = new Singleton();
            }
        }
    }
    return instance;
}
```

Builder Pattern

- **Separates** the construction of a complex object from its representation
  - The same construction process can create different representations
- Provides control over steps of the construction process

Example: Computer Class

```java
public class Computer {
    private String RAM;
    private boolean isGraphicsCardEnabled;

    public String getRAM() { return RAM; }
    public boolean isGraphicsCardEnabled() {
        return isGraphicsCardEnabled;
    }
    public Computer(String ram, boolean isGraphicsCardEnabled) {
        this.RAM = ram;
        this.isGraphicsCardEnabled = isGraphicsCardEnabled;
    }
}
```

Example: ComputerBuilder Class

```java
public class ComputerBuilder {
    private String RAM;
    private boolean isGraphicsCardEnabled;

    public ComputerBuilder(String ram){ this.RAM = ram; }
    public ComputerBuilder setGraphicsCardEnabled(
                             boolean isGraphicsCardEnabled) {
        this.isGraphicsCardEnabled = isGraphicsCardEnabled;
        return this; }

    public Computer build(){
        return new Computer(this.RAM, this.isGraphicsCardEnabled);
    }
}
```

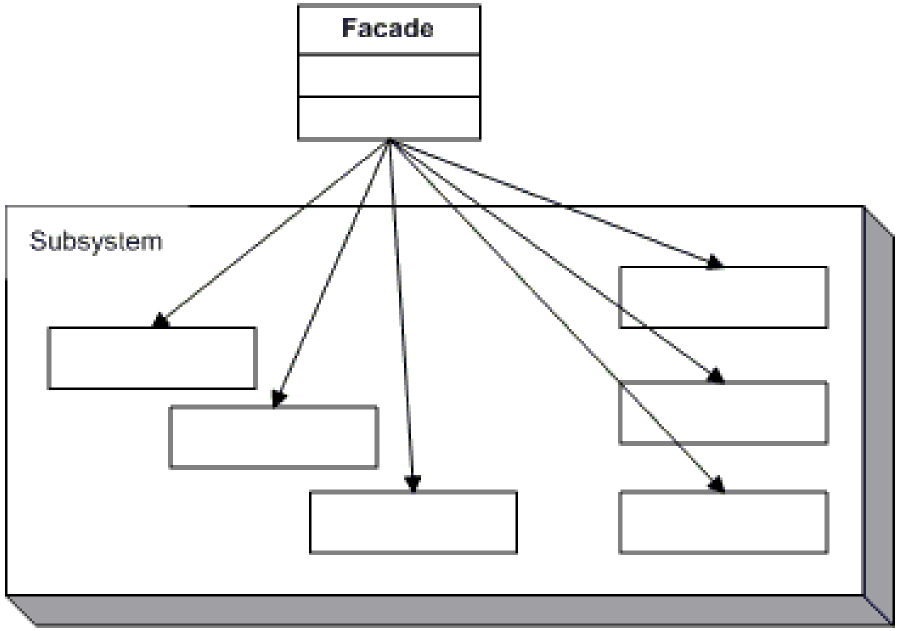
Structural Patterns

Purposes

- Describe ways to assemble **objects** to implement a **new functionality**
- Ease the design by identifying a simple way to realize the **relationship** between entities
- All about Class and Object composition
  - **Inheritance** to compose interfaces
  - Ways to compose objects to obtain **new functionality**

Façade Pattern

- Provides a **unified interface** to a set of interfaces in a subsystem
- Defines a **higher-level interface** that makes the subsystem easier to use



Façade Example

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

Façade Example

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

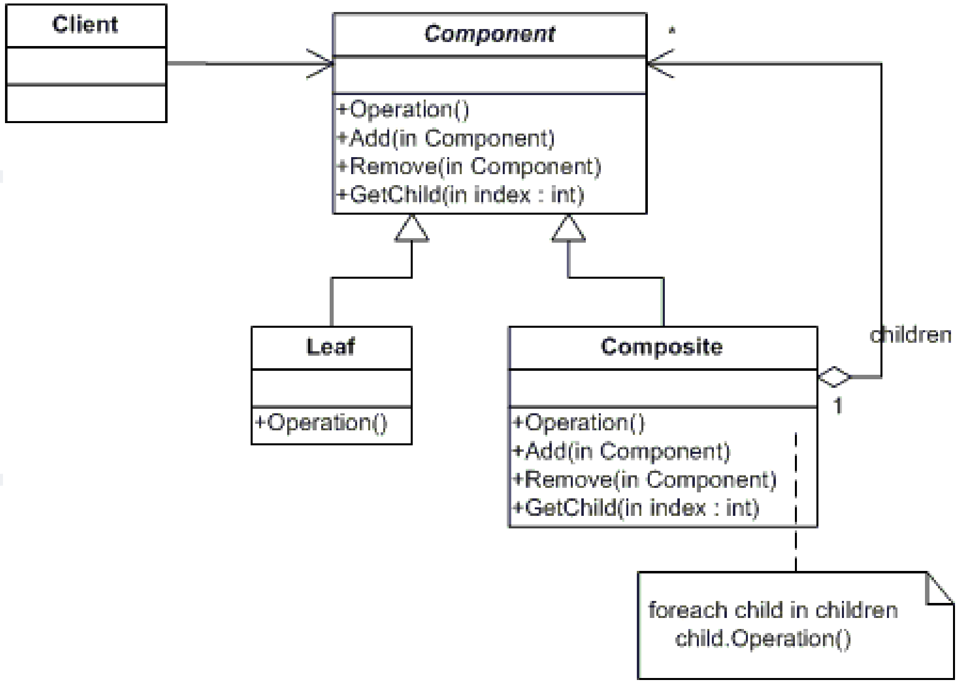
Façade Example

```java
public class ShapeMaker {
    private Shape rectangle;
    private Shape square;
    public ShapeMaker() {
        rectangle = new Rectangle();
        square = new Square(); }

    public void drawRectangle(){
        rectangle.draw(); }
    public void drawSquare(){
        square.draw(); }
}
```

Composite Pattern

- Allows to **combine** different types of objects in tree structures
- Gives the possibility to treat the **same object(s)**
- Used when
  - You have different objects that you want to **treat the same way**
  - You want to present a **hierarchy** of objects



The Component Abstract Class

```
abstract class Component {
  protected String name;

  public Component(String name) {
      this.name = name; }

  public abstract void add(Component c);
  public abstract void remove(Component c);
  public abstract void display(int depth);
}
```

The Composite Class

```
class Composite extends Component {
  private List<Component> children = new ArrayList<Component>();
  public Composite(String name) { super(name); }
  @Override
  public void add(Component component) {
       children.add(component); }
  @Override
  public void remove(Component component) {
       children.Remove(component); }
```

The Composite Class

```
@Override
public void display(int depth) {
    System.out.println(printNameInDepth(depth, name);
    foreach (Component component : children) {
      component.display(depth + 2);
    }
}

public void printNameInDepth(int depth, String name) {
    for(int i = 0; i < depth; i++)
       System.out.print("-");
    System.out.print(name);
}
```

The Leaf Class

```java
class Leaf extends Component {
  public Leaf(String name) { super(name); }

  @Override
  public void add(Component c) {
    System.out.println("Cannot add to a leaf"); }
  @Override
  public void Remove(Component c) {
    System.out.println("Cannot remove from a leaf"); }
  @Override
  public void Display(int depth) {
    System.out.println(printNameInDepth(depth, name); }
}
```

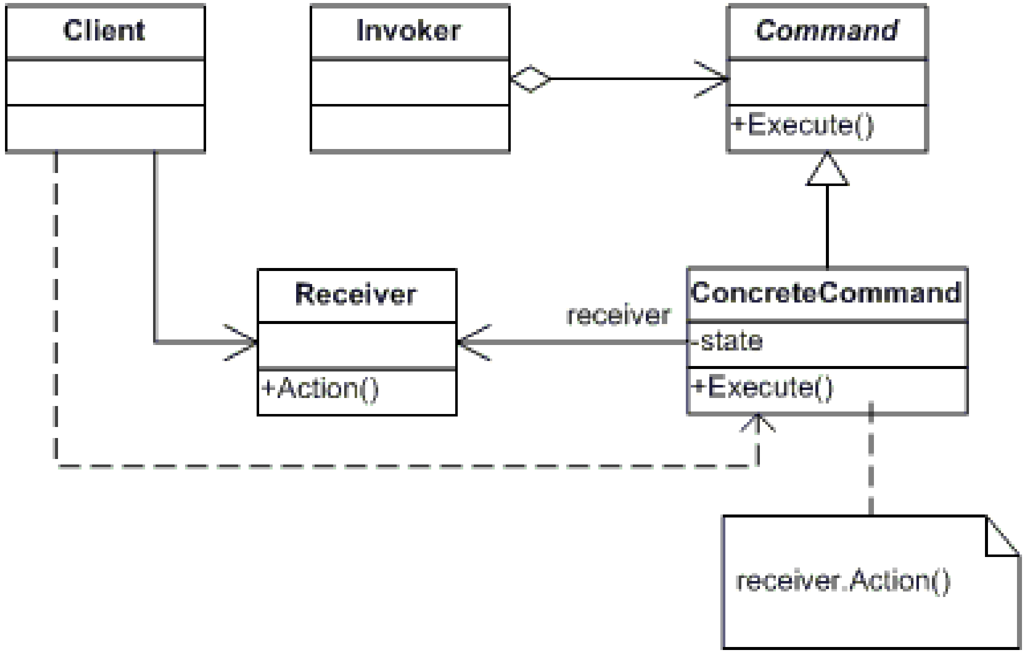
Behavioral Patterns

Purposes

- Concerned with the **interaction** between objects
  - Either with the **assignment of responsibilities** between objects
  - Or **encapsulating behavior** in an object and delegating requests to it
- Increases **flexibility** in carrying out cross-classes communication

Command Pattern

- An object **encapsulates** all the information needed to call a method at a later time
  - Lets you **parameterize** clients with different requests, queue or log requests, and support undoable operations



The Command Abstract Class

```
abstract class Command {
  protected Receiver receiver;

  public Command(Receiver receiver) {
    this.receiver = receiver; }

  public abstract void execute();
}
```

Concrete Command Class

```
class ConcreteCommand extends Command {
  public ConcreteCommand(Receiver receiver) {
       super(receiver); }

  @Override
  public void execute() {
    receiver.action(); }
}
```

The Receiver Class

```
class Receiver {
  public void action() {
    System.out.println("Called Receiver.action()");
  }
}
```

The Invoker Class

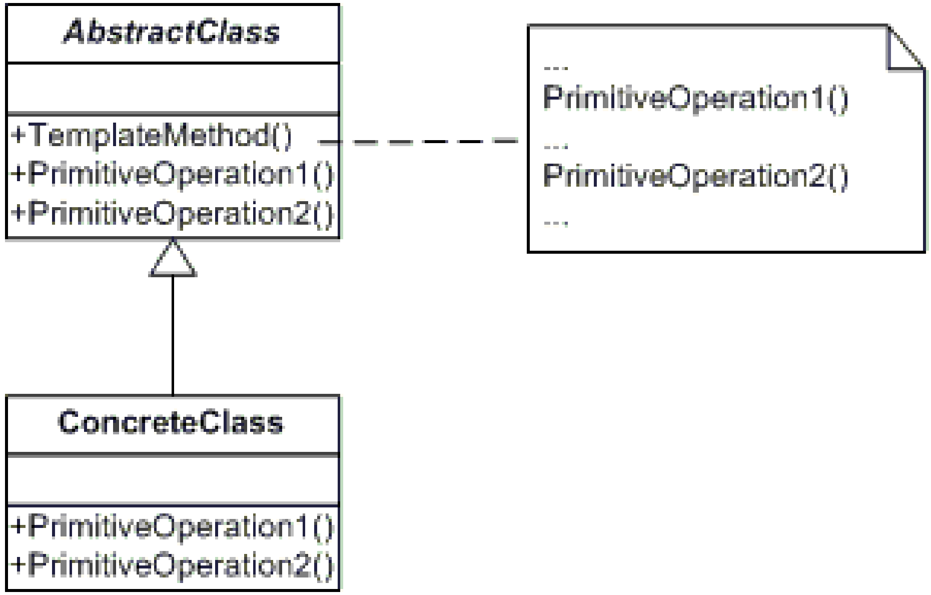
```
class Invoker {
  private Command command;

  public void setCommand(Command command) {
    this.command = command; }

  public void ExecuteCommand() {
    command.execute(); }
}
```

Template Pattern

- Define the **skeleton** of an algorithm in a method, leaving some implementation to its subclasses
- Allows the subclasses to **redefine** the implementation of some of the **parts** of the algorithm, but not its structure



The Abstract Class

```
abstract class AbstractClass {
  public abstract void primitiveOperation1();
  public abstract void primitiveOperation2();

  public void templateMethod() {
    primitiveOperation1();
    primitiveOperation2();
    System.out.println(""); }
}
```

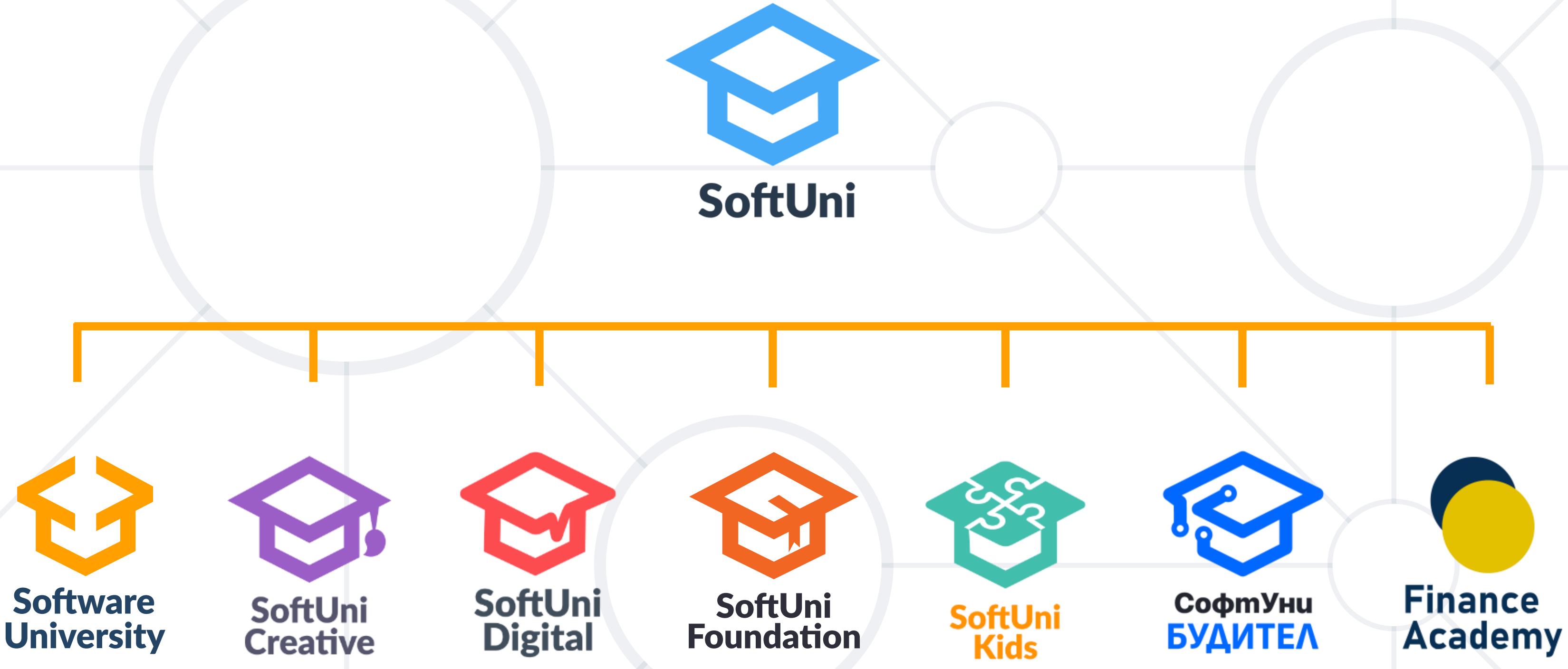
A Concrete Class

```
class ConcreteClassA extends AbstractClass {
  @Override
  public void primitiveOperation1() {
    System.out.println("ConcreteClassA.primitiveOperation1()"); }
  @Override
  public void primitiveOperation2() {
  System.out.println("ConcreteClassA.primitiveOperation2()");
}
```

Summary

- Design Patterns
  - Provide solution to common problems
  - Add additional layers of abstraction
- Three main types of Design Patterns
  - Creational
  - Structural
  - Behavioral

Questions?



SoftUni

Software University

SoftUni Creative

SoftUni Digital

SoftUni Foundation

SoftUni Kids

СофтУни БУДИТЕЛ

Finance Academy

SoftUni Diamond Partners

SmartIT

createX

VIVACOM

INDEAVR
Serving the high achievers

Coca-Cola HBC
Bulgaria

Postbank
Решения за твоето утре

AMBITIONED

SOFTWARE
GROUP

SUPER
HOSTING
.BG

DRAFT
KINGS

PHAR
VISION

Trainings @ Software University (SoftUni)

- Software University – High-Quality Education, Profession and Job for Software Developers
  - softuni.bg
- Software University Foundation
  - softuni.foundation
- Software University @ Facebook
  - facebook.com/SoftwareUniversity

Software
University

License