

# Design Patterns



**SoftUni Team**  
**Technical Trainers**



**SoftUni**



**Software University**

<https://softuni.bg>

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

sli.do

**#java-advanced**



# Design Patterns

# What Are Design Patterns?

- **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?



- 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



- 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

- 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



- 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



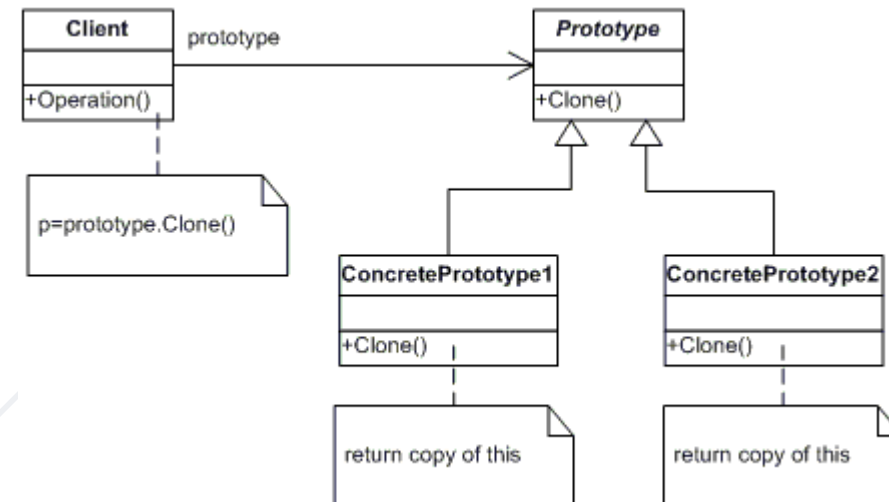
Singleton
-instance : Singleton
-Singleton() +Instance() : Singleton

# Double-Check Singleton Example

```
class SerializableSingleton implements Serializable {  
    private static SerializableSingleton instance;  
    private SerializableSingleton() {}  
    public static synchronized SerializableSingleton getInstance() {  
        if(instance == null) {  
            instance = new SerializableSingleton();  
        }  
        return instance;  
    }  
}
```



- Factory for **cloning** new instances from a prototype
  - Create new objects by copying this prototype
  - Instead of using the "new" keyword
- **Cloneable** interface acts as Prototype



# The Prototype Abstract Class

```
public abstract class Prototype {  
    private String id;  
  
    public Prototype(String id) {  
        this.id = id; }  
  
    public String getId() { return this.id; }  
    public abstract Prototype clone();  
}
```

# A Concrete Prototype Class

```
class ConcretePrototype extends Prototype {  
    public ConcretePrototype(string id) {  
        super(id);  
    }  
    @Override  
    public Prototype clone() {  
        return (Prototype)this.clone(); }  
}
```

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

# Example: Computer Class

```
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

```
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**

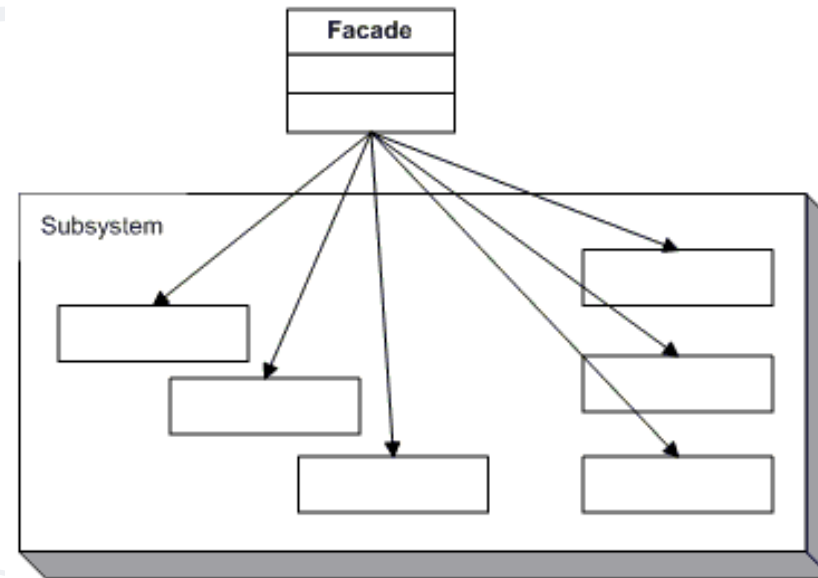
# Purposes

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





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



# The Façade Class (1)

```
class Facade {  
    private SubSystemOne one;  
    private SubSystemTwo two;  
  
    public Facade() {  
        one = new SubSystemOne();  
        two = new SubSystemTwo();  
    }  
}
```

# The Façade Class (2)

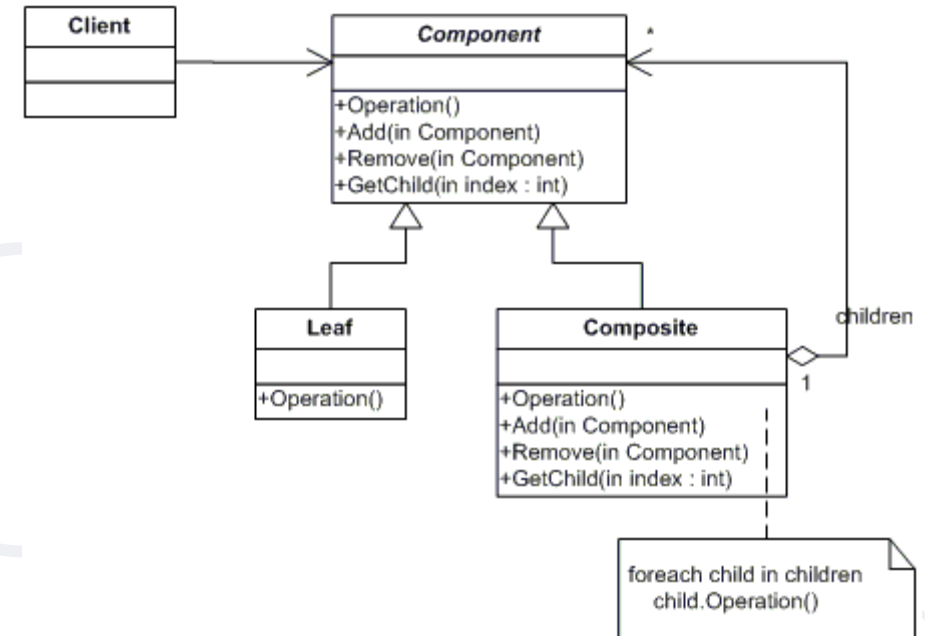
```
public void MethodA() {  
    System.out.println("\nMethodA() ---- ");  
    one.MethodOne();  
    two.MethodTwo(); }  
  
public void MethodB() {  
    System.out.println("\nMethodB() ---- ");  
    two.MethodTwo(); }  
}
```

```
class SubSystemOne {  
    public void MethodOne() {  
        System.out.println(" SubSystemOne Method"); }  
}
```

```
class SubSystemTwo {  
    public void MethodTwo() {  
        System.out.println(" SubSystemTwo Method"); }  
}
```

# 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 **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 (1)

```
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 (2)

```
@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

```
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); }  
}
```



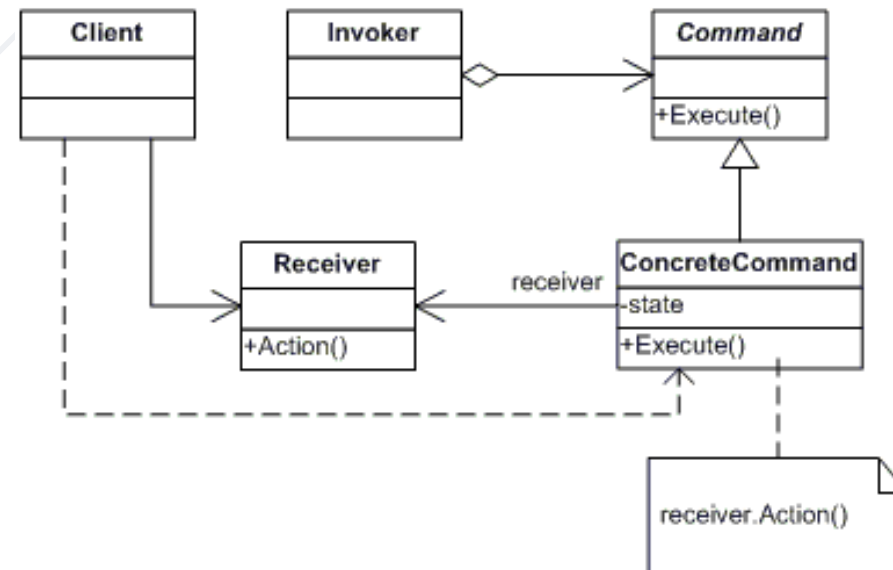
# Behavioral Patterns

# Purposes

- Concerned with **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



- 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(); }  
}
```

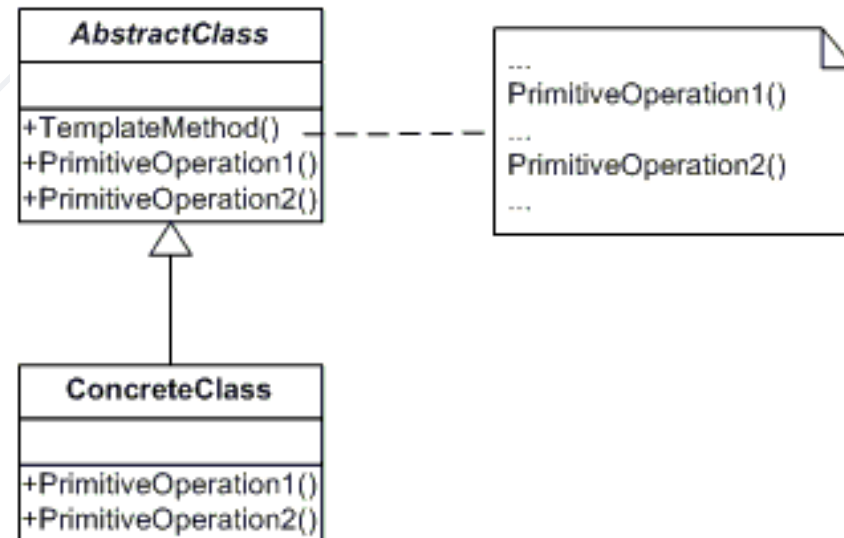
```
class Receiver {  
    public void action() {  
        System.out.println("Called Receiver.action()");  
    }  
}
```

# The Invoker Class

```
class Invoker {  
    private Command command;  
  
    public void setCommand(Command command) {  
        this.command = command; }  
  
    public void ExecuteCommand() {  
        command.execute(); }  
}
```



- 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



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

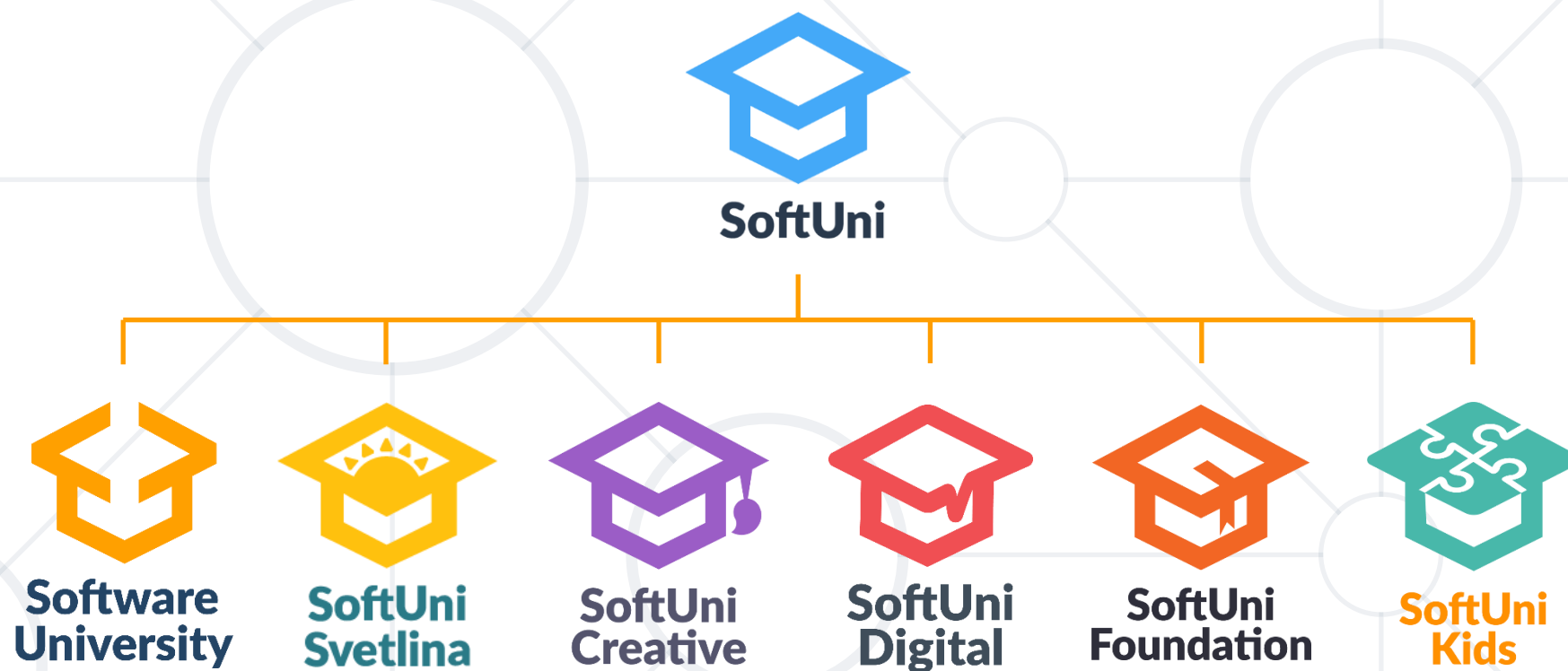
# A Concrete Class

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

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



# Questions?



# SoftUni Diamond Partners

**SUPER  
HOSTING  
.BG**

**INDEAVR**  
Serving the high achievers

 **SmartIT**

  
SOFTWARE

**zühlke**  
empowering ideas

 **INFRAGISTICS®**



**Coca-Cola HBC  
Bulgaria**



**Postbank**

*Решения за твоето утре*



 **DRAFT  
KINGS**



**SOFTWARE  
GROUP**



- Software University – High-Quality Education, Profession and Job for Software Developers

- [softuni.bg](http://softuni.bg)

- Software University Foundation

- [softuni.foundation](http://softuni.foundation)

- Software University @ Facebook

- [facebook.com/SoftwareUniversity](https://facebook.com/SoftwareUniversity)

- Software University Forums

- [forum.softuni.bg](http://forum.softuni.bg)





- This course (slides, examples, demos, exercises, homework, documents, videos and other assets) is **copyrighted content**
- Unauthorized copy, reproduction or use is illegal
- © SoftUni – <https://about.softuni.bg>
- © Software University – <https://softuni.bg>

