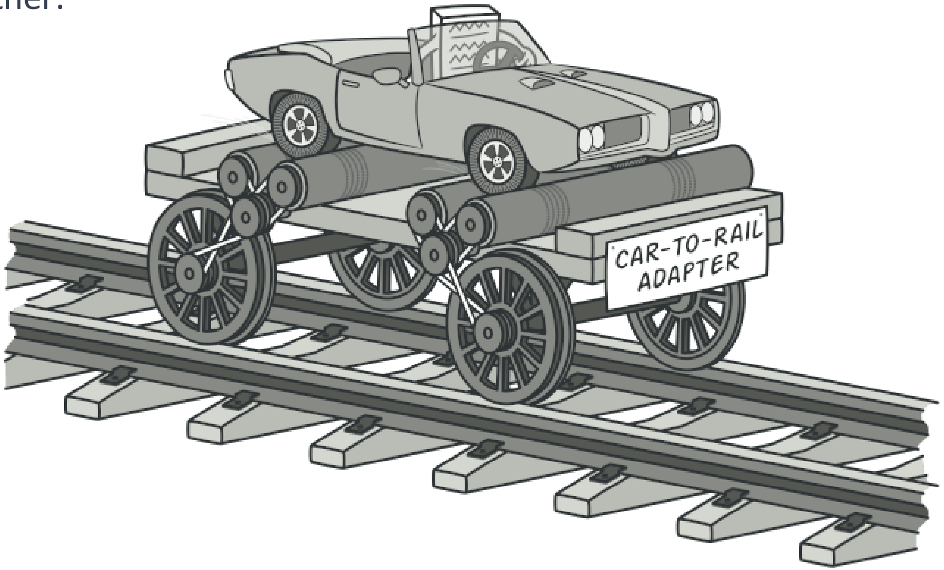
Design Patterns
Structural

Structural Pattern – Adapter

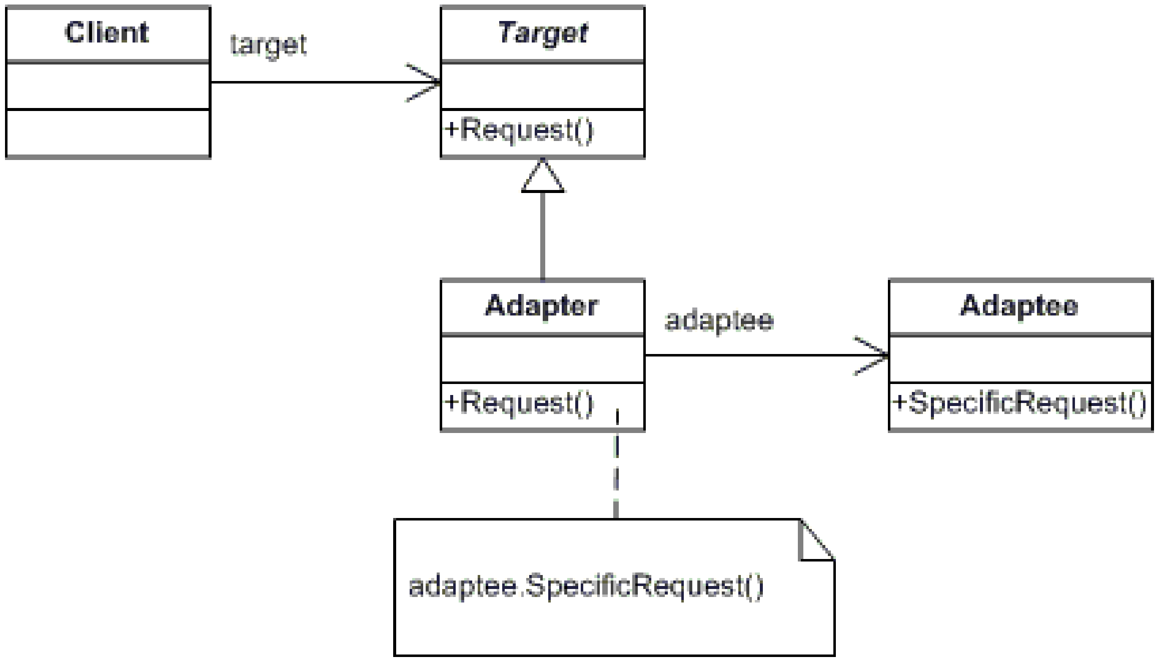
The Adapter design pattern is to create an intermediary abstracted layer or class that acts as a translator between two incompatible or disparate interfaces. This is done in a way that allows classes with incompatible interfaces to work together.



Adapter – When to use

- **Incompatible Interfaces**: Use the Adapter pattern when you need to make a class with a non-matching interface work with others without changing its source code.
- **Legacy or External Code Integration**: The pattern is helpful when integrating legacy systems, third-party libraries, or external APIs with interface mismatches.
- **Refactoring**: An Adapter can facilitate communication with complex legacy code while refactoring it into smaller, more focused classes.

Adapter – UML


Client
target
Target
+Request()
Adapter
adaptee
Adaptee
+Request()
+SpecificRequest()
adaptee.SpecificRequest()

Adapter – Participants

- **Target Interface**: This is the interface that the existing system and classes expect to work with. The client interacts with the Target interface.
- **Adapter**: The Adapter is the class that gets integrated into the existing system. It adapts the interface of the Adaptee to the Target interface.
- **Adaptee**: This is the class (or interface) whose capabilities we need in the existing system but its interface is not compatible with the existing system.
- **Client**: This is the class that interacts with the Target interface. It's unaware of the Adapter and the Adaptee. From the Client's perspective, it's interacting with the Target interface only.

Adapter – Implementation step 1

Define the Target interface: The **Target** interface represents what the client can work with.

```java
public interface Target {
        void request();
}
```

Adapter – Implementation step 2

Implement the Adaptee: The **Adaptee** class is an existing class that provides some useful behavior, but its interface is not compatible with the rest of our code.

```java
public class Adaptee {
    public void specificRequest() {
        System.out.println("Specific Request!");
    }
}
```

Adapter – Implementation step 3

Create the Adapter: The **Adapter** class needs to implement the **Target** interface and should have a reference to an **Adaptee** object to make the **Adaptee**'s functionality work with the **Target** interface.

```java
public class Adapter implements Target {
    private Adaptee adaptee;

    public Adapter(Adaptee adaptee) {
        this.adaptee = adaptee;
    }

    @Override
    public void request() {
        adaptee.specificRequest();
    }
}
```

Adapter – Implementation step 4

Use the Adapter in the Client: The **Client** class works with objects that implement the **Target** interface.

```java
public class Client {
    private Target target;

    public Client(Target target) {
        this.target = target;
    }

    public void doSomething() {
        target.request();
    }

    public static void main(String[] args) {
        // Create an instance of Adaptee
        Adaptee adaptee = new Adaptee();

        // Create an instance of Adapter and pass the Adaptee object to it
        Target adapter = new Adapter(adaptee);

        // Use the Adapter instance to create a Client object
        Client client = new Client(adapter);

        // Perform an operation on the client
        client.doSomething(); // Outputs: "Specific Request!"
    }
}
```

Adapter – Pros and Cons

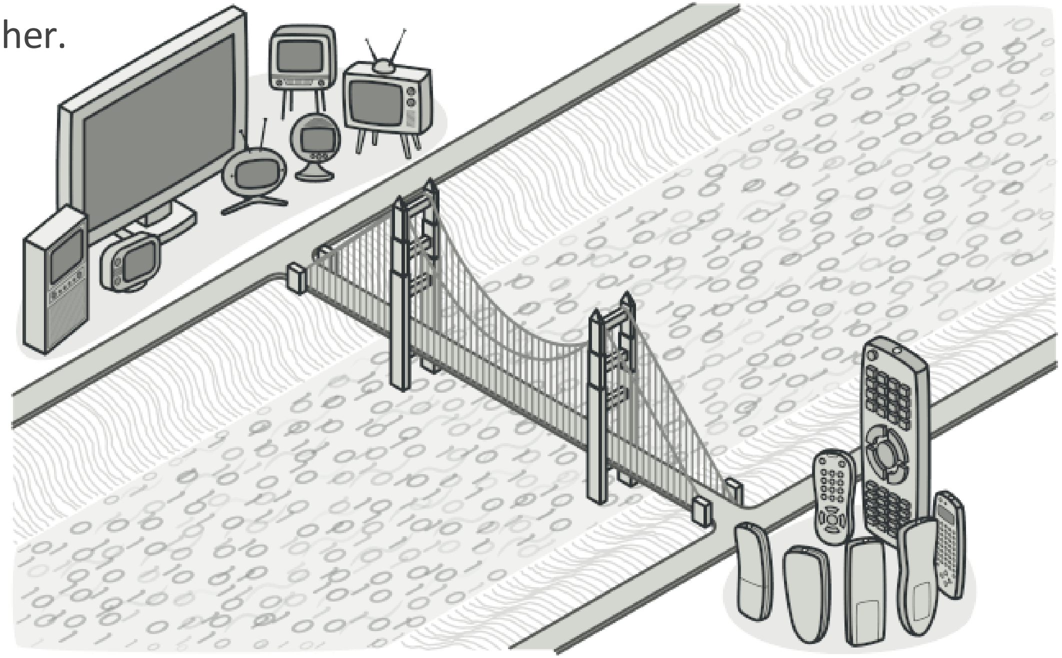
- **Single Responsibility Principle**: Decouples interface or data conversion from core business logic, ensuring each class has one responsibility.
- **Open/Closed Principle**: New types of adapters can be added without altering existing client code, as long as they adhere to the client interface.
- **Code Complexity**: The downside of the Adapter pattern is potential increase in complexity due to new interfaces and classes. Sometimes, modifying the service class directly can be simpler.

Adapter – Use case in Java

- java.util.Arrays#asList()
- java.util.Collections#list()
- java.util.Collections#enumeration()
- java.io.InputStreamReader(InputStream) (returns a Reader object)
- java.io.OutputStreamWriter(OutputStream) (returns a Writer object)

Structural Pattern – Bridge

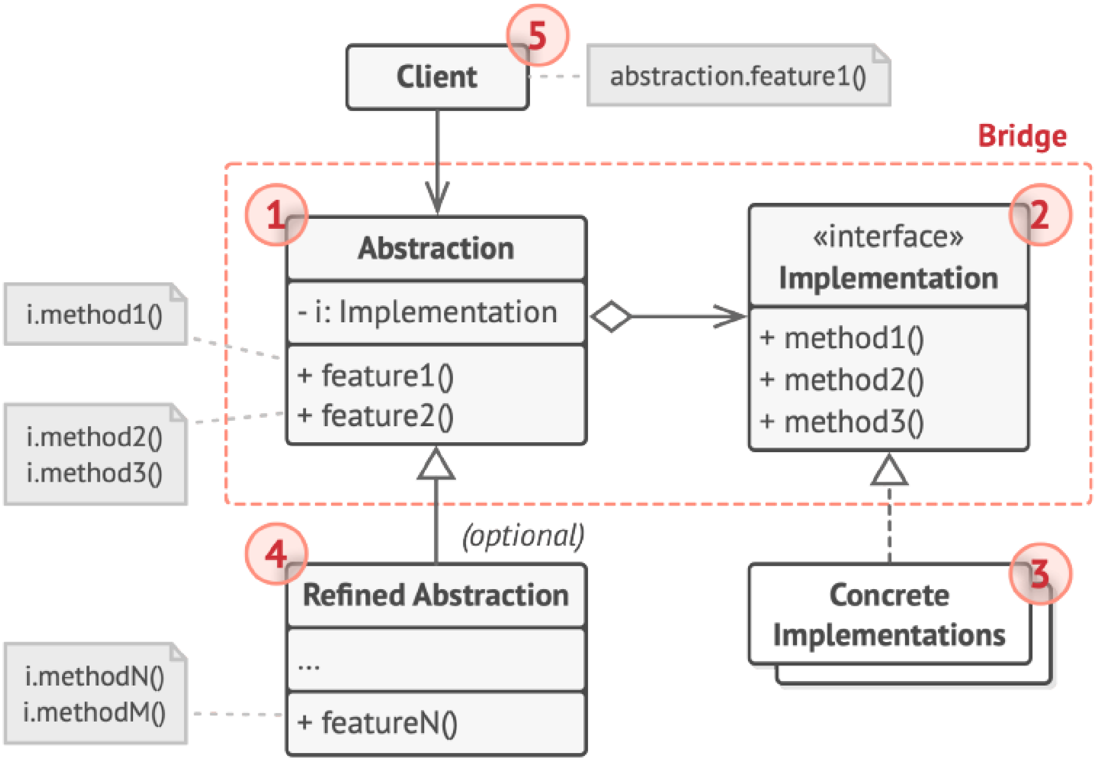
Bridge is a structural design pattern that lets you split a large class or a set of closely related classes into two separate hierarchies—abstraction and implementation—which can be developed independently of each other.



Bridge – When to use

- **When you want to avoid a permanent binding between an abstraction and its implementation**: This is particularly useful when the implementation must be selected or switched at runtime.
- **When both the abstractions and their implementations should be extensible by subclassing**: In this case, the Bridge pattern lets you combine the different abstractions and implementations and extend them independently.
- **When changes in the implementation of an abstraction should have no impact on clients**: That is, their code should not need to be recompiled.
- **When you want to hide the implementation of an abstraction completely from clients**: In other words, when you want to share an implementation among multiple objects (reference counting), and this fact should be hidden from the client.

Bridge – UML


5
Client
abstraction.feature1()
Bridge
1
Abstraction
- i: Implementation
+ feature1()
+ feature2()
2
«interface»
Implementation
+ method1()
+ method2()
+ method3()
i.method1()
i.method2()
i.method3()
(optional)
4
Refined Abstraction
...
+ featureN()
3
Concrete
Implementations
i.methodN()
i.methodM()

Bridge – Participants

- **Abstraction**: Defines the client interface and maintains a reference to an Implementor object.
- **RefinedAbstraction**: Offers more specific implementations of the Abstraction. This is an optional component.
- **Implementation**: Establishes the interface for concrete implementation classes, which can be quite different from the Abstraction's interface.
- **ConcreteImplementor**: Provides the actual implementation of the Implementor interface. It defines concrete behaviors.

Bridge – Implementation step 1

Define the Implementor: Define a Color interface, which is the **Implementor** in this scenario:

```
interface Implementor {
    void operationImpl();
}
```

Bridge – Implementation step 2

Define Concrete Implementors: define a couple of **ConcreteImplementors**, **ConcreteImplementorsA** and **ConcreteImplementorsB**

```java
// ConcreteImplementors
class ConcreteImplementorA implements Implementor {
    public void operationImpl() {
        System.out.println("ConcreteImplementorA's implementation.");
    }
}

class ConcreteImplementorB implements Implementor {
    public void operationImpl() {
        System.out.println("ConcreteImplementorB's implementation.");
    }
}
```

Bridge – Implementation step 3

Define the Abstraction:

```java
abstract class Abstraction {
    protected Implementor implementor;

    public Abstraction(Implementor implementor) {
        this.implementor = implementor;
    }

    public abstract void operation();
}
```

Bridge – Implementation step 3

Define Refined Abstractions:

```java
// RefinedAbstraction
class RefinedAbstraction extends Abstraction {
    public RefinedAbstraction(Implementor implementor) {
        super(implementor);
    }

    public void operation() {
        implementor.operationImpl();
    }
}
```

Bridge – Implementation step 5

Usage:

```java
public class Client {
    public static void main(String[] args) {
        Implementor implementorA = new ConcreteImplementorA();
        Abstraction abstractionA = new RefinedAbstraction(implementorA);

        abstractionA.operation(); // Output: ConcreteImplementorA's implementation.

        Implementor implementorB = new ConcreteImplementorB();
        Abstraction abstractionB = new RefinedAbstraction(implementorB);

        abstractionB.operation(); // Output: ConcreteImplementorB's implementation.
    }
}
```

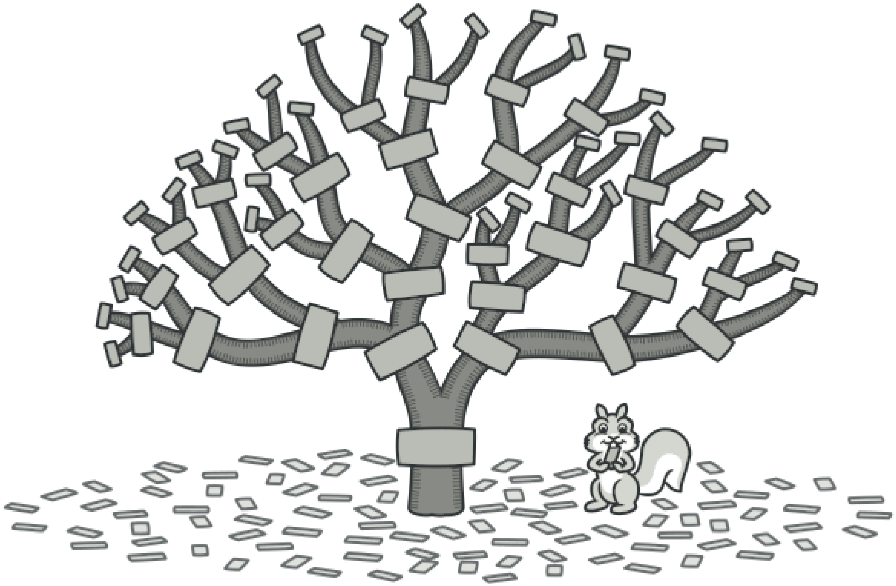
Bridge – Pros and Cons

- **Decoupling**: Separates the interface from the implementation, allowing each to vary independently.
- **Extensibility**: Supports independent extension of the abstraction and implementation hierarchies.
- **Hiding Implementation**: The pattern hides implementation details from clients.
- **Implementation Sharing**: Useful for sharing an implementation among multiple objects.

- **Increased Complexity**: The pattern can add complexity compared to simple inheritance.
- **Setup Difficulty**: It can be more challenging to set up and organize your code to use this pattern.
- **Conceptual Difficulty**: Understanding and implementing the pattern can be difficult for beginners.

Structural Pattern – Composite

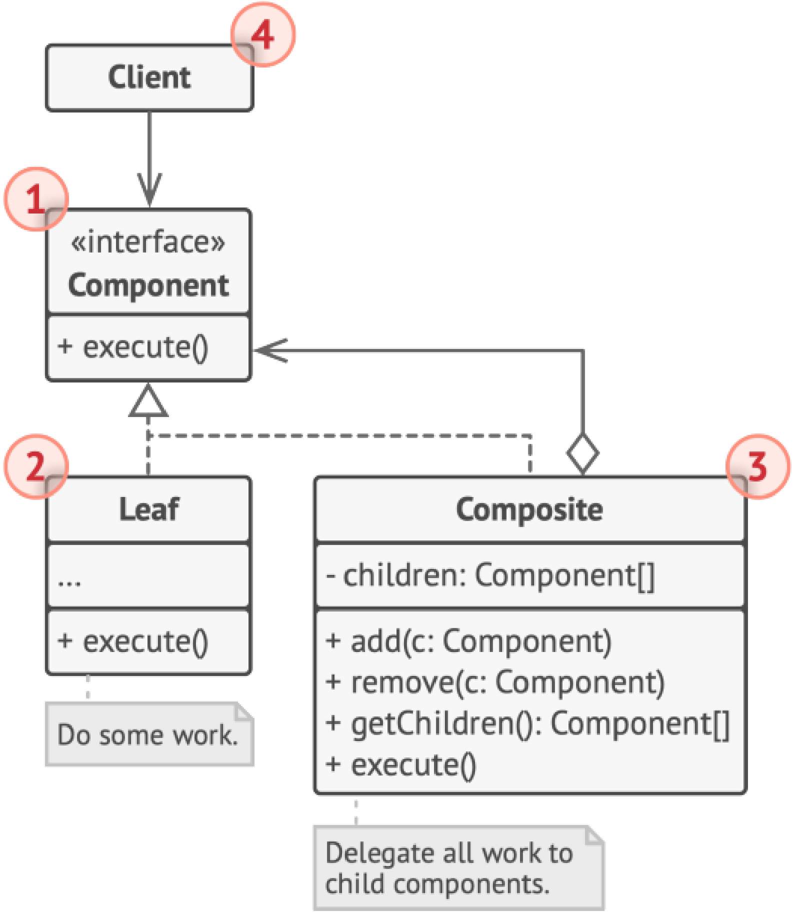
Composite is a structural design pattern that lets you compose objects into tree structures and then work with these structures as if they were individual objects.



Composite – When to use

- **Part-Whole Hierarchies**: Use Composite when you need to represent part-whole relationships in a hierarchical object structure.
- **Uniform Treatment**: Composite is ideal when individual objects and compositions should be treated uniformly by client code.
- **Simplification**: The pattern simplifies the client interaction with complex structures by treating them as a single entity.

Composite – UML



Composite – Participants

- **Component**: This is a base interface (or abstract class) that defines the common operations for both simple and complex objects in the hierarchy.
- **Leaf**: This class represents the end objects of a composition. A leaf has no children and implements all Component operations.
- **Composite**: This class can store children Components. It implements Component operations and delegates them to its children, if necessary.
- **Client**: The Client manipulates objects in the hierarchy using the Component interface.

Composite – Implementation step 1

Define the Component:

```
abstract class Component {
	protected String name;

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

	abstract void display();
}
```

Composite – Implementation step 2

Define the Leaf:

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

	void display() {
		System.out.println("Leaf: " + name);
	}
}
```

Composite – Implementation step 3

Define the Composite:

```
class Composite extends Component {
    private List<Component> children = new ArrayList<>();

    public Composite(String name) {
        super(name);
    }

    void addComponent(Component component) {
        children.add(component);
    }

    void removeComponent(Component component) {
        children.remove(component);
    }

    void display() {
        System.out.println("Composite: " + name);
        for (Component child : children) {
            child.display();
        }
    }
}
```

Composite – Implementation step 4

Use the Composite: use the Composite Pattern in the main method

```
public class Client {
    public static void main(String[] args) {
        Composite root = new Composite("root");
        Leaf leaf1 = new Leaf("leaf1");
        Composite composite1 = new Composite("composite1");

        root.addComponent(leaf1);
        root.addComponent(composite1);

        Leaf leaf2 = new Leaf("leaf2");
        Leaf leaf3 = new Leaf("leaf3");
        composite1.addComponent(leaf2);
        composite1.addComponent(leaf3);

        root.display();
    }
}
```

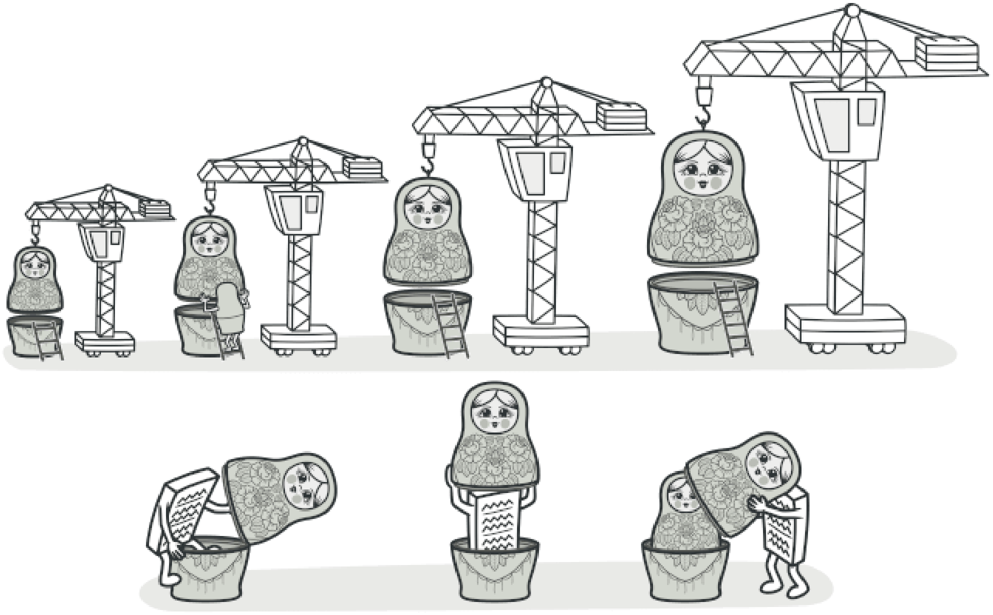
Composite – Pros and Cons

- You can work with complex tree structures more conveniently: use polymorphism and recursion to your advantage.
- *Open/Closed Principle*. You can introduce new element types into the app without breaking the existing code, which now works with the object tree.

- It might be difficult to provide a common interface for classes whose functionality differs too much. In certain scenarios, you'd need to overgeneralize the component interface, making it harder to comprehend.

Structural Pattern – Decorator

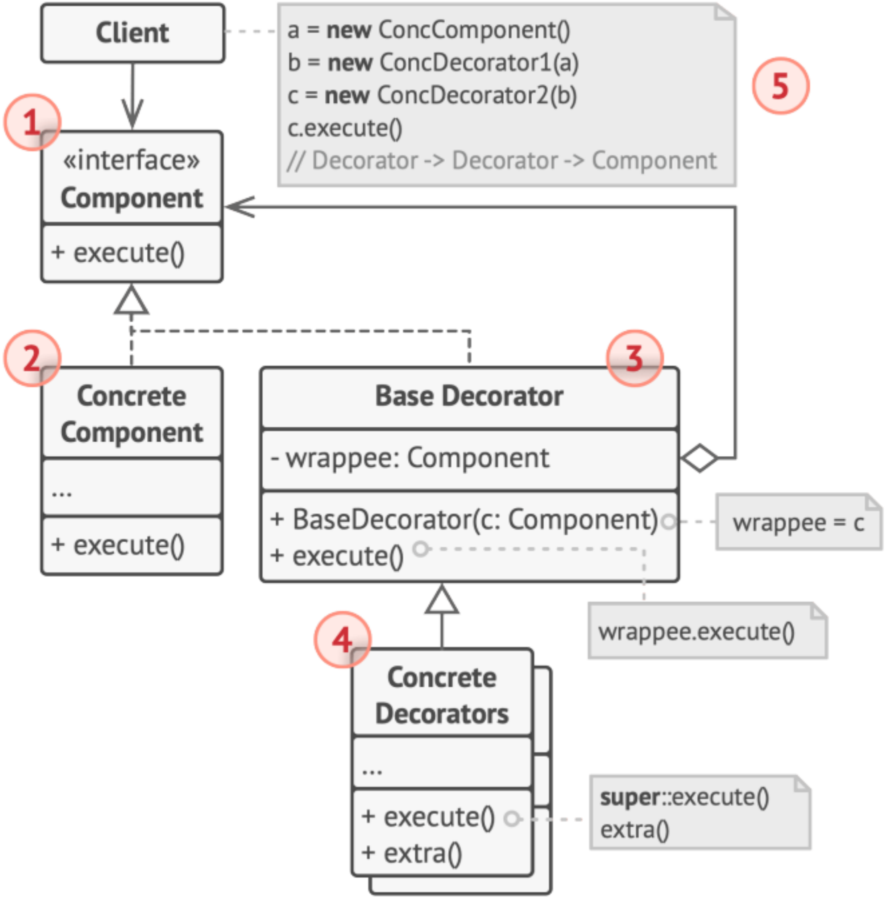
Decorator is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.



Decorator – When to use

- when you need to be able to assign extra behaviors to objects at runtime without breaking the code that uses these objects.
- **Part-Whole Hierarchies**: Use Composite when you need to represent part-whole relationships in a hierarchical object structure.
- **Uniform Treatment**: Composite is ideal when individual objects and compositions should be treated uniformly by client code.
- **Simplification**: The pattern simplifies the client interaction with complex structures by treating them as a single entity.

Decorator – UML



Decorator – Participants

- **Component**: An interface defining operations for dynamically extensible objects. Used by both Concrete Component and Decorator.
- **Concrete Component**: A class that implements Component, signifying objects to which we can add behaviors.
- **Decorator**: An abstract class holding a Component reference. It conforms to Component's interface and facilitates behavior extension.
- **Concrete Decorator**: Decorator subclasses that extend Component's behavior by adding new state or methods.

Decorator – Implementation step 1

Define the Component Interface:

```
// Component
interface Component {
    void operation();
}
```

Decorator – Implementation step 2

Create a Concrete Component:

```java
class ConcreteComponent implements Component {
    public void operation() {
        System.out.println("Performing operation in ConcreteComponent");
    }
}
```

Decorator – Implementation step 3

Create a Base Decorator:

```
abstract class Decorator implements Component {
	protected Component component;

	public Decorator(Component component) {
		this.component = component;
	}

	public void operation() {
		component.operation();
	}
}
```

Decorator – Implementation step 4

Create Concrete Decorators:

```java
class ConcreteDecoratorA extends Decorator {
    public ConcreteDecoratorA(Component component) {
        super(component);
    }

    public void operation() {
        System.out.println("Performing operation in ConcreteDecoratorA");
        super.operation();
    }
}

// ConcreteDecoratorB
class ConcreteDecoratorB extends Decorator {
    public ConcreteDecoratorB(Component component) {
        super(component);
    }

    public void operation() {
        System.out.println("Performing operation in ConcreteDecoratorB");
        super.operation();
    }
}
```

Decorator – Implementation step 5

Use the Decorator

```
public class Client {
    public static void main(String[] args) {
        Component component = new ConcreteComponent();
        component = new ConcreteDecoratorA(component);
        component = new ConcreteDecoratorB(component);
        component.operation();
    }
}
```

Decorator – Pros and Cons

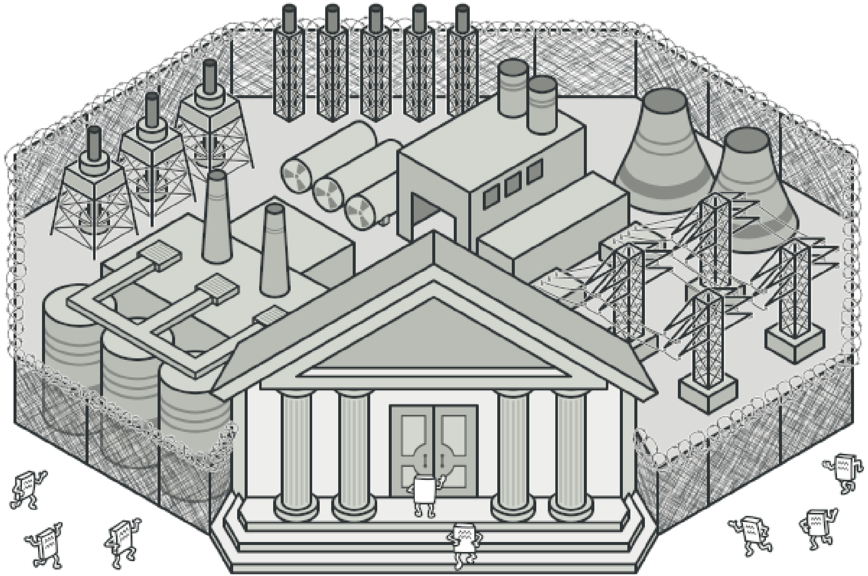
- You can extend an object's behavior without making a new subclass.
- You can add or remove responsibilities from an object at runtime.
- You can combine several behaviors by wrapping an object into multiple decorators.
- *Single Responsibility Principle*. You can divide a monolithic class that implements many possible variants of behavior into several smaller classes.
- It's hard to remove a specific wrapper from the wrappers stack.
- It's hard to implement a decorator in such a way that its behavior doesn't depend on the order in the decorators stack.
- The initial configuration code of layers might look pretty ugly.

Decorator – Use case in Java

- All subclasses of java.io.InputStream, OutputStream, Reader and Writer have constructors that accept objects of their own type.
- java.util.Collections, methods checkedXXX(), synchronizedXXX() and unmodifiableXXX().
- javax.servlet.http.HttpServletRequestWrapper and HttpServletResponseWrapper

Structural Pattern – Facade

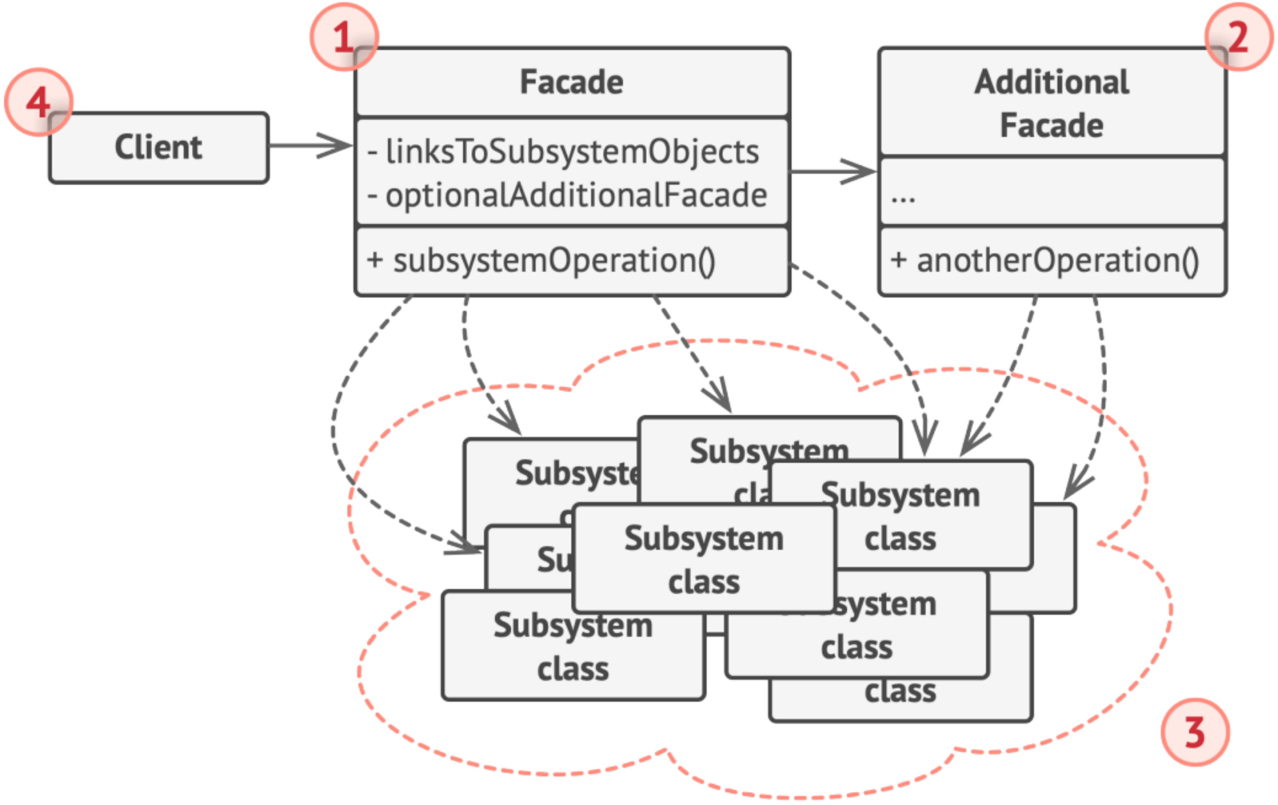
Facade is a structural design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes.



Facade – When to use

- Use the Facade pattern when you need to have a limited but straightforward interface to a complex subsystem.
- Use the Facade when you want to structure a subsystem into layers.

Facade – UML



1
2
3
4
Client
Facade
- linksToSubsystemObjects
- optionalAdditionalFacade
+ subsystemOperation()
Additional
Facade
...
+ anotherOperation()
Subsystem
class
Subsystem
class
Subsystem
class
Subsystem
class

Façade – Participants

- **Facade**: A class that provides a simplified interface to a complex subsystem.
- **Subsystems**: The classes that implement the underlying complexity. They aren't aware of the facade.
- **Client**: Code that uses the facade instead of interacting with the subsystems directly.

Façade – Implementation step 1

Step 1 Define Subsystem Classes:

```java
class Subsystem1 {
    public void operation1() {
        System.out.println("Subsystem1: Operation1");
    }
}

class Subsystem2 {
    public void operation2() {
        System.out.println("Subsystem2: Operation2");
    }
}

class Subsystem3 {
    public void operation3() {
        System.out.println("Subsystem3: Operation3");
    }
}
```

Façade – Implementation step 2

Step 2 Define Facade Classe:

```java
class Facade {
    private Subsystem1 subsystem1;
    private Subsystem2 subsystem2;
    private Subsystem3 subsystem3;
    private AdditionalFacade additionalFacade;

    public Facade() {
        this.subsystem1 = new Subsystem1();
        this.subsystem2 = new Subsystem2();
        this.subsystem3 = new Subsystem3();
        this.additionalFacade = new AdditionalFacade();
    }

    public void operation() {
        subsystem1.operation1();
        subsystem2.operation2();
        subsystem3.operation3();

        additionalFacade.additionalOperation();
    }
}
```

Façade – Implementation step 4

Step 2 Define Additional Facade :

```java
// Additional Facade
class AdditionalFacade {


    public AdditionalFacade() {

    }

    public void additionalOperation() {
        System.out.println("Additional operation in AdditionalFacade");
        facade.operation();
    }
}
```

Façade – Implementation step 3

Step 2 Use the Façade in client:

```
public class Main {
    public static void main(String[] args) {
        Facade facade = new Facade();
        Facade.operation();
    }
}
```

Façade – Pros and Cons

- You can isolate your code from the complexity of a subsystem.
- A facade class could be coupled to all classes of an app.

Facade – Use case in Java

- SLF4J