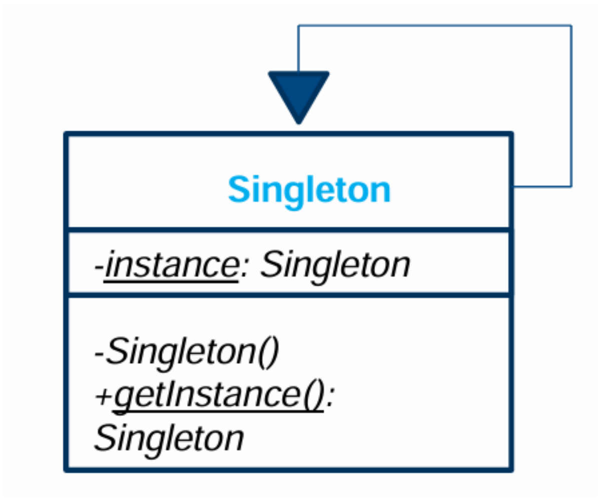
Creational Patterns

Singleton:

Esures that only one instance of an object is created.

So the class is responsible for ensuring that only one instance of an object is created

UML:

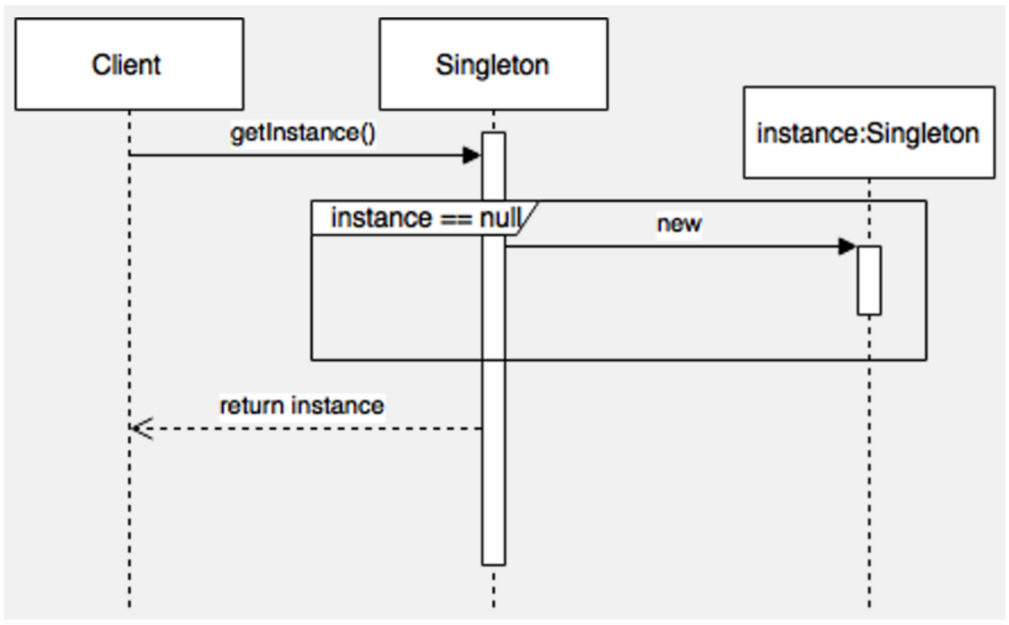


public class Singleton {

public static final Singleton C_INSTANCE = new Singleton();

}

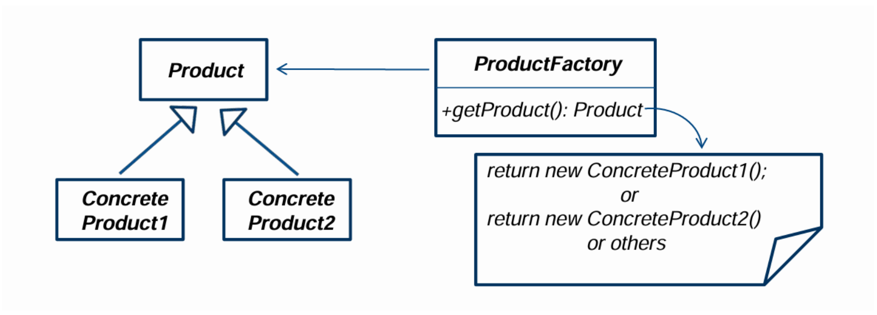
You can directly access the instance with Singleton.C_INSTANCE and sice its final we have no problem with declaring it as final

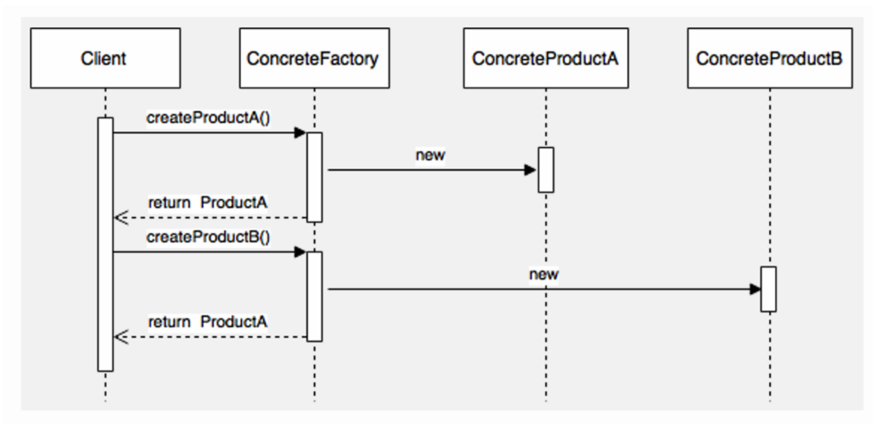


Static Factory Pattern

Hides object instanstation and lowers code coupling with implementation.

Defines an abstract interface which lets subclasses decide which class should be instantiated





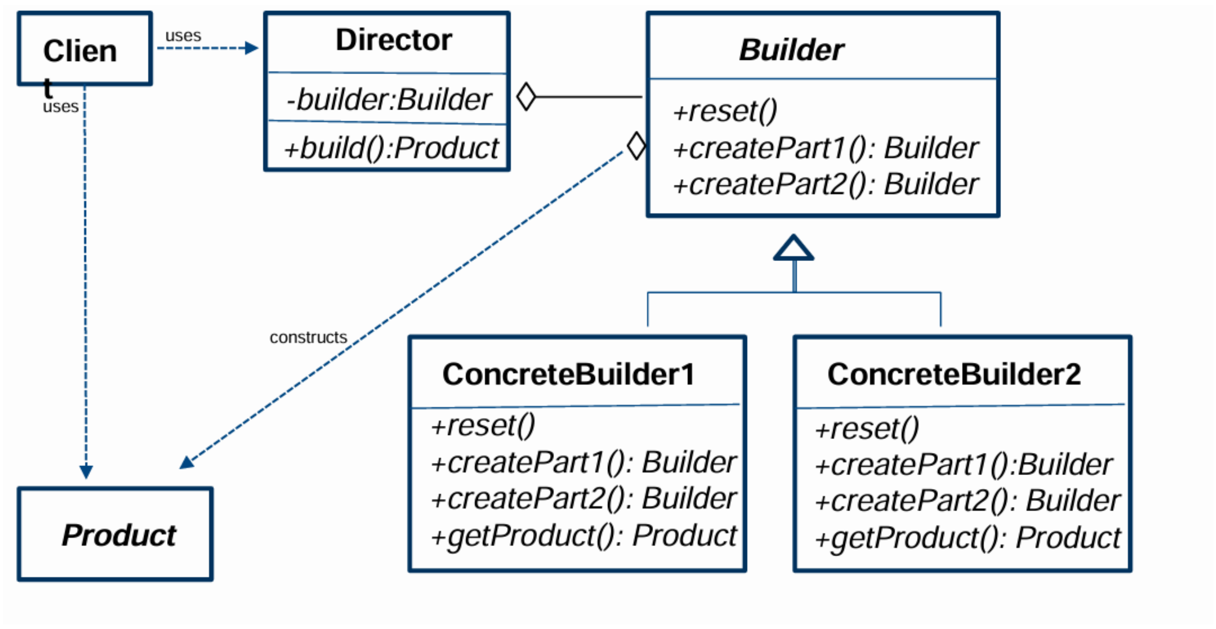
```
// Step 1: Define the Product interface

interface Product {

    void use();

}


// Step 2: Create Concrete Products with protected constructors to hide
instantiation

class ConcreteProductA implements Product {

    protected ConcreteProductA() {}  // Protected constructor to restrict direct
instantiation


    @Override

    public void use() {

        System.out.println("Using ConcreteProductA");

    }

}


class ConcreteProductB implements Product {

    protected ConcreteProductB() {}  // Protected constructor to restrict direct
instantiation
```

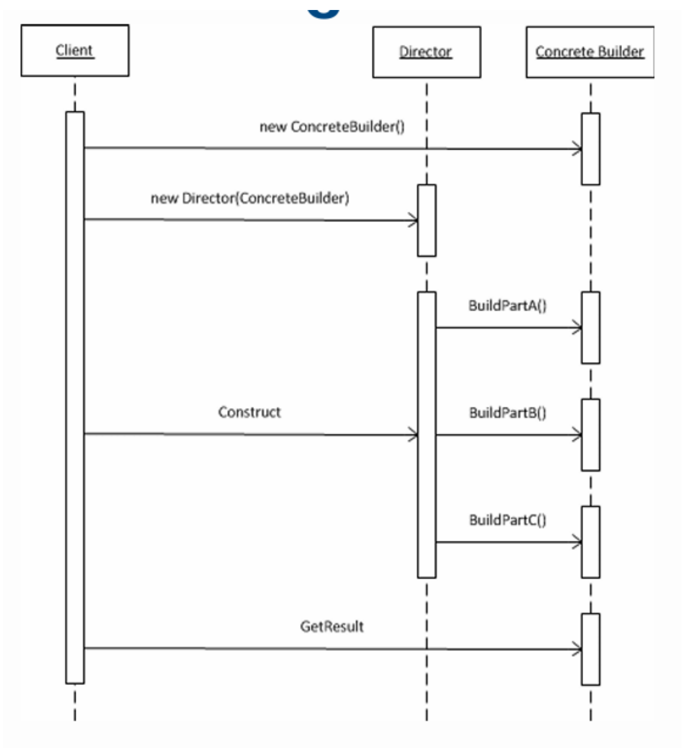
```
    @Override
    public void use() {
        System.out.println("Using ConcreteProductB");
    }
}

// Step 3: Define a single Factory with a non-static factory method
class ProductFactory {
    // Enum to specify product types
    public enum ProductType {
        PRODUCT_A,
        PRODUCT_B
    }

    // Non-static factory method that creates a product based on the type parameter
    public Product createProduct(ProductType type) {
        switch (type) {
            case PRODUCT_A:
                return new ConcreteProductA();
            case PRODUCT_B:
                return new ConcreteProductB();
            default:
                throw new IllegalArgumentException("Unknown product type");
        }
    }
}
```

Builder

Is there to eliminate the Telescoping Constructor Antipattern (many constructor with optional arguments)

Separates the construction of an complex object from it’s representation. Builds the object as a configurable step by step part instead of one shot. Client has control over the creation process





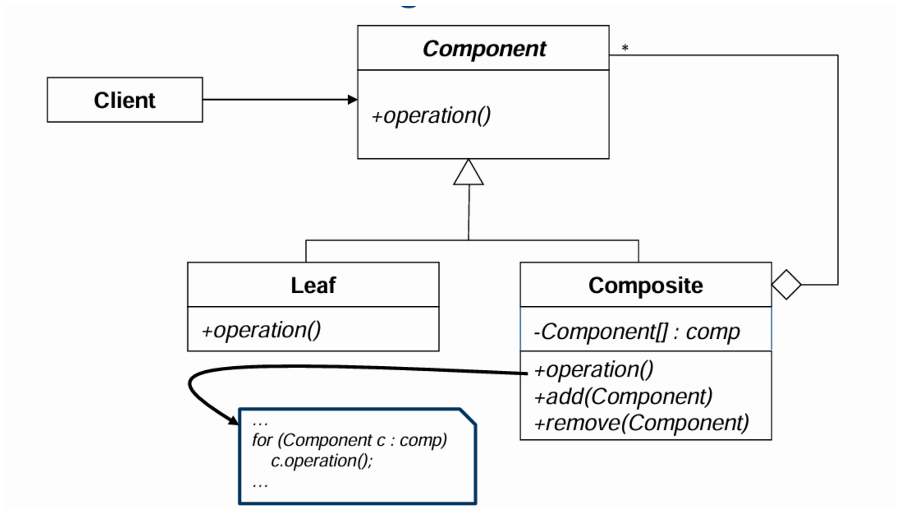
```
// Step 1: Define the Product class with a nested static Builder class
class Product {
    // Fields for the Product
    private final String part1;  // Mandatory
    private final String part2;  // Mandatory
    private String optionalPart1; // Optional
    private String optionalPart2; // Optional
    // Private constructor to prevent direct instantiation
    private Product(Builder builder) {
        this.part1 = builder.part1;
        this.part2 = builder.part2;
        this.optionalPart1 = builder.optionalPart1;
        this.optionalPart2 = builder.optionalPart2;
    }

    @Override
    public String toString() {
        return "Product [Part1 = " + part1 + ", Part2 = " + part2
                + ", OptionalPart1 = " + optionalPart1 + ", OptionalPart2 =
" + optionalPart2 + "]";
    }

    // Step 2: Define the static nested Builder class
    public static class Builder {
        // Mandatory fields
        private final String part1;
        private final String part2;

        // Optional fields with default values
        private String optionalPart1 = "Default Optional Part1";
        private String optionalPart2 = "Default Optional Part2";

        // Constructor for Builder with mandatory parameters
        public Builder(String part1, String part2) {
            this.part1 = part1;
            this.part2 = part2;
        }

        // Method to set optionalPart1 and return the Builder object
        public Builder setOptionalPart1(String optionalPart1) {
            this.optionalPart1 = optionalPart1;
            return this;
        }
        // Method to set optionalPart2 and return the Builder object
        public Builder setOptionalPart2(String optionalPart2) {
            this.optionalPart2 = optionalPart2;
            return this;
        }

        // Build method to create the Product instance with the current
Builder state
        public Product build() {
            return new Product(this);  // Pass the builder itself to
Product constructor
        }
    }
}

// Step 3: Using the Builder in a Client
public class BuilderPatternDemo {
    public static void main(String[] args) {
        // Create a Product with mandatory parameters and one optional
parameter
        Product product = new Product.Builder("Engine", "Wheels") //
Mandatory parts
                .setOptionalPart1("Leather Seats")  // Optional part1
                .build();

        // Output the product to verify its creation
        System.out.println(product);
        // Output: Product [Part1 = Engine, Part2 = Wheels, OptionalPart1 =
Leather Seats, OptionalPart2 = Default Optional Part2]

        // Create another Product with all parameters
        Product product2 = new Product.Builder("Engine", "Wheels") //
Mandatory parts
                .setOptionalPart1("Leather Seats")
                .setOptionalPart2("Sunroof")
                .build();
        System.out.println(product2);

        // Output: Product [Part1 = Engine, Part2 = Wheels, OptionalPart1 =
Leather Seats, OptionalPart2 = Sunroof]

    }

}
```

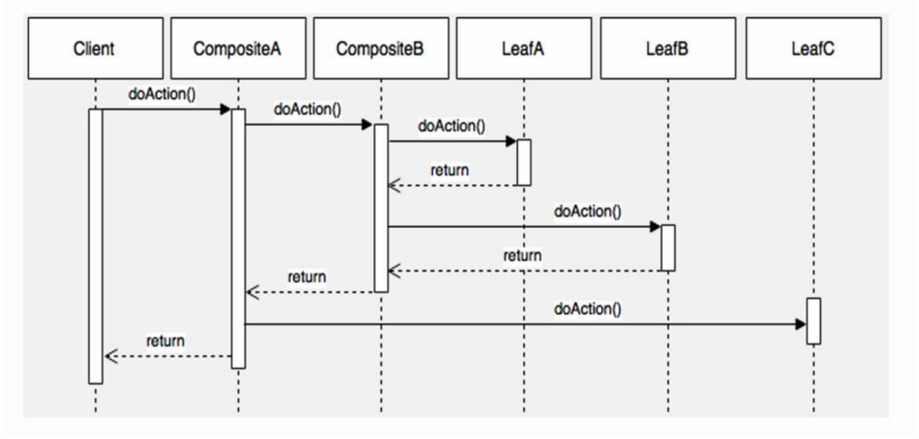
Structural Patterns

Composite Pattern

Allows to compose objects into a tree structure and lets clients treat individual objects and compositional objects uniformly

Every element in the structure operates with a uniform interface, adding new components is easy and client code remains unchanged



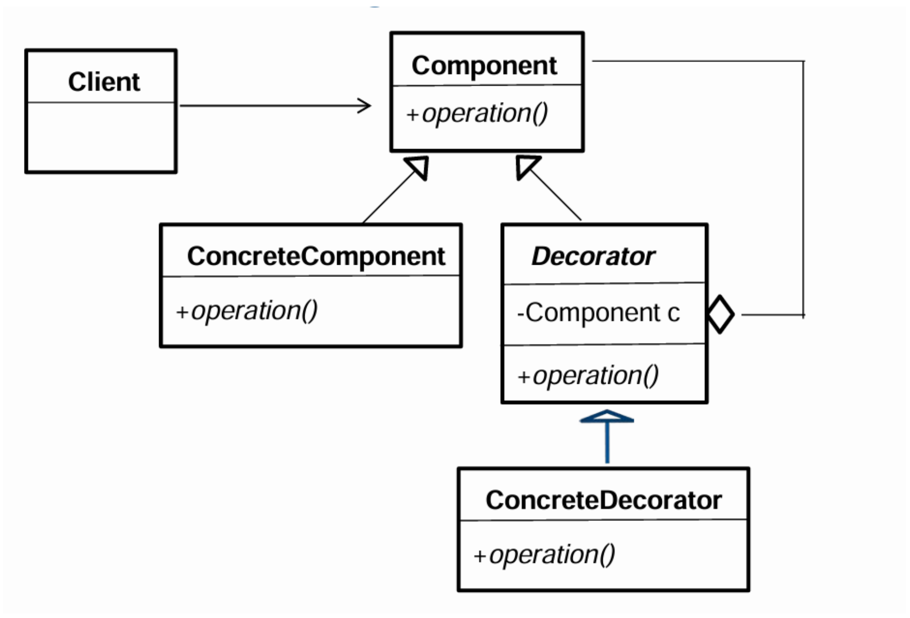


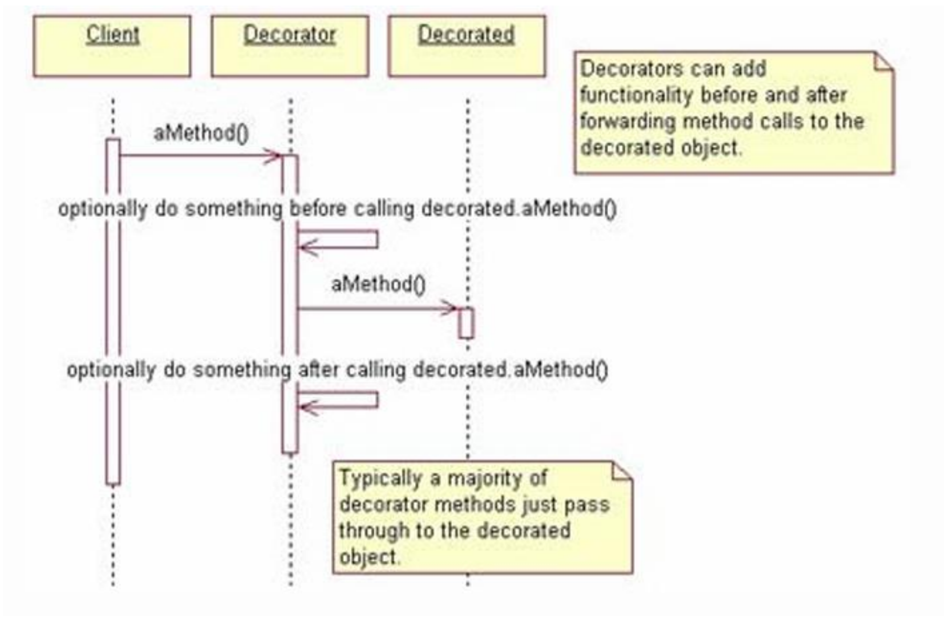
```
import java.util.ArrayList;

import java.util.List;


// Step 1: Component Interface
```

```
interface FileSystemComponent {

    void showDetails(); // This operation will be used by both files and
folders

}


// Step 2: Leaf Class (File)

class File implements FileSystemComponent {

    private String name;

    private int size;


    public File(String name, int size) {

        this.name = name;

        this.size = size;

    }


    @Override

    public void showDetails() {

        System.out.println("File: " + name + ", Size: " + size + "KB");

    }

}


// Step 3: Composite Class (Folder)

class Folder implements FileSystemComponent {

    private String name;

    private List<FileSystemComponent> components = new ArrayList<>();


    public Folder(String name) {

        this.name = name;

    }


    // Add a file or folder to this folder

    public void addComponent(FileSystemComponent component) {

        components.add(component);

    }


    // Remove a file or folder from this folder
    public void removeComponent(FileSystemComponent component) {
        components.remove(component);
    }

    @Override
    public void showDetails() {
        System.out.println("Folder: " + name);
        for (FileSystemComponent component : components) {
            component.showDetails();  // Recursive call to show details of
contained components
        }
    }
}

// Step 4: Client Code to Test the Composite Pattern
public class CompositePatternDemo {
    public static void main(String[] args) {
        // Creating individual files
        File file1 = new File("Document.txt", 15);
        File file2 = new File("Image.jpg", 45);
        File file3 = new File("Video.mp4", 300);

        // Creating a folder and adding files to it
        Folder folder1 = new Folder("Media");
        folder1.addComponent(file1);
        folder1.addComponent(file2);

        // Creating a subfolder and adding files to it
        Folder folder2 = new Folder("Movies");
        folder2.addComponent(file3);

        // Adding subfolder to main folder
        folder1.addComponent(folder2);

        // Displaying the structure
        folder1.showDetails();
    }
}
```

Decorator Pattern

When objects are alike in some fundamental way the decorator pattern avoids the complexity of having many derived classes, the decorator is an object that modifies the behavior or adds functionality to another object





Client
Decorator
Decorated
Decorators can add functionality before and after forwarding method calls to the decorated object.
aMethod()
optionally do something before calling decorated.aMethod()
aMethod()
optionally do something after calling decorated.aMethod()
Typically a majority of decorator methods just pass through to the decorated object.

```
// Component interface
interface Coffee {
    double cost();
}
```

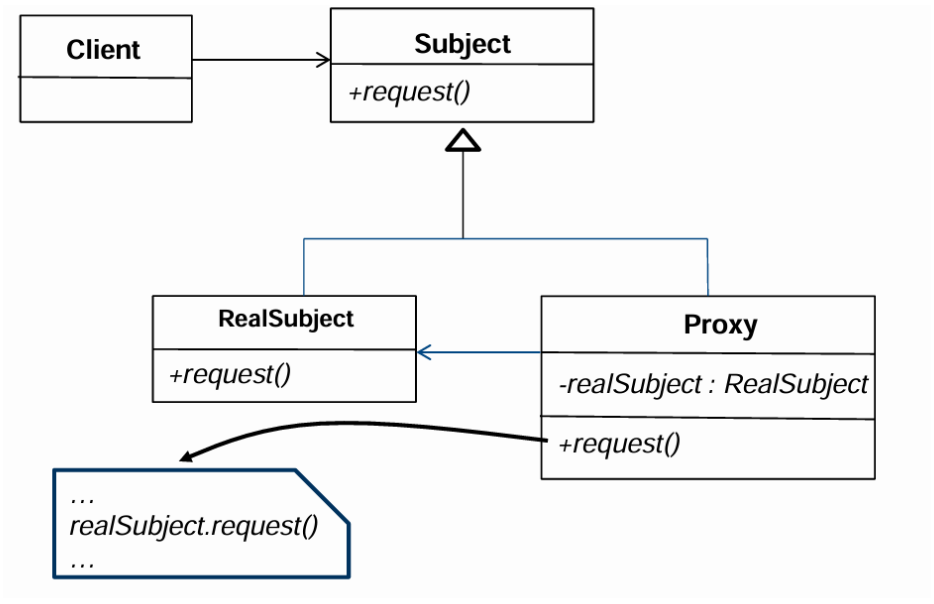
```java
// Concrete component
class SimpleCoffee implements Coffee {
    public double cost() {
        return 5;
    }
}

// Decorator class implements Coffee interface
class CoffeeDecorator implements Coffee {
    protected Coffee decoratedCoffee;

    public CoffeeDecorator(Coffee coffee) {
        this.decoratedCoffee = coffee;
    }

    public double cost() {
        return decoratedCoffee.cost();
    }
}

// Concrete decorator adds extra functionality
class MilkDecorator extends CoffeeDecorator {
    public MilkDecorator(Coffee coffee) {
        super(coffee);
    }

    public double cost() {
        return decoratedCoffee.cost() + 2;
    }
}

class SugarDecorator extends CoffeeDecorator {
    public SugarDecorator(Coffee coffee) {
        super(coffee);
    }
    public double cost() {
        return decoratedCoffee.cost() + 1;
    }
}

// Main class to test
public class Main {
    public static void main(String[] args) {
        Coffee coffee = new SimpleCoffee();
        System.out.println("Cost of Simple Coffee: " + coffee.cost());

        coffee = new MilkDecorator(coffee);
        System.out.println("Cost of Coffee with Milk: " + coffee.cost());

        coffee = new SugarDecorator(coffee);
        System.out.println("Cost of Coffee with Milk and Sugar: " +
coffee.cost());
    }
}
```

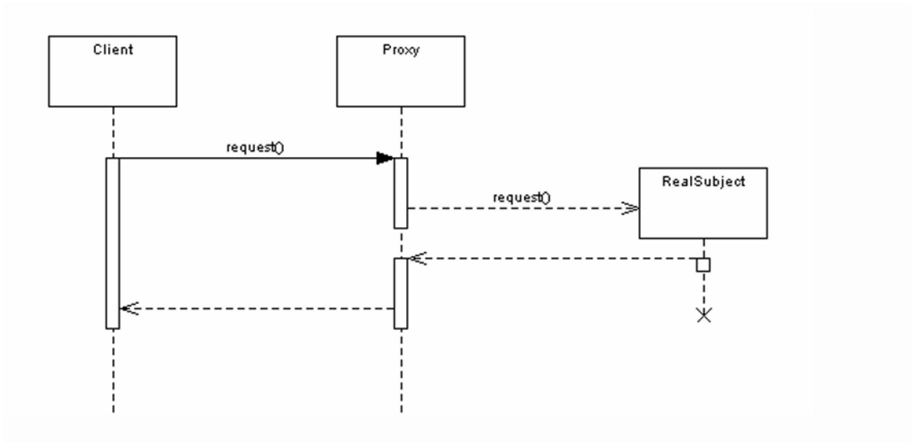
Proxy Pattern

Provides a surrogate for another object for speed access or security. It provides an interface same as the one from the real subject and it could also be responsible for creating and deleting it.

There are a few types of proxies:

- Remote Proxy: Local represative of object in different address space
- Virtual Proxy: creates object on demand
- Protection Proxy: controls access to original object
- Smart Reference: replaces bare pointer by counting references
- Firewall Proxy: protects clients from bad clients
- Cache Proxy: provides temporary storage
- Synchronization Proxy: provides multiple access to a target



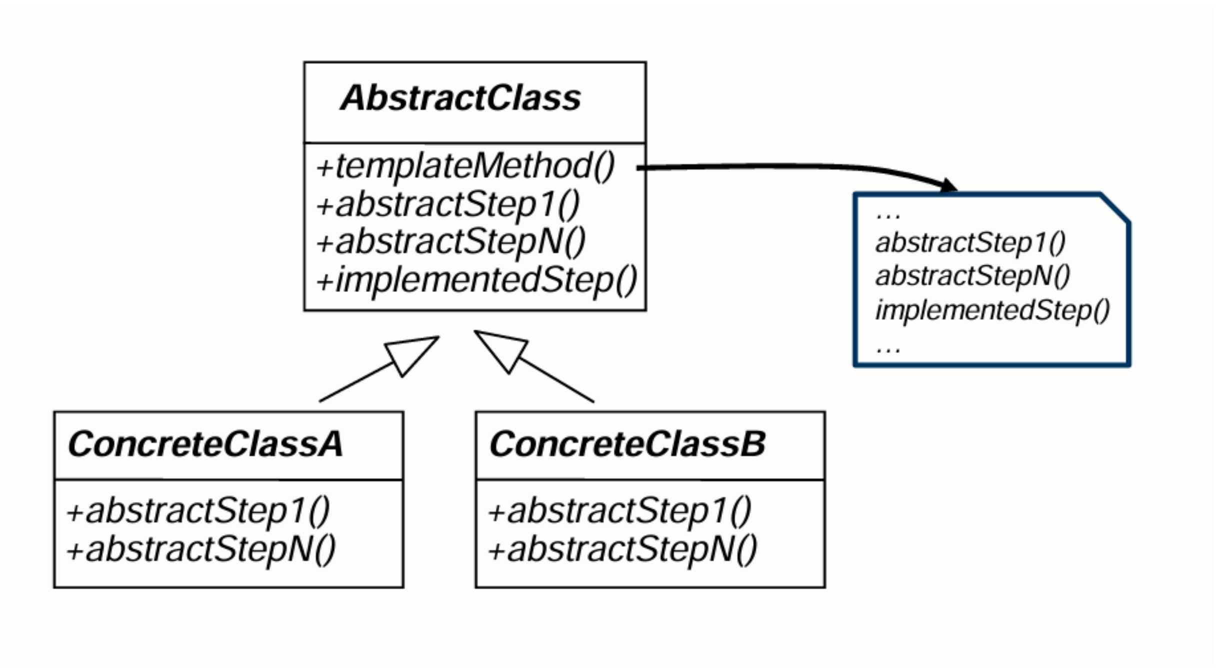


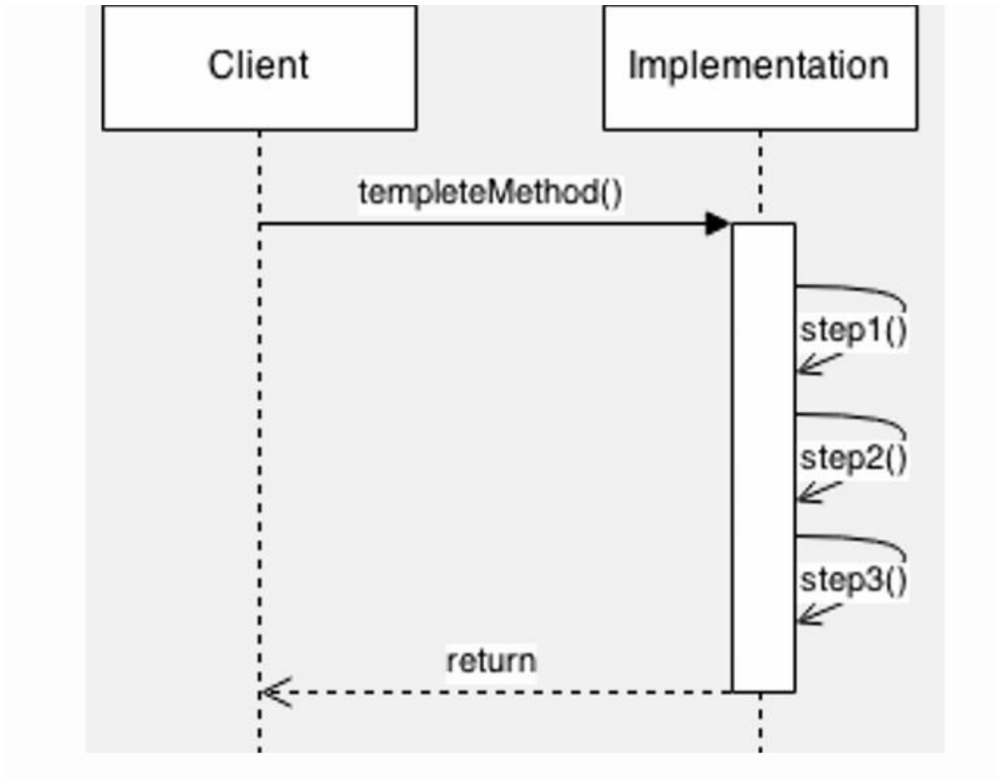
```
// Subject Interface
interface Subject {
    void request();
}


// RealSubject: The actual object we want to access
class RealSubject implements Subject {
    public RealSubject() {
        // Simulate expensive object creation
        System.out.println("RealSubject: Expensive object created!");
    }


    public void request() {
        System.out.println("RealSubject: Request processed.");
    }
}

// Proxy: Controls access to the RealSubject
class Proxy implements Subject {
    private RealSubject realSubject;

    public void request() {
        if (realSubject == null) {
            realSubject = new RealSubject();  // Lazy initialization
        }
        realSubject.request();  // Delegate the request to the real object
    }
}

// Main class to test the Proxy pattern
public class ProxyPatternExample {
    public static void main(String[] args) {
        Subject subject = new Proxy();

        System.out.println("Making the first request...");
        subject.request();  // This will create RealSubject and make the
request

        System.out.println("\nMaking the second request...");
        subject.request();  // This will use the existing RealSubject
without creating a new one
    }
}
```

Behavioral Patterns

Template Pattern

Defines the Skeleton of an algorithm as a sequence of methods which allows to always have the same sequence with different implementations. Uses the Hollywood principle: Don’t calls us, we call you



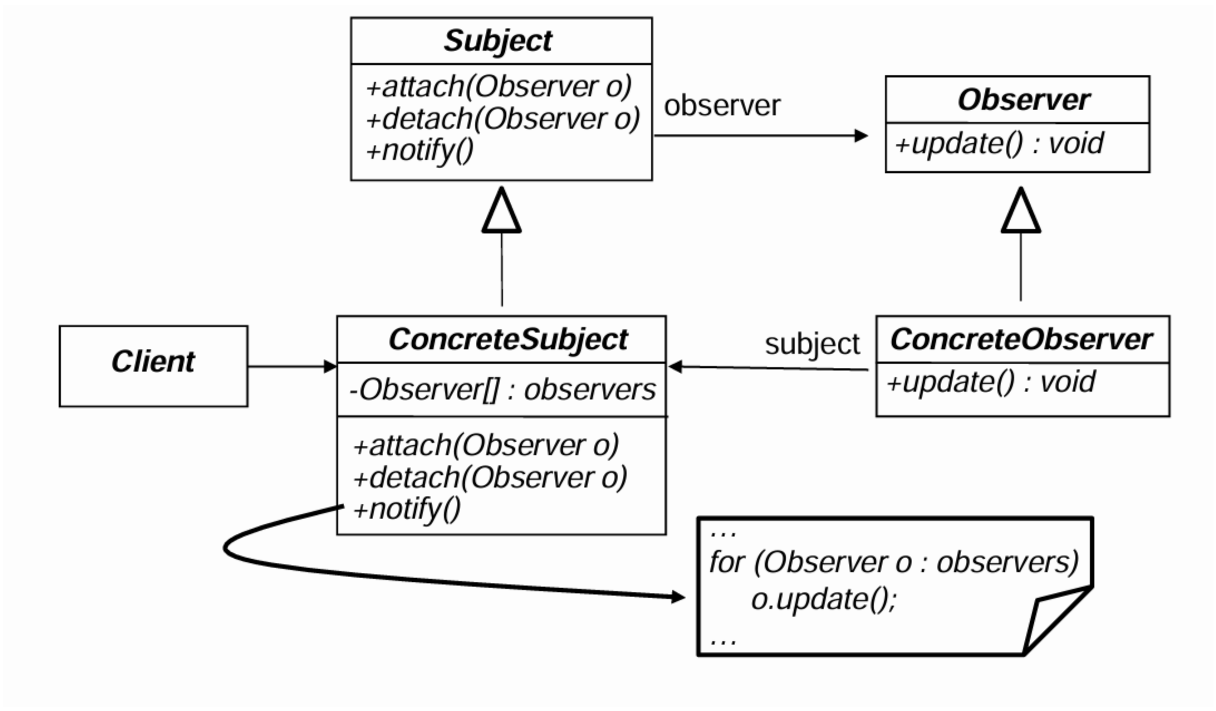


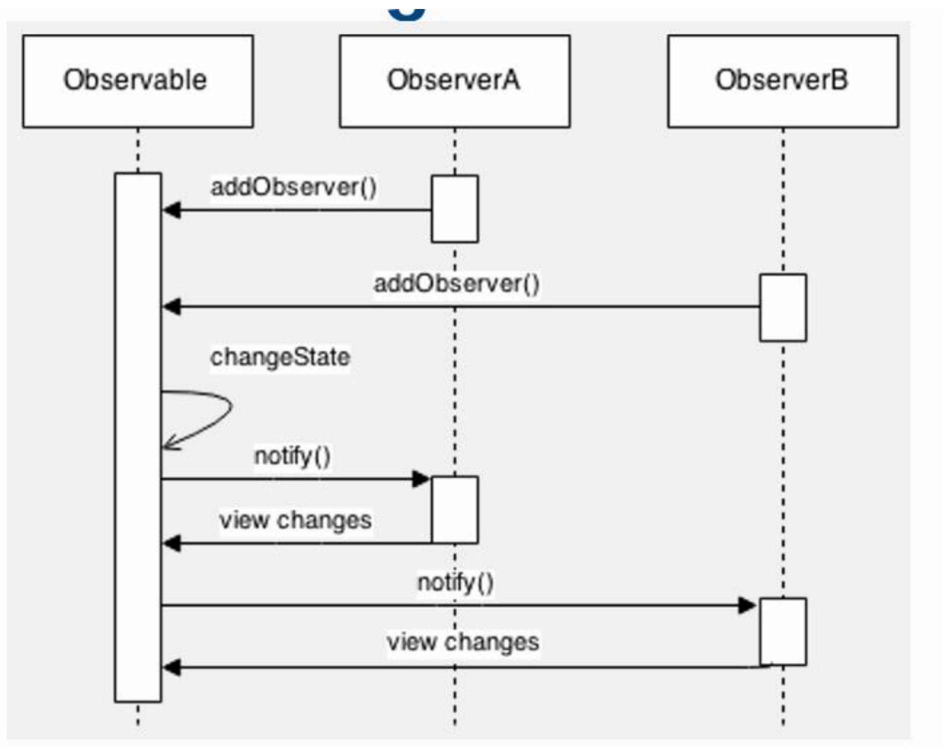
```
// Abstract class defining the template method
abstract class Game {

    // Template method that defines the structure of the algorithm
    public final void play() {
        startGame();
        playTurns();
        endGame();
```

```java
    }

    // Abstract methods to be implemented by concrete classes
    protected abstract void startGame();
    protected abstract void playTurns();
    protected abstract void endGame();
}

// Concrete class implementing the template
class Chess extends Game {

    @Override
    protected void startGame() {
        System.out.println("Starting a game of Chess!");
    }

    @Override
    protected void playTurns() {
        System.out.println("Players take turns moving pieces.");
    }

    @Override
    protected void endGame() {
        System.out.println("Game Over. Checkmate or stalemate!");
    }
}

// Another concrete class
class Football extends Game {

    @Override
    protected void startGame() {
        System.out.println("Starting a game of Football!");
    }
    @Override
    protected void playTurns() {
        System.out.println("Players take turns scoring goals.");
    }

    @Override
    protected void endGame() {
        System.out.println("Game Over. Final score is displayed!");
    }
}

// Client code
public class Main {
    public static void main(String[] args) {
        // Play Chess
        Game chess = new Chess();
        chess.play();

        System.out.println("\n");

        // Play Football
        Game football = new Football();
        football.play();
    }
}
```

Observer Pattern

Implements an one to many relationship one subject is being observed by many observers which can be attached or removed from the list of observers.





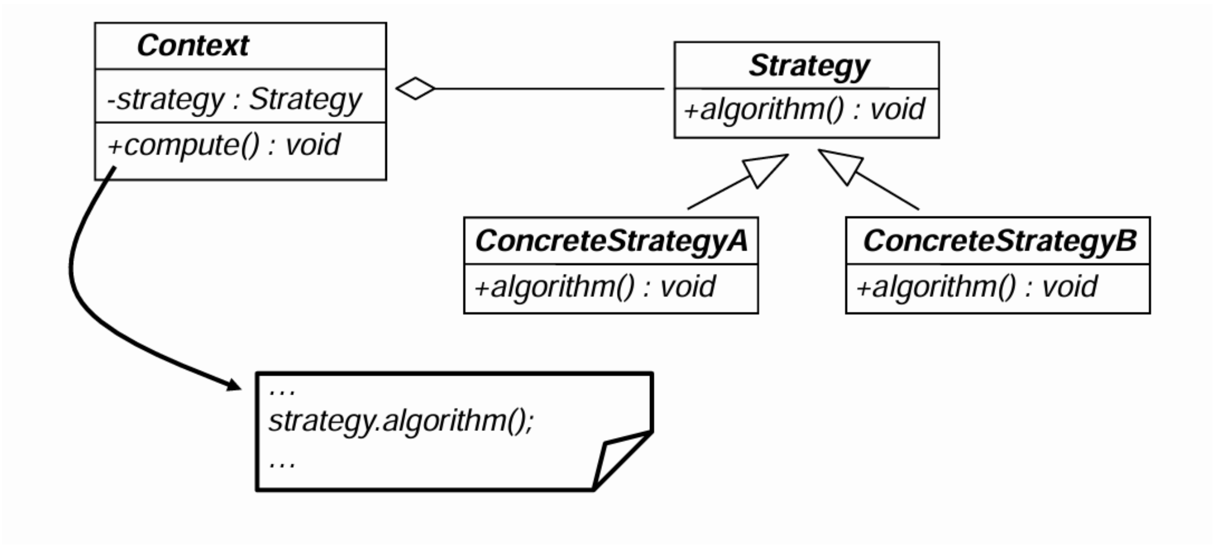
```
import java.util.ArrayList;

import java.util.List;


// Observer interface

interface Observer {

    void update(String message);

}
```

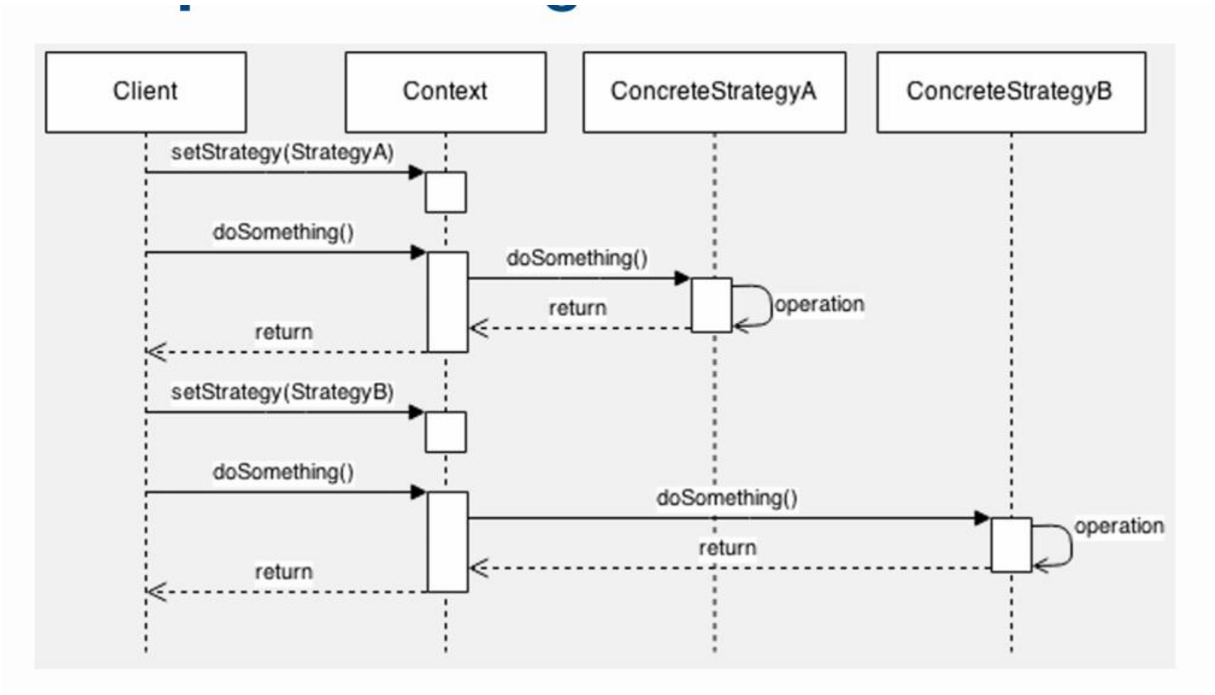
```java
// Subject interface (also called Publisher)
interface Subject {
    void registerObserver(Observer observer);
    void removeObserver(Observer observer);
    void notifyObservers();
}

// Concrete class for Subject (also called ConcretePublisher)
class NewsAgency implements Subject {
    private List<Observer> observers = new ArrayList<>();
    private String latestNews;

    @Override
    public void registerObserver(Observer observer) {
        observers.add(observer);
    }

    @Override
    public void removeObserver(Observer observer) {
        observers.remove(observer);
    }

    @Override
    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(latestNews);
        }
    }

    public void setLatestNews(String news) {
        this.latestNews = news;
        notifyObservers();  // Notify all observers of the change
    }
}
```

```
// Concrete Observer (Subscriber)
class NewsChannel implements Observer {
    private String channelName;

    public NewsChannel(String channelName) {
        this.channelName = channelName;
    }

    @Override
    public void update(String message) {
        System.out.println(channelName + " received news update: " +
message);
    }
}

// Main class to demonstrate the Observer Pattern
public class Main {
    public static void main(String[] args) {
        // Create a subject (News Agency)
        NewsAgency newsAgency = new NewsAgency();

        // Create observers (News Channels)
        NewsChannel channel1 = new NewsChannel("CNN");
        NewsChannel channel2 = new NewsChannel("BBC");

        // Register observers with the subject
        newsAgency.registerObserver(channel1);
        newsAgency.registerObserver(channel2);

        // Change the state of the subject and notify observers
        newsAgency.setLatestNews("Breaking News: Observer Pattern
Explained!");

        System.out.println("\n--- Now removing BBC channel ---\n");

        // Remove an observer and update again
        newsAgency.removeObserver(channel2);


        // Change the state again and notify the remaining observers

        newsAgency.setLatestNews("Latest Update: Observer Pattern is very
useful.");

    }

}
```

Strategy Pattern

Defines a group of classes the represent a set of available behaviours that can be plugged into an application. It allows changing the behaviour of an object without changing the object





```
// Strategy interface
```

```
interface PaymentStrategy {

    void pay(int amount);

}


// Concrete strategy for Credit Card payment

class CreditCardPayment implements PaymentStrategy {

    private String cardNumber;


    public CreditCardPayment(String cardNumber) {

        this.cardNumber = cardNumber;

    }


    @Override

    public void pay(int amount) {

        System.out.println("Paid " + amount + " using Credit Card (" +
cardNumber + ")");

    }

}


// Concrete strategy for PayPal payment

class PayPalPayment implements PaymentStrategy {

    private String email;


    public PayPalPayment(String email) {

        this.email = email;

    }


    @Override

    public void pay(int amount) {

        System.out.println("Paid " + amount + " using PayPal (" + email +
")");

    }

}


// Context class that uses the strategy

class ShoppingCart {
    private PaymentStrategy paymentStrategy;

    public void setPaymentStrategy(PaymentStrategy paymentStrategy) {
        this.paymentStrategy = paymentStrategy;
    }

    public void checkout(int amount) {
        if (paymentStrategy != null) {
            paymentStrategy.pay(amount);
        } else {
            System.out.println("No payment strategy selected!");
        }
    }
}

// Main class to demonstrate the Strategy Pattern
public class Main {
    public static void main(String[] args) {
        // Create the context (ShoppingCart)
        ShoppingCart cart = new ShoppingCart();

        // Set strategy to CreditCardPayment and checkout
        cart.setPaymentStrategy(new CreditCardPayment("1234-5678-9876-
5432"));
        cart.checkout(100);  // Paid 100 using Credit Card (1234-5678-9876-
5432)

        System.out.println("\n");

        // Set strategy to PayPalPayment and checkout
        cart.setPaymentStrategy(new PayPalPayment("user@example.com"));
        cart.checkout(200);  // Paid 200 using PayPal (user@example.com)
    }
}
```