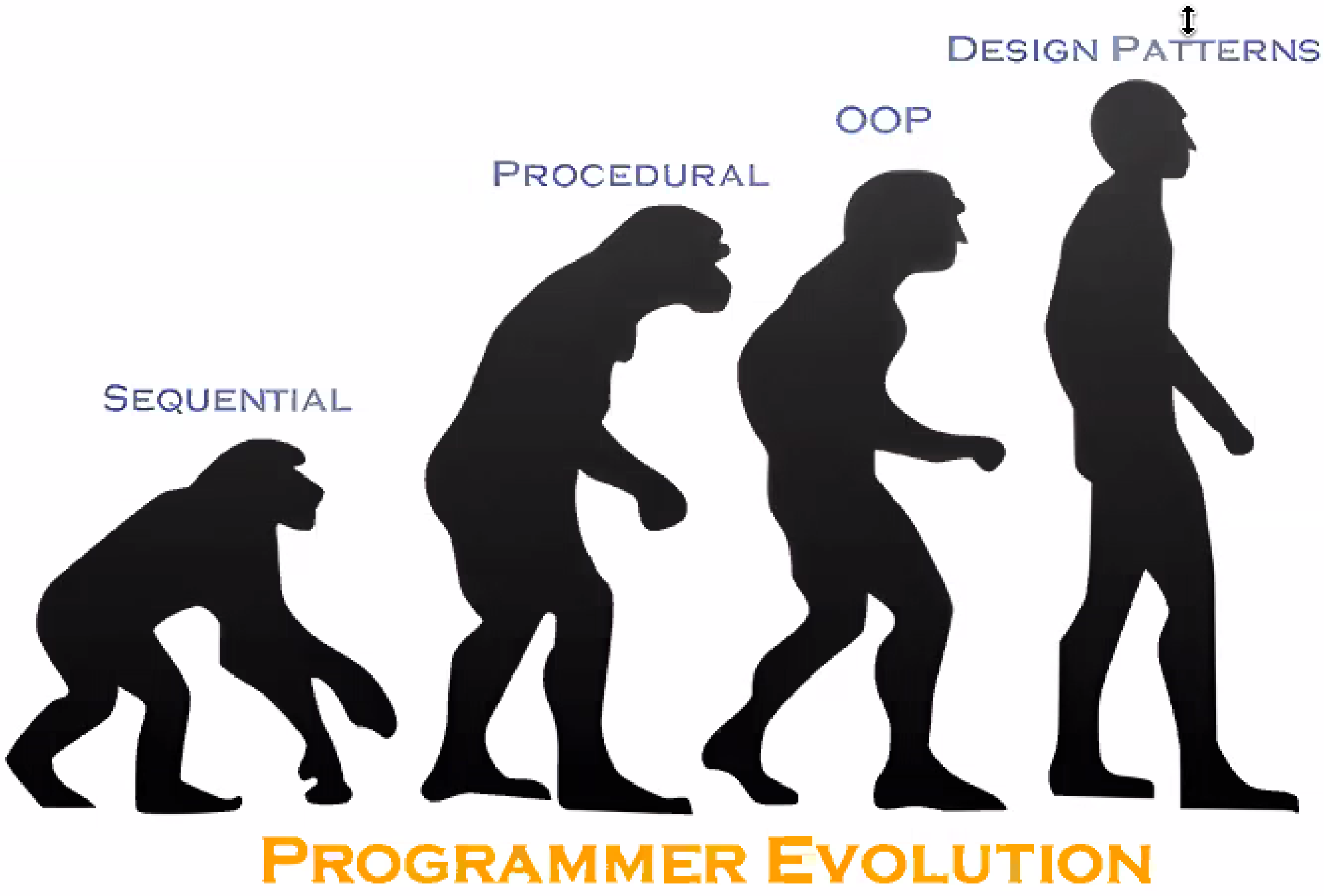
OODP I

Object-Oriented Design Patterns

Lecturer : Dr.Tharaka Sankalpa

Object-Oriented Design Patterns

*A design pattern is a **reusable solution to a common problem** in software design. **It provides a template** for how to solve issues in a way that can be adapted to different situations.*



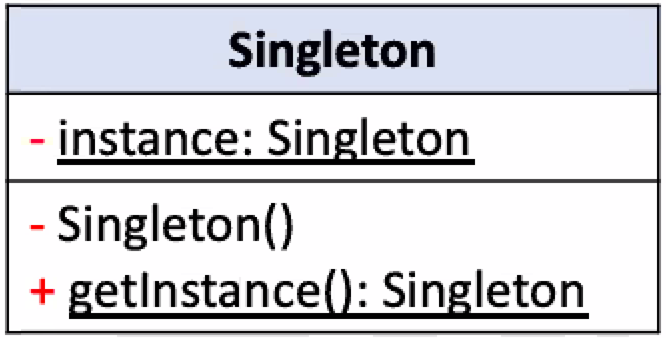
Creational Patterns *Focus on creating objects in a flexible way*
• **Singleton Design Pattern**
• **Factory Design Pattern**

Structural Patterns *Focus on how objects are arranged to build larger structures and simplify complex systems.*
• **Adapter Design Pattern**
• **Facade Design Pattern**
• **Proxy Design Pattern**

Behavioral Patterns *Focus on how objects work together and share tasks.*
• **Strategy Design Pattern**
• **Template Method Design Pattern**
• **Command Design Pattern**
• **Iterator Design Pattern**
• **State Design Pattern**
• **Observer Design Pattern**

- **Singleton Design Pattern**

Single Instance: *Only one instance of the class is created.*
Global Access: *Provides a way to access that single instance from anywhere in the application.*



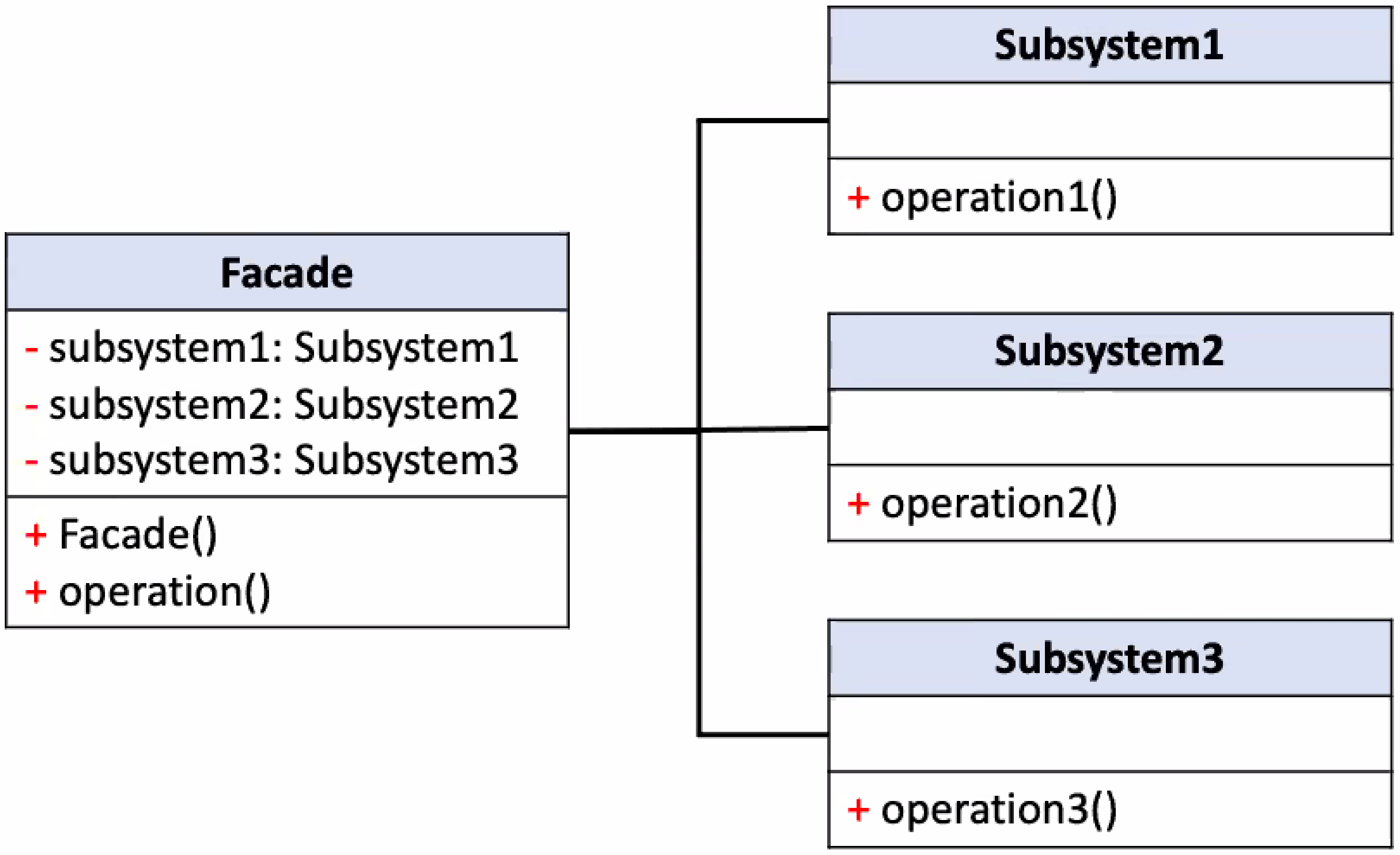
```java
public class A {
    private static A a;

    private A() { }

    public static A getA() {
        if (a == null) {
            a = new A();
        }
        return a;
    }
}

class Test {
    public static void main(String[] args) {

        A a1 = A.getA();
        A a2 = A.getA();
        System.out.println(a1 == a2);
    }
}
```

+Singleton
- instance: Singleton
- Singleton() + getInstance(): Singleton

+A
- a: A
- A() + getA(): A

- **Facade Design Pattern**

Simplified Interface: *Provides a single, simplified interface to a complex subsystem.*
Encapsulation: *Hides the complexities of the subsystem from the client.*

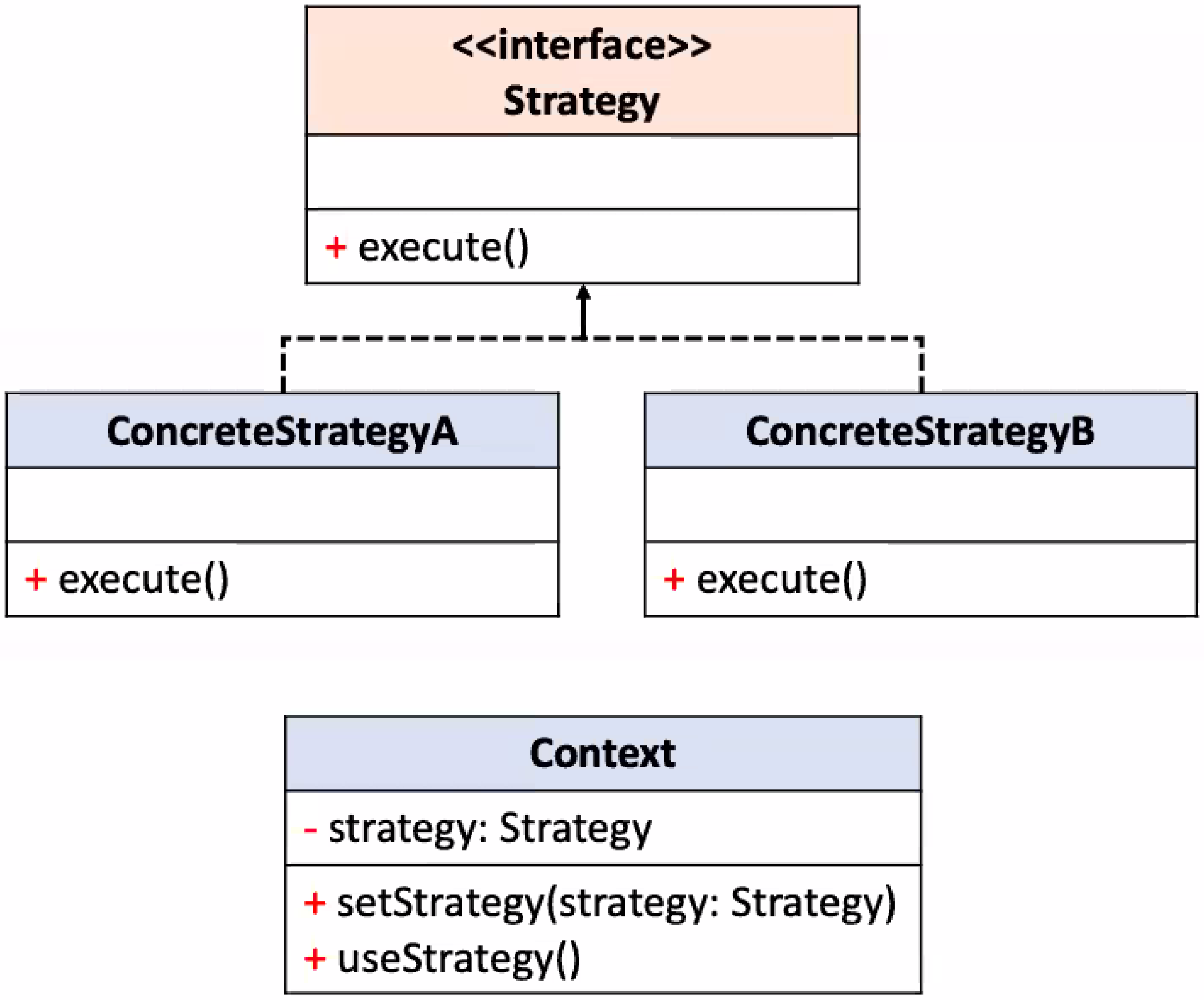


```java
class A {
   public void a1() {
      System.out.println("A a1");
   }

   public void a2() {
      System.out.println("A a2");
   }
}

class B {
   public void b1() {
      System.out.println("B b1");
   }
}

class C {
   public void c1() {
      System.out.println("C c1");
   }
}
```

```java
class X {
   private A a;
   private B b;
   private C c;

   public X() {
      a = new A();
      b = new B();
      c = new C();
   }

   public void x1() {
      a.a1();
      b.b1();
      a.a2();
      c.c1();
   }
}

class Test {
   public static void main(String[] args) {
      X x = new X();
      x.x1();
   }
}
```

- **Strategy Design Pattern**

Interchangeable: *Makes algorithms interchangeable without changing the code that uses them.*
Flexible: Lets you switch algorithms at runtime based on the situation.



```java
interface Strategy{
  public abstract boolean check(String text);
}

class SahanStrategy implements Strategy{
  @Override
  public boolean check(String text) {
    return text.contains("J");
  }
}

class KasunStrategy implements Strategy{
  @Override
  public boolean check(String text) {

    boolean b = false;

    for (int i = 0; i < text.length(); i++) {
      if(text.charAt(i)=='J'){
        b = true;
        break;
      }
    }
    return b;
  }
}
```

```java
class Context{
  private Strategy strategy;

  public void setStrategy(Strategy strategy) {
    this.strategy = strategy;
  }

  public void useStrategy(String text){
    strategy.check(text);
  }
}

class Test {
  public static void main(String[] args) {

    String text = "Hello Java";

    Context context = new Context();

    context.setStrategy(new SahanStrategy());
    context.useStrategy(text);

    context.setStrategy(new KasunStrategy());
    context.useStrategy(text);
  }
}
```

Strategy Interface

java

```java
interface Strategy {
    public abstract boolean check(String text);
}
```

- **Purpose:** Defines a common interface for all strategy implementations.
- **Method:** `check(String text)` which is an abstract method to be implemented by different strategies.

SahanStrategy Class

java

```java
class SahanStrategy implements Strategy {
    @Override
    public boolean check(String text) {
        return text.contains("J");
    }
}
```

- **Purpose:** A concrete implementation of the `Strategy` interface.
- **Method:** `check(String text)` checks if the text contains the letter "J".

KasunStrategy Class

java

Copy code

```java
class KasunStrategy implements Strategy {
    @Override
    public boolean check(String text) {
        boolean b = false;
        for (int i = 0; i < text.length(); i++) {
            if (text.charAt(i) == 'J') {
                b = true;
                break;
            }
        }
        return b;
    }
}
```

- **Purpose:** Another concrete implementation of the `Strategy` interface.
- **Method:** `check(String text)` iterates through the text to see if it contains the letter "J" and returns `true` if found, otherwise `false`.

Context Class

java

```java
class Context {
    private Strategy strategy;

    public void setStrategy(Strategy strategy) {
        this.strategy = strategy;
    }

    public void useStrategy(String text) {
        strategy.check(text);
    }
}
```

- **Purpose:** Maintains a reference to a `Strategy` object and allows changing the strategy at runtime.
- **Methods:**
    - `setStrategy(Strategy strategy)`: Sets the current strategy.
    - `useStrategy(String text)`: Uses the current strategy to perform the `check` operation on the given text.

Test Class

java

Copy code

```java
class Test {
    public static void main(String[] args) {
        String text = "Hello Java";

        Context context = new Context();

        context.setStrategy(new SahanStrategy());
        context.useStrategy(text);

        context.setStrategy(new KasunStrategy());
        context.useStrategy(text);
    }
}
```

- **Purpose:** Demonstrates the usage of the `Context` class with different strategies.
- **Steps:**
  - A `Context` object is created.
  - The strategy is set to `SahanStrategy`, and the `check` method is called.
  - The strategy is then changed to `KasunStrategy`, and the `check` method is called again.

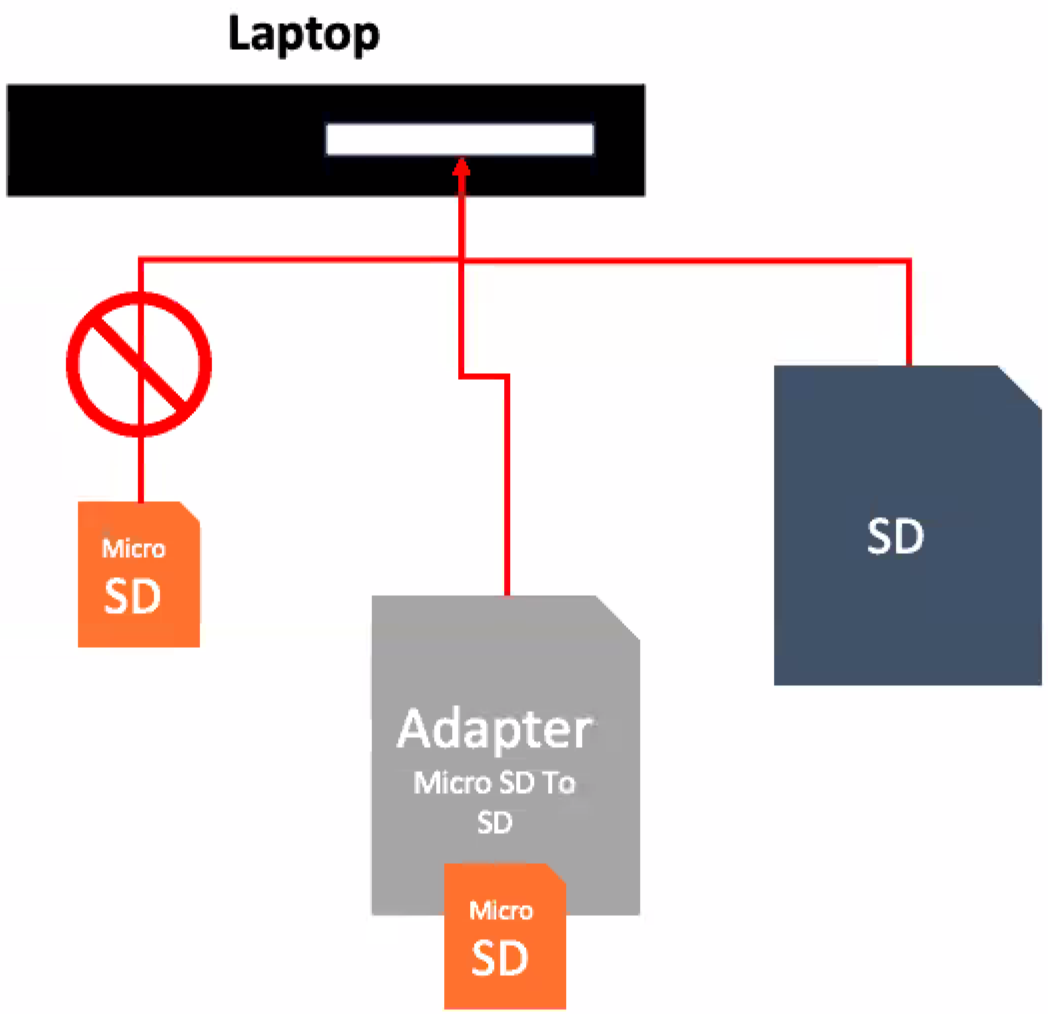
Key Points

1. **Strategy Pattern:** This pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. It lets the algorithm vary independently from clients that use it.
2. **Interface and Implementation:** `` `Strategy` `` is the interface, and `` `SahanStrategy` `` and `` `KasunStrategy` `` are concrete implementations of this interface.
3. **Context Class:** Uses a `` `Strategy` `` object to call the `` `check` `` method, allowing the behavior to change dynamically.
4. **Test Class:** Demonstrates how to use different strategies with the same context.

In this example, the `` `Context` `` class can use either `` `SahanStrategy` `` or `` `KasunStrategy` `` to check if a given text contains the letter "J". The strategy can be switched at runtime by calling `` `setStrategy` `` with a different strategy object.

- **Adapter Design Pattern**

Interface Conversion: *Adapts one interface to another, making incompatible interfaces compatible.*
Wrapper Class: *The adapter class wraps the original class and provides a new interface that the client expects.*

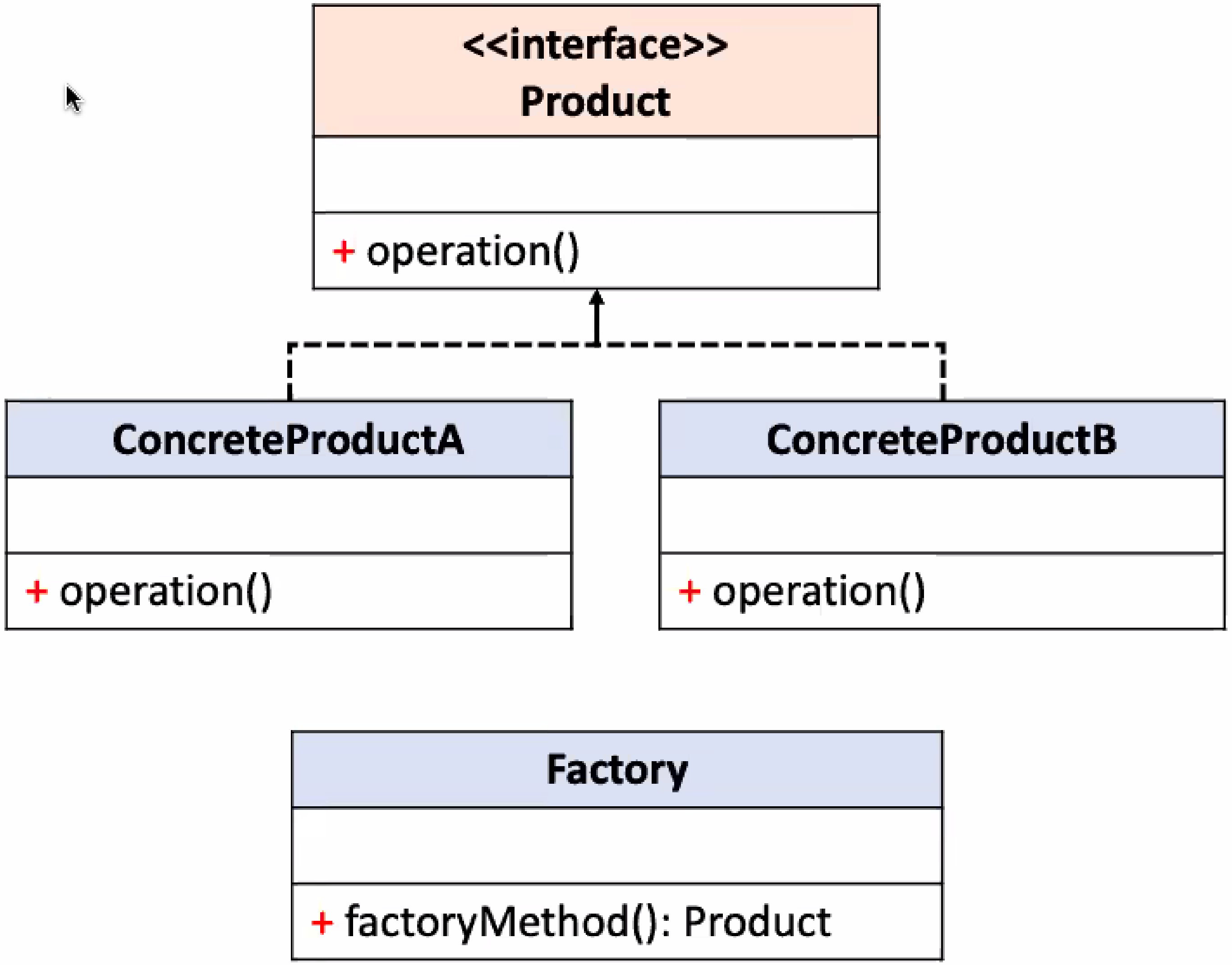


```java
class Laptop {
    private SD sd;

    public void setSd(SD sd) {
        this.sd = sd;
    }

    public void viewFiles() {
        this.sd.readSDCard();
    }
}

interface SD {
    public abstract void readSDCard();
}

class SonySD implements SD {
    @Override
    public void readSDCard() {
        System.out.println("Reading Sony SD Card");
    }
}

interface MicroSD {
    public abstract void readMicroSDCard();
}

class SamsungMicroSD implements MicroSD {
    @Override
    public void readMicroSDCard() {
        System.out.println("Reading Samsung Micro SD Card");
    }
}
```

```java
class Adapter implements SD {

    private MicroSD microSD;

    public Adapter(MicroSD microSD) {
        this.microSD = microSD;
    }

    @Override
    public void readSDCard() {
        this.microSD.readMicroSDCard();
    }
}

class Test {

    public static void main(String[] args) {

        Laptop laptop = new Laptop();

        SonySD sonySD = new SonySD();
        laptop.setSd(sonySD);
        laptop.viewFiles();

        SamsungMicroSD samsungMicroSD = new SamsungMicroSD();
        //laptop.setSd(samsungMicroSD);
        Adapter adapter = new Adapter(samsungMicroSD);
        laptop.setSd(adapter);
        laptop.viewFiles();
    }
}
```

- **Factory Design Pattern**

Object Creation: Encapsulates object creation to avoid exposing the instantiation logic to the client.
Decoupling: Decouples the client code from the specific classes it needs to instantiate.

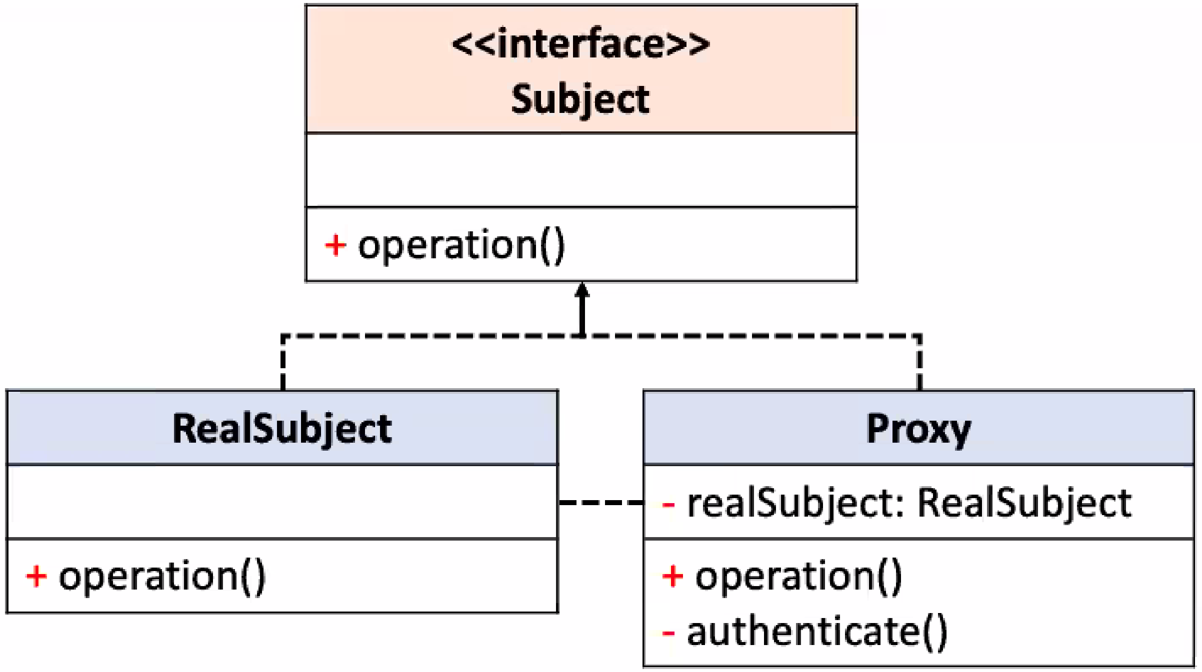


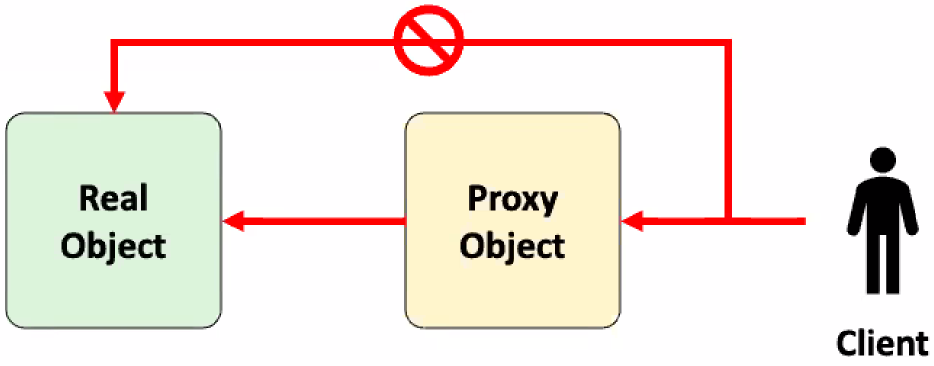
```java
interface Gun {
    public abstract void fire();
}

class AK47 implements Gun {
    @Override
    public void fire() {
        System.out.println("Trrr");
    }
}

class Sniper implements Gun {
    @Override
    public void fire() {
        System.out.println("Booom");
    }
}
```

```java
class GunFactory {

    public Gun makeGun(String name) {
        if (name.equals("AK47")) {
            return new AK47();
        } else if (name.equals("Sniper")) {
            return new Sniper();
        } else {
            return null;
        }
    }
}

class Test {
    public static void main(String[] args) {
        //AK47 ak47 = new AK47();
        //ak47.fire();
        GunFactory gunFactory = new GunFactory();

        Gun g1 = gunFactory.makeGun("AK47");
        g1.fire();

        Gun g2 = gunFactory.makeGun("Sniper");
        g2.fire();
    }
}
```

- **Proxy Design Pattern**

Control Access: The proxy controls and manages access to the real object.



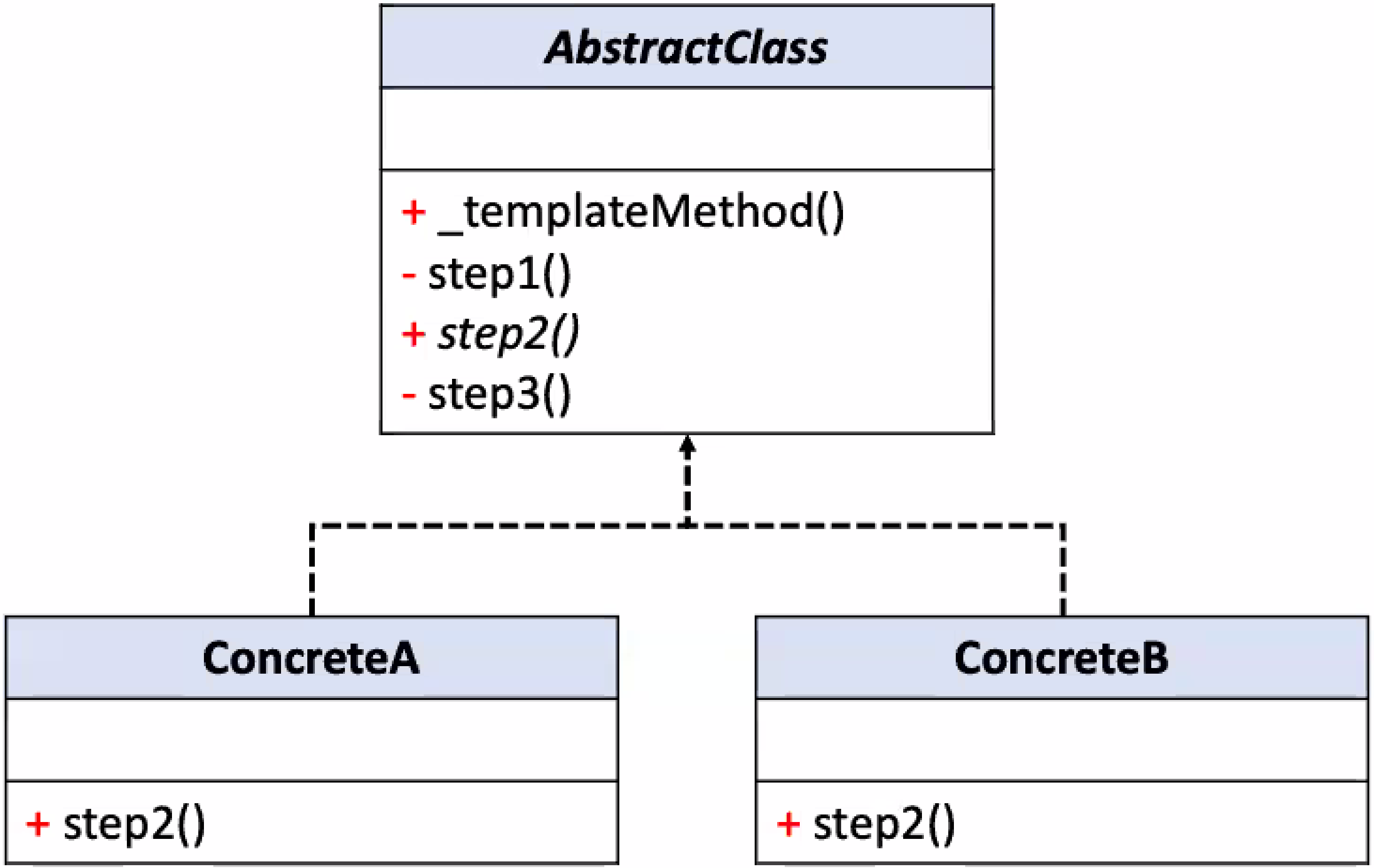


```java
interface Database {
   public abstract void search(String query);
}

class RealDatabase implements Database{
   @Override
   public void search(String query) {
      System.out.println("Database Search "+query);
   }
}

class ProxyDatabase implements Database{

   private RealDatabase realDatabase;

   private String password;

   public ProxyDatabase(String password) {
      this.password = password;
      this.realDatabase = new RealDatabase();
   }

   @Override
   public void search(String query) {
      if(authenticate()){
         this.realDatabase.search(query);
      }else{
         System.out.println("Access Denied!");
      }
   }

   private boolean authenticate(){
      if(this.password.equals("123")){
         return true;
      }else{
         return false;
      }
   }
}
```

```java
class Test {
   public static void main(String[] args) {
      ProxyDatabase proxyDatabase = new ProxyDatabase("123");
      proxyDatabase.search("SELECT * FROM product");
   }
}
```

- **Template Method Design Pattern**

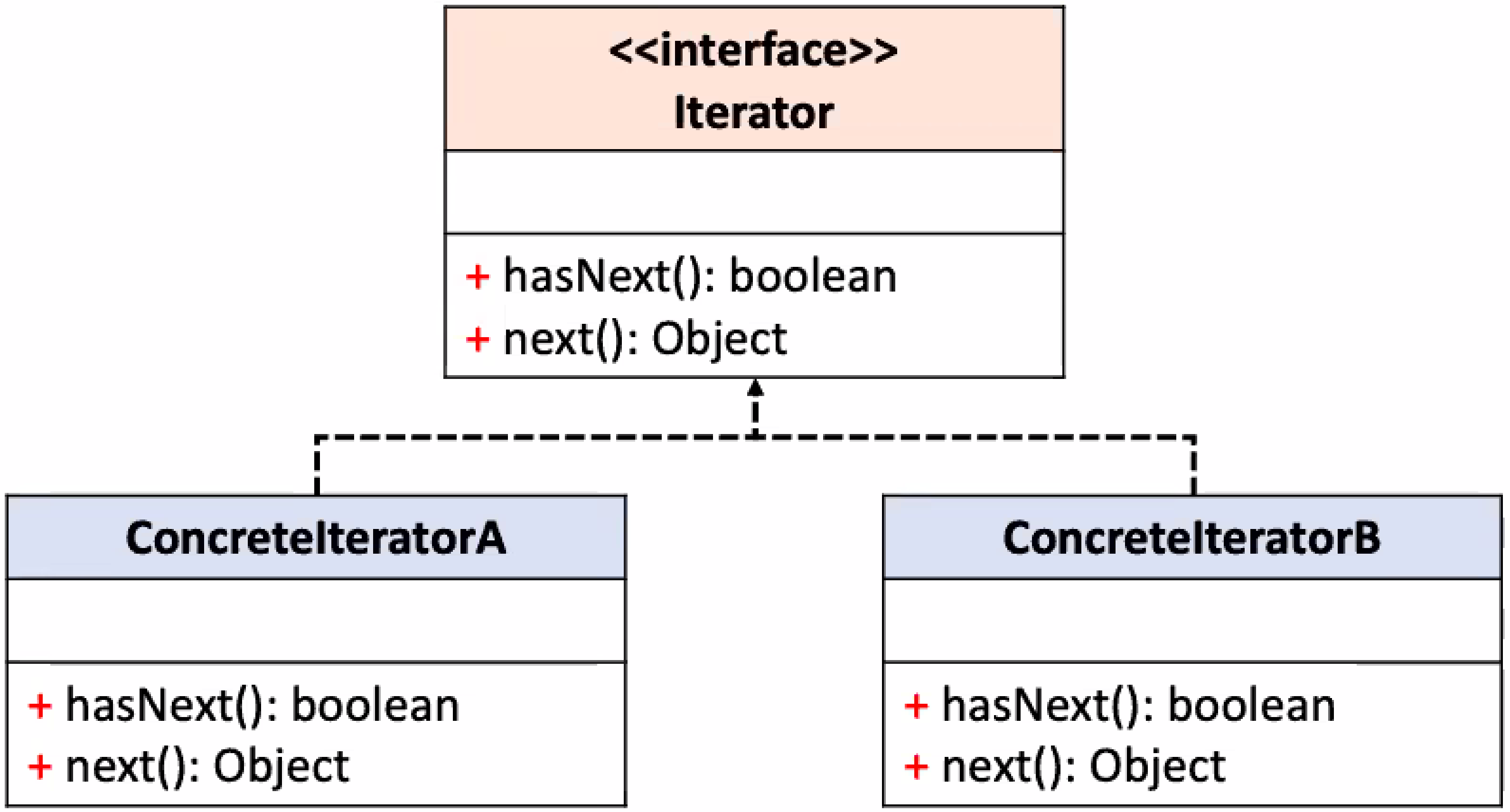
Algorithm Framework: Sets up the main steps of an algorithm, letting subclasses fill in the details.
Consistency: Keeps the core structure of the algorithm the same, preventing changes to key parts.



```java
abstract class FruitJuice{

    public final void prepare(){
        select();
        addIngredients();
        blend();
        serve();
    }

    public abstract void select();

    public abstract void addIngredients();

    private void blend(){
        System.out.println("Blend");
    }

    private void serve(){
        System.out.println("Serve");
    }
}

class AppleJuice extends FruitJuice{

    @Override
    public void select() {
        System.out.println("Select Apple");
    }

    @Override
    public void addIngredients() {
        System.out.println("Add Water & Sugar");
    }
}

class OrangeJuice extends FruitJuice{

    @Override
    public void select() {
        System.out.println("Select Orange");
    }

    @Override
    public void addIngredients() {
        System.out.println("Add Water & Mint");
    }
}

class Test {
    public static void main(String[] args) {

        FruitJuice fruitJuice1 = new AppleJuice();
        fruitJuice1.prepare();

        FruitJuice fruitJuice2 = new OrangeJuice();
        fruitJuice2.prepare();

    }
}
```

- **Iterator Design Pattern**

Sequential Access: *Allows you to access elements one by one in a collection.*
Separate Traversal Logic: *Keeps the logic of how to traverse a collection separate from the collection itself.*



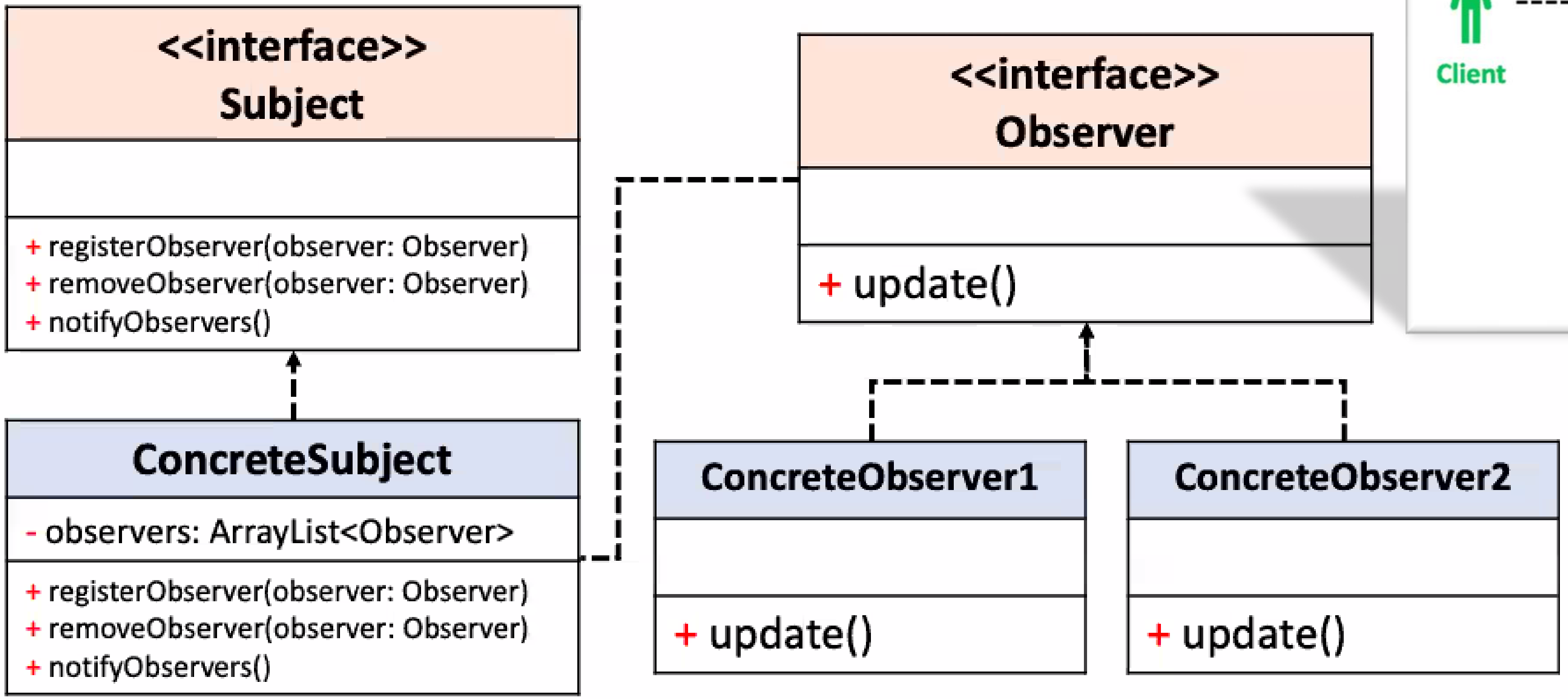
```java
import java.util.ArrayList;

interface Iterator{
    public abstract boolean hasNext();
    public abstract Object next();
}

class ArrayIterator implements Iterator{
    private Object array[];
    private int index;

    public ArrayIterator(Object[] array) {
        this.array = array;
    }

    @Override
    public boolean hasNext() {
        return index < array.length;
    }

    @Override
    public Object next() {
        return array[index++];
    }
}
```

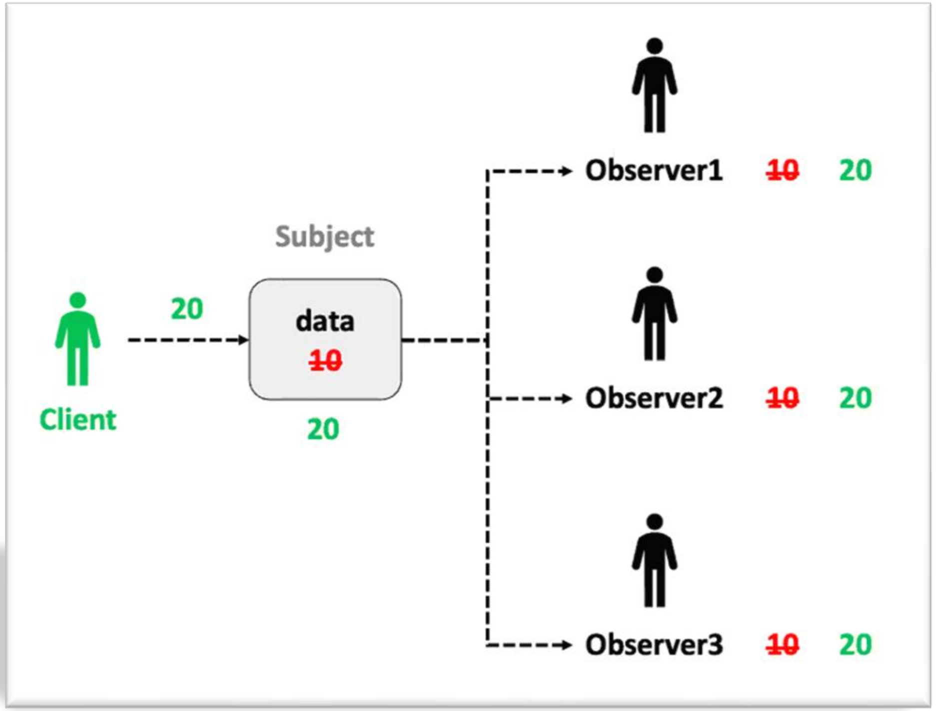
```java
class ArrayListIterator implements Iterator{
    private ArrayList list;
    private int index;

    public ArrayListIterator(ArrayList list) {
        this.list = list;
    }

    @Override
    public boolean hasNext() {
        return index < list.size();
    }

    @Override
    public Object next() {
        return list.get(index++);
    }
}
```

```java
class Test {
    public static void main(String[] args) {

        //Array
        String array1[] = {"Java","PHP","C#"};

        //ArrayList
        ArrayList<String> list1 = new ArrayList<>();
        list1.add("Java");
        list1.add("PHP");
        list1.add("C#");

        Iterator iterator1 = new ArrayIterator(array1);
        Iterator iterator2 = new ArrayListIterator(list1);

        while(iterator1.hasNext()){
            System.out.println(iterator1.next());
        }

        while(iterator2.hasNext()){
            System.out.println(iterator2.next());
        }
    }
}
```

- **Observer Design Pattern**

Defines a One-to-Many Dependency: When one object changes state, all its dependents (observers) are notified and updated automatically.





```java
class Subject{
   private int x;
   private ArrayList<Observer> observerList;

   public Subject() {
      this.observerList = new ArrayList<>();
   }

   public void registerObserver(Observer observer){
      this.observerList.add(observer);
   }

   public void removeObserver(Observer observer){
      this.observerList.remove(observer);
   }

   public void notifyObservers(){
      for (Observer observer : observerList) {
         observer.update(this.x);
      }
   }

   public void setX(int x) {
      this.x = x;
      notifyObservers();
   }

   public int getX() {
      return x;
   }
}
```

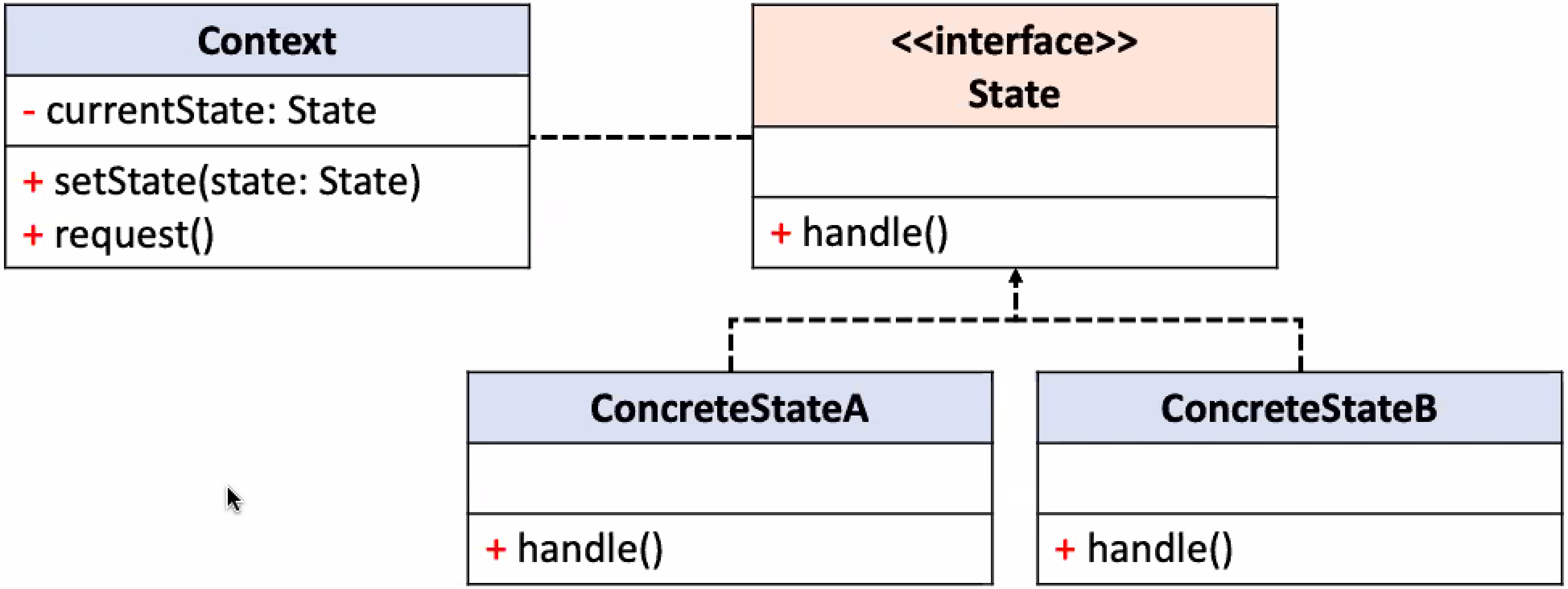
```java
interface Observer{
   public abstract void update(int x);
}

class ConcreteObserver1 implements Observer{
   @Override
   public void update(int x) {
      System.out.println("Observer 1: "+x);
   }
}

class ConcreteObserver2 implements Observer{
   @Override
   public void update(int x) {
      System.out.println("Observer 2: "+x);
   }
}

class Test {
   public static void main(String[] args) {
      Subject subject = new Subject();

      Observer observer1 = new ConcreteObserver1();
      Observer observer2 = new ConcreteObserver2();

      subject.registerObserver(observer1);
      subject.registerObserver(observer2);

      subject.setX(50);
   }
}
```

```java
interface Subject {
    public void registerObserver(Observer observer);
    public void removeObserver(Observer observer);
    public void notifyObservers();
}

class ConcreteSubject implements Subject {
    private int x;
    private ArrayList<Observer> observerList;

    public ConcreteSubject() {
        this.observerList = new ArrayList<>();
    }

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

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

    @Override
    public void notifyObservers() {
        for (Observer observer : observerList) {
            observer.update(this.x);
        }
    }

    public void setX(int x) {
        this.x = x;
        notifyObservers();
    }

    public int getX() {
        return x;
    }
}
```

```java
interface Observer {
    public abstract void update(int x);
}

class ConcreteObserver1 implements Observer {
    @Override
    public void update(int x) {
        System.out.println("Observer 1: " + x);
    }
}

class ConcreteObserver2 implements Observer {
    @Override
    public void update(int x) {
        System.out.println("Observer 2: " + x);
    }
}

class Test {
    public static void main(String[] args) {
        ConcreteSubject subject = new ConcreteSubject();

        Observer observer1 = new ConcreteObserver1();
        Observer observer2 = new ConcreteObserver2();

        subject.registerObserver(observer1);
        subject.registerObserver(observer2);

        subject.setX(50);
    }
}
```

- **State Design Pattern**

Encapsulates State-Specific Behavior: *Allows an object to alter its behavior when its internal state changes.*

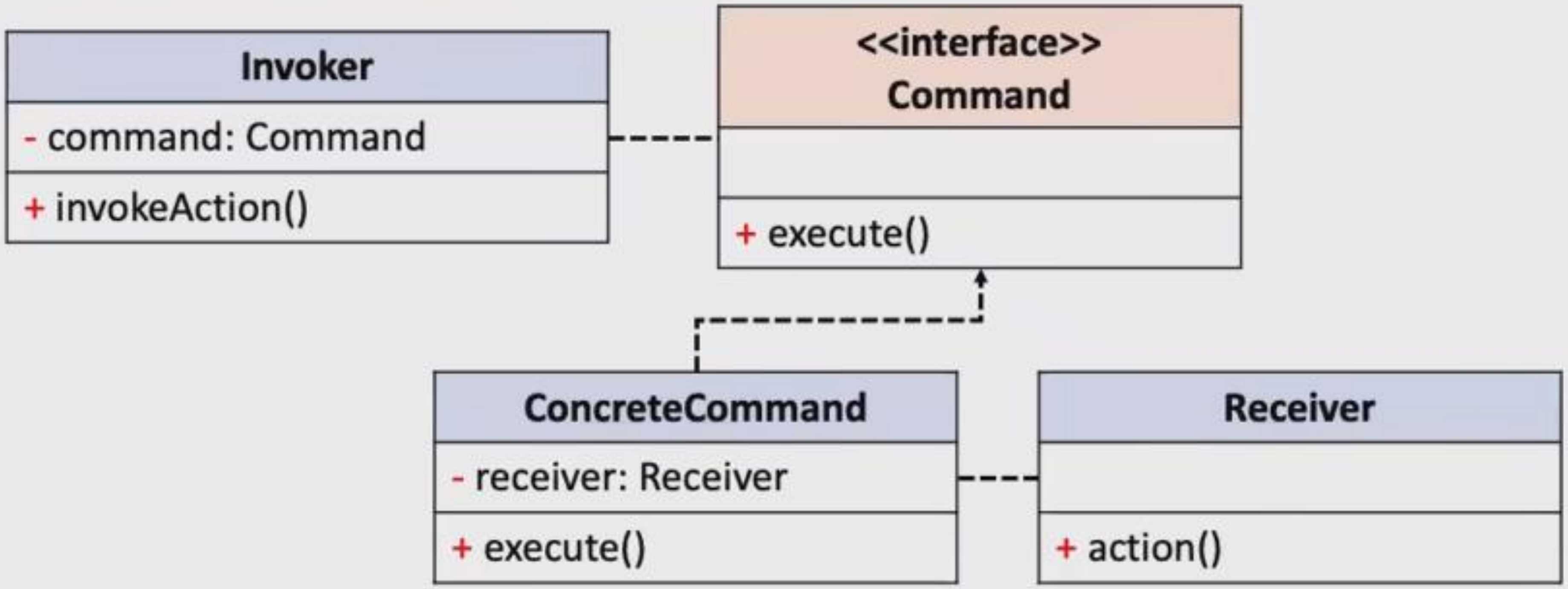


```java
interface Gun{
   public abstract void fire();
}

class AK47 implements Gun{
   @Override
   public void fire() {
      System.out.println("Trrr");
   }
}

class Sniper implements Gun{
   @Override
   public void fire() {
      System.out.println("Boom");
   }
}

class Man{
   private Gun gun;

   public void setGun(Gun gun) {
      this.gun = gun;
   }

   public void shoot(){
      this.gun.fire();
   }
}
```

```java
class Test {
   public static void main(String[] args) {
      Man man = new Man();

      Gun ak47 = new AK47();
      Gun sniper = new Sniper();

      man.setGun(ak47);
      man.shoot();

      man.setGun(sniper);
      man.shoot();
   }
}
```

- **Command Design Pattern**

Decouples Sender and Receiver: Separates the object that invokes the operation from the one that knows how to perform it.



```java
interface Command {
  public abstract void execute();
}

class BallaBuranawa implements Command {
  private Balla balla;

  public BallaBuranawa(Balla balla) {
    this.balla = balla;
  }

  @Override
  public void execute() {
    this.balla.buranawa();
  }
}

class BallaDuwanawa implements Command {
  private Balla balla;

  public BallaDuwanawa(Balla balla) {
    this.balla = balla;
  }

  @Override
  public void execute() {
    this.balla.duwanawa();
  }
}
```

```java
class Balla {
  public void buranawa() {
    System.out.println("Balla Buranawa");
  }

  public void duwanawa() {
    System.out.println("Balla Duwanawa");
  }
}

class Invoker {

  private Command command;

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

  public void invokeAction() {
    command.execute();
  }
}

class Test {
  public static void main(String[] args) {
    Balla balla = new Balla();
    //balla.buranawa();
    //balla.duwanawa();

    Command command1 = new BallaBuranawa(balla);
    Command command2 = new BallaDuwanawa(balla);
    //command1.execute();
    //command2.execute();

    Invoker invoker1 = new Invoker();

    invoker1.setCommand(command1);
    invoker1.invokeAction();

    invoker1.setCommand(command2);
    invoker1.invokeAction();
  }
}
```