# Java Training
# Design Patterns

# Agenda

- Homework review
- Creational Design Patterns
- Behavioral Design Patterns
- Structural Design Patterns

# Design Patterns (GoF)

**Creational Design Patterns**

- Singleton
- Factory & Factory method
- Abstract Factory
- Builder
- Prototype

**Structural Design Patterns**

- Decorator
- Adapter
- Façade
- Bridge
- Proxy
- Composite

**Behavioral Design Patterns**

- Iterator
- Strategy
- State
- Observer
- Command
- Memento
- Chain of Responsibility
- Mediator

# Creational Design Patterns

# Factory

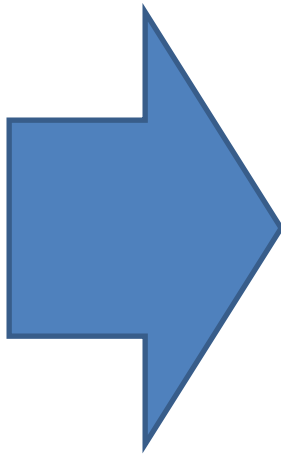
Design principle:

Program to an interface,
not an implementation.

But every time you use **new**,
that's exactly what you are
doing.

# What's wrong with "new"?

```
public Pizza orderPizza() {

    Pizza pizza = new CheesePizza();

    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();

    return pizza;
}
```



```
public Pizza orderPizza(String type) {

    Pizza pizza = null;

    if (type.equals("cheese")) {
        pizza = new CheesePizza();
    } else if (type.equals("pepperoni")) {
        pizza = new PepperoniPizza();
    } else if (type.equals("clam")) {
        pizza = new ClamPizza();
    }

    pizza.prepare();
    pizza.bake();
    pizza.cut();
    pizza.box();

    return pizza;
}
```

1. The code is not closed to modification!

2. The code is changed as new as new classes are added

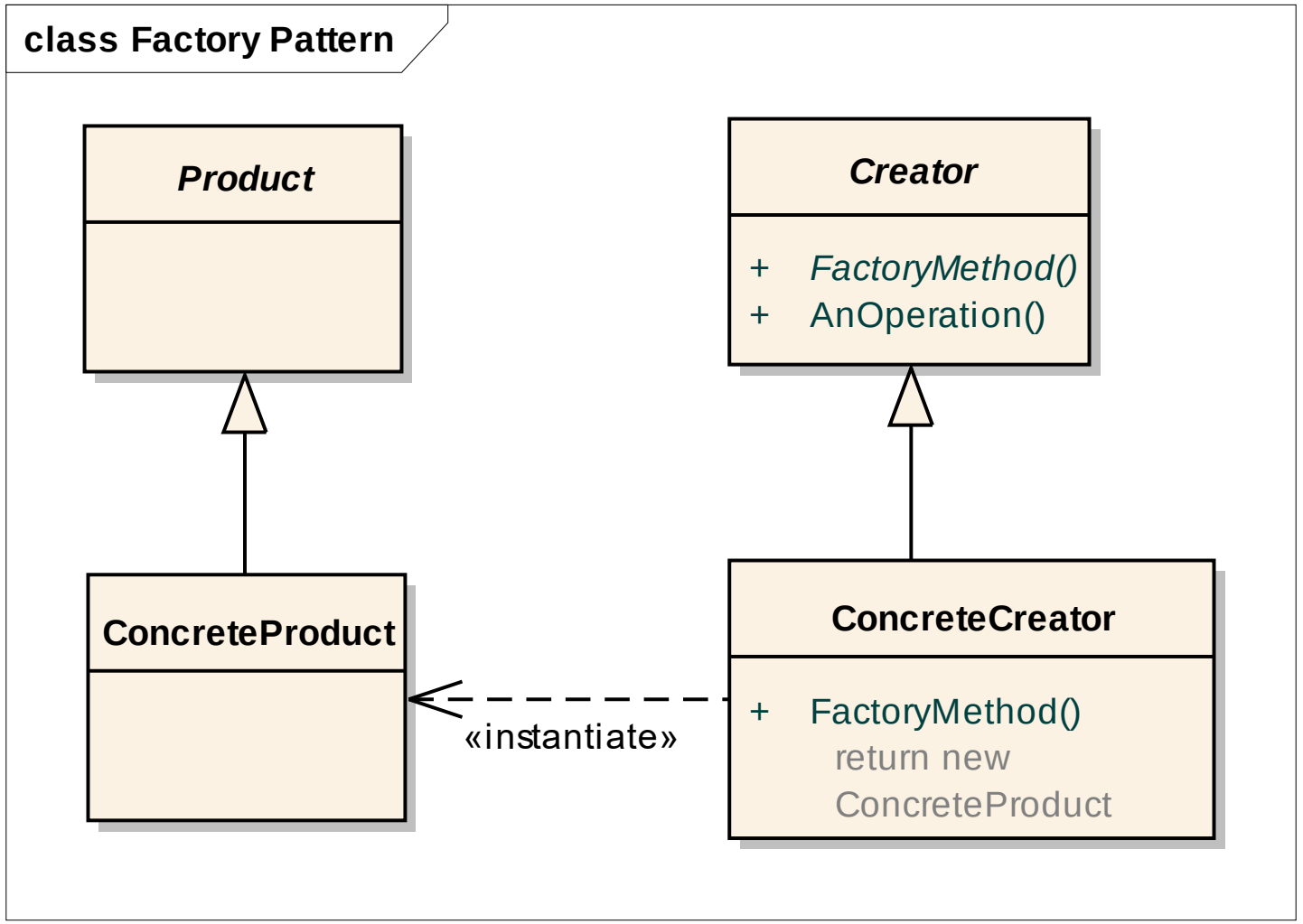
# Factory Method Example

```
public class PizzaStore {
    SimplePizzaFactory factory;

    public PizzaStore(SimplePizzaFactory factory) {
        this.factory = factory;
    }

    public Pizza orderPizza(String type) {
        Pizza pizza;
        pizza = factory.createPizza(type);
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();

        return pizza;
    }

}
```

The code is still parameterized

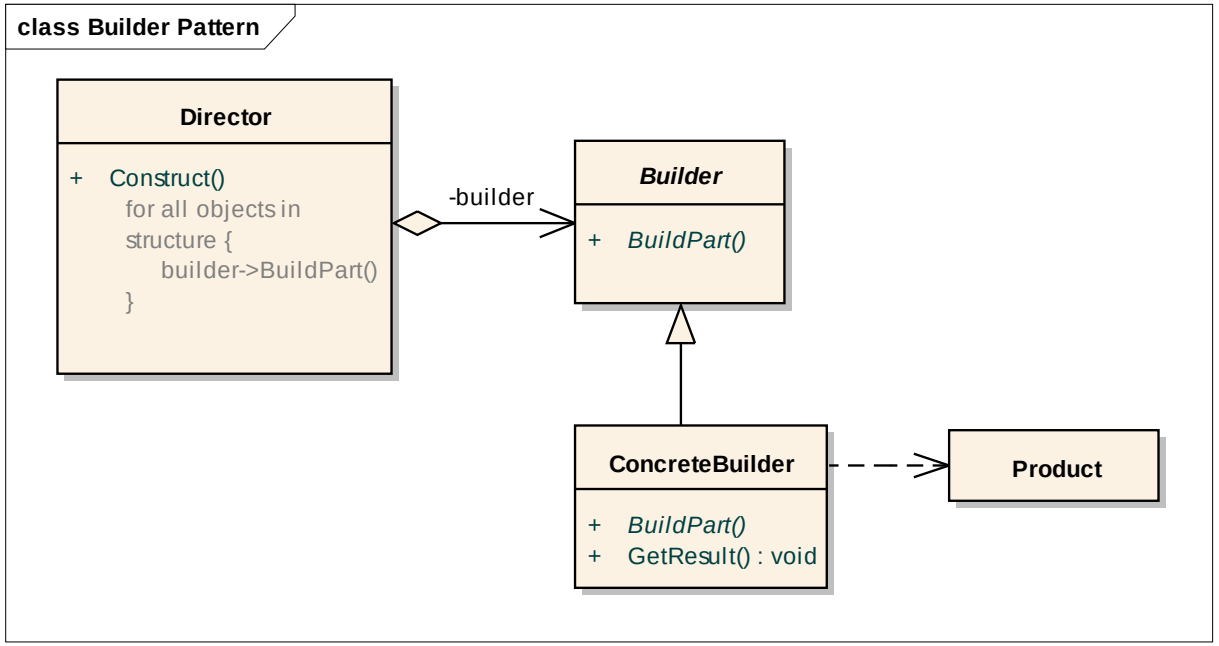
```
public class SimplePizzaFactory {

    public Pizza createPizza(String type) {

        Pizza pizza = null;

        if (type.equals("cheese")) {
            pizza = new CheesePizza();
        } else if (type.equals("pepperoni")) {
            pizza = new PepperoniPizza();
        } else if (type.equals("clam")) {
            pizza = new ClamPizza();
        } else if (type.equals("veggie")) {
            pizza = new VeggiePizza();
        }

        return pizza;
    }
}
```

# Factory Method UML

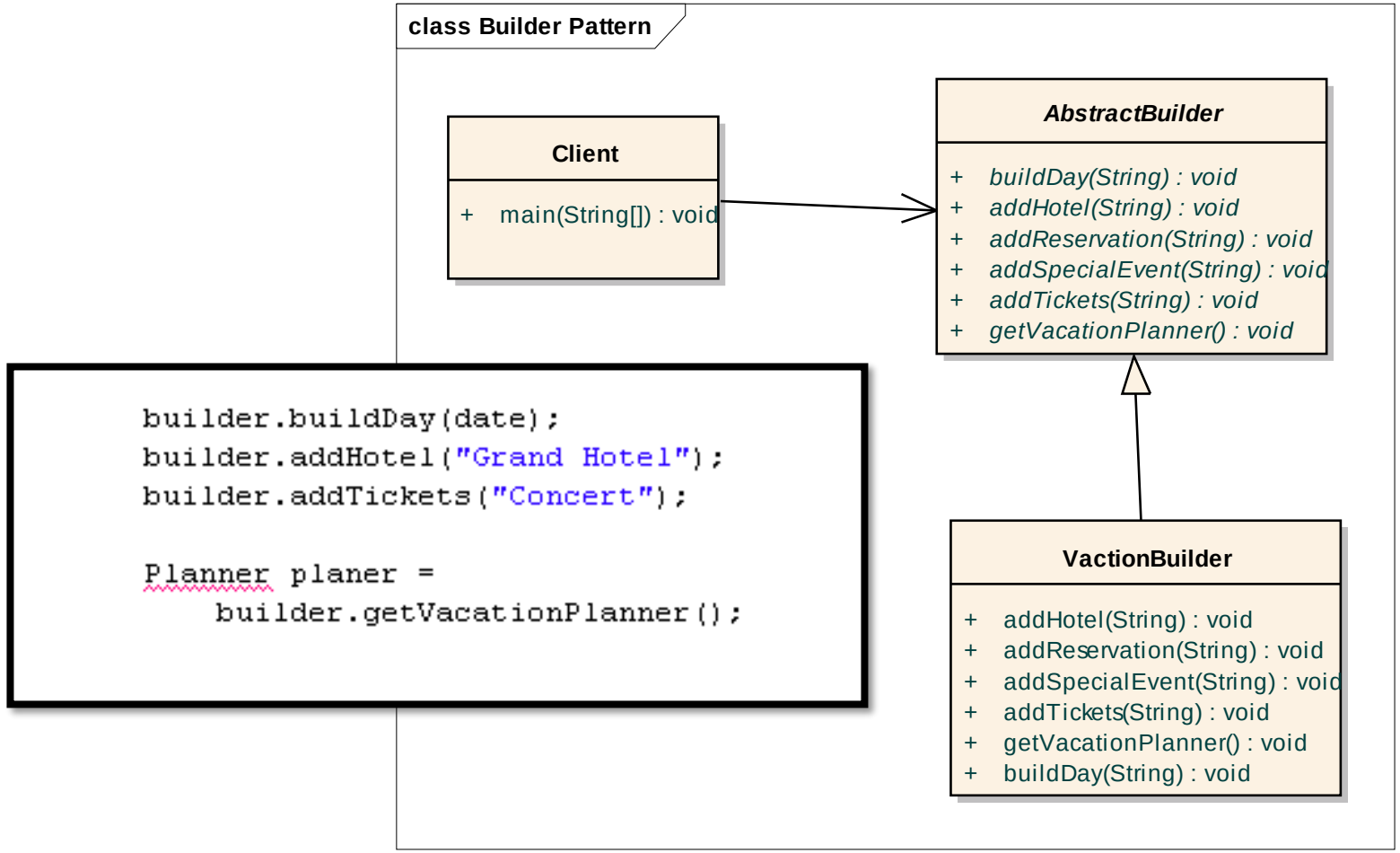


# Builder Pattern UML

**The Builder Pattern** separates the construction of a complex object from its representation so that the same construction process can create different representations



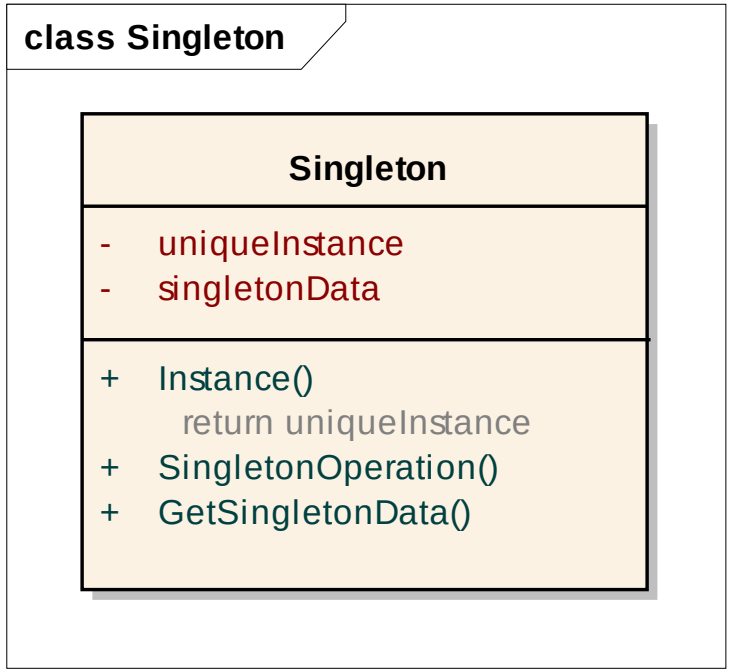
# Example: Vacation Builder




```
builder.buildDay(date);
builder.addHotel("Grand Hotel");
builder.addTickets("Concert");

Planner planer =
    builder.getVacationPlanner();
```

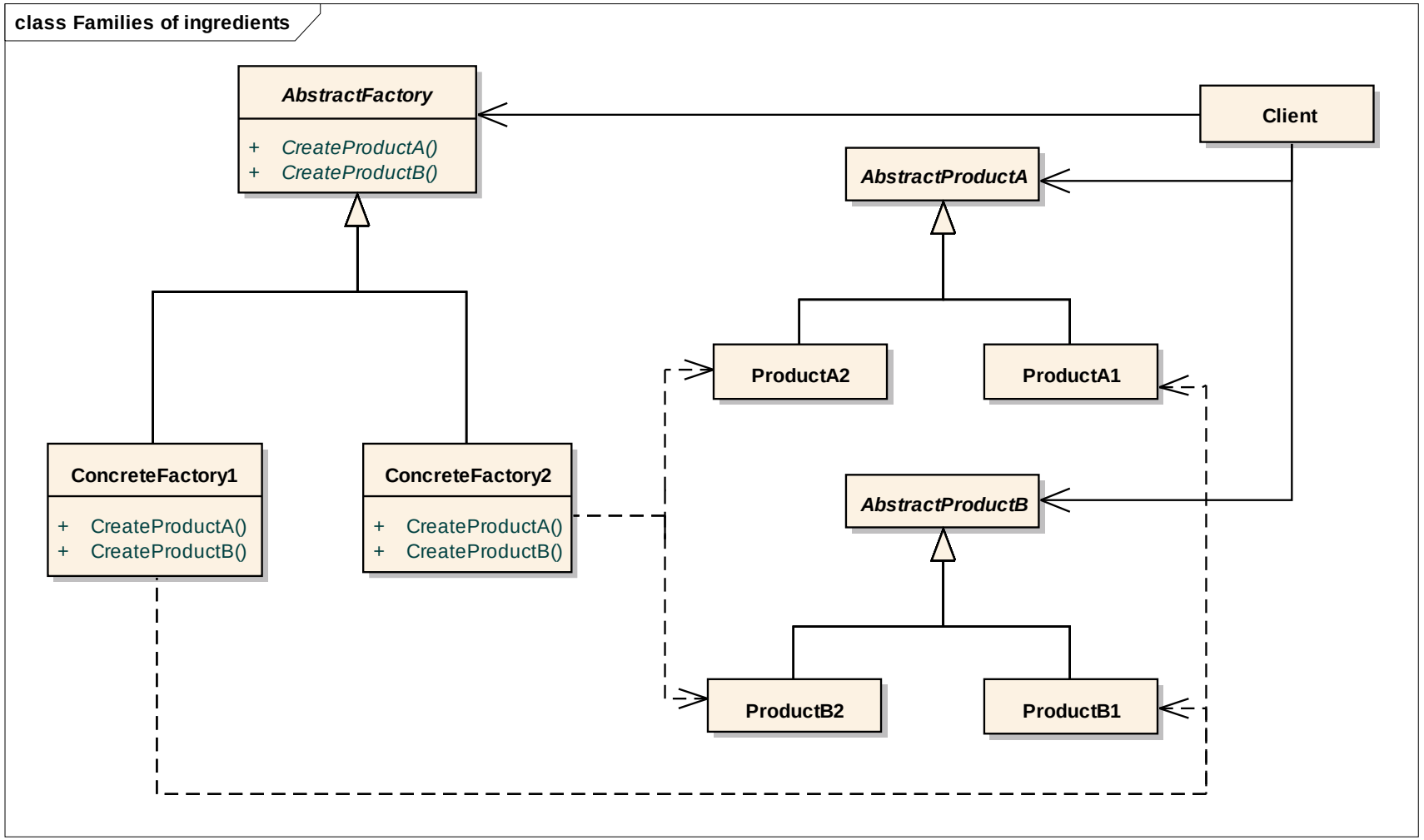
# Singleton Pattern

```
// NOTE: This is not thread safe!

public class Singleton {
    private static Singleton uniqueInstance;

    // other useful instance variables here

    private Singleton() {}

    public static Singleton getInstance() {
        if (uniqueInstance == null) {
            uniqueInstance = new Singleton();
        }
        return uniqueInstance;
    }

    // other useful methods here
}
```



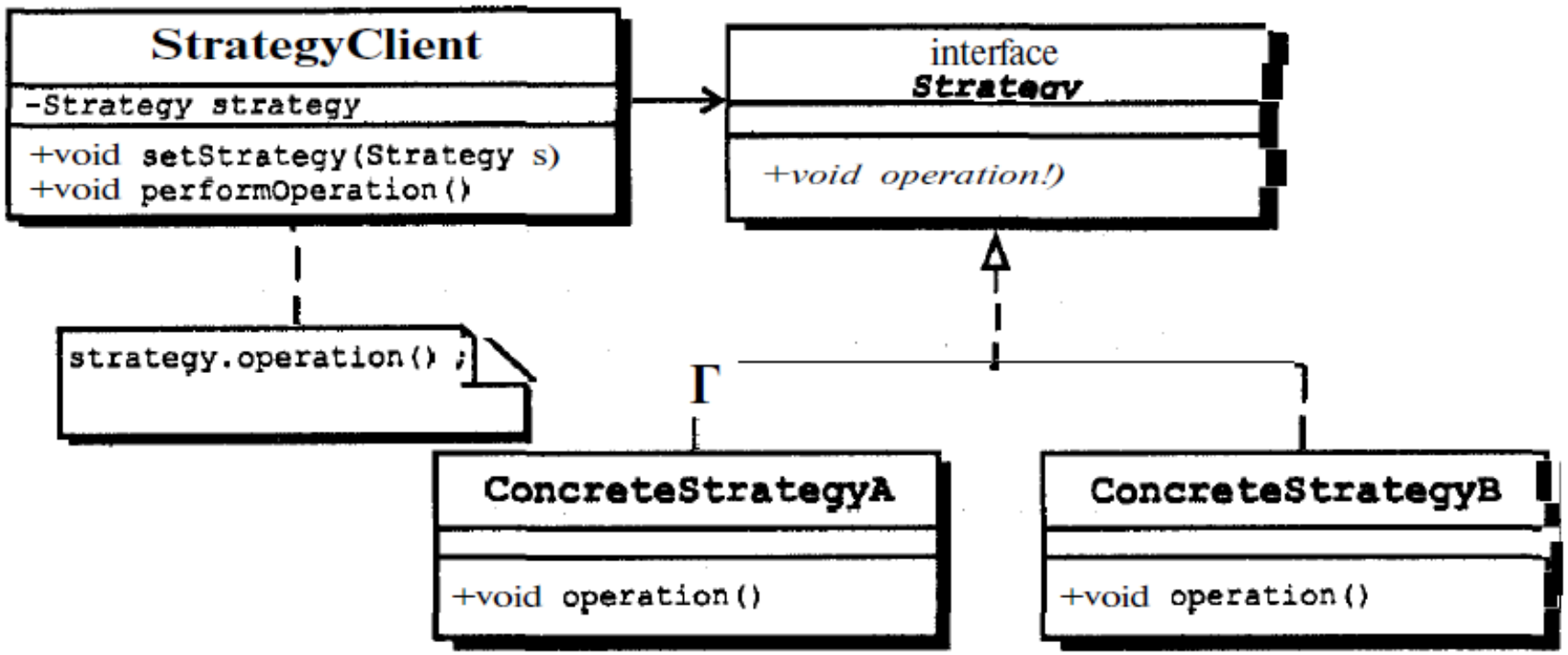

```
public class Singleton {
    private static Singleton uniqueInstance = new Singleton();

    private Singleton() {}

    public static Singleton getInstance() {
        return uniqueInstance;
    }
}
```

# Abstract Factory UML



# Behavioral Design Patterns

# Strategy UML



# Strategy Example

```
interface IStrategy {
    int execute(int a, int b);
}

// Implements the algorithm using the strategy interface
class ConcreteStrategyAdd implements IStrategy {

    public int execute(int a, int b) {
        System.out.println("Called ConcreteStrategyAdd's execute()");
        return a + b;  // Do an addition with a and b
    }
}

class ConcreteStrategySubtract implements IStrategy {

    public int execute(int a, int b) {
        System.out.println("Called ConcreteStrategySubtract's execute()");
        return a - b;  // Do a subtraction with a and b
    }
}

class ConcreteStrategyMultiply implements IStrategy {

    public int execute(int a, int b) {
        System.out.println("Called ConcreteStrategyMultiply's execute()");
        return a * b;   // Do a multiplication with a and b
    }
}
```

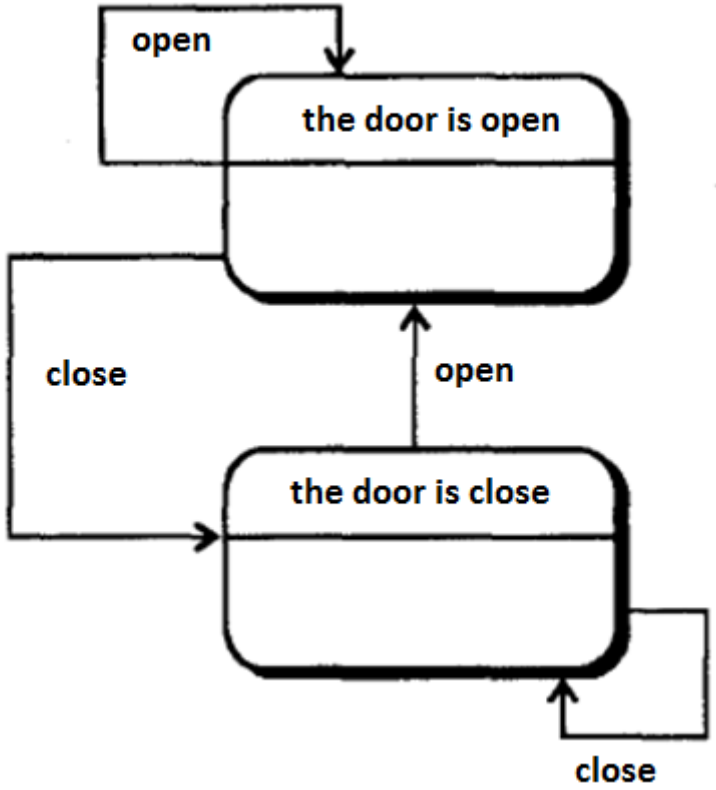
# Strategy

```
class Context {

    private IStrategy strategy;

    // Constructor
    public Context(IStrategy strategy) {
        this.strategy = strategy;
    }

    public int executeStrategy(int a, int b) {
        return strategy.execute(a, b);
    }
}
```

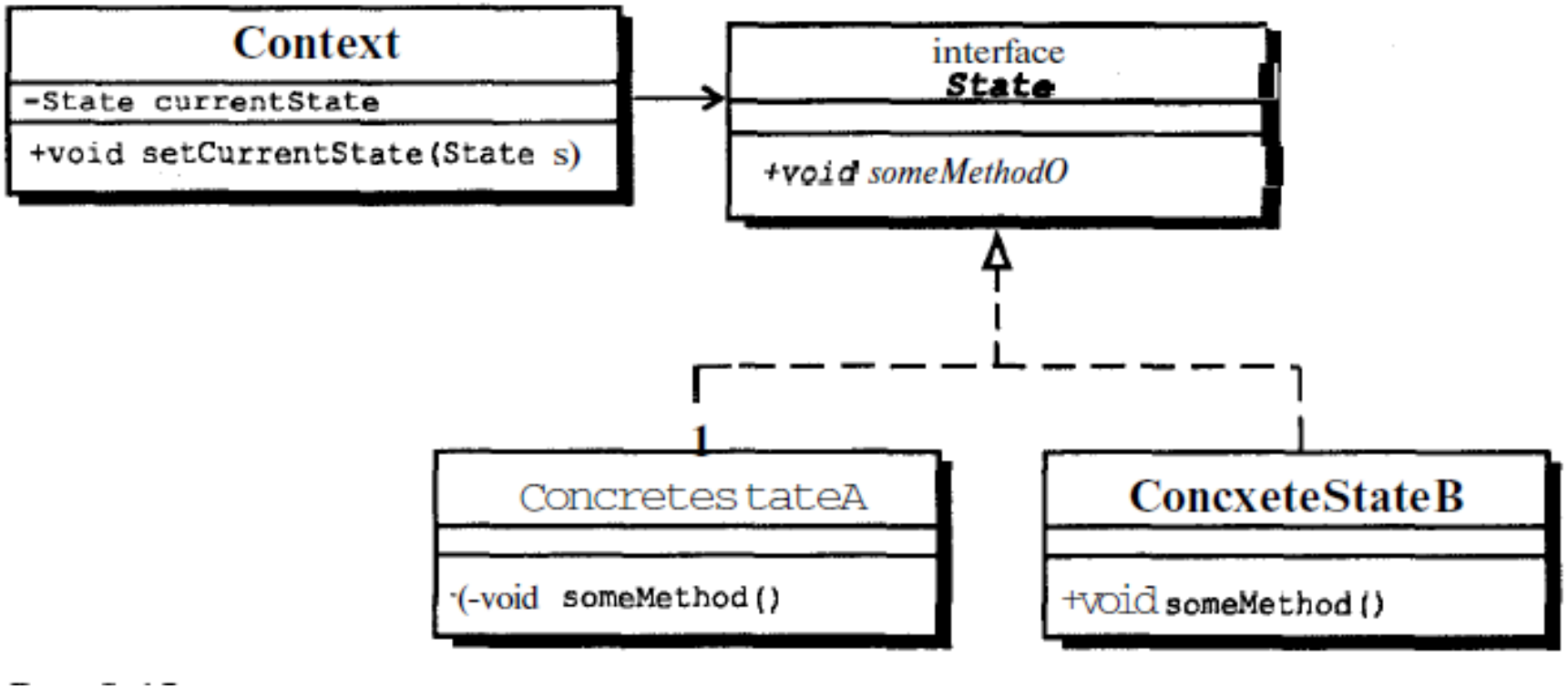
```
class StrategyExample {

    public static void main(String[] args) {

        Context context;

        // Three contexts following different strategies
        context = new Context(new ConcreteStrategyAdd());
        int resultA = context.executeStrategy(3,4);

        context = new Context(new ConcreteStrategySubtract());
        int resultB = context.executeStrategy(3,4);

        context = new Context(new ConcreteStrategyMultiply());
        int resultC = context.executeStrategy(3,4);

        System.out.println("Result A : " + resultA );
        System.out.println("Result B : " + resultB );
        System.out.println("Result C : " + resultC );
    }
}
```

# State

Allows an object to alter its behavior when its internal state changes.



# State UML


Context
-State currentState
+void setCurrentState(State s)
interface
State
+void someMethodO
ConcretestateA
(-void someMethod()
ConcxeteStateB
+void someMethod()

# State

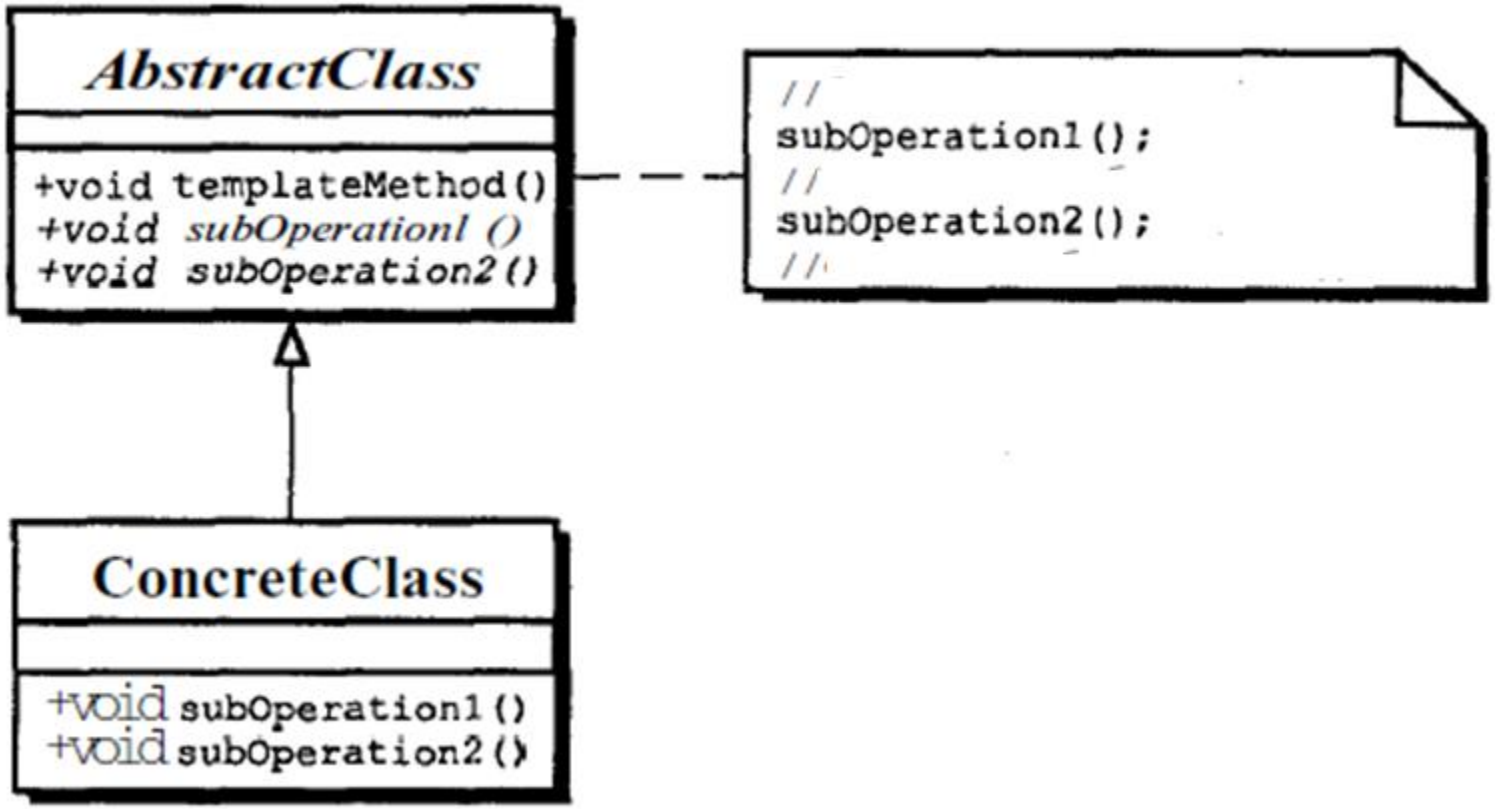
```java
public interface DoorState {

    void action();

    DoorState getNextState();
}
```

```java
public class Context {

    private DoorState currentState = new ClosedState();

    public void action() {
        currentState.action();
        DoorState nextState = currentState.getNextState();
        setCurrentState(nextState);
    }

    public void setCurrentState(DoorState currentState) {
        this.currentState = currentState;
    }
}
```

```java
public class OpenedState implements DoorState {

    private static DoorState nextState = new ClosedState();

    @Override
    public void action() {
        System.out.println("The was closed.");
    }

    @Override
    public DoorState getNextState() {
        return nextState;
    }
}
```

```java
public class ClosedState implements DoorState {

    private static DoorState nextState = new OpenedState();

    @Override
    public void action() {
        System.out.println("The was opened.");
    }

    @Override
    public DoorState getNextState() {
        return nextState;
    }
```

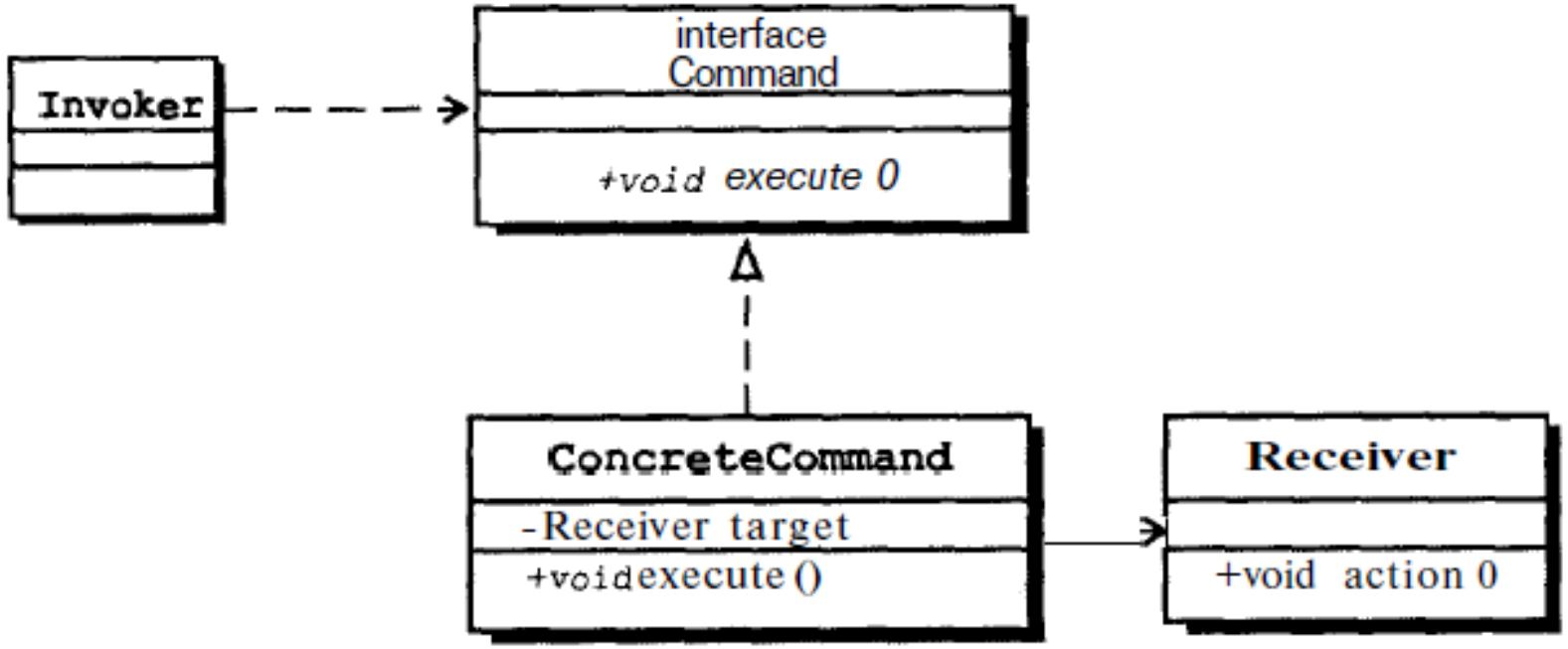
# Template method UML



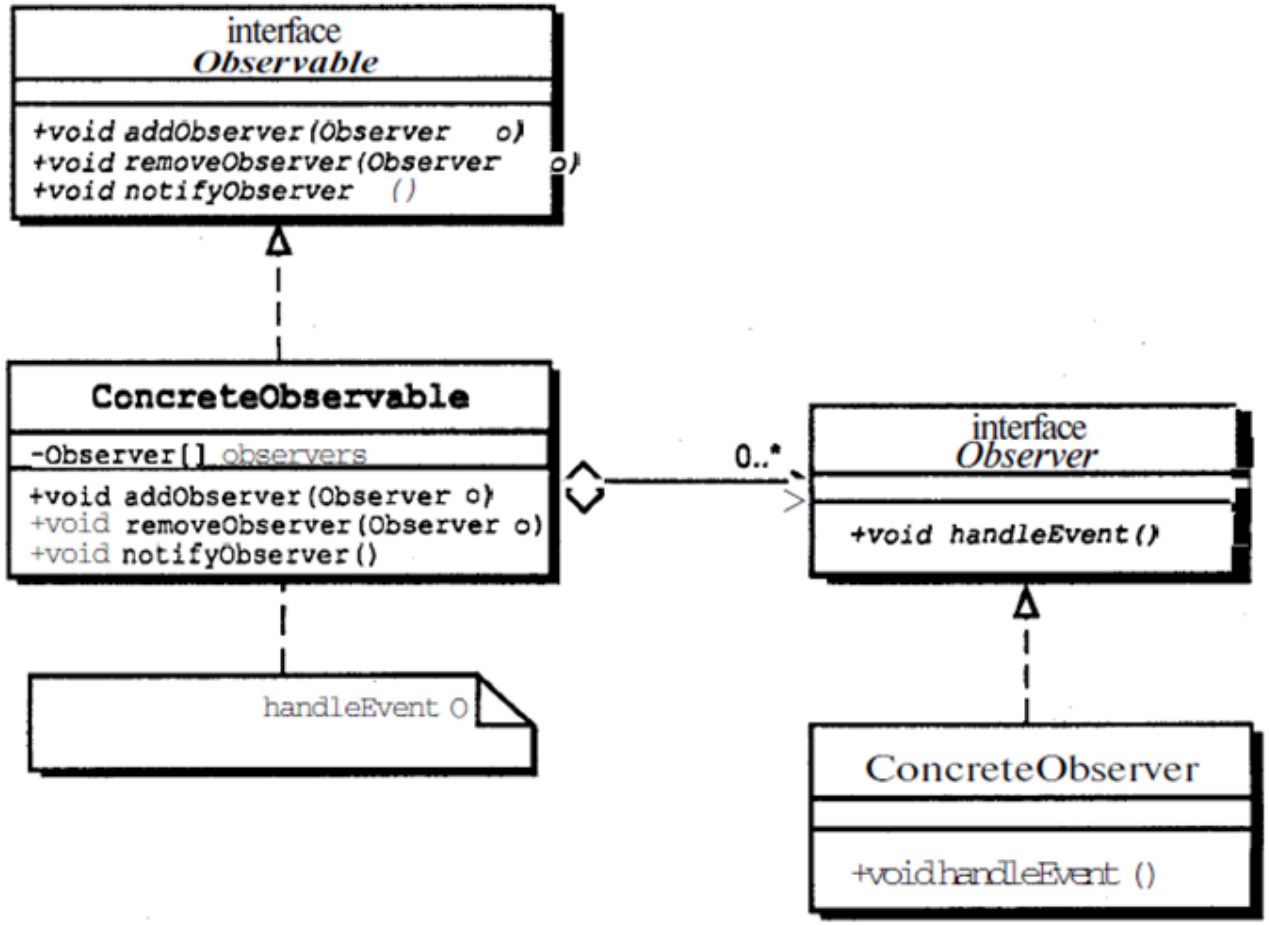
# Example: Template Method

```
abstract class CheckBackground {

    public abstract void checkBank();
    public abstract void checkCredit();
    public abstract void checkLoan();
    public abstract void checkStock();
    public abstract void checkIncome();

    //work as template method
    public void check() {
        checkBank();
        checkCredit();
        checkLoan();
        checkStock();
        checkIncome();
    }
}
```

# Command

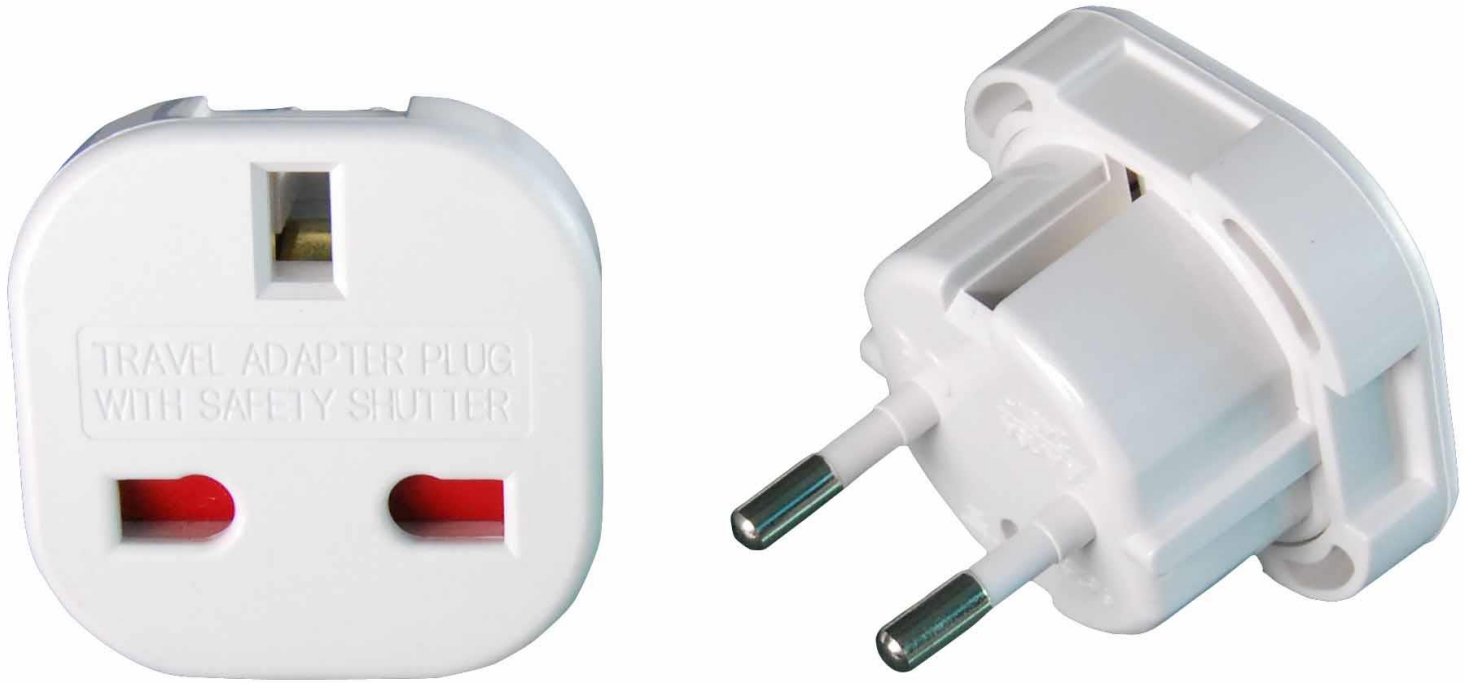

Invoker
interface
Command
+void execute 0
ConcreteCommand
-Receiver target
+voidexecute ()
Receiver
+void action 0

# Observer

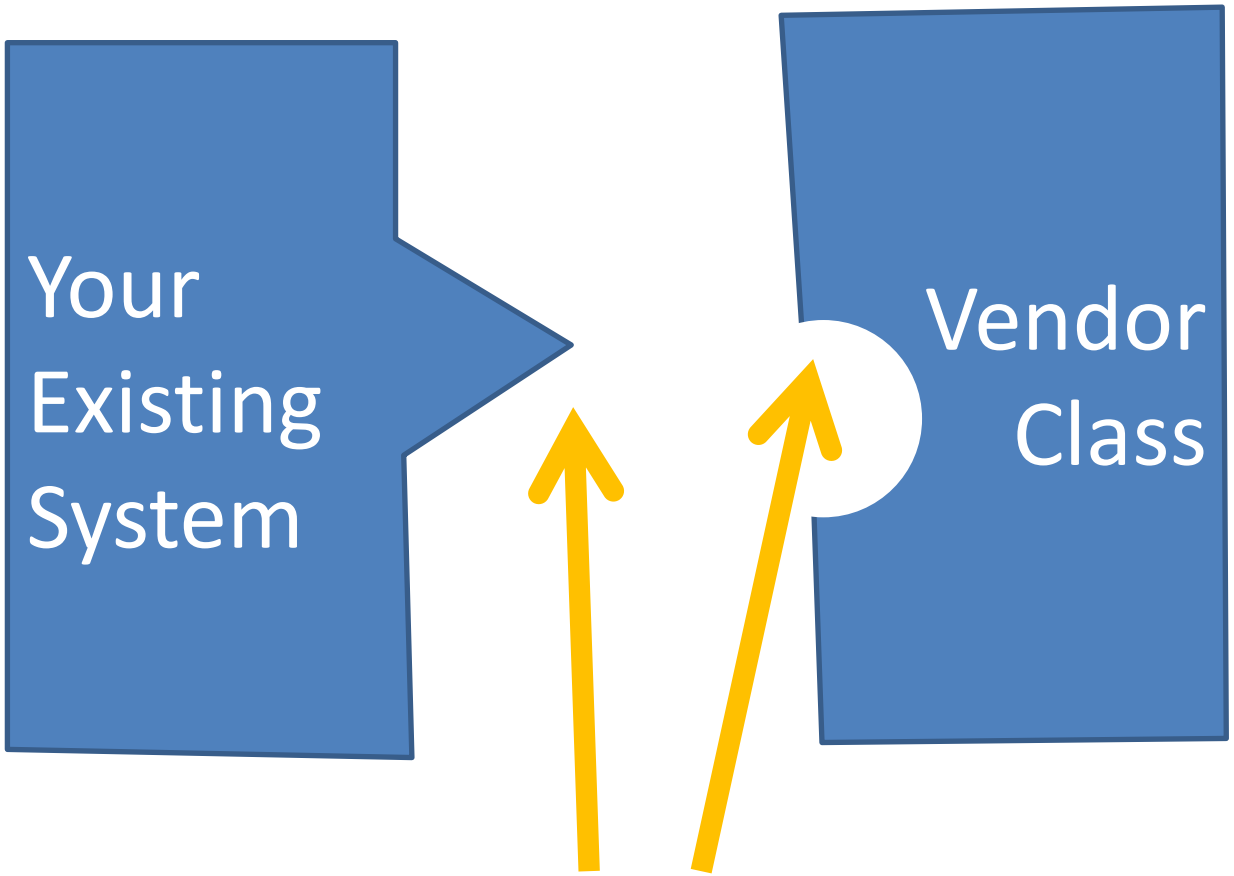

interface
Observable
+void addObserver(Observer o)
+void removeObserver(Observer o)
+void notifyObserver ()
ConcreteObservable
-Observer[] observers
+void addObserver(Observer o)
+void removeObserver(Observer o)
+void notifyObserver()
handleEvent ()
0..*
interface
Observer
+void handleEvent()
ConcreteObserver
+void handleEvent ()

# Structural Design Patterns

# Adapter Pattern

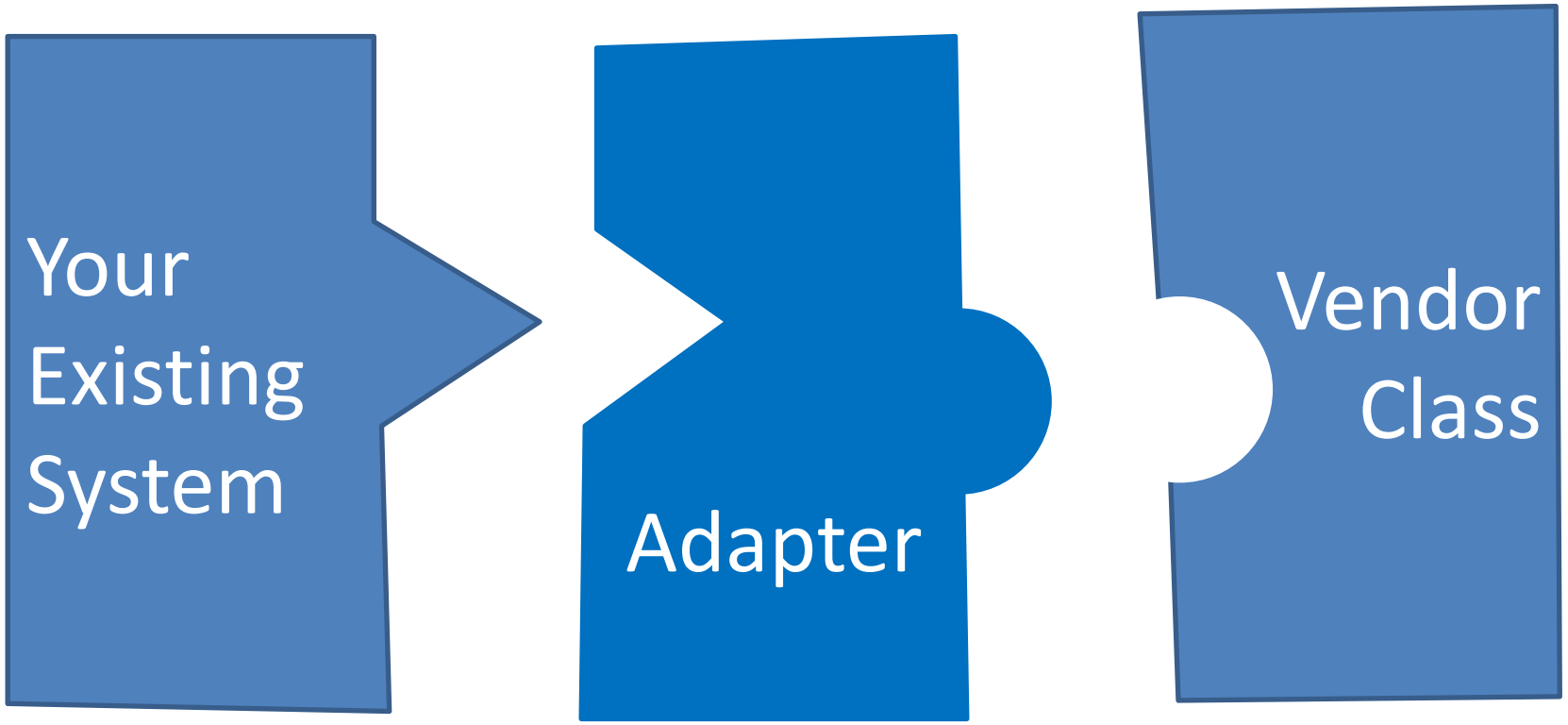


# Problem to solve

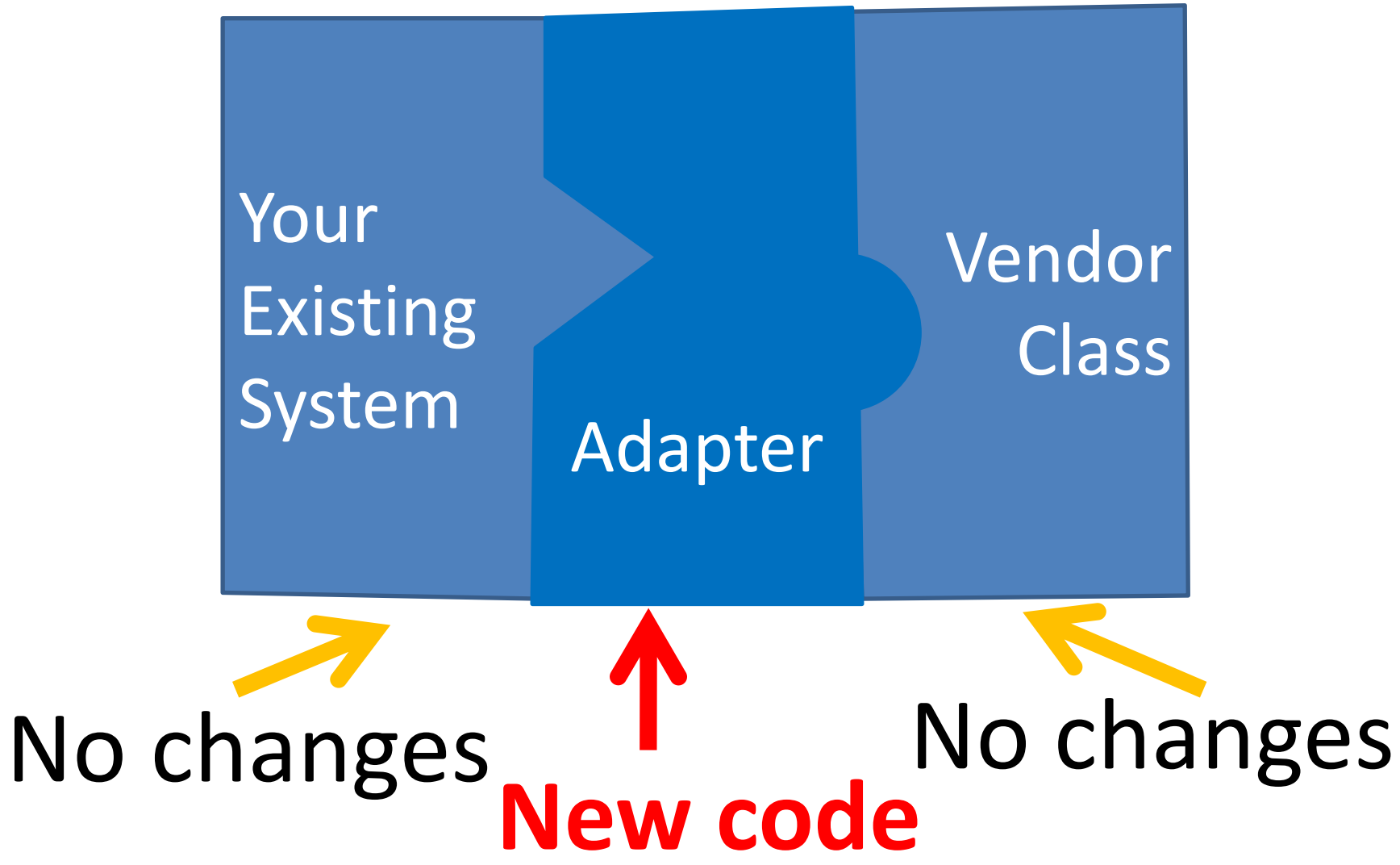



The interfaces doesn’t match

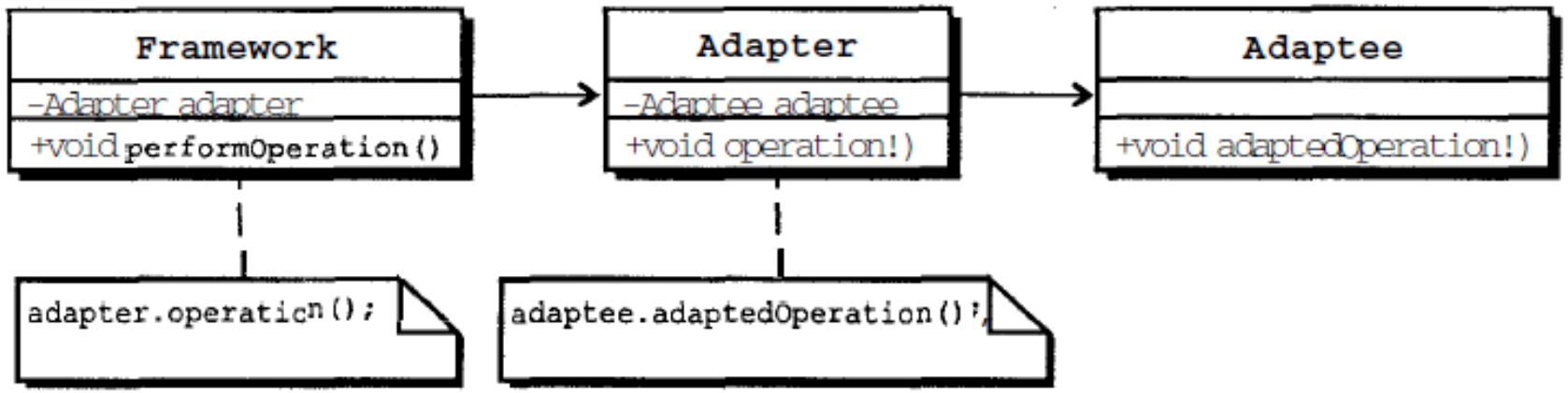
# Adapter



# Adapter


Your
Existing
System
Adapter
Vendor
Class
No changes
New code
No changes

# Adapter UML

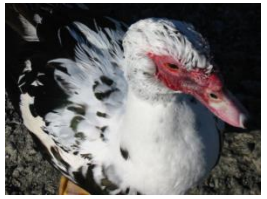

Framework
-Adapter adapter
+void performOperation()
Adapter
-Adaptee adaptee
+void operation!)
Adaptee
+void adaptedOperation!)
adapter.operation();
adaptee.adaptedOperation();

# Adapter Example

```java
public interface Duck {
    public void quack();
    public void fly();
}
```

```java
public class MallardDuck implements Duck {
    public void quack() {
        System.out.println("Quack");
    }

    public void fly() {
        System.out.println("I'm flying");
    }
}
```

```java
public interface Turkey {
    public void gobble();
    public void fly();
}
```

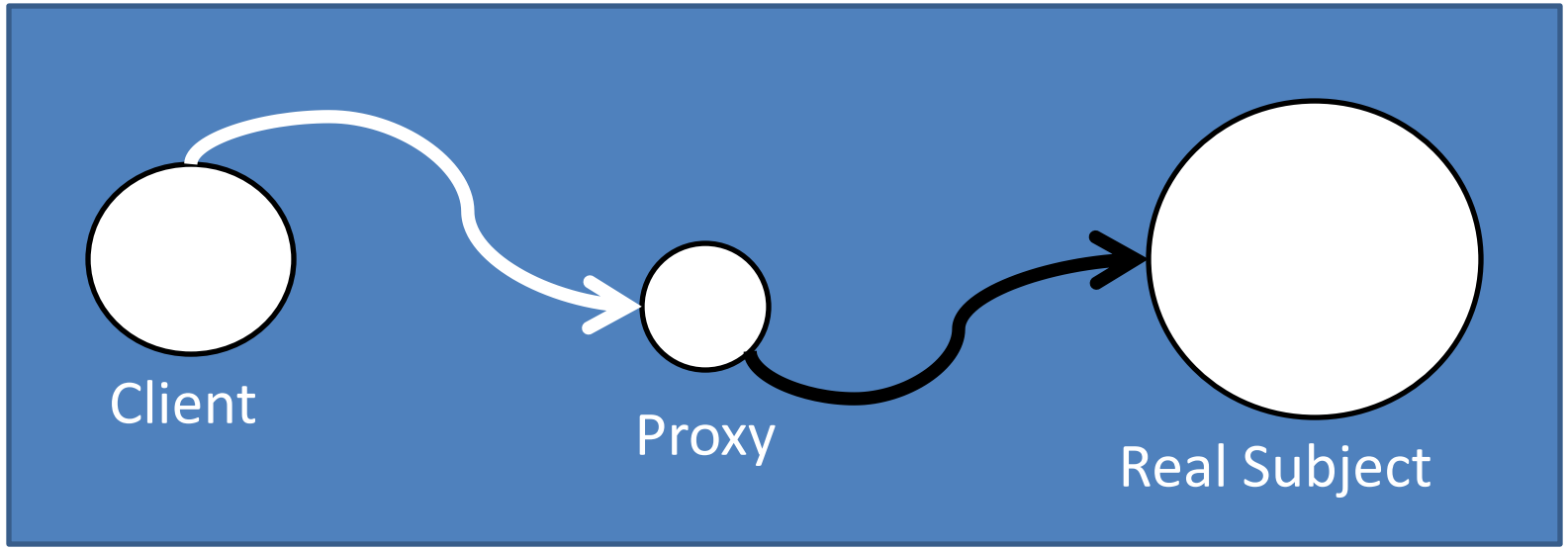


```java
public class WildTurkey implements Turkey {
    public void gobble() {
        System.out.println("Gobble gobble");
    }

    public void fly() {
        System.out.println("I'm flying a short distance");
    }
}
```

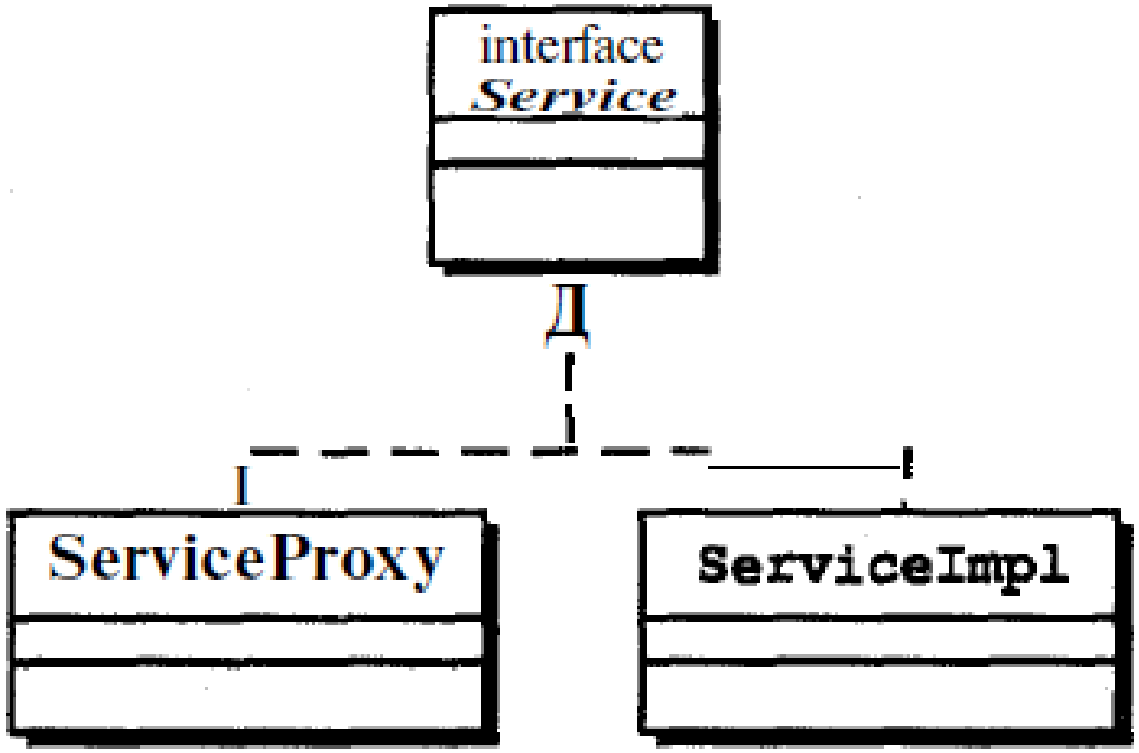
# Use adapter

```
public class TurkeyAdapter implements Duck {
    Turkey turkey;

    public TurkeyAdapter(Turkey turkey) {
        this.turkey = turkey;
    }

    public void quack() {
        turkey.gobble();
    }

    public void fly() {
        for(int i=0; i < 5; i++) {
            turkey.fly();
        }
    }
}
```

# Proxy



# Proxy UML



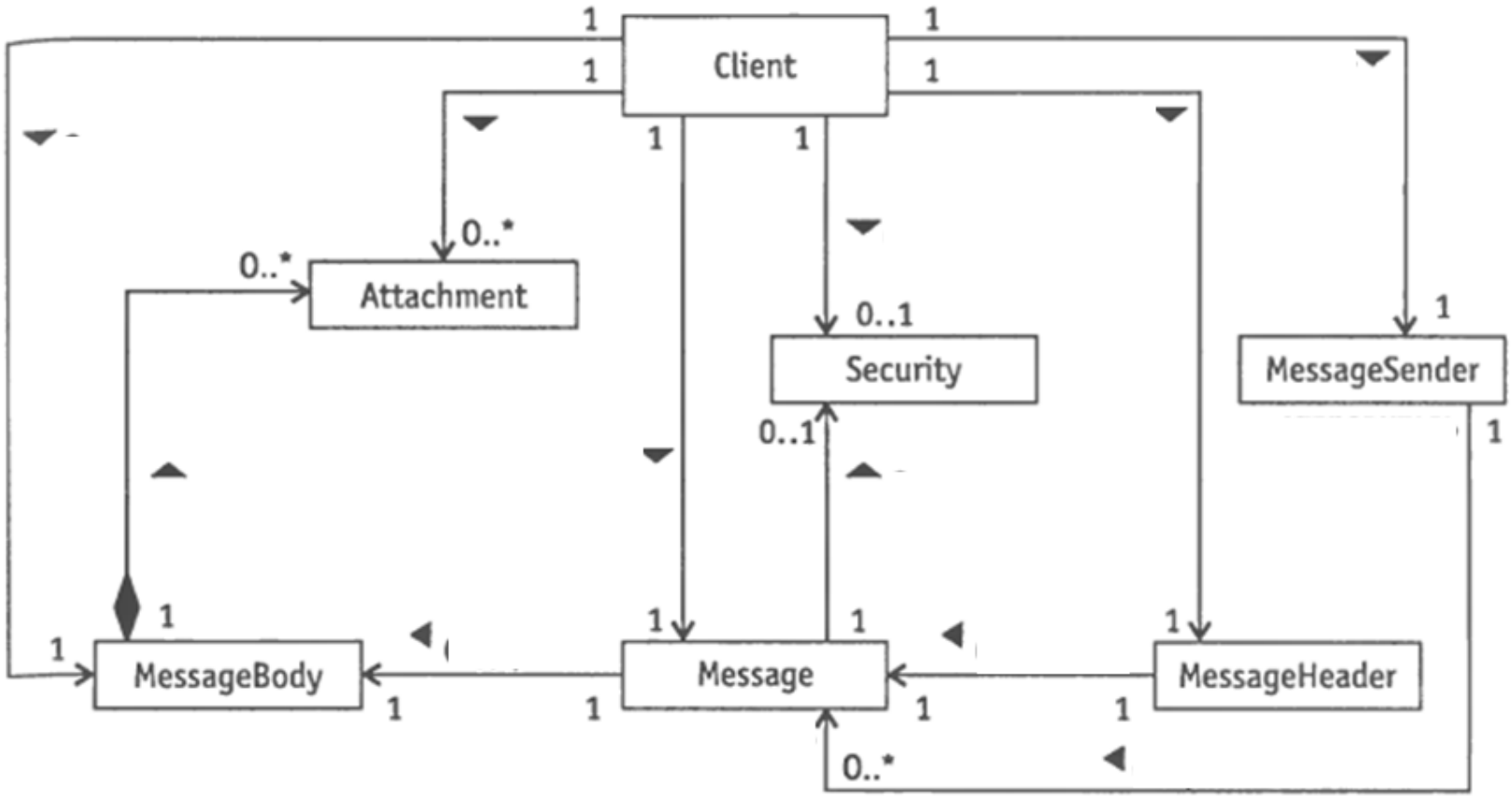
# Proxy

```java
public interface Service {

    void doSomething();
}
```

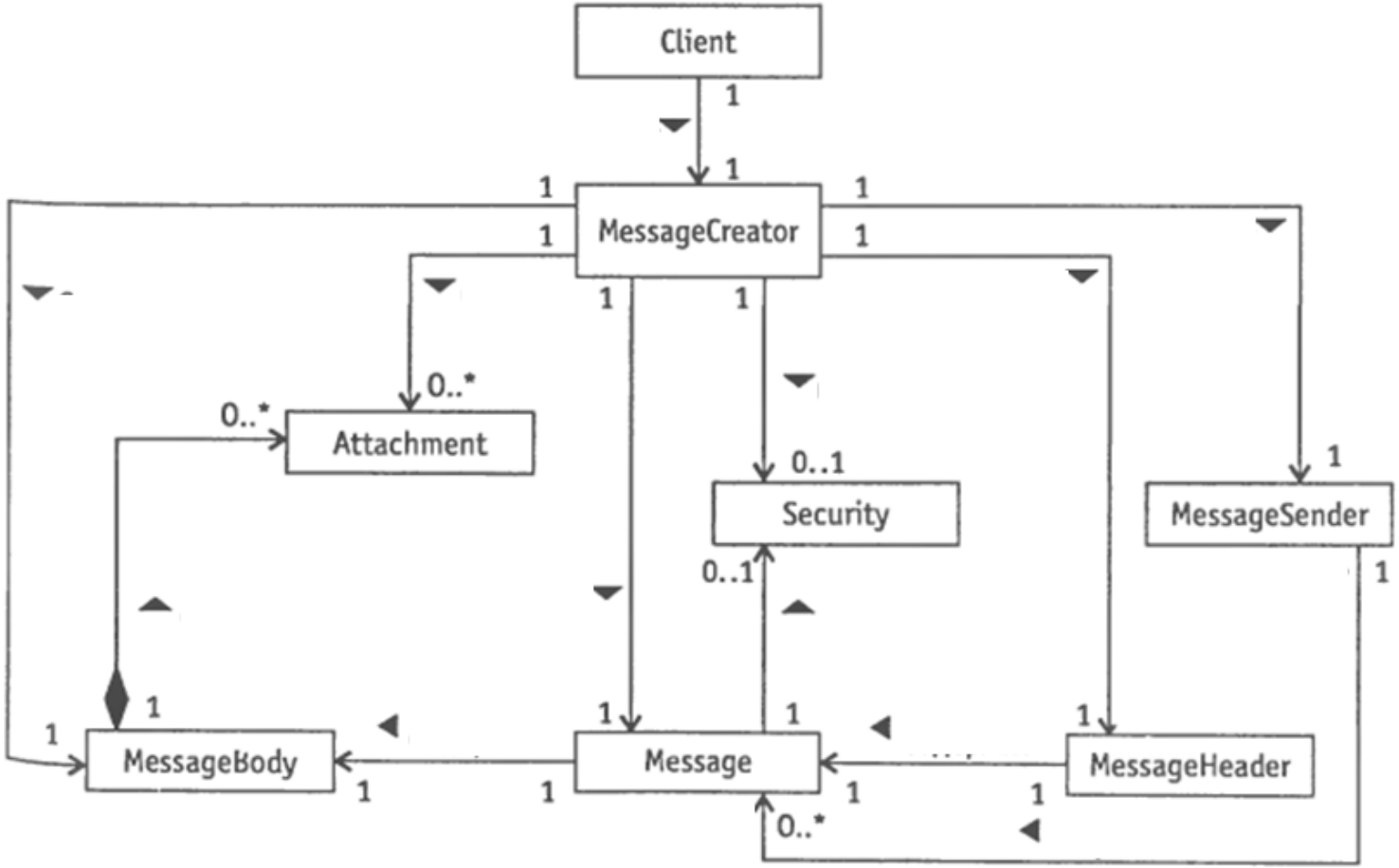
```java
public class ServiceImpl implements Service {

    @Override
    public void doSomething() {
        //some code here
    }
}
```

```java
public class ServiceProxy implements Service {

    private Service service = new ServiceImpl();

    @Override
    public void doSomething() {
        service.doSomething();
    }
}
```

# Façade


Client
Attachment
Security
MessageSender
MessageBody
Message
MessageHeader
1
0..*
0..1

# Façade UML


Client
MessageCreator
Attachment
Security
MessageSender
MessageBody
Message
MessageHeader
1
0..*
0..1

# Façade Example

```
public class HomeTheaterFacade {
    Amplifier amp;
    Tuner tuner;
    DvdPlayer dvd;
    CdPlayer cd;
    Projector projector;
    TheaterLights lights;
    Screen screen;
    PopcornPopper popper;

    public HomeTheaterFacade(Amplifier amp,
                Tuner tuner,
                DvdPlayer dvd,
                CdPlayer cd,
                Projector projector,
                Screen screen,
                TheaterLights lights,
                PopcornPopper popper) {

        this.amp = amp;
        this.tuner = tuner;
        this.dvd = dvd;
        this.cd = cd;
        this.projector = projector;
        this.screen = screen;
        this.lights = lights;
        this.popper = popper;
    }
```

```
    public void watchMovie(String movie) {
        popper.on();
        popper.pop();
        lights.dim(10);
        screen.down();
        projector.on();
        projector.wideScreenMode();
        amp.on();
        amp.setDvd(dvd);
        amp.setSurroundSound();
        amp.setVolume(5);
        dvd.on();
        dvd.play(movie);
    }


    public void endMovie() {
        popper.off();
        lights.on();
        screen.up();
        projector.off();
        amp.off();
        dvd.stop();
        dvd.eject();
        dvd.off();
    }
```

# Books

- Head First Design Patterns: A Brain-Friendly Guide, E. Freeman
- Patterns in Java, M. Grand

# Useful Links

- https://sourcemaking.com/design_patterns

# Thanks