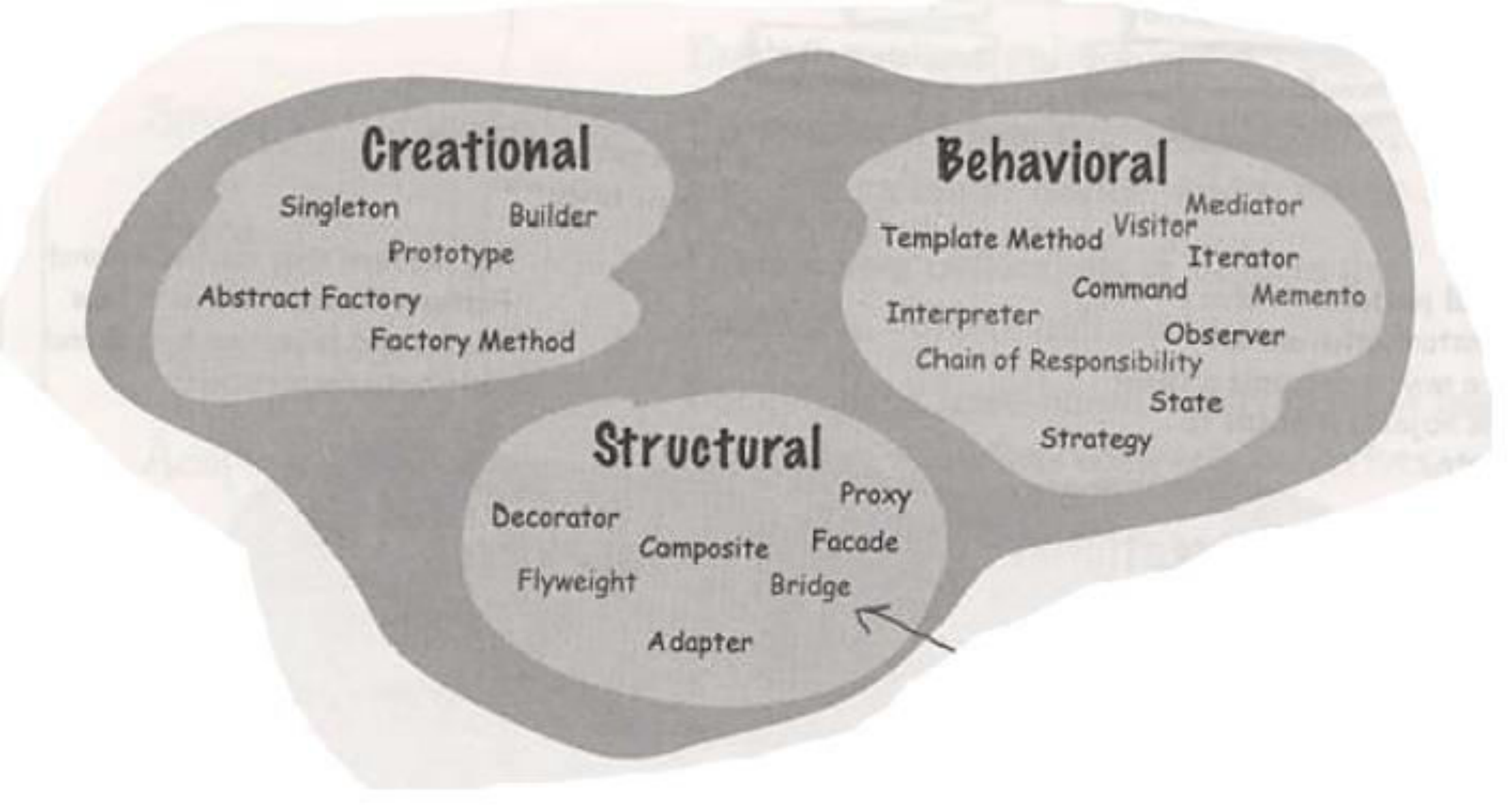
Design Patterns

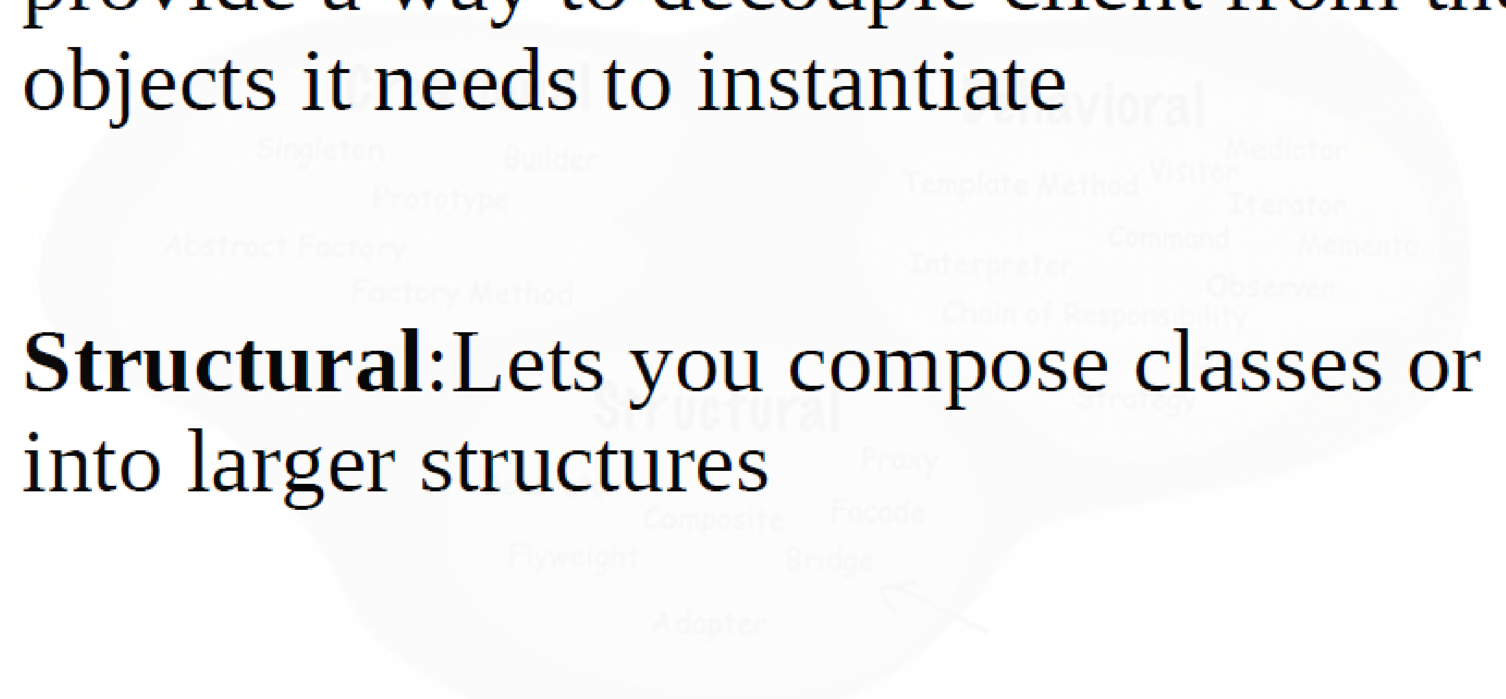
Software Architecture (SA)
3rd Year – Semester 1
By Udara Samaratunge

Fundamental Design Patterns (Gang Of Four - GOF)



By Udara Samaratunge

- **Creational**: Involve object initialization and provide a way to decouple client from the objects it needs to instantiate
- **Structural**:Lets you compose classes or objects into larger structures
- **Behavioral**: Concerned with how classes and objects interact or distribute responsibility



Gang of Four Patterns

Creational :

- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton

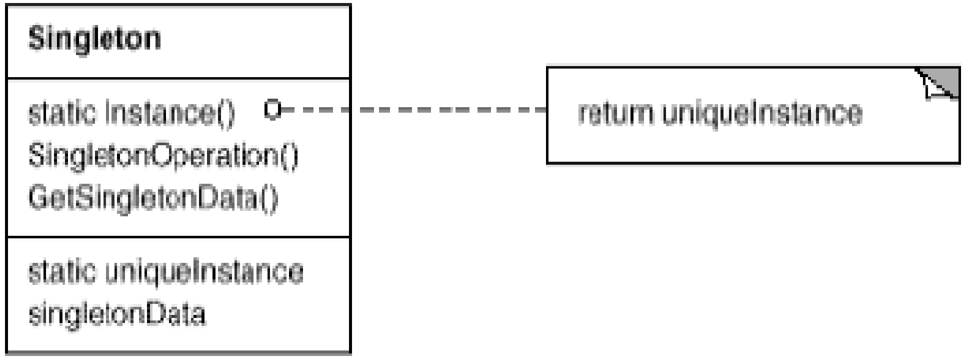
Behavioral :

- Strategy
- Observer
- Command
- Interpreter
- Iterator
- Mediator
- Memento
- Chain of Responsibility
- State
- Template Method
- Visitor

Structural :

- Adapter
- Bridge
- Composite
- Decorator
- Facade
- Flyweight
- Proxy

Singleton Pattern



Singleton

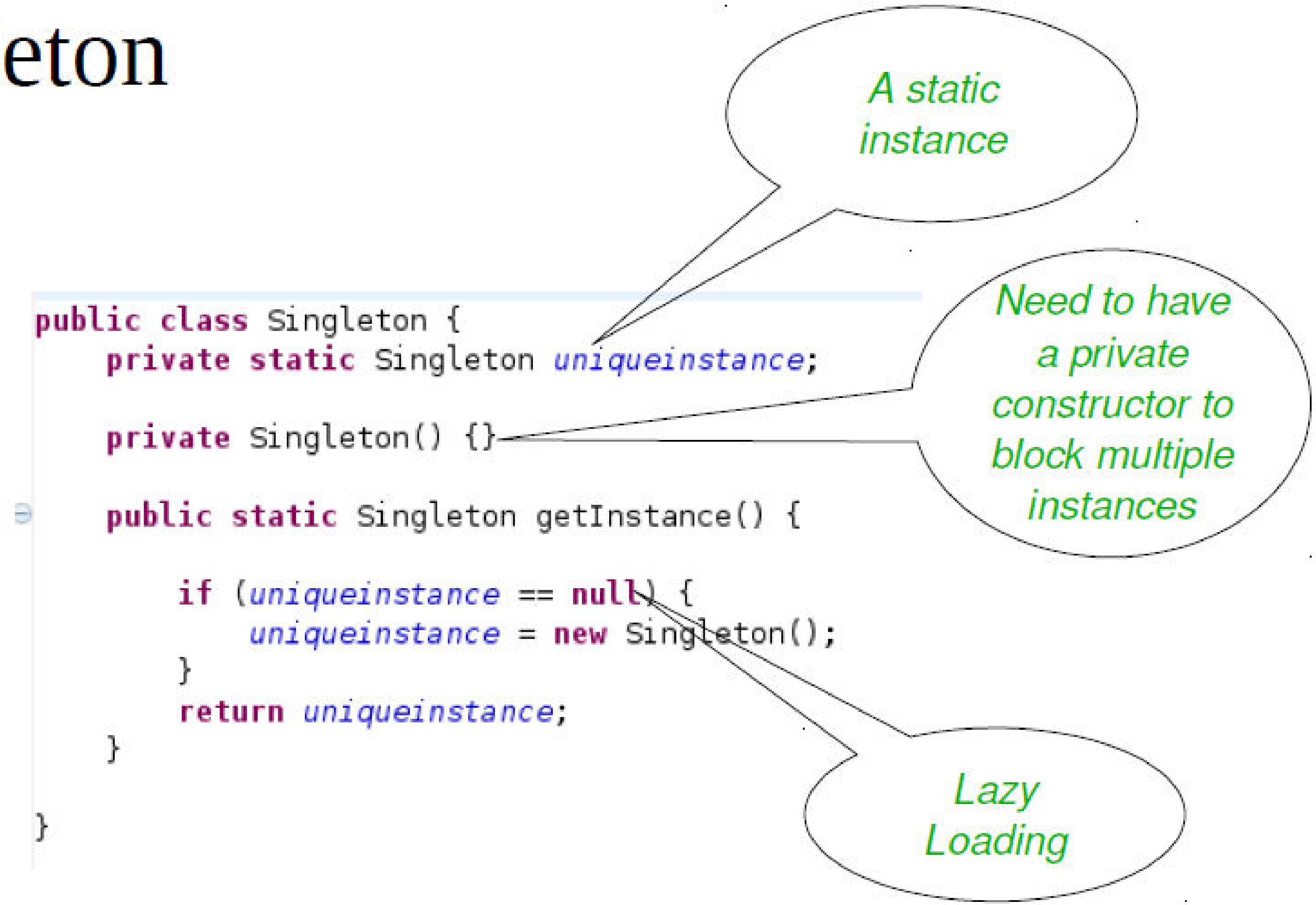
static Instance()
SingletonOperation()
GetSingletonData()

static uniqueInstance
singletonData

return uniqueInstance

Ensure a class only has one instance, and provide a global point of access to it

Singleton



```
public class Singleton {
    private static Singleton uniqueinstance;

    private Singleton() {}

    public static Singleton getInstance() {

        if (uniqueinstance == null) {
            uniqueinstance = new Singleton();
        }
        return uniqueinstance;
    }

}
```

A static instance

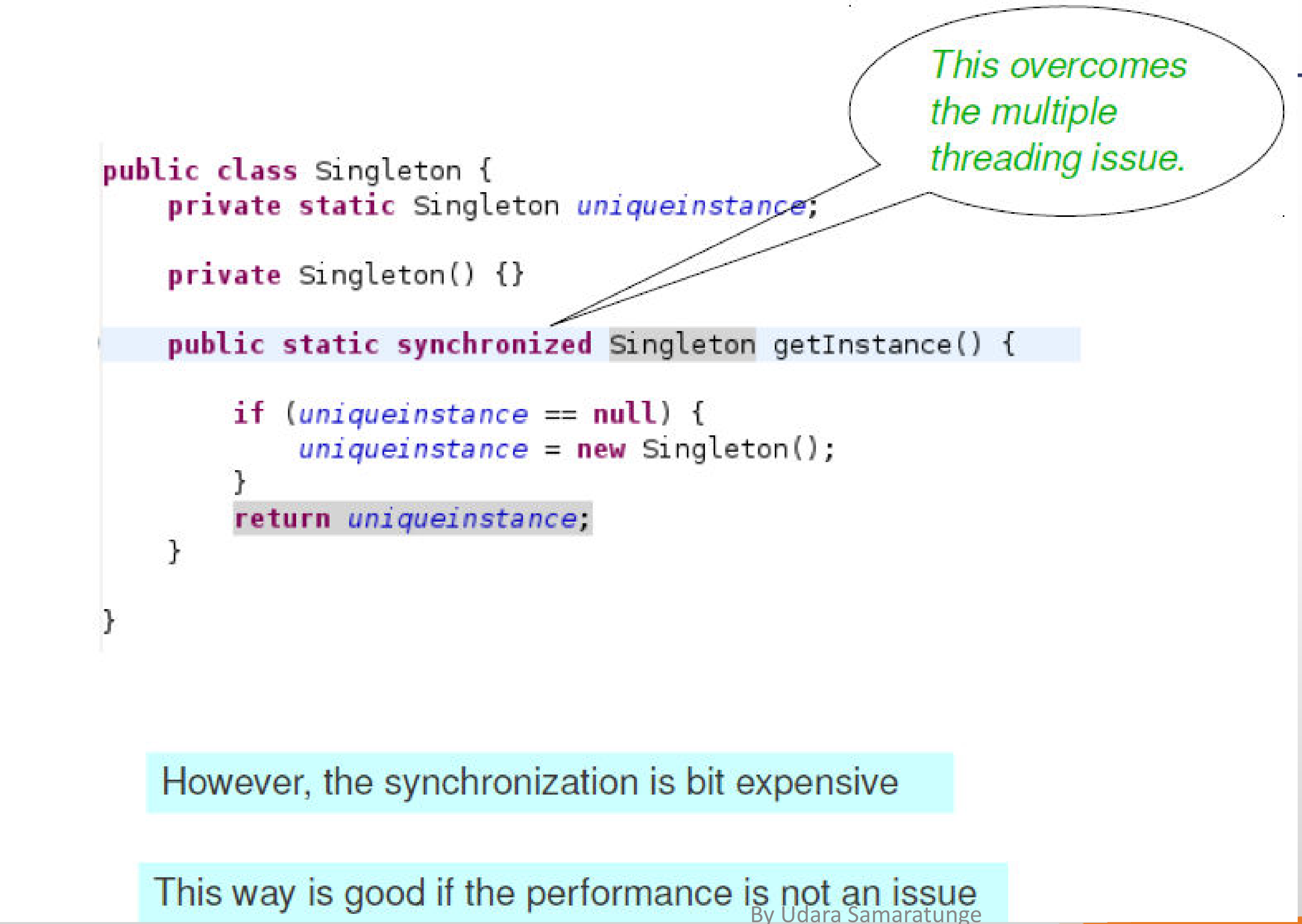
Need to have a private constructor to block multiple instances

Lazy Loading

This ensures only one object instance is ever created.

However, this is good only for a single-threaded application

Singleton



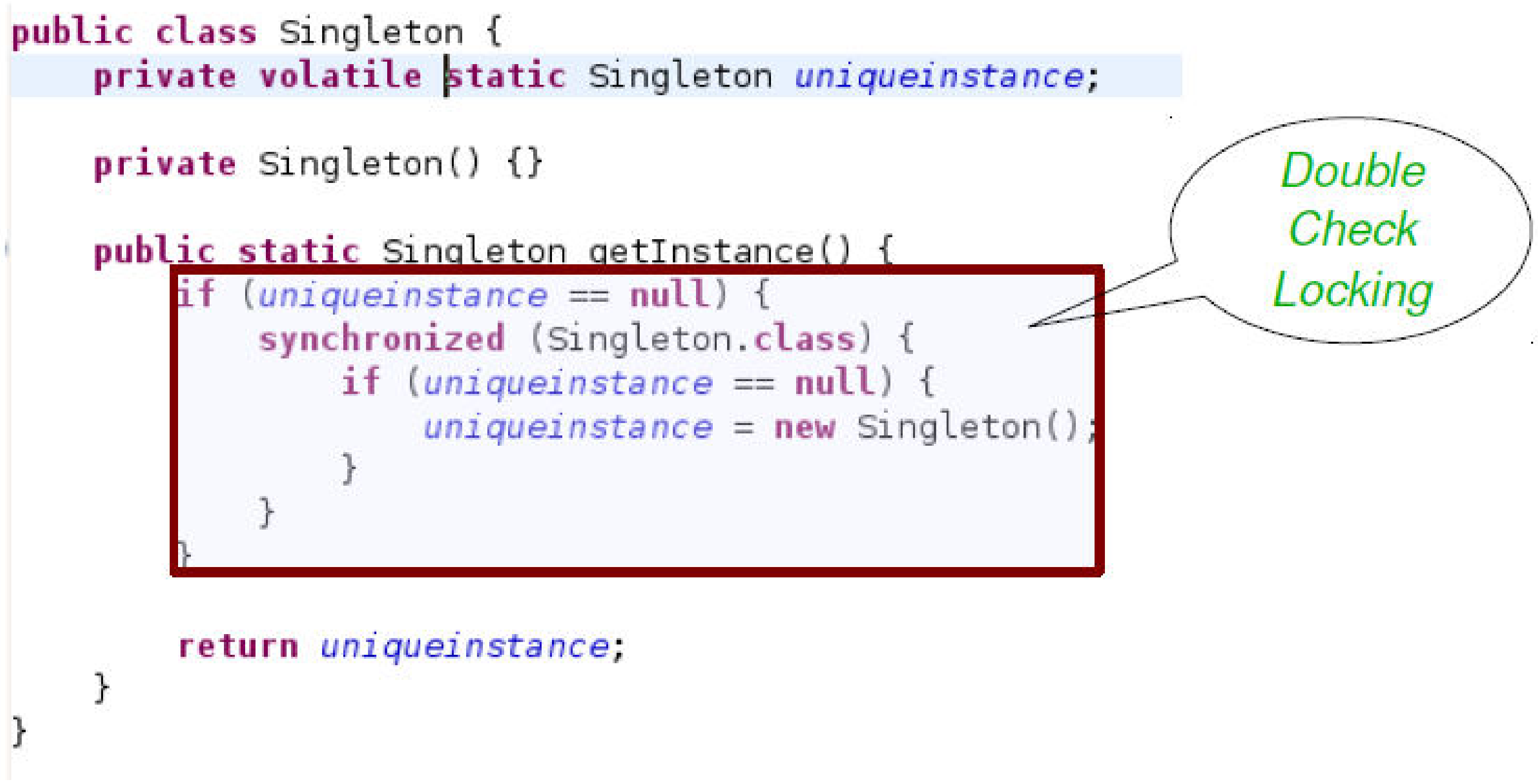
```
public class Singleton {
    private static Singleton uniqueinstance;

    private Singleton() {}

    public static synchronized Singleton getInstance() {

        if (uniqueinstance == null) {
            uniqueinstance = new Singleton();
        }
        return uniqueinstance;
    }

}
```

This overcomes the multiple threading issue.

However, the synchronization is bit expensive

This way is good if the performance is not an issue

Singleton – with Double Check Lock



```
public class Singleton {
    private volatile static Singleton uniqueinstance;

    private Singleton() {}

    public static Singleton getInstance() {
        if (uniqueinstance == null) {
            synchronized (Singleton.class) {
                if (uniqueinstance == null) {
                    uniqueinstance = new Singleton();
                }
            }
        }

        return uniqueinstance;
    }
}
```

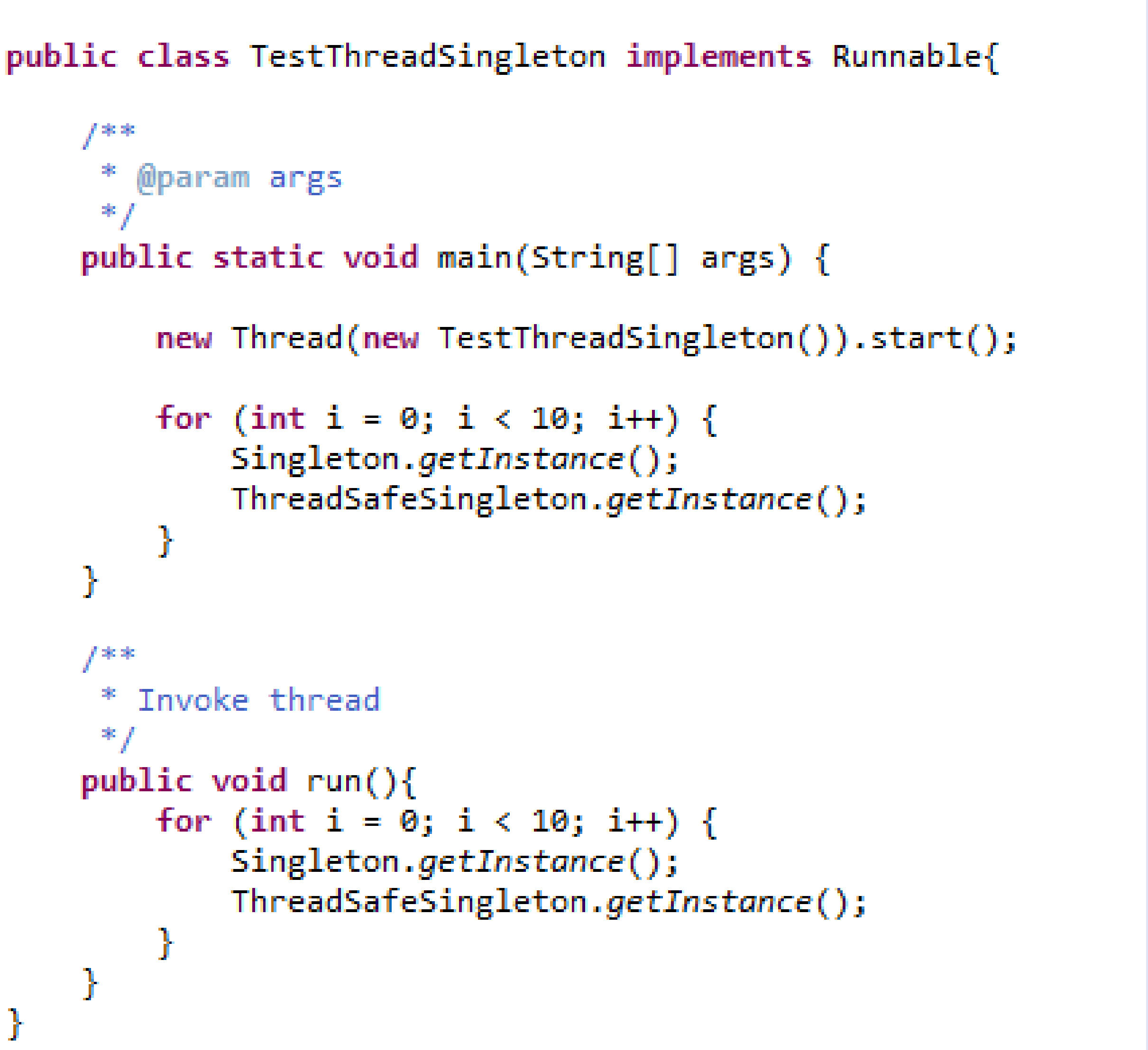
Double Check Locking

Here the object is created and synchronized at the first time only. If the Double Check Is not there, two threads can get into the synchronized block one after the other

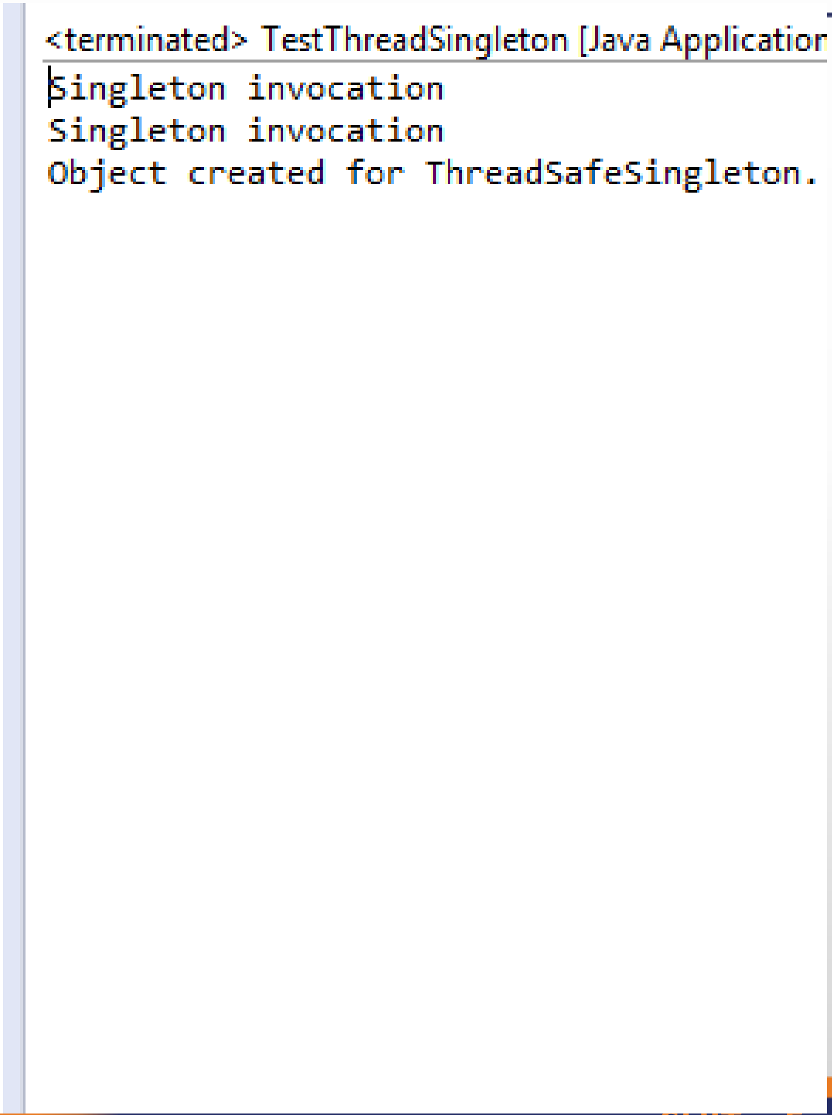
This way is good if the application is keen on its performance

Reference: http://www.ibm.com/developerworks/java/library/j-dcl.html

Thread-safe singleton output



```
public class TestThreadSingleton implements Runnable{

    /**
     * @param args
     */
    public static void main(String[] args) {

        new Thread(new TestThreadSingleton()).start();

        for (int i = 0; i < 10; i++) {
            Singleton.getInstance();
            ThreadSafeSingleton.getInstance();
        }
    }

    /**
     * Invoke thread
     */
    public void run(){
        for (int i = 0; i < 10; i++) {
            Singleton.getInstance();
            ThreadSafeSingleton.getInstance();
        }
    }
}
```



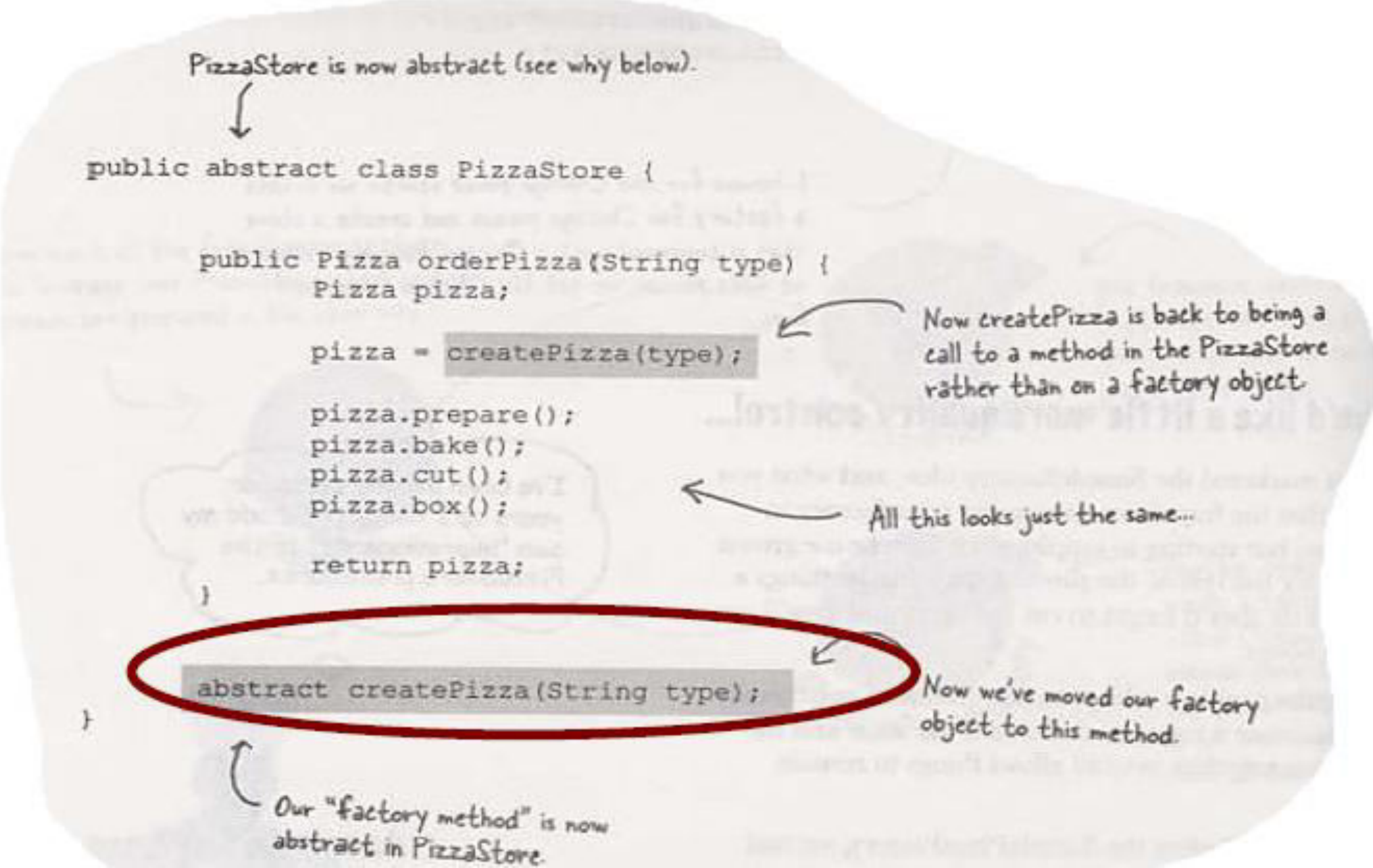
<terminated> TestThreadSingleton [Java Applicatior

```
Singleton invocation
Singleton invocation
Object created for ThreadSafeSingleton.
```

Factory Pattern

The Factory Method

The factory method pattern encapsulates the object creation by letting subclasses to decide what objects to create



PizzaStore is now abstract (see why below).

```
public abstract class PizzaStore {

    public Pizza orderPizza(String type) {
        Pizza pizza;

        pizza = createPizza(type);

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

        return pizza;
    }

    abstract createPizza(String type);

}
```

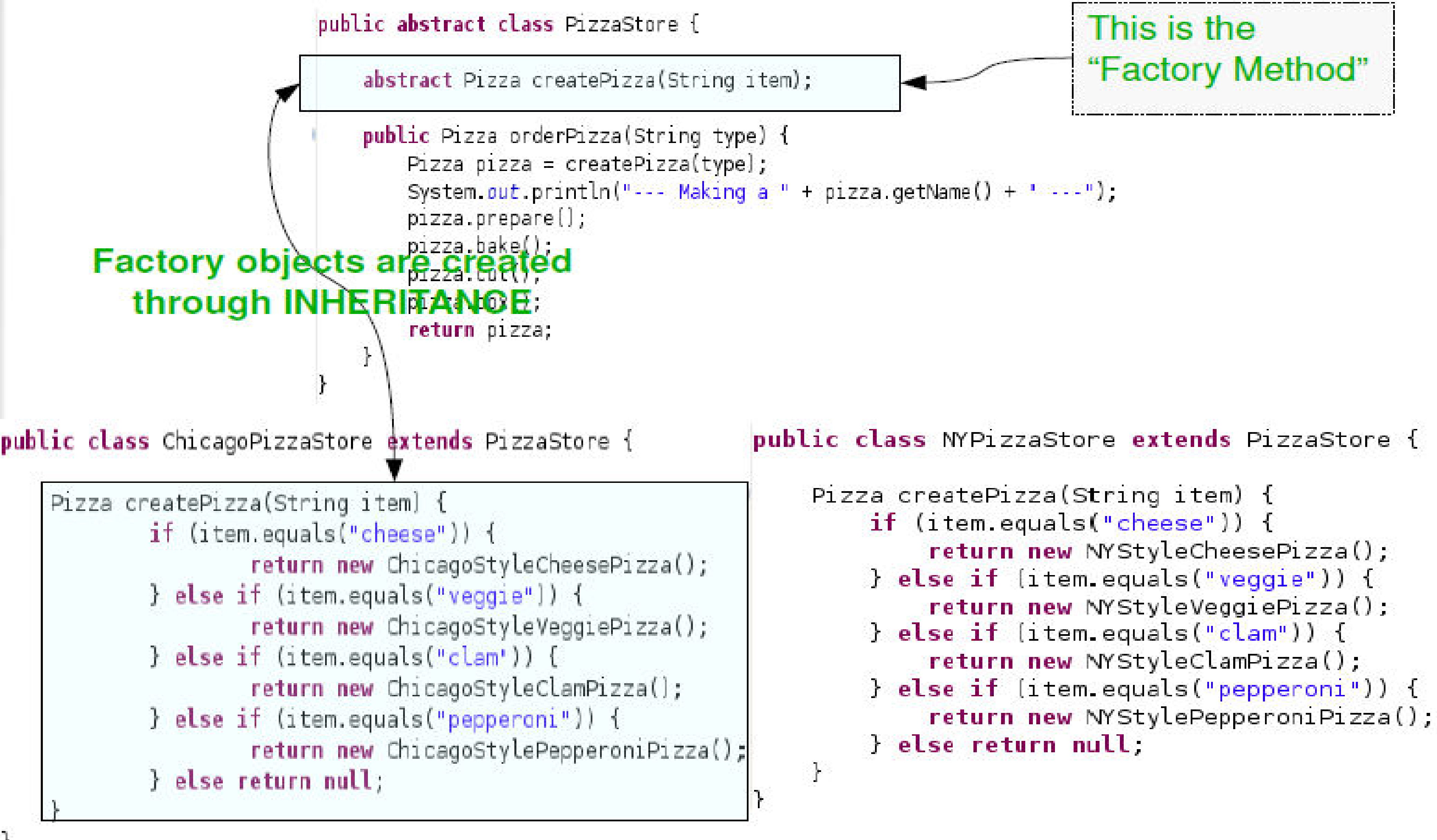
Now createPizza is back to being a call to a method in the PizzaStore rather than on a factory object.

All this looks just the same...

Now we've moved our factory object to this method.

Our "factory method" is now abstract in PizzaStore.

The Creator Classes



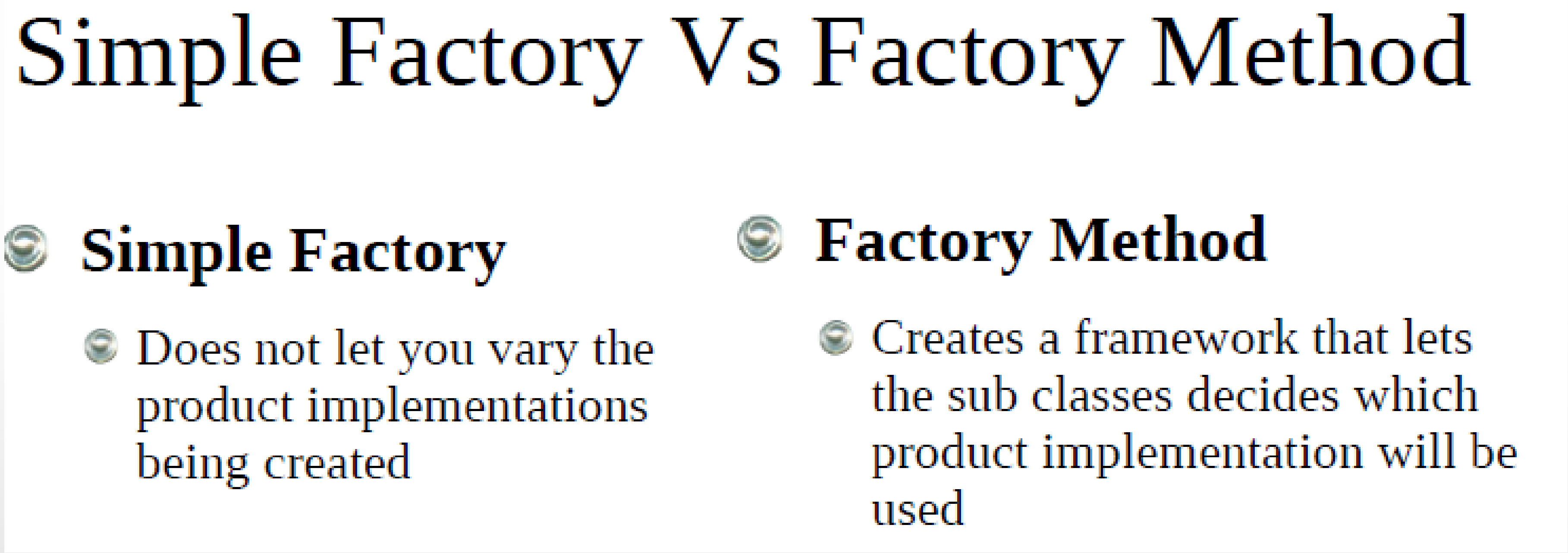
```
public abstract class PizzaStore {

    abstract Pizza createPizza(String item);

    public Pizza orderPizza(String type) {
        Pizza pizza = createPizza(type);
        System.out.println("--- Making a " + pizza.getName() + " ---");
        pizza.prepare();
        pizza.bake();
        pizza.cut();
        pizza.box();
        return pizza;
    }
}
```

This is the “Factory Method”

Factory objects are created through INHERITANCE

```
public class ChicagoPizzaStore extends PizzaStore {

Pizza createPizza(String item) {
        if (item.equals("cheese")) {
                return new ChicagoStyleCheesePizza();
        } else if (item.equals("veggie")) {
                return new ChicagoStyleVeggiePizza();
        } else if (item.equals("clam")) {
                return new ChicagoStyleClamPizza();
        } else if (item.equals("pepperoni")) {
                return new ChicagoStylePepperoniPizza();
        } else return null;
}
}
```

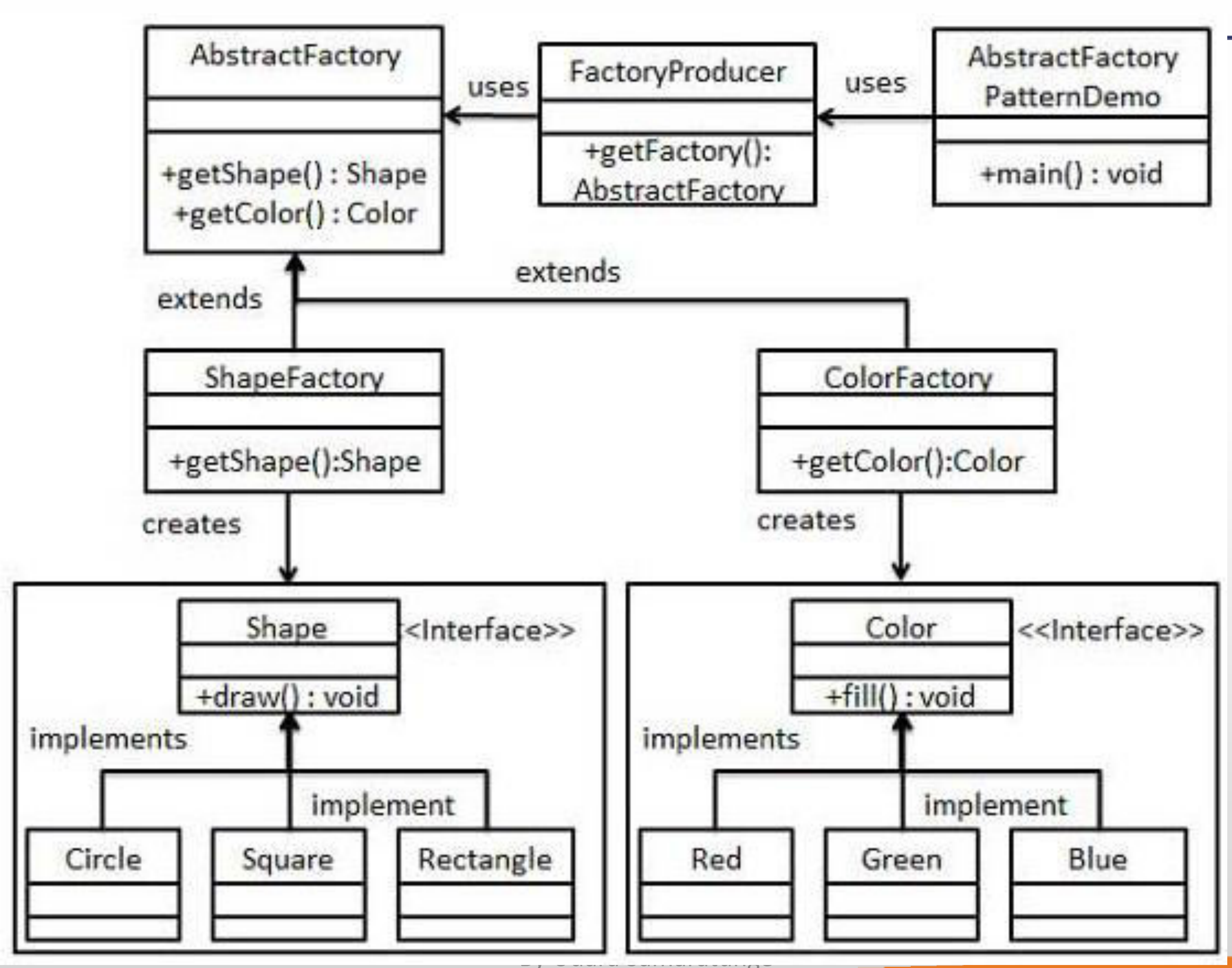
```
public class NYPizzaStore extends PizzaStore {

    Pizza createPizza(String item) {
        if (item.equals("cheese")) {
            return new NYStyleCheesePizza();
        } else if (item.equals("veggie")) {
            return new NYStyleVeggiePizza();
        } else if (item.equals("clam")) {
            return new NYStyleClamPizza();
        } else if (item.equals("pepperoni")) {
            return new NYStylePepperoniPizza();
        } else return null;
    }
}
```



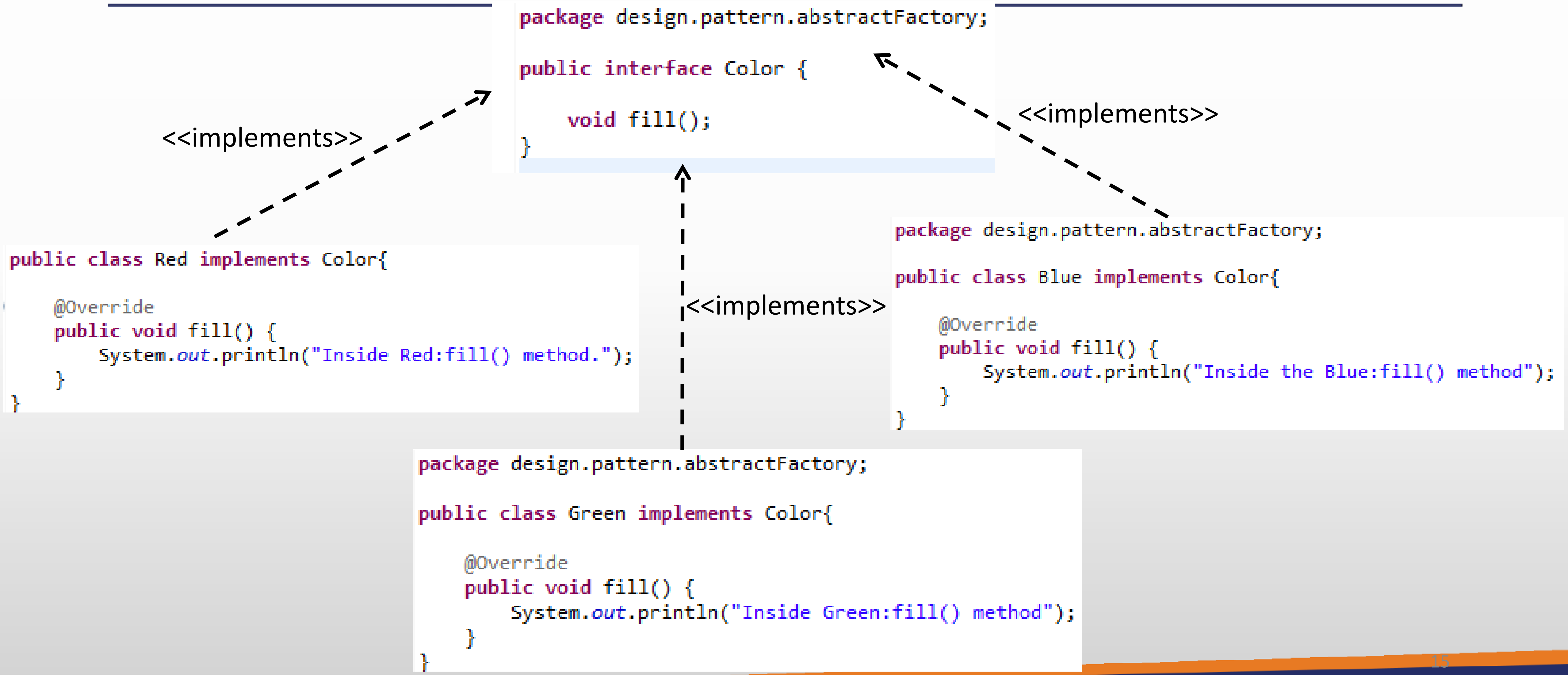
Simple Factory Vs Factory Method

- **Simple Factory**
  - Does not let you vary the product implementations being created
- **Factory Method**
  - Creates a framework that lets the sub classes decides which product implementation will be used

Abstract Factory Pattern



Abstract Factory Pattern



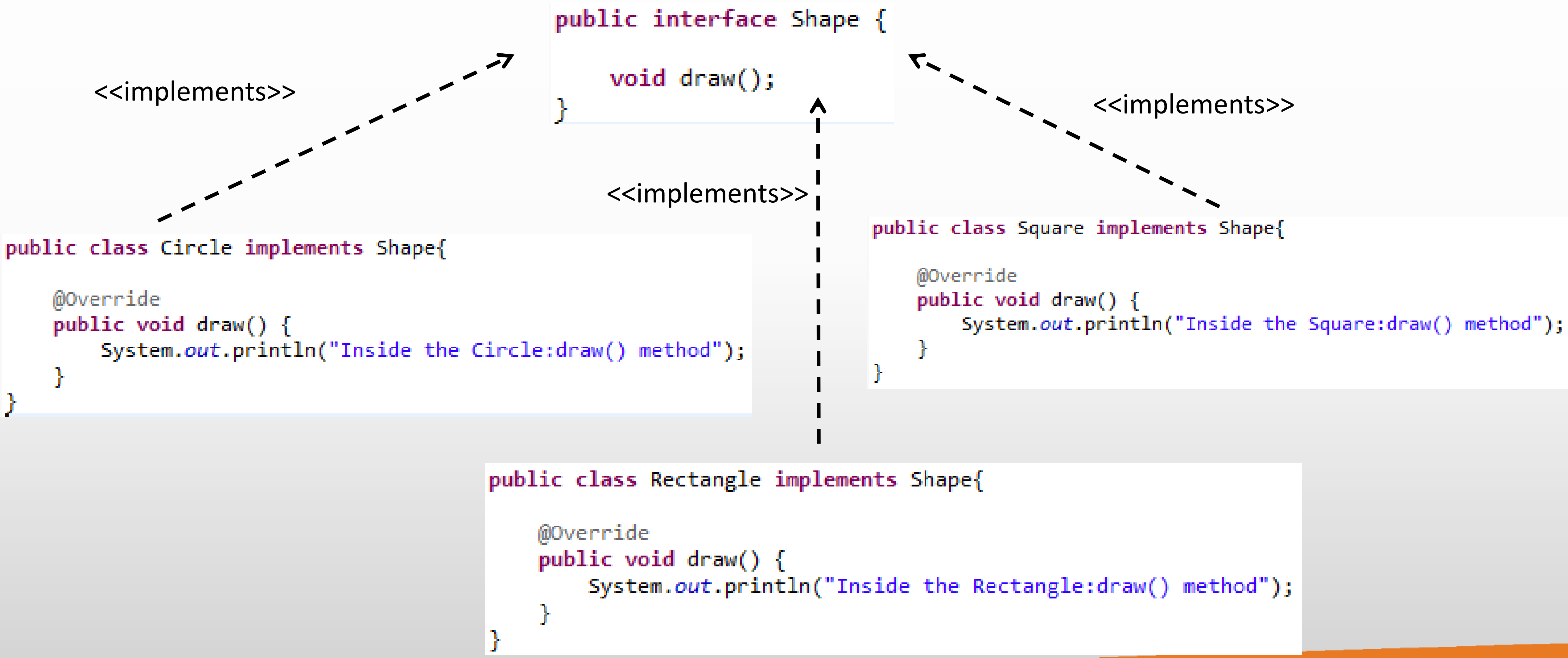
```java
package design.pattern.abstractFactory;

public interface Color {

    void fill();
}
```

```java
public class Red implements Color{

    @Override
    public void fill() {
        System.out.println("Inside Red:fill() method.");
    }
}
```

```java
package design.pattern.abstractFactory;

public class Blue implements Color{

    @Override
    public void fill() {
        System.out.println("Inside the Blue:fill() method");
    }
}
```

```java
package design.pattern.abstractFactory;

public class Green implements Color{

    @Override
    public void fill() {
        System.out.println("Inside Green:fill() method");
    }
}
```

Abstract Factory Pattern



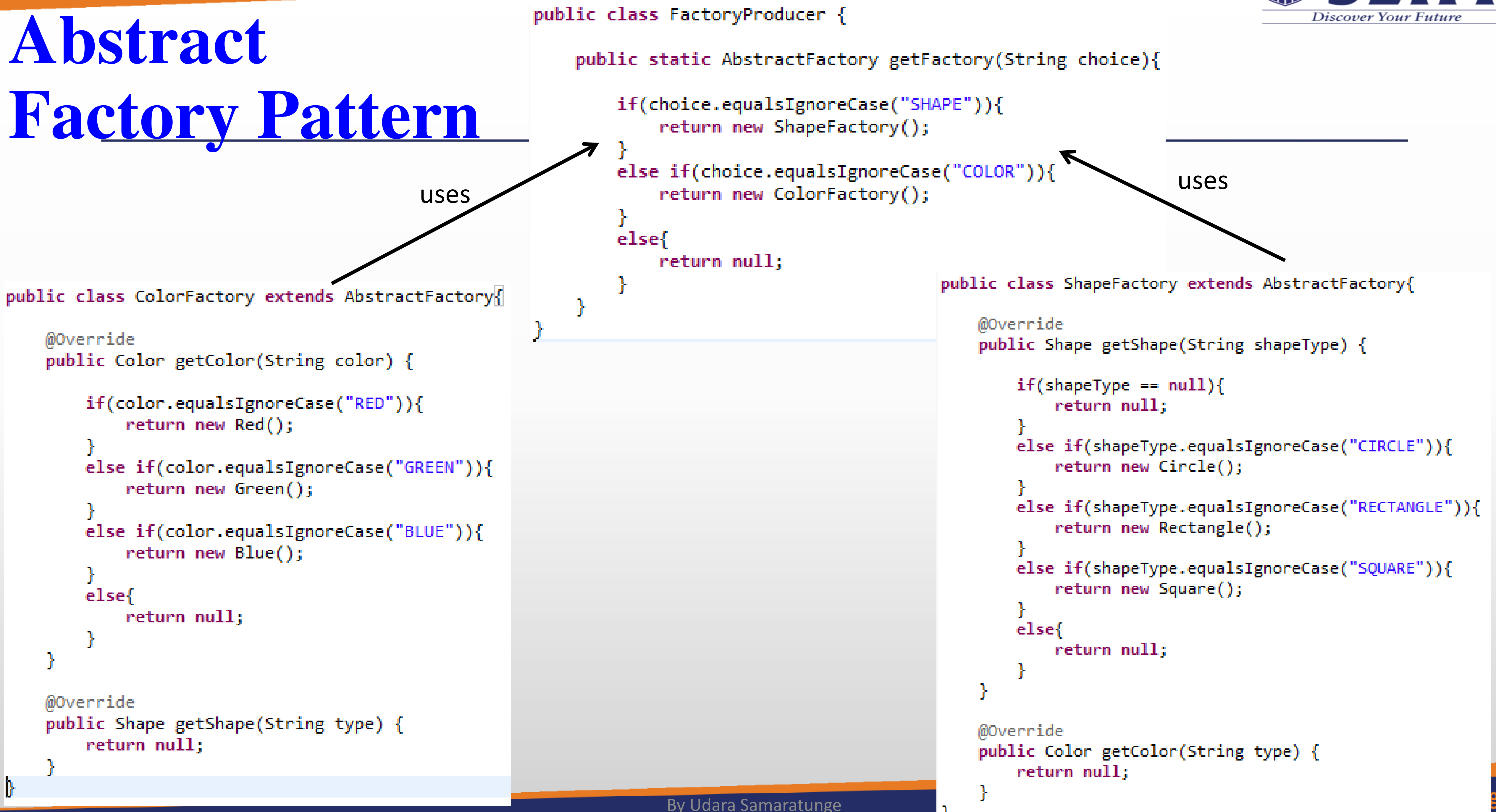
```java
public interface Shape {

    void draw();
}
```

```java
public class Circle implements Shape{

    @Override
    public void draw() {
        System.out.println("Inside the Circle:draw() method");
    }
}
```

```java
public class Square implements Shape{

    @Override
    public void draw() {
        System.out.println("Inside the Square:draw() method");
    }
}
```

```java
public class Rectangle implements Shape{

    @Override
    public void draw() {
        System.out.println("Inside the Rectangle:draw() method");
    }
}
```

Abstract Factory Pattern



uses

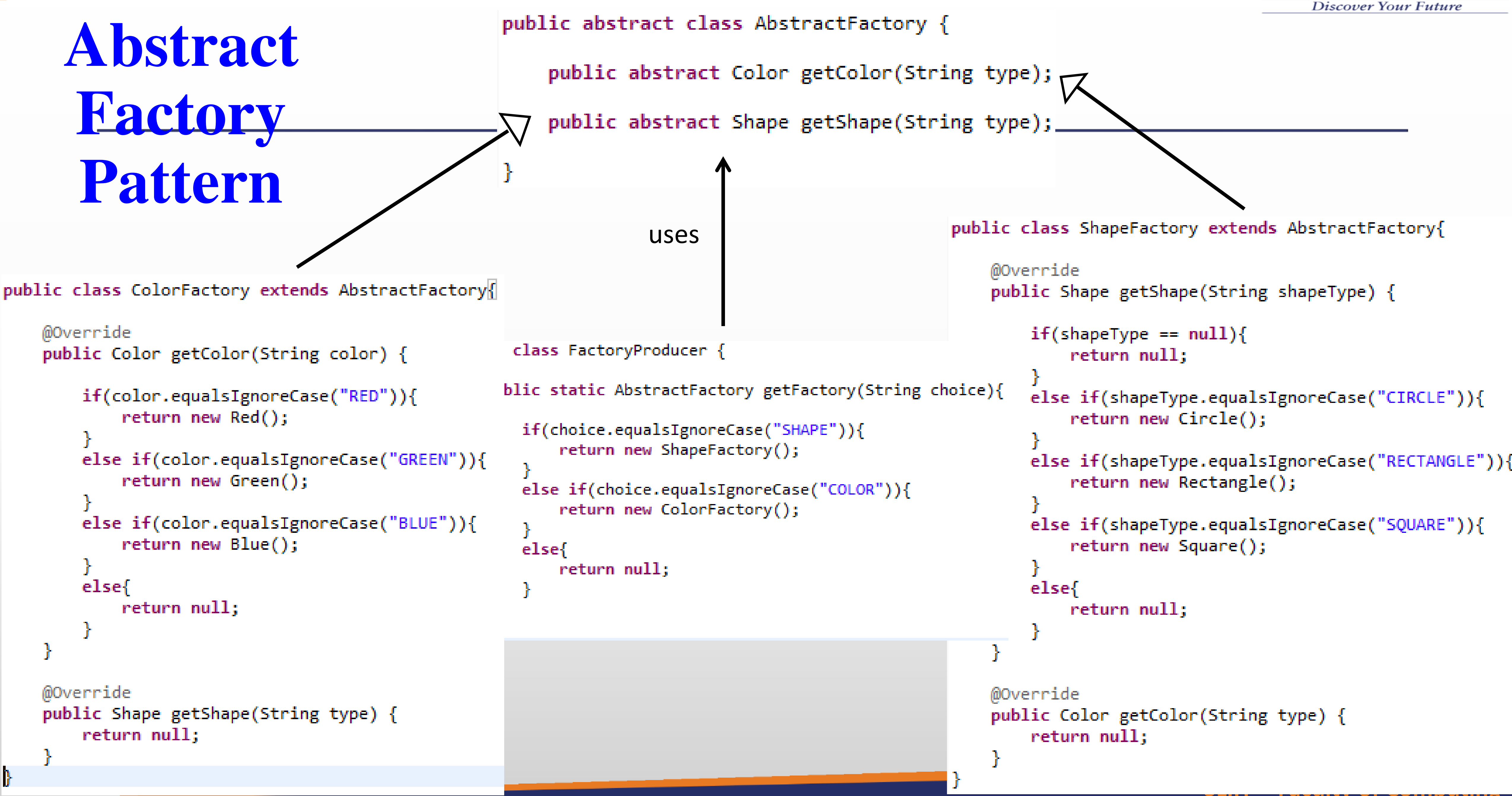
uses

```
public class FactoryProducer {

    public static AbstractFactory getFactory(String choice){

        if(choice.equalsIgnoreCase("SHAPE")){
            return new ShapeFactory();
        }
        else if(choice.equalsIgnoreCase("COLOR")){
            return new ColorFactory();
        }
        else{
            return null;
        }
    }
}
```

```
public class ColorFactory extends AbstractFactory{

    @Override
    public Color getColor(String color) {

        if(color.equalsIgnoreCase("RED")){
            return new Red();
        }
        else if(color.equalsIgnoreCase("GREEN")){
            return new Green();
        }
        else if(color.equalsIgnoreCase("BLUE")){
            return new Blue();
        }
        else{
            return null;
        }
    }

    @Override
    public Shape getShape(String type) {
        return null;
    }
}
```

```
public class ShapeFactory extends AbstractFactory{

    @Override
    public Shape getShape(String shapeType) {

        if(shapeType == null){
            return null;
        }
        else if(shapeType.equalsIgnoreCase("CIRCLE")){
            return new Circle();
        }
        else if(shapeType.equalsIgnoreCase("RECTANGLE")){
            return new Rectangle();
        }
        else if(shapeType.equalsIgnoreCase("SQUARE")){
            return new Square();
        }
        else{
            return null;
        }
    }

    @Override
    public Color getColor(String type) {
        return null;
    }
}
```

Abstract Factory Pattern



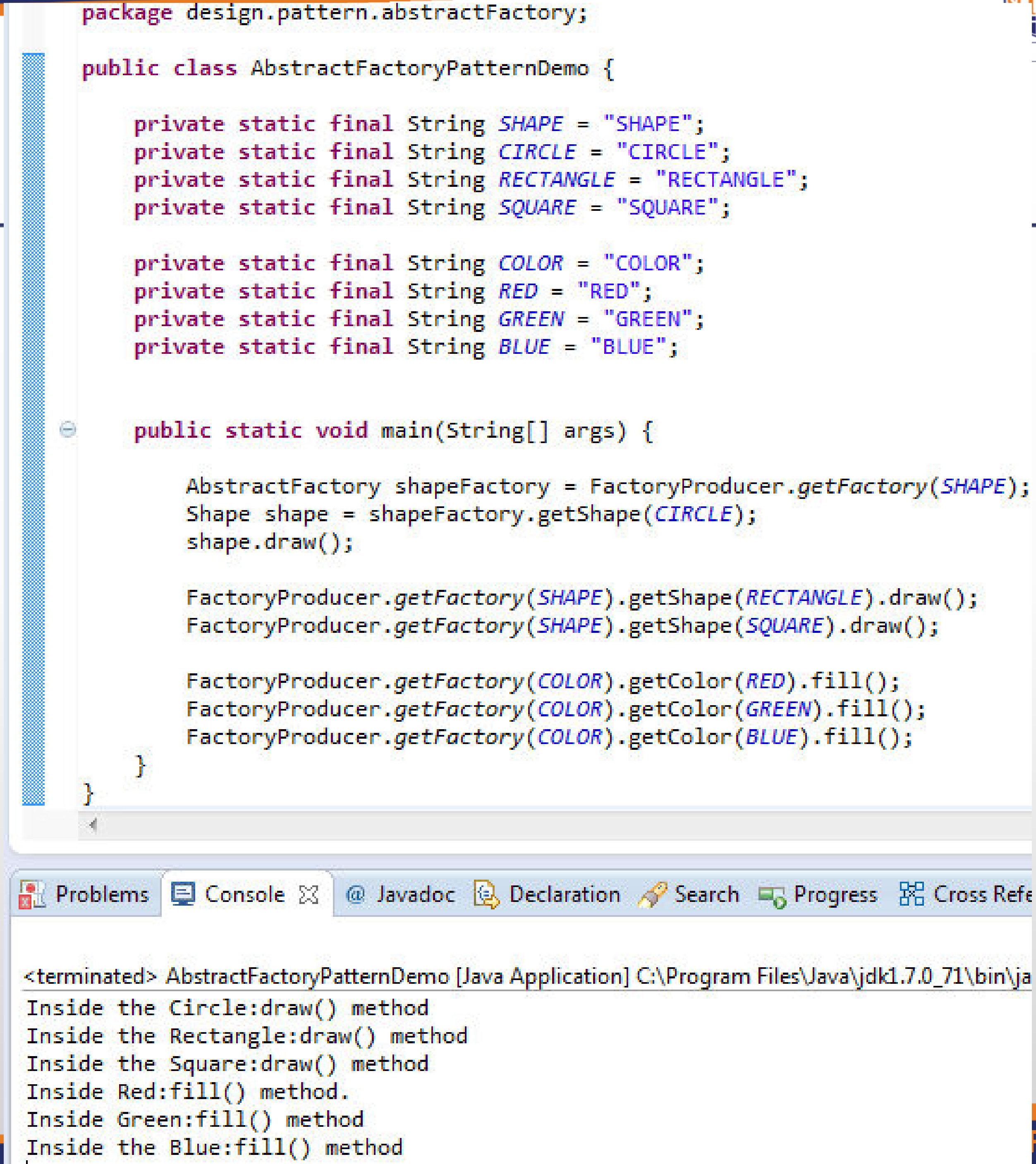
```
public abstract class AbstractFactory {

    public abstract Color getColor(String type);

    public abstract Shape getShape(String type);

}
```

```
public class ColorFactory extends AbstractFactory{

    @Override
    public Color getColor(String color) {

        if(color.equalsIgnoreCase("RED")){
            return new Red();
        }
        else if(color.equalsIgnoreCase("GREEN")){
            return new Green();
        }
        else if(color.equalsIgnoreCase("BLUE")){
            return new Blue();
        }
        else{
            return null;
        }
    }

    @Override
    public Shape getShape(String type) {
        return null;
    }
}
```

```
 class FactoryProducer {

blic static AbstractFactory getFactory(String choice){

   if(choice.equalsIgnoreCase("SHAPE")){
       return new ShapeFactory();
   }
   else if(choice.equalsIgnoreCase("COLOR")){
       return new ColorFactory();
   }
   else{
       return null;
   }
```

```
public class ShapeFactory extends AbstractFactory{

    @Override
    public Shape getShape(String shapeType) {

        if(shapeType == null){
            return null;
        }
        else if(shapeType.equalsIgnoreCase("CIRCLE")){
            return new Circle();
        }
        else if(shapeType.equalsIgnoreCase("RECTANGLE")){
            return new Rectangle();
        }
        else if(shapeType.equalsIgnoreCase("SQUARE")){
            return new Square();
        }
        else{
            return null;
        }
    }

    @Override
    public Color getColor(String type) {
        return null;
    }
}
```

Abstract Factory Pattern



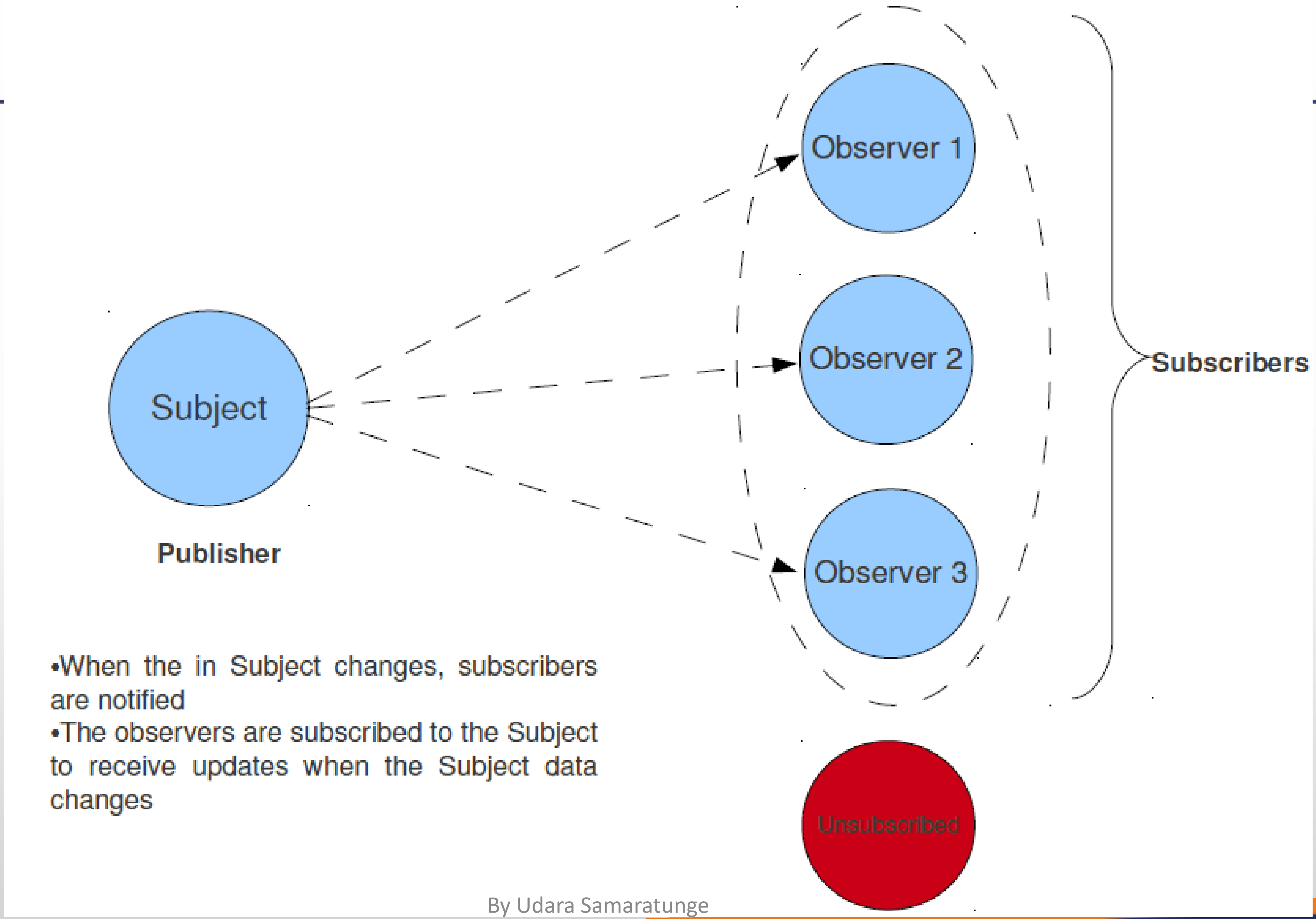
```java
package design.pattern.abstractFactory;

public class AbstractFactoryPatternDemo {

    private static final String SHAPE = "SHAPE";
    private static final String CIRCLE = "CIRCLE";
    private static final String RECTANGLE = "RECTANGLE";
    private static final String SQUARE = "SQUARE";

    private static final String COLOR = "COLOR";
    private static final String RED = "RED";
    private static final String GREEN = "GREEN";
    private static final String BLUE = "BLUE";


    public static void main(String[] args) {

        AbstractFactory shapeFactory = FactoryProducer.getFactory(SHAPE);
        Shape shape = shapeFactory.getShape(CIRCLE);
        shape.draw();

        FactoryProducer.getFactory(SHAPE).getShape(RECTANGLE).draw();
        FactoryProducer.getFactory(SHAPE).getShape(SQUARE).draw();

        FactoryProducer.getFactory(COLOR).getColor(RED).fill();
        FactoryProducer.getFactory(COLOR).getColor(GREEN).fill();
        FactoryProducer.getFactory(COLOR).getColor(BLUE).fill();
    }
}
```

<terminated> AbstractFactoryPatternDemo [Java Application] C:\Program Files\Java\jdk1.7.0_71\bin\ja

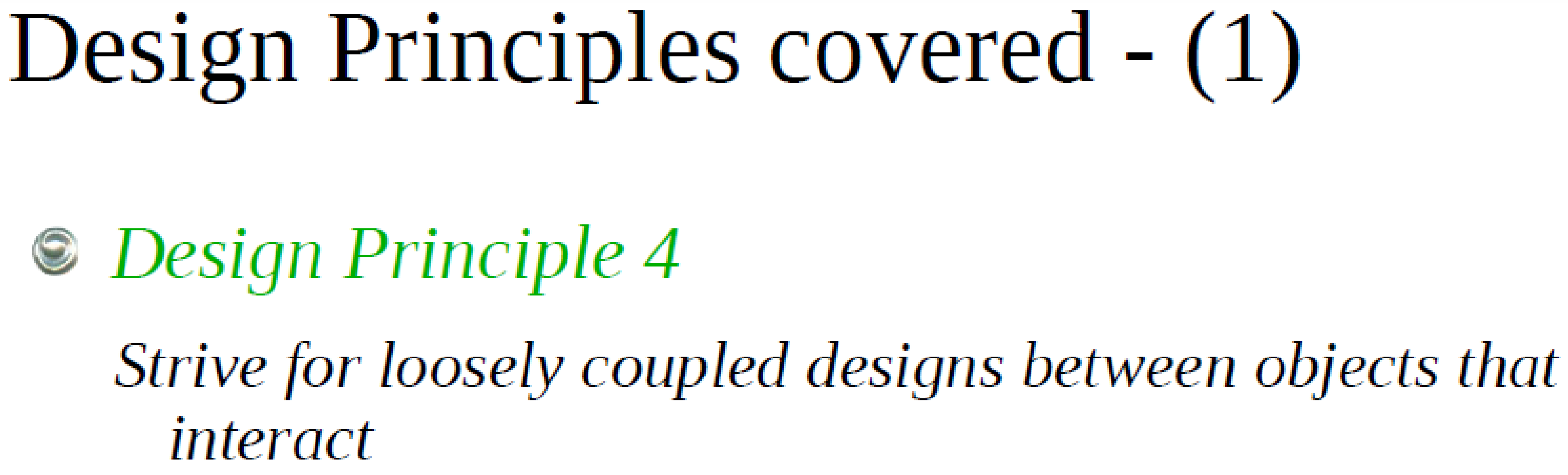
```
Inside the Circle:draw() method
Inside the Rectangle:draw() method
Inside the Square:draw() method
Inside Red:fill() method.
Inside Green:fill() method
Inside the Blue:fill() method
```

Observer Pattern

The Observer Pattern



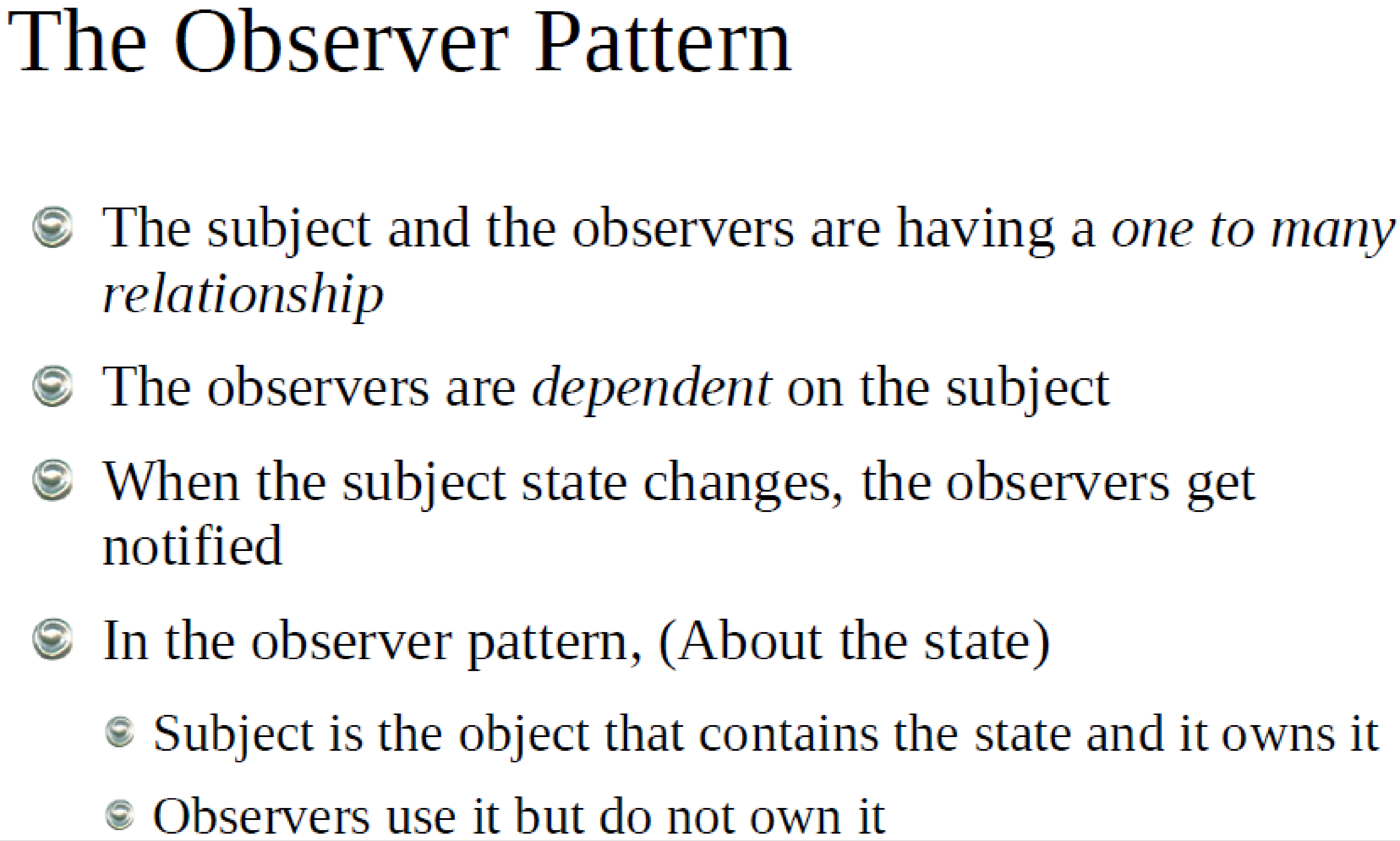
- When the in Subject changes, subscribers are notified
- The observers are subscribed to the Subject to receive updates when the Subject data changes



Design Principles covered - (1)

- *Design Principle 4*

  Strive for loosely coupled designs between objects that interact

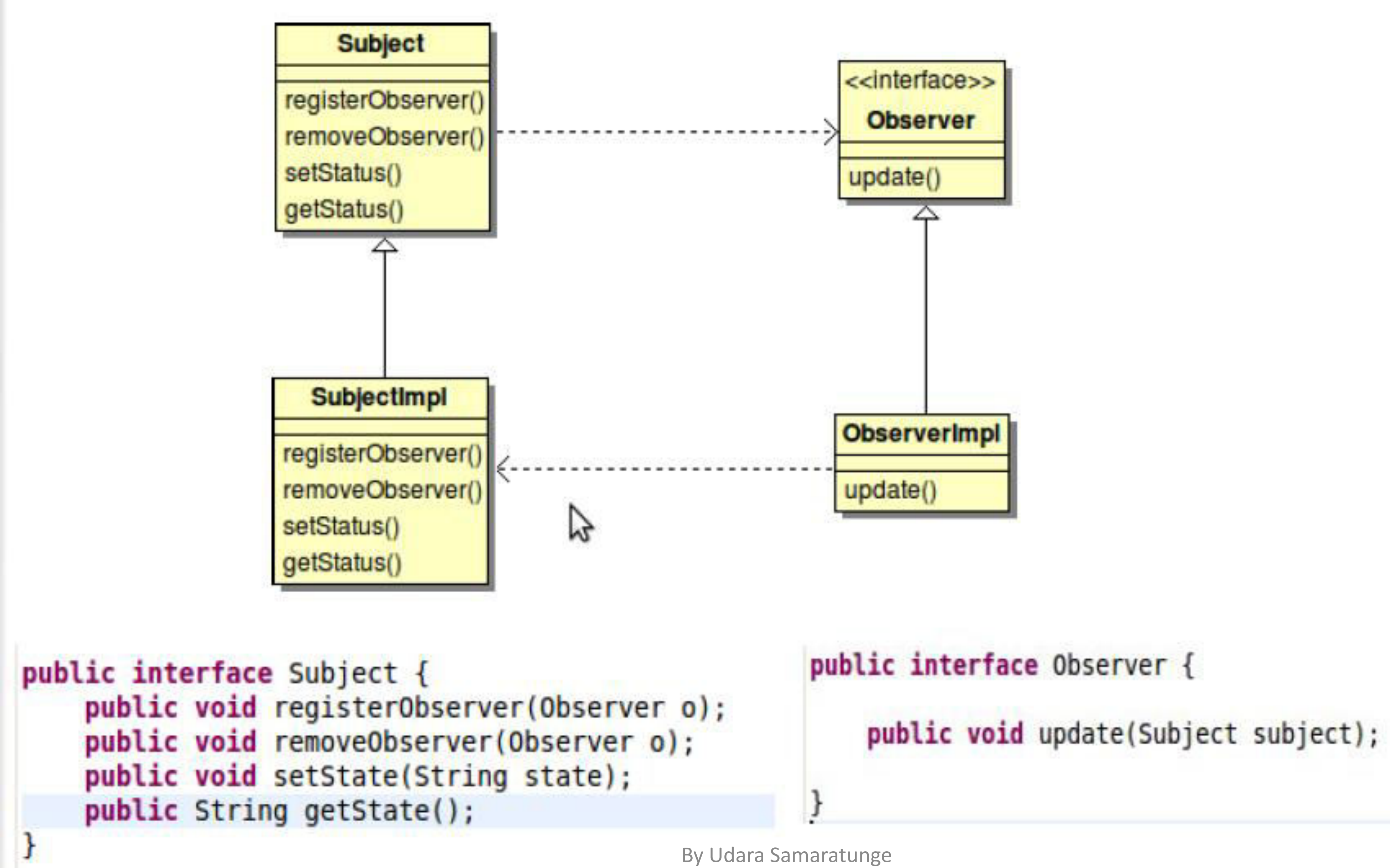


The Observer Pattern

- The subject and the observers are having a *one to many relationship*
- The observers are *dependent* on the subject
- When the subject state changes, the observers get notified
- In the observer pattern, (About the state)
  - Subject is the object that contains the state and it owns it
  - Observers use it but do not own it

The Observer Pattern - Explained

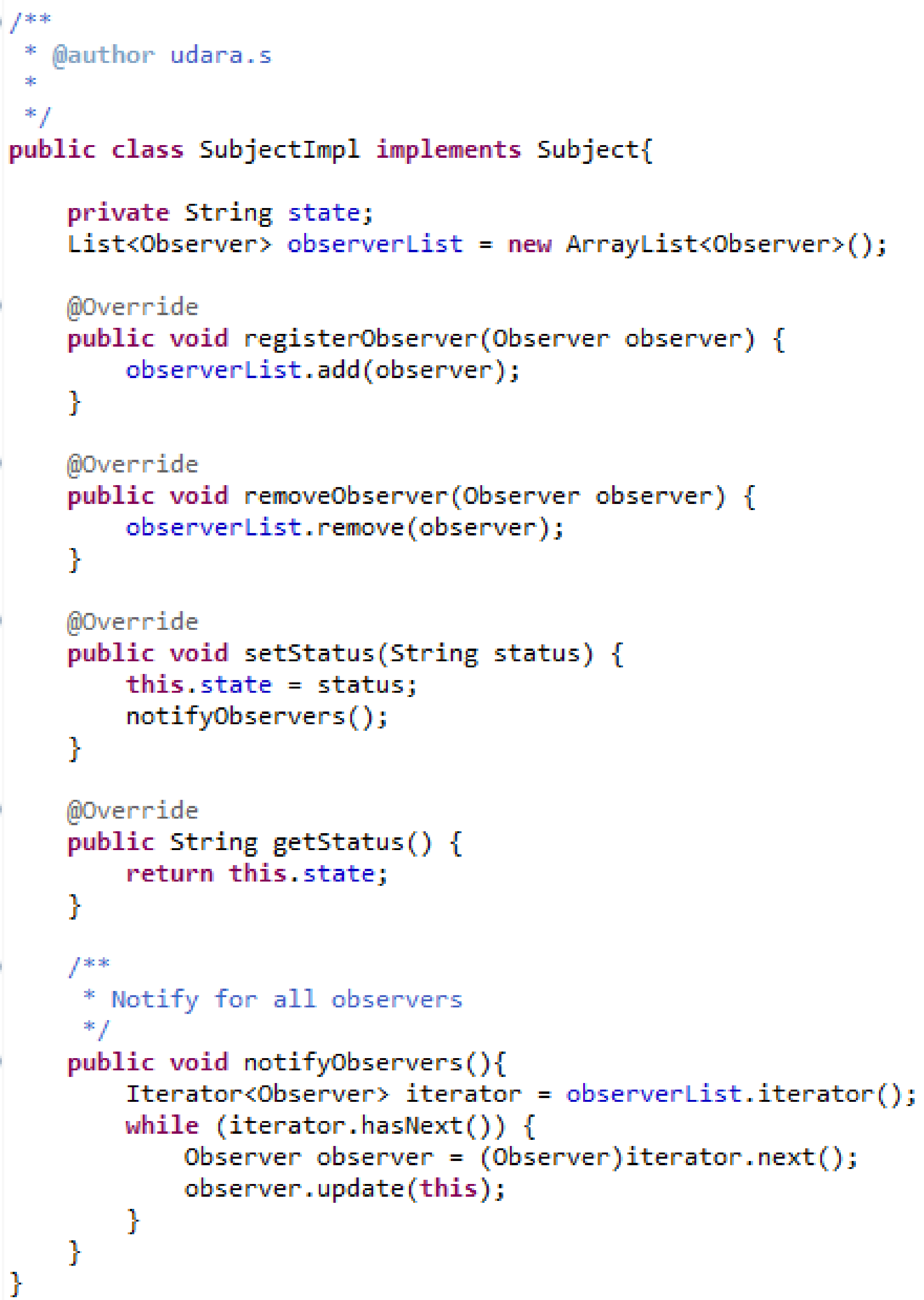
- **Subject**: Maintains a list of Observer references. Subject also provides an interface for attaching and detaching Observer objects.
- **Observer**: Defines an updating interface for objects that should be notified of changes in a subject.
- **ConcreteSubject**: Stores state of interest to ConcreteObserver objects and sends notifications to its observers upon state changes.
- **ConcreteObserver**: Maintains a reference to a ConcreteSubject object and a state that should stay consistent with the subject's.



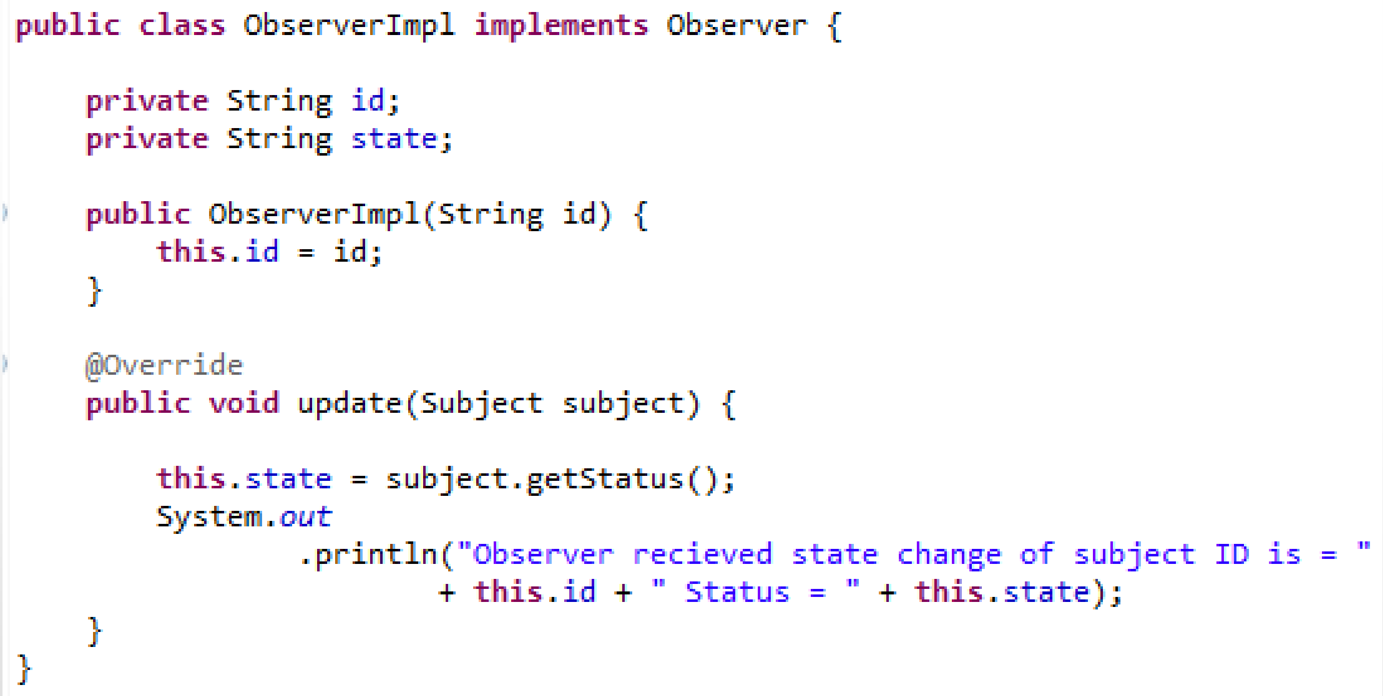
```
public interface Subject {
    public void registerObserver(Observer o);
    public void removeObserver(Observer o);
    public void setState(String state);
    public String getState();
}
```

```
public interface Observer {

    public void update(Subject subject);

}
```

Observer Pattern

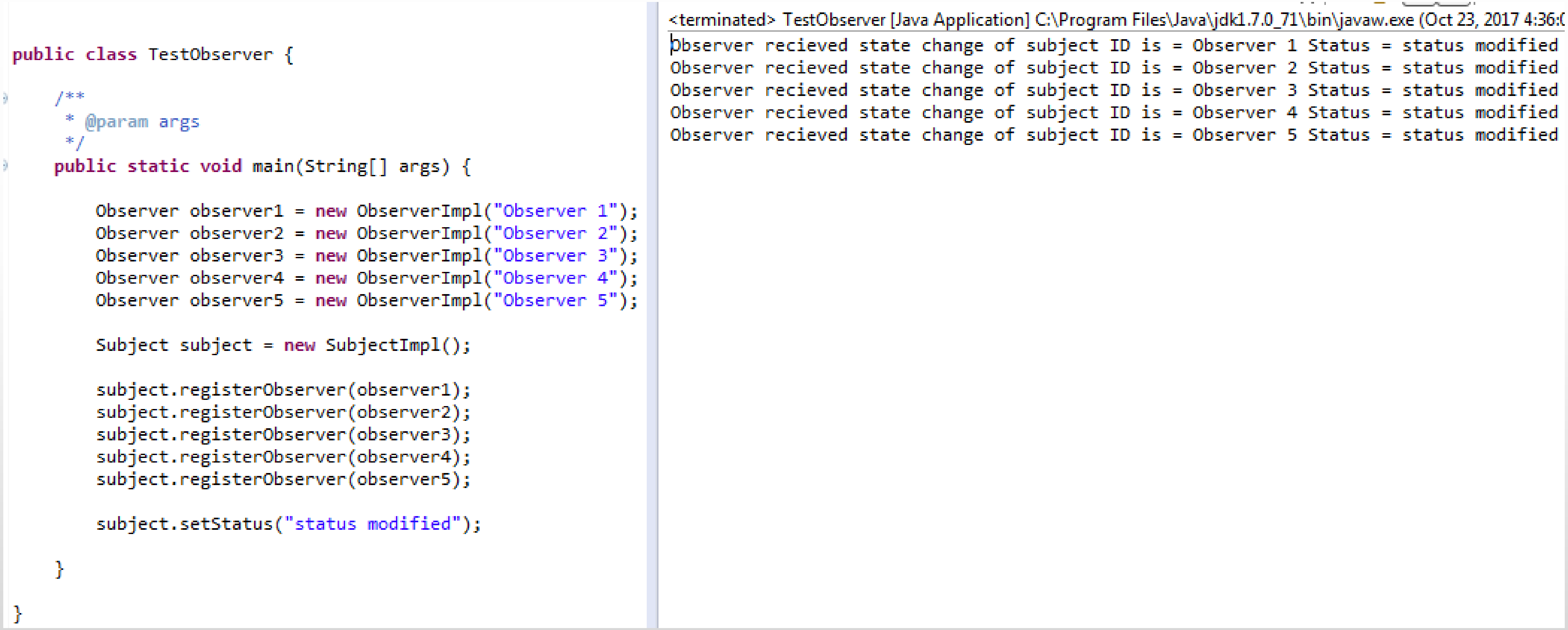


```
/**
 * @author udara.s
 *
 */
public class SubjectImpl implements Subject{

    private String state;
    List<Observer> observerList = new ArrayList<Observer>();

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

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

    @Override
    public void setStatus(String status) {
        this.state = status;
        notifyObservers();
    }

    @Override
    public String getStatus() {
        return this.state;
    }

    /**
     * Notify for all observers
     */
    public void notifyObservers(){
        Iterator<Observer> iterator = observerList.iterator();
        while (iterator.hasNext()) {
            Observer observer = (Observer)iterator.next();
            observer.update(this);
        }
    }
}
```



```
public class ObserverImpl implements Observer {

    private String id;
    private String state;

    public ObserverImpl(String id) {
        this.id = id;
    }

    @Override
    public void update(Subject subject) {

        this.state = subject.getStatus();
        System.out
                .println("Observer recieved state change of subject ID is = "
                        + this.id + " Status = " + this.state);
    }
}
```

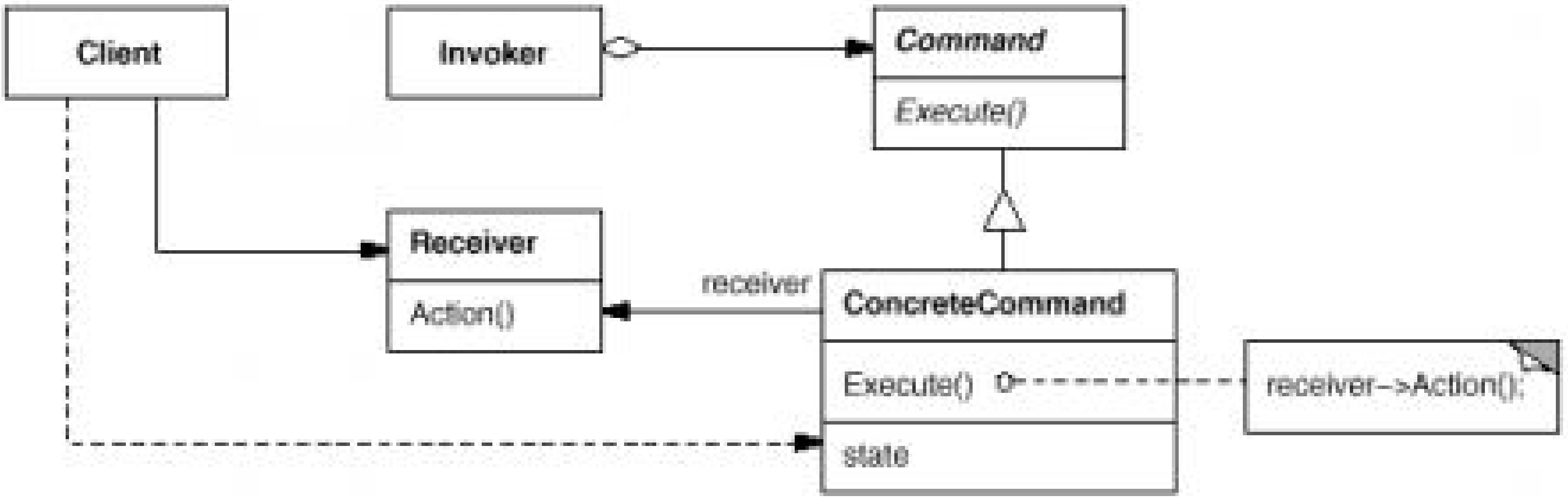
Observer Pattern Output



```
public class TestObserver {

    /**
     * @param args
     */
    public static void main(String[] args) {

        Observer observer1 = new ObserverImpl("Observer 1");
        Observer observer2 = new ObserverImpl("Observer 2");
        Observer observer3 = new ObserverImpl("Observer 3");
        Observer observer4 = new ObserverImpl("Observer 4");
        Observer observer5 = new ObserverImpl("Observer 5");

        Subject subject = new SubjectImpl();

        subject.registerObserver(observer1);
        subject.registerObserver(observer2);
        subject.registerObserver(observer3);
        subject.registerObserver(observer4);
        subject.registerObserver(observer5);

        subject.setStatus("status modified");

    }

}
```

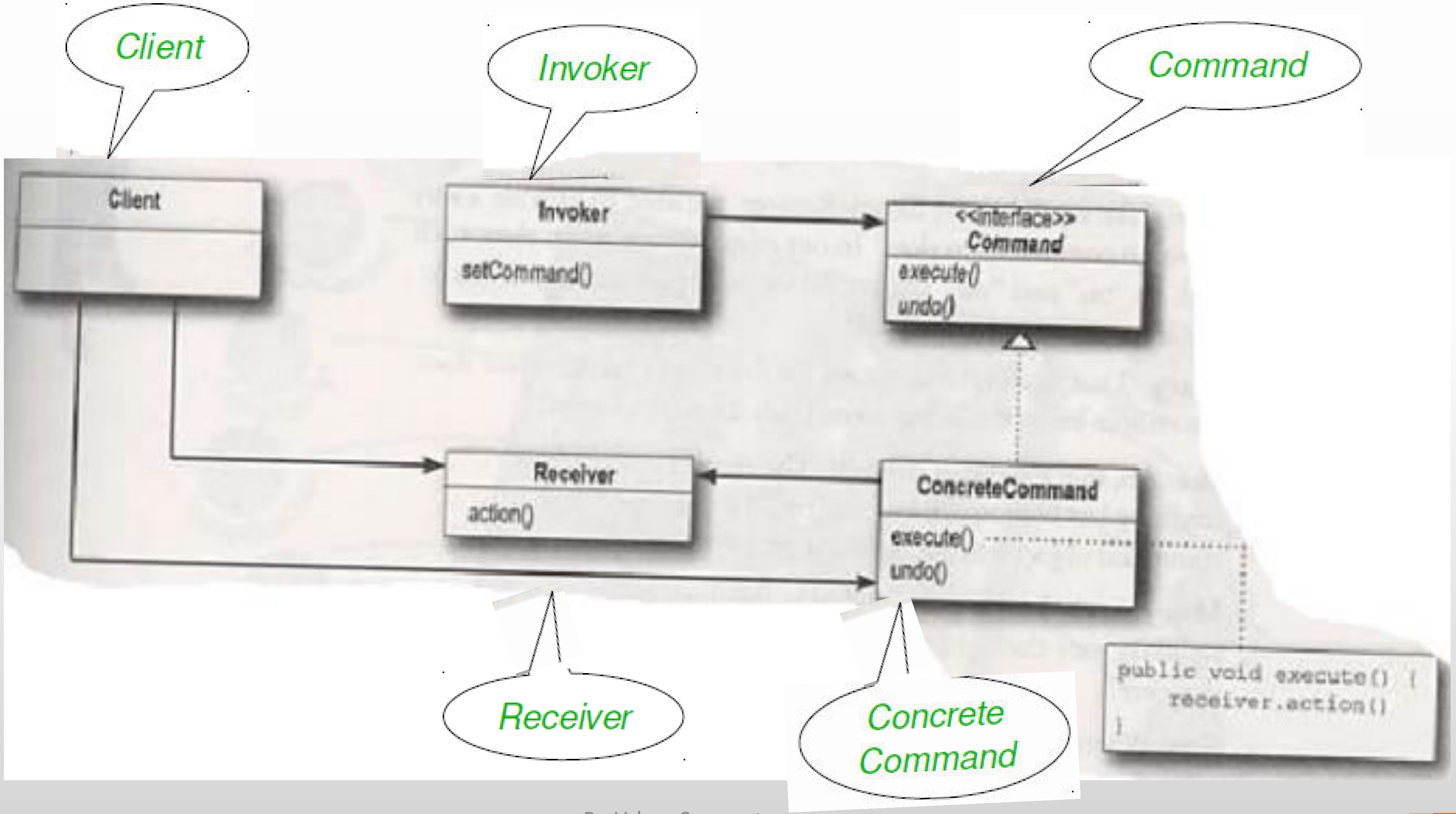
```
<terminated> TestObserver [Java Application] C:\Program Files\Java\jdk1.7.0_71\bin\javaw.exe (Oct 23, 2017 4:36:(
Observer recieved state change of subject ID is = Observer 1 Status = status modified
Observer recieved state change of subject ID is = Observer 2 Status = status modified
Observer recieved state change of subject ID is = Observer 3 Status = status modified
Observer recieved state change of subject ID is = Observer 4 Status = status modified
Observer recieved state change of subject ID is = Observer 5 Status = status modified
```

Command Pattern



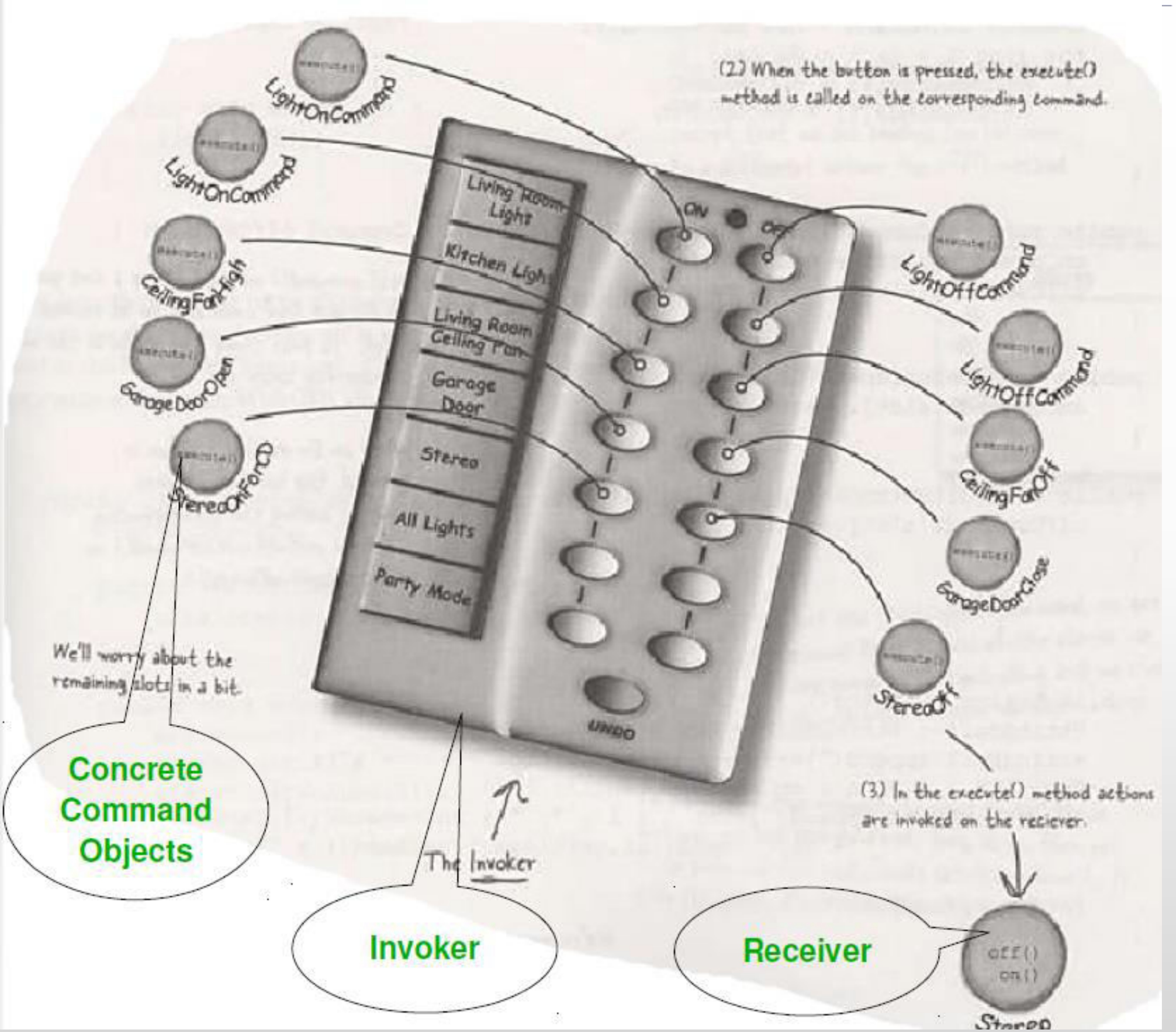
Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations

Command Pattern



```
public void execute() {
    receiver.action()
}
```

Command Pattern



(2) When the button is pressed, the execute() method is called on the corresponding command.

We'll worry about the remaining slots in a bit.

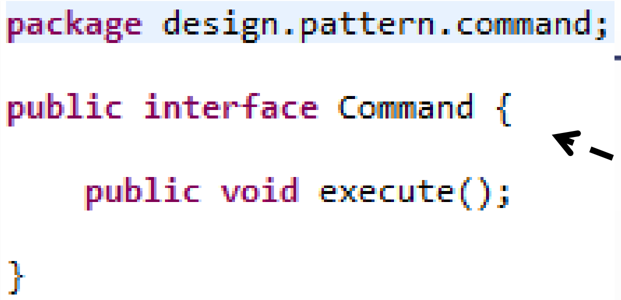
(3) In the execute() method actions are invoked on the reciever.

Concrete Command Objects

Invoker

Receiver

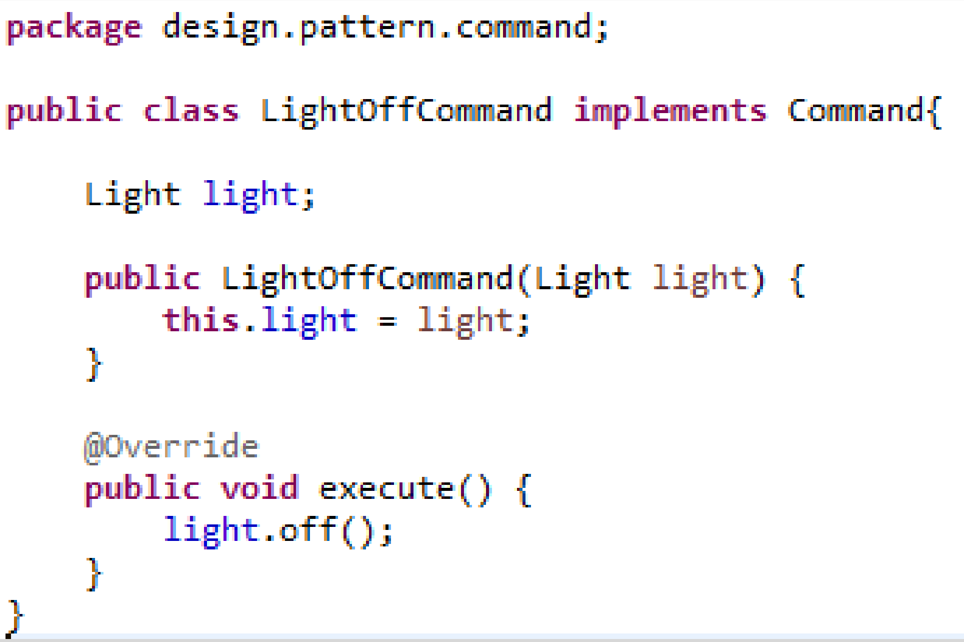
Command & Concrete Command



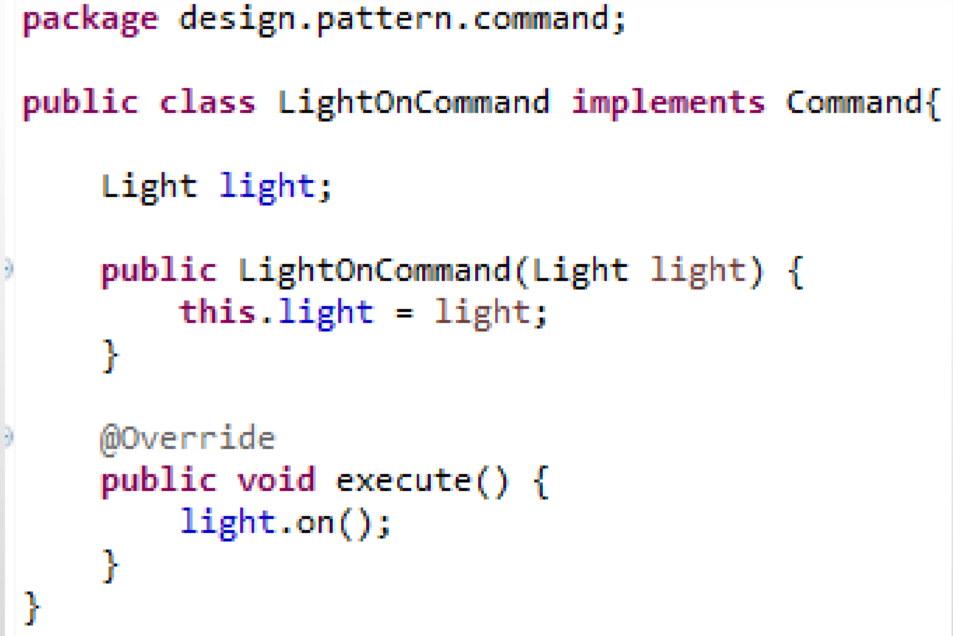
```java
package design.pattern.command;

public interface Command {

    public void execute();

}
```

<<implements>>

<<implements>>

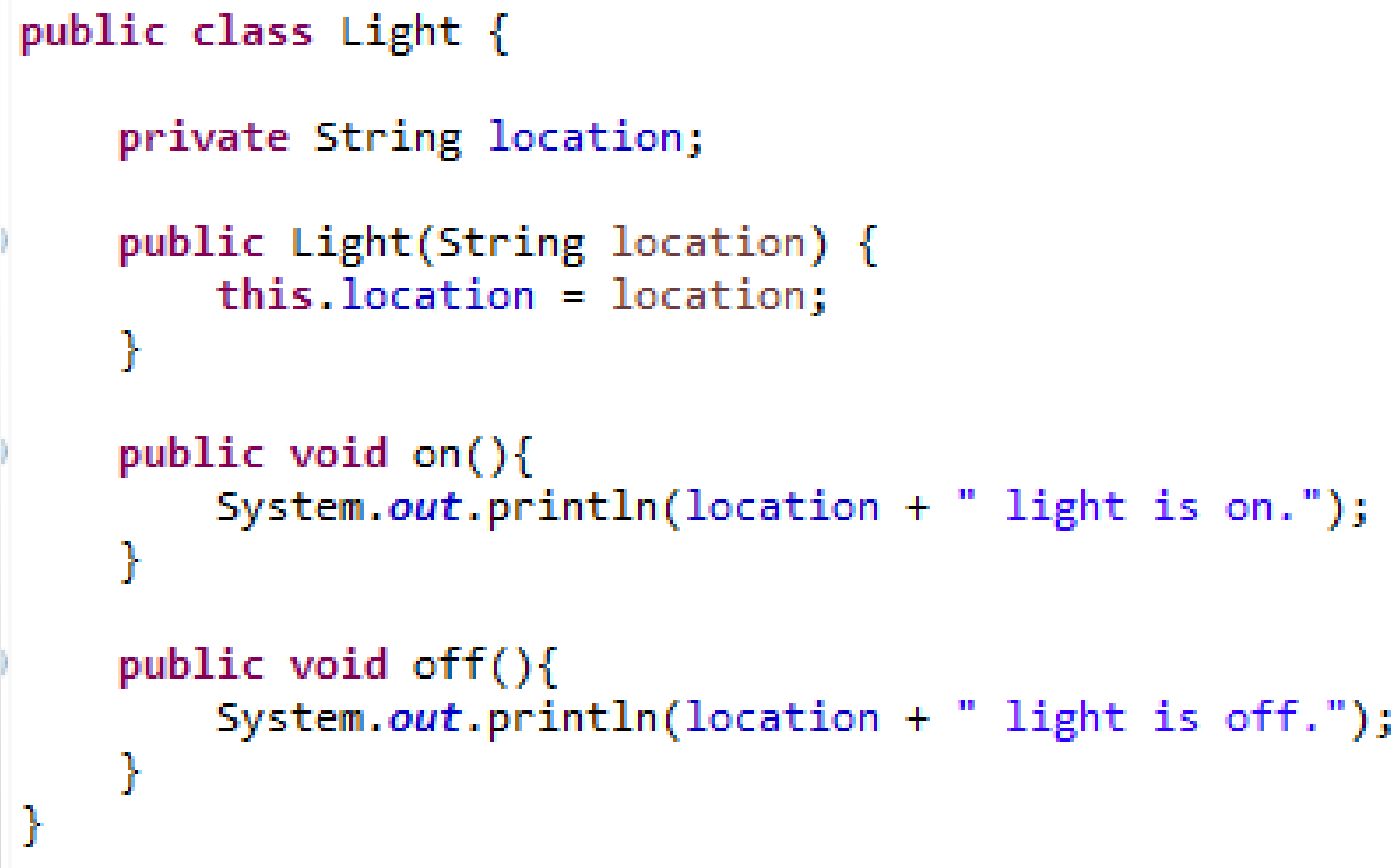


```java
package design.pattern.command;

public class LightOffCommand implements Command{

    Light light;

    public LightOffCommand(Light light) {
        this.light = light;
    }

    @Override
    public void execute() {
        light.off();
    }
}
```



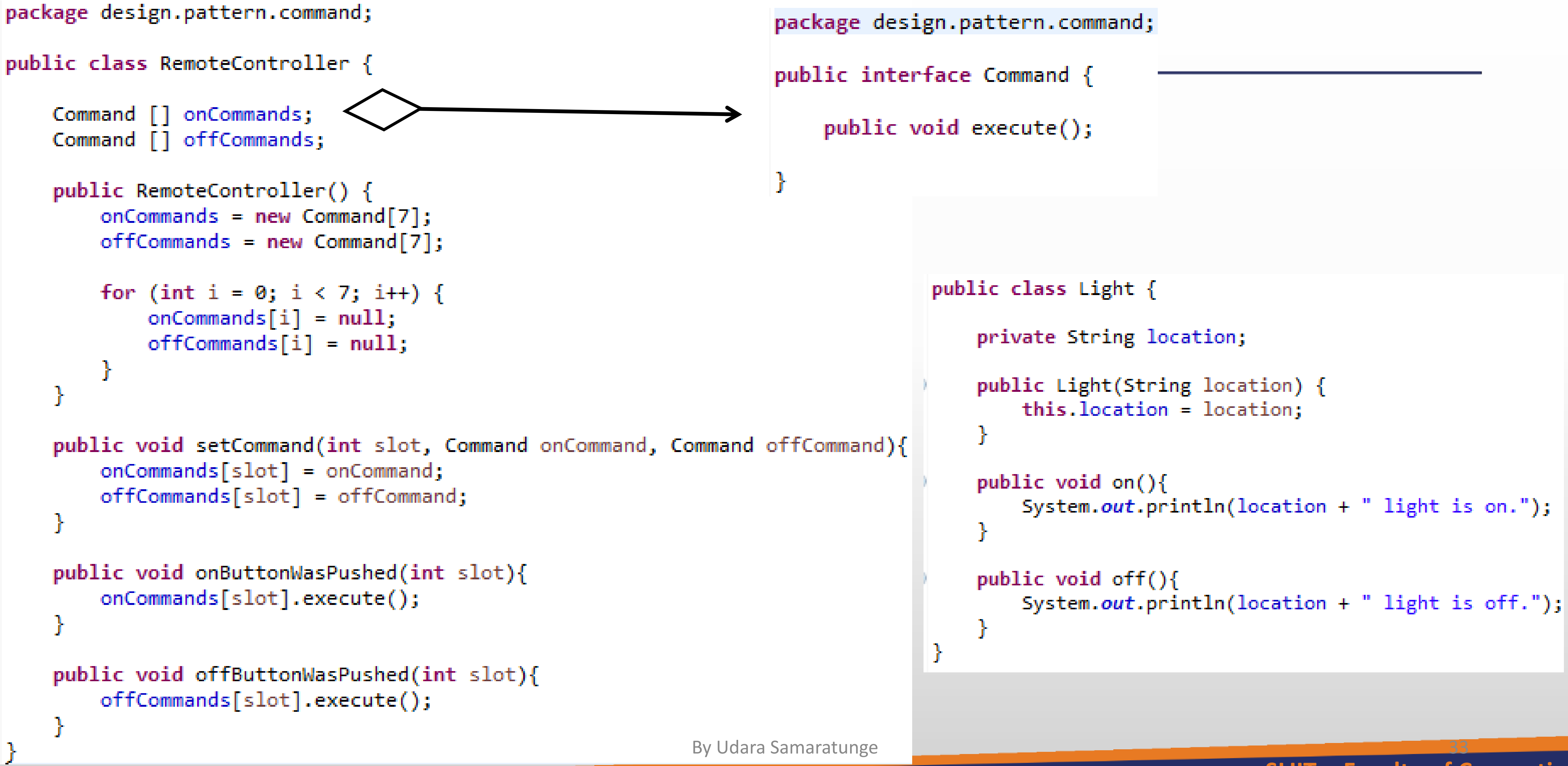
```java
package design.pattern.command;

public class LightOnCommand implements Command{

    Light light;

    public LightOnCommand(Light light) {
        this.light = light;
    }

    @Override
    public void execute() {
        light.on();
    }
}
```

Receiver



```
public class Light {

    private String location;

    public Light(String location) {
        this.location = location;
    }

    public void on(){
        System.out.println(location + " light is on.");
    }

    public void off(){
        System.out.println(location + " light is off.");
    }
}
```

Command Pattern

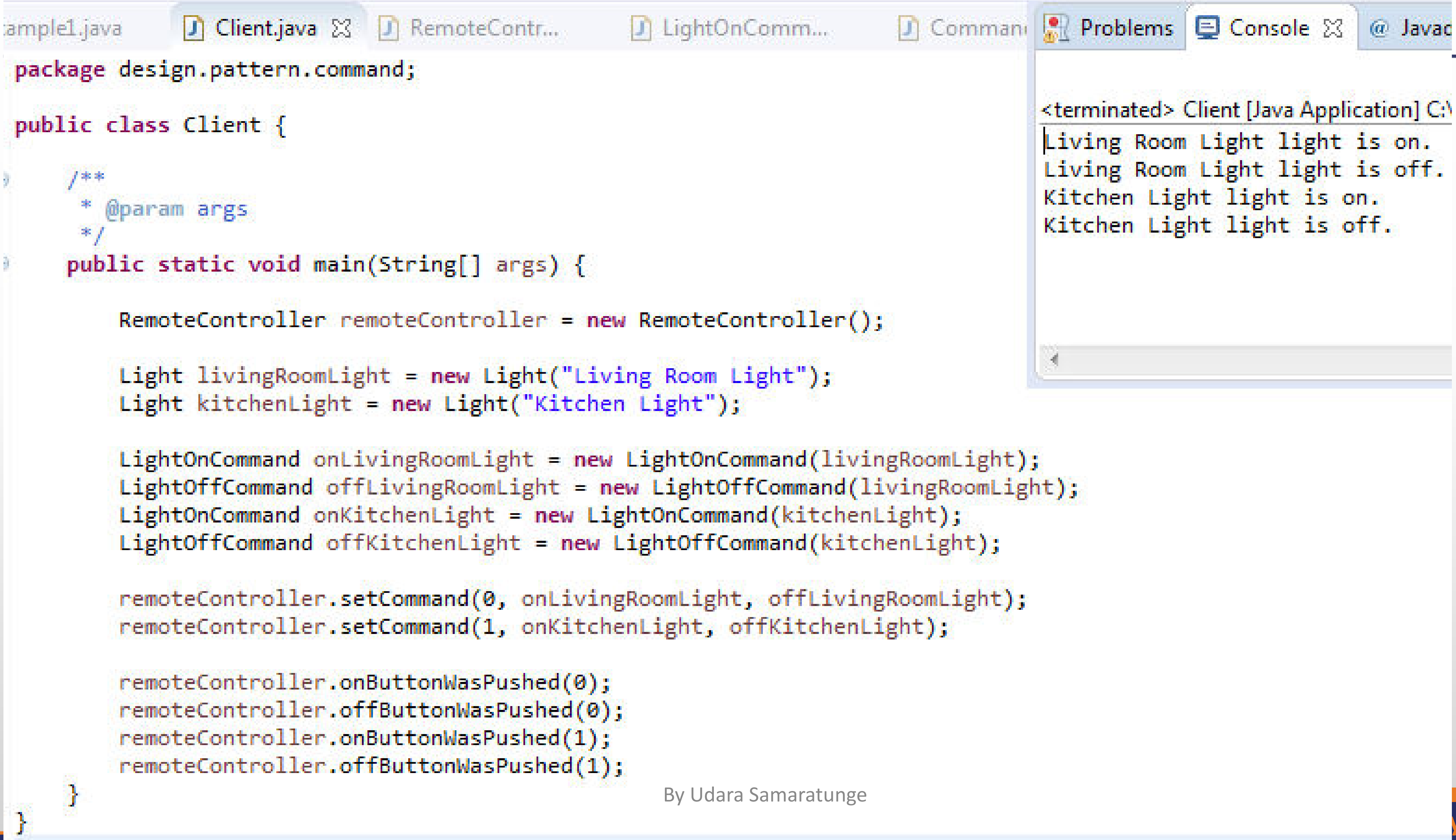


```java
package design.pattern.command;

public class RemoteController {

    Command [] onCommands;
    Command [] offCommands;

    public RemoteController() {
        onCommands = new Command[7];
        offCommands = new Command[7];

        for (int i = 0; i < 7; i++) {
            onCommands[i] = null;
            offCommands[i] = null;
        }
    }

    public void setCommand(int slot, Command onCommand, Command offCommand){
        onCommands[slot] = onCommand;
        offCommands[slot] = offCommand;
    }

    public void onButtonWasPushed(int slot){
        onCommands[slot].execute();
    }

    public void offButtonWasPushed(int slot){
        offCommands[slot].execute();
    }
}
```

```java
package design.pattern.command;

public interface Command {

    public void execute();

}
```

```java
public class Light {

    private String location;

    public Light(String location) {
        this.location = location;
    }

    public void on(){
        System.out.println(location + " light is on.");
    }

    public void off(){
        System.out.println(location + " light is off.");
    }
}
```

Invoker

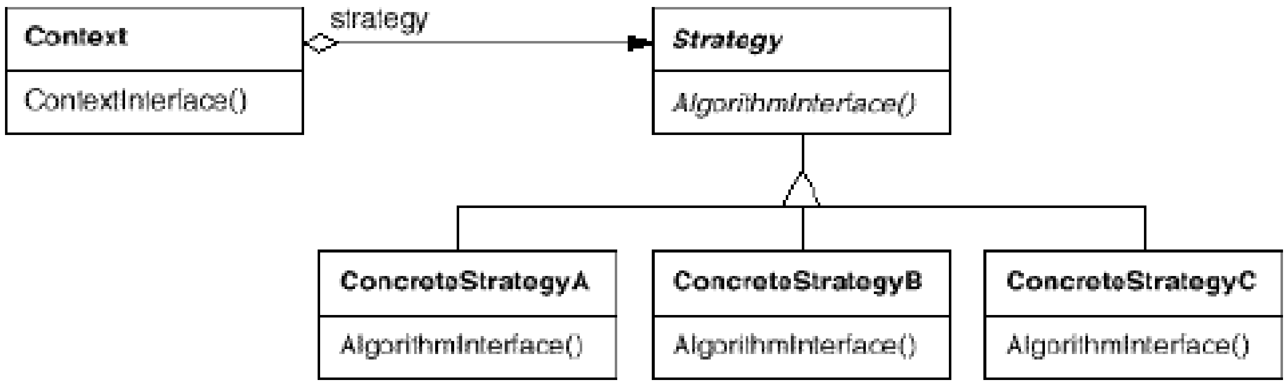


```
package design.pattern.command;

public class Client {

    /**
     * @param args
     */
    public static void main(String[] args) {

        RemoteController remoteController = new RemoteController();

        Light livingRoomLight = new Light("Living Room Light");
        Light kitchenLight = new Light("Kitchen Light");

        LightOnCommand onLivingRoomLight = new LightOnCommand(livingRoomLight);
        LightOffCommand offLivingRoomLight = new LightOffCommand(livingRoomLight);
        LightOnCommand onKitchenLight = new LightOnCommand(kitchenLight);
        LightOffCommand offKitchenLight = new LightOffCommand(kitchenLight);

        remoteController.setCommand(0, onLivingRoomLight, offLivingRoomLight);
        remoteController.setCommand(1, onKitchenLight, offKitchenLight);

        remoteController.onButtonWasPushed(0);
        remoteController.offButtonWasPushed(0);
        remoteController.onButtonWasPushed(1);
        remoteController.offButtonWasPushed(1);
    }
}
```

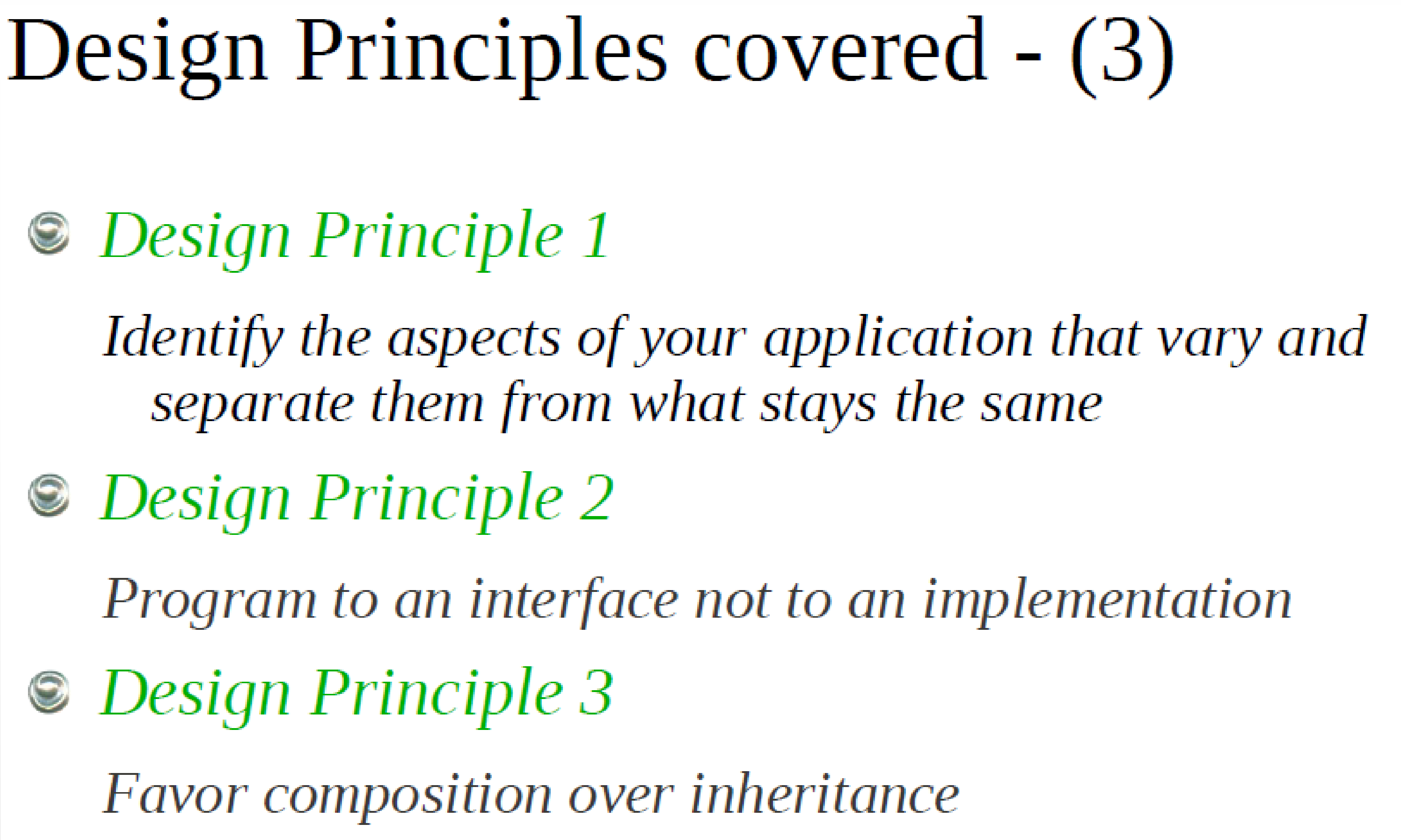
```
<terminated> Client [Java Application] C:\
Living Room Light light is on.
Living Room Light light is off.
Kitchen Light light is on.
Kitchen Light light is off.
```

Strategy Pattern

Strategy Pattern



Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.



Design Principles covered - (3)

- *Design Principle 1*

  Identify the aspects of your application that vary and separate them from what stays the same

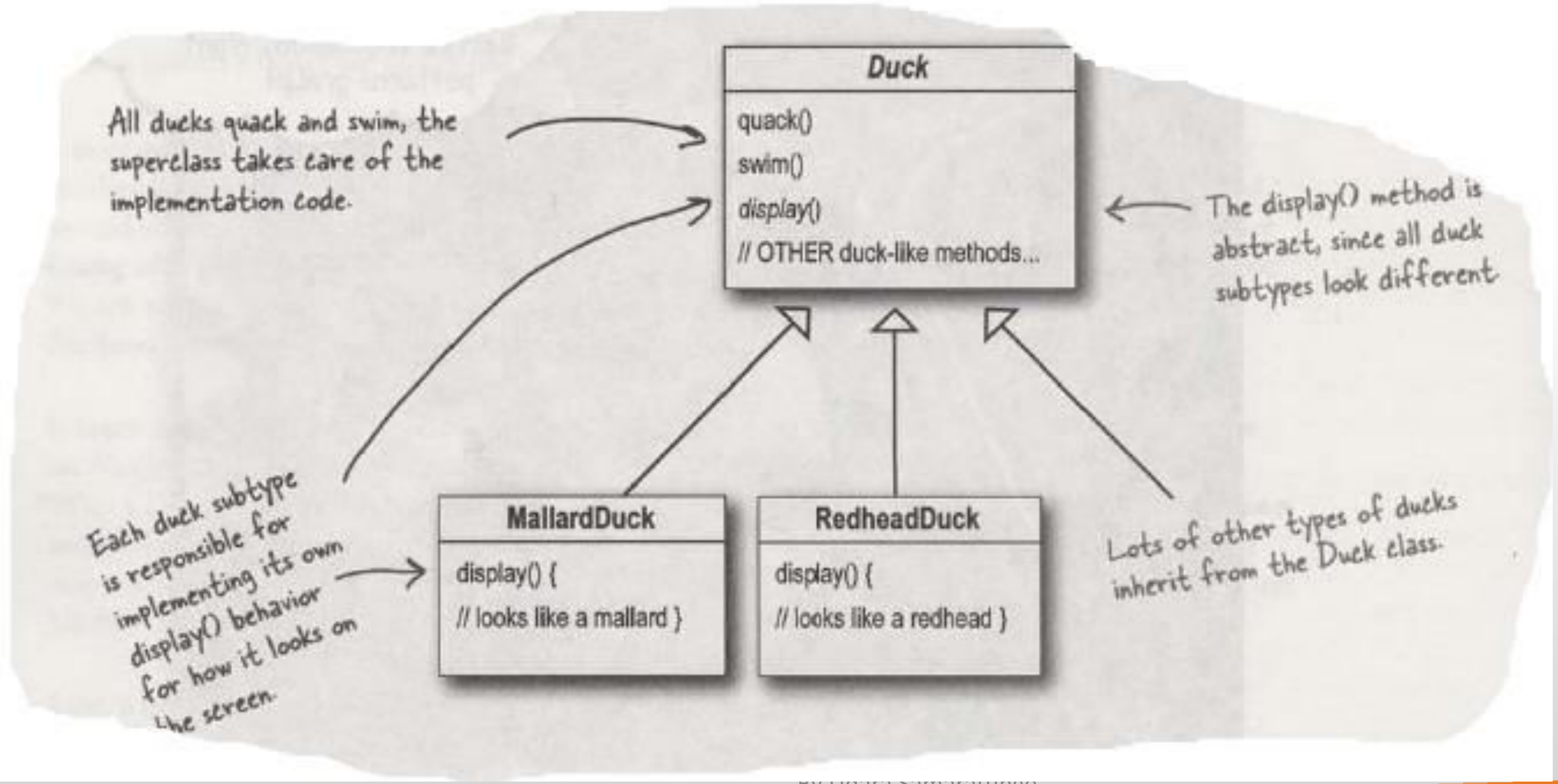
- *Design Principle 2*

  Program to an interface not to an implementation

- *Design Principle 3*

  Favor composition over inheritance

The Duck Simulation



Duck

quack()

swim()

display()

// OTHER duck-like methods...

MallardDuck

display() {

// looks like a mallard }

RedheadDuck

display() {

// looks like a redhead }

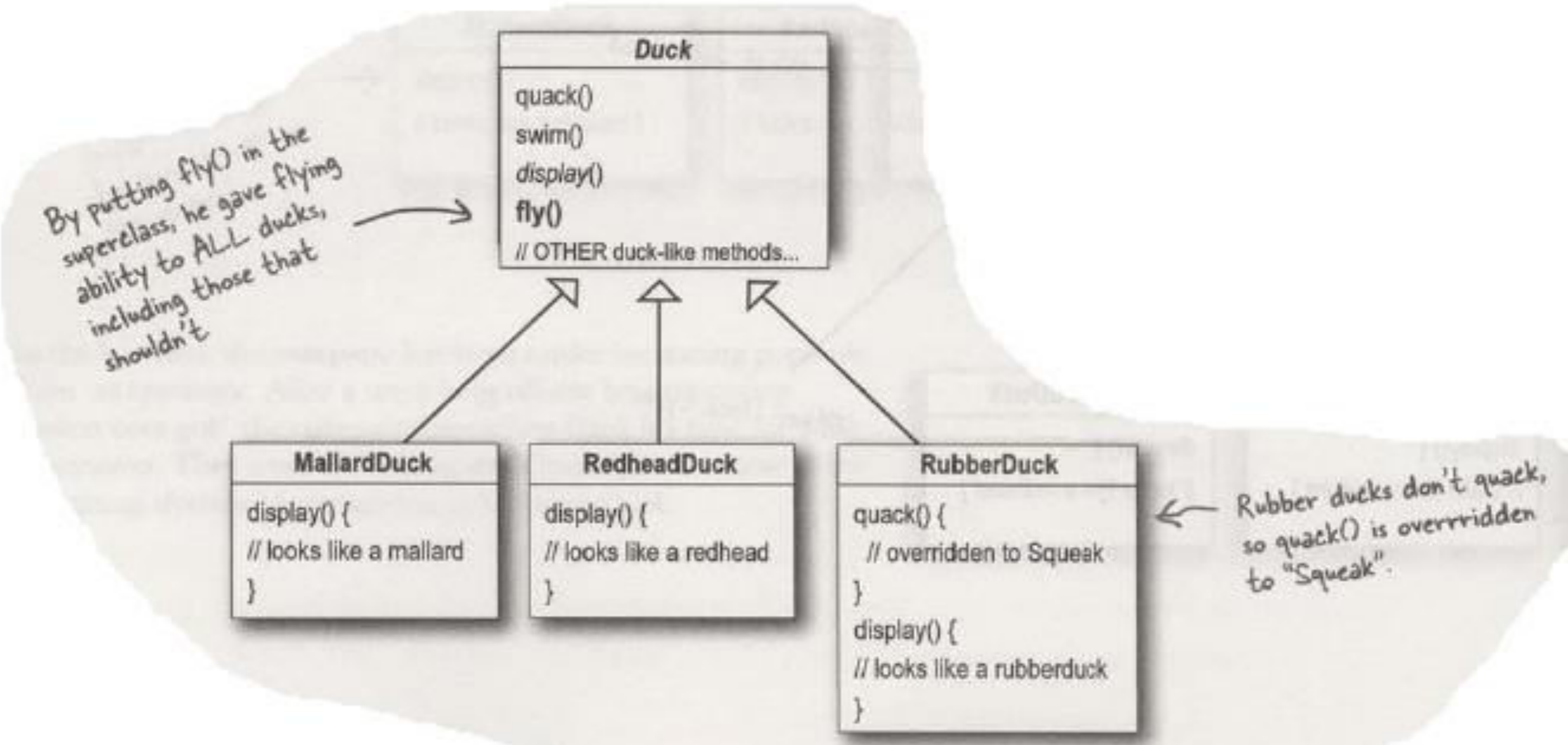
All ducks quack and swim, the superclass takes care of the implementation code.

The display() method is abstract, since all duck subtypes look different.

Each duck subtype is responsible for implementing its own display() behavior for how it looks on the screen.

Lots of other types of ducks inherit from the Duck class.

How about Inheritance?



Duck

```
quack()
swim()
display()
fly()
// OTHER duck-like methods...
```

MallardDuck

```
display() {
// looks like a mallard
}
```

RedheadDuck

```
display() {
// looks like a redhead
}
```

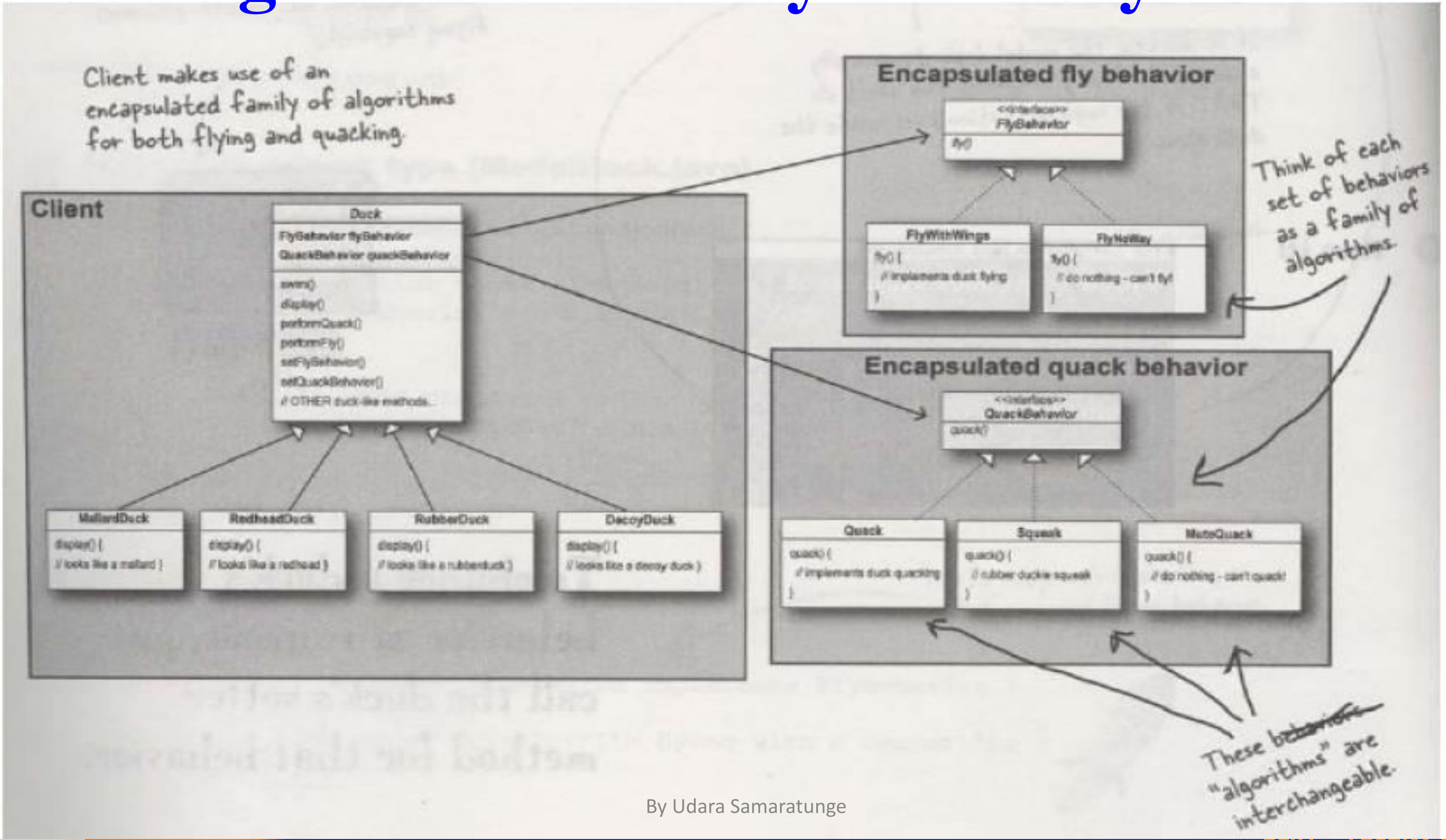
RubberDuck

```
quack() {
  // overridden to Squeak
}
display() {
// looks like a rubberduck
}
```

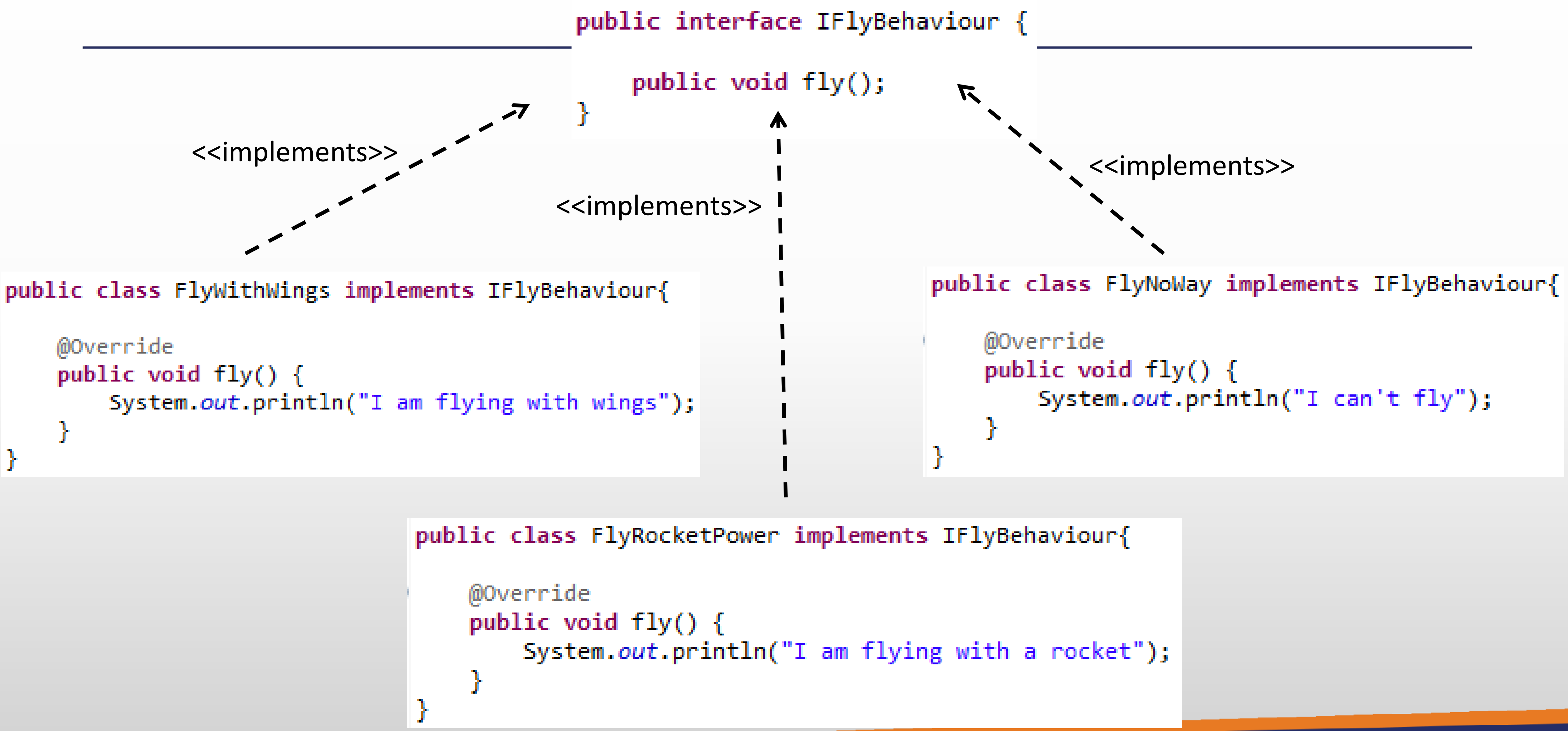
By putting fly() in the superclass, he gave flying ability to ALL ducks, including those that shouldn't

Rubber ducks don't quack, so quack() is overridden to "Squeak".

Setting the behavior dynamically


Client makes use of an encapsulated family of algorithms for both flying and quacking.
Client
Duck
FlyBehavior flyBehavior
QuackBehavior quackBehavior
swim()
display()
performQuack()
performFly()
setFlyBehavior()
setQuackBehavior()
// OTHER duck-like methods...
MallardDuck
display() {
// looks like a mallard }
RedheadDuck
display() {
// looks like a redhead }
RubberDuck
display() {
// looks like a rubberduck }
DecoyDuck
display() {
// looks like a decoy duck }
Encapsulated fly behavior
<<interface>>
FlyBehavior
fly()
FlyWithWings
fly() {
// implements duck flying
}
FlyNoWay
fly() {
// do nothing - can't fly!
}
Think of each set of behaviors as a family of algorithms.
Encapsulated quack behavior
<<interface>>
QuackBehavior
quack()
Quack
quack() {
// implements duck quacking
}
Squeak
quack() {
// rubber duckie squeak
}
MuteQuack
quack() {
// do nothing - can't quack!
}
These ~~behaviors~~ "algorithms" are interchangeable.

Strategy Pattern Implementation



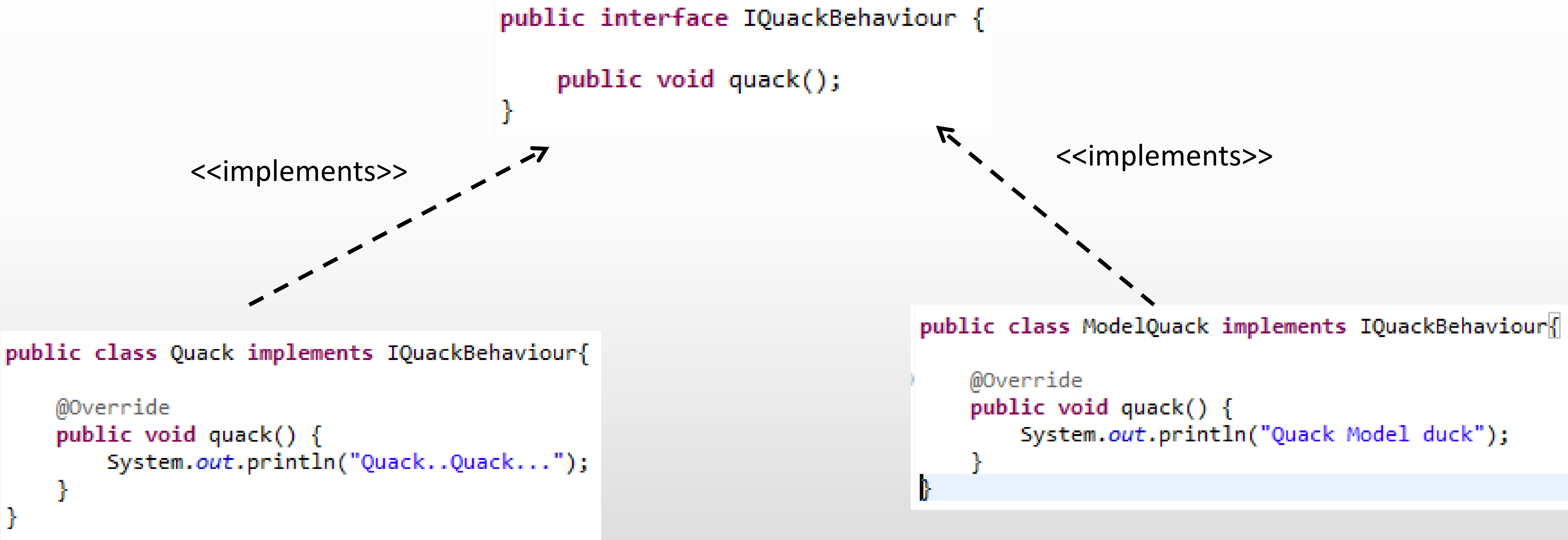
```java
public interface IFlyBehaviour {

    public void fly();
}
```

```java
public class FlyWithWings implements IFlyBehaviour{

    @Override
    public void fly() {
        System.out.println("I am flying with wings");
    }
}
```

```java
public class FlyNoWay implements IFlyBehaviour{

    @Override
    public void fly() {
        System.out.println("I can't fly");
    }
}
```

```java
public class FlyRocketPower implements IFlyBehaviour{

    @Override
    public void fly() {
        System.out.println("I am flying with a rocket");
    }
}
```

Strategy Pattern Implementation



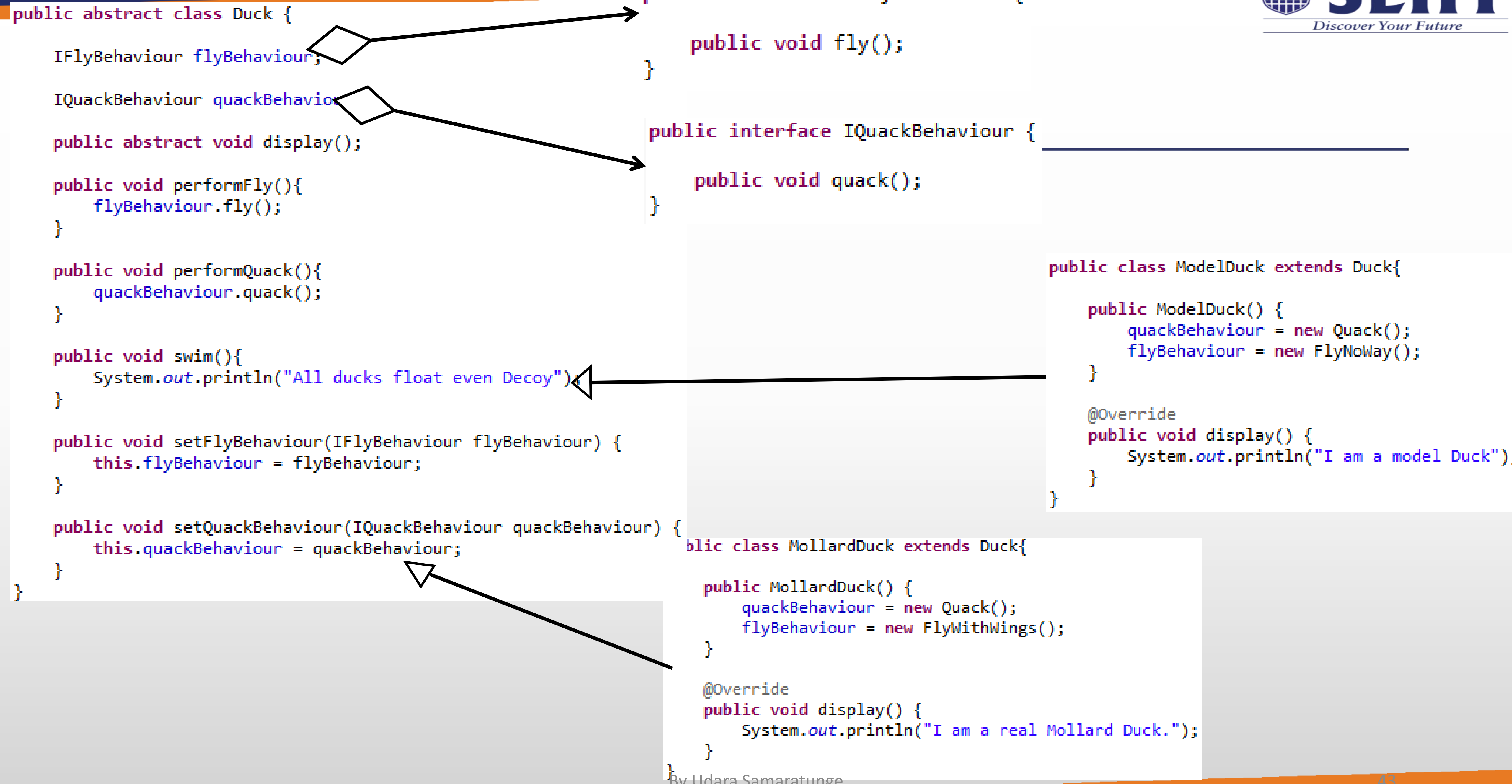
```
public interface IQuackBehaviour {

    public void quack();
}
```

<<implements>>

<<implements>>

```
public class Quack implements IQuackBehaviour{

    @Override
    public void quack() {
        System.out.println("Quack..Quack...");
    }
}
```

```
public class ModelQuack implements IQuackBehaviour{

    @Override
    public void quack() {
        System.out.println("Quack Model duck");
    }
}
```



```
public abstract class Duck {

    IFlyBehaviour flyBehaviour;

    IQuackBehaviour quackBehavio

    public abstract void display();

    public void performFly(){
        flyBehaviour.fly();
    }

    public void performQuack(){
        quackBehaviour.quack();
    }

    public void swim(){
        System.out.println("All ducks float even Decoy");
    }

    public void setFlyBehaviour(IFlyBehaviour flyBehaviour) {
        this.flyBehaviour = flyBehaviour;
    }

    public void setQuackBehaviour(IQuackBehaviour quackBehaviour) {
        this.quackBehaviour = quackBehaviour;
    }
}
```

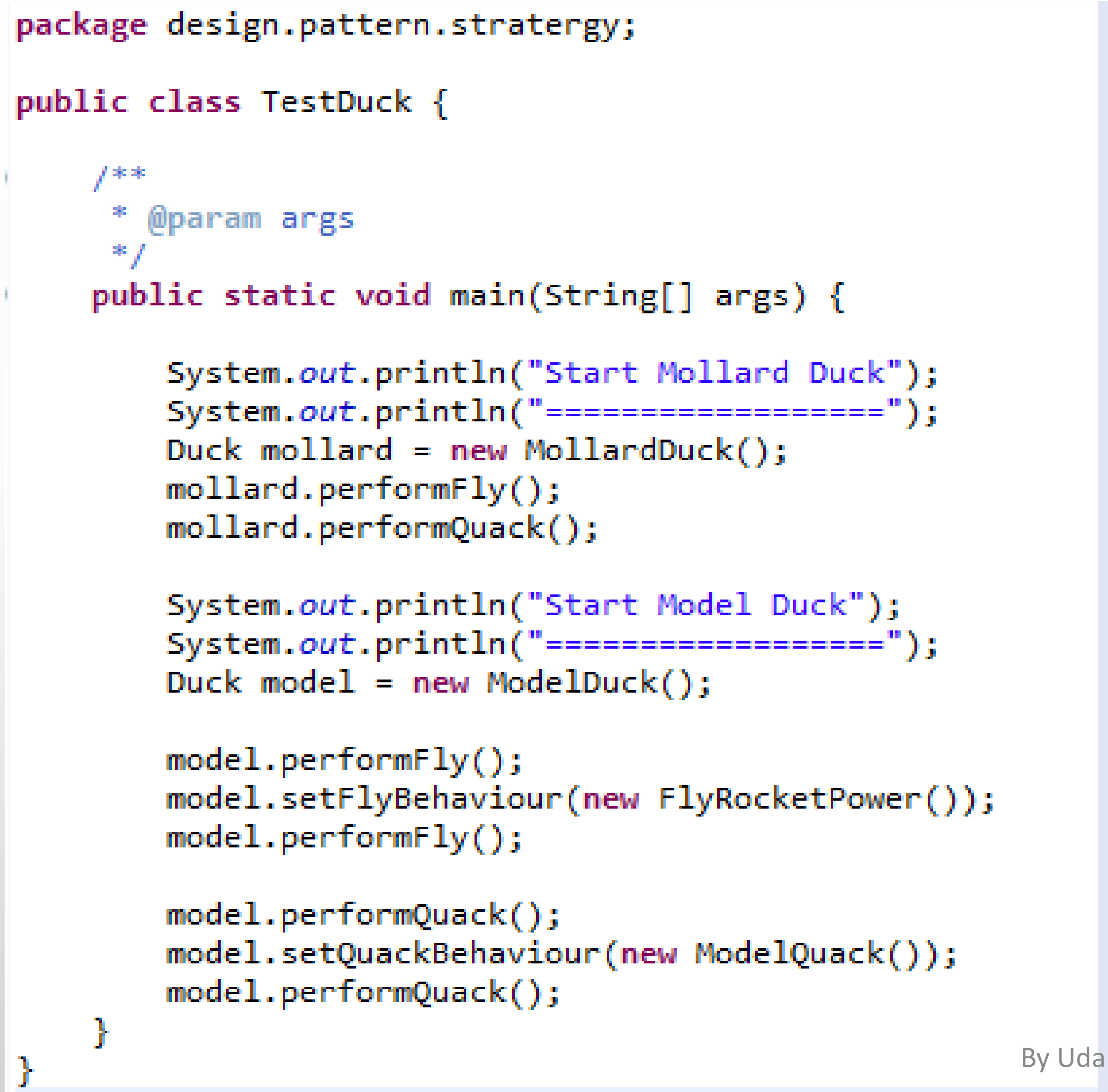
```
public interface IFlyBehaviour {

    public void fly();
}
```

```
public interface IQuackBehaviour {

    public void quack();
}
```

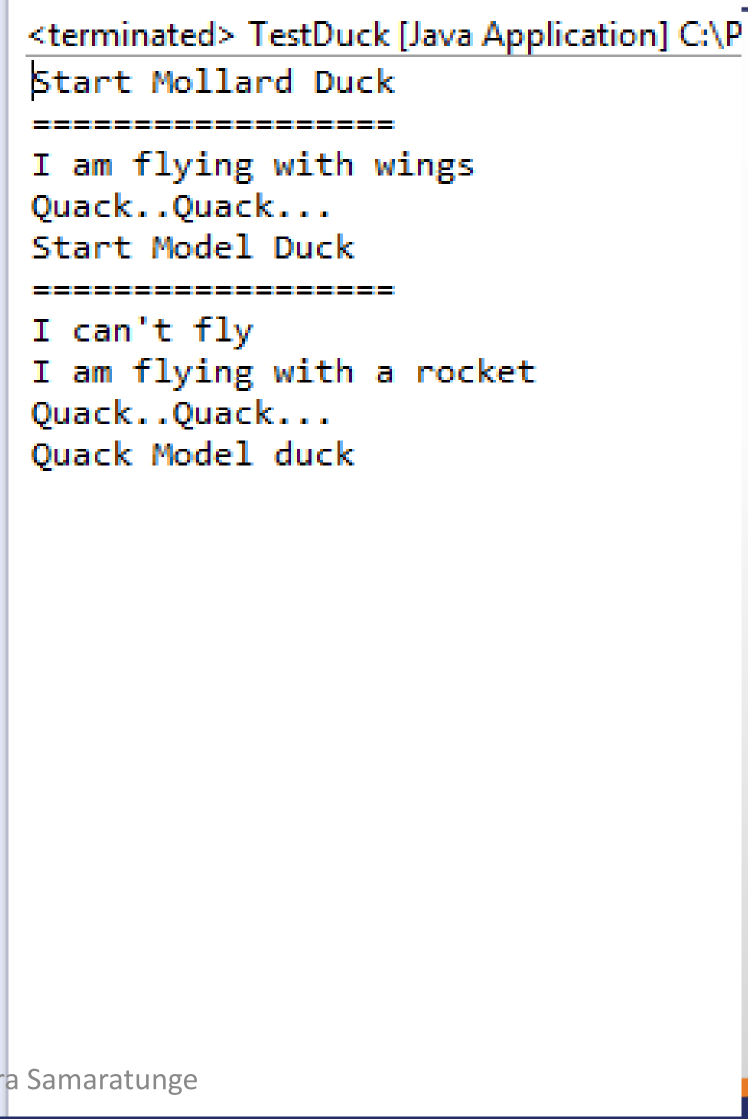
```
public class ModelDuck extends Duck{

    public ModelDuck() {
        quackBehaviour = new Quack();
        flyBehaviour = new FlyNoWay();
    }

    @Override
    public void display() {
        System.out.println("I am a model Duck")
    }
}
```

```
blic class MollardDuck extends Duck{

    public MollardDuck() {
        quackBehaviour = new Quack();
        flyBehaviour = new FlyWithWings();
    }

    @Override
    public void display() {
        System.out.println("I am a real Mollard Duck.");
    }
}
```

Strategy Pattern Implementation



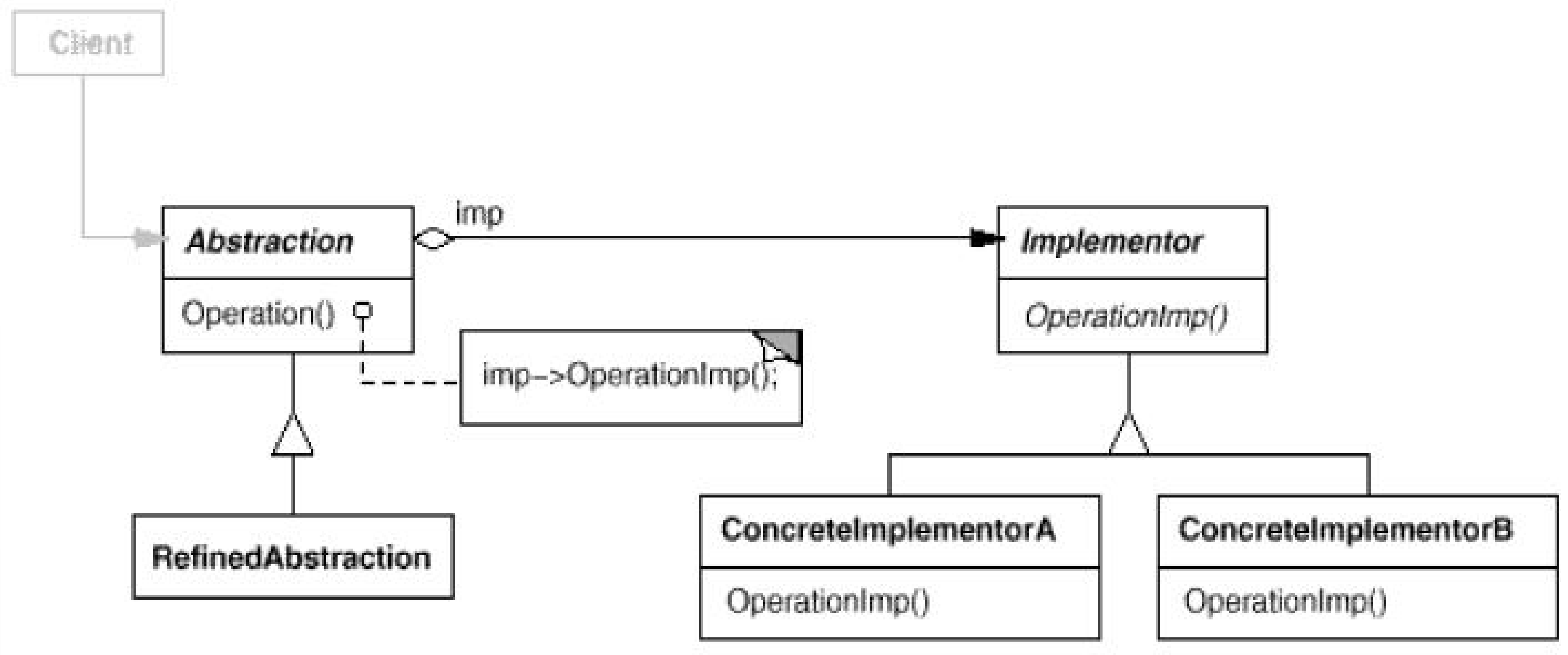
```
package design.pattern.stratergy;

public class TestDuck {

    /**
     * @param args
     */
    public static void main(String[] args) {

        System.out.println("Start Mollard Duck");
        System.out.println("==================");
        Duck mollard = new MollardDuck();
        mollard.performFly();
        mollard.performQuack();

        System.out.println("Start Model Duck");
        System.out.println("==================");
        Duck model = new ModelDuck();

        model.performFly();
        model.setFlyBehaviour(new FlyRocketPower());
        model.performFly();

        model.performQuack();
        model.setQuackBehaviour(new ModelQuack());
        model.performQuack();
    }
}
```



```
<terminated> TestDuck [Java Application] C:\P
Start Mollard Duck
==================
I am flying with wings
Quack..Quack...
Start Model Duck
==================
I can't fly
I am flying with a rocket
Quack..Quack...
Quack Model duck
```

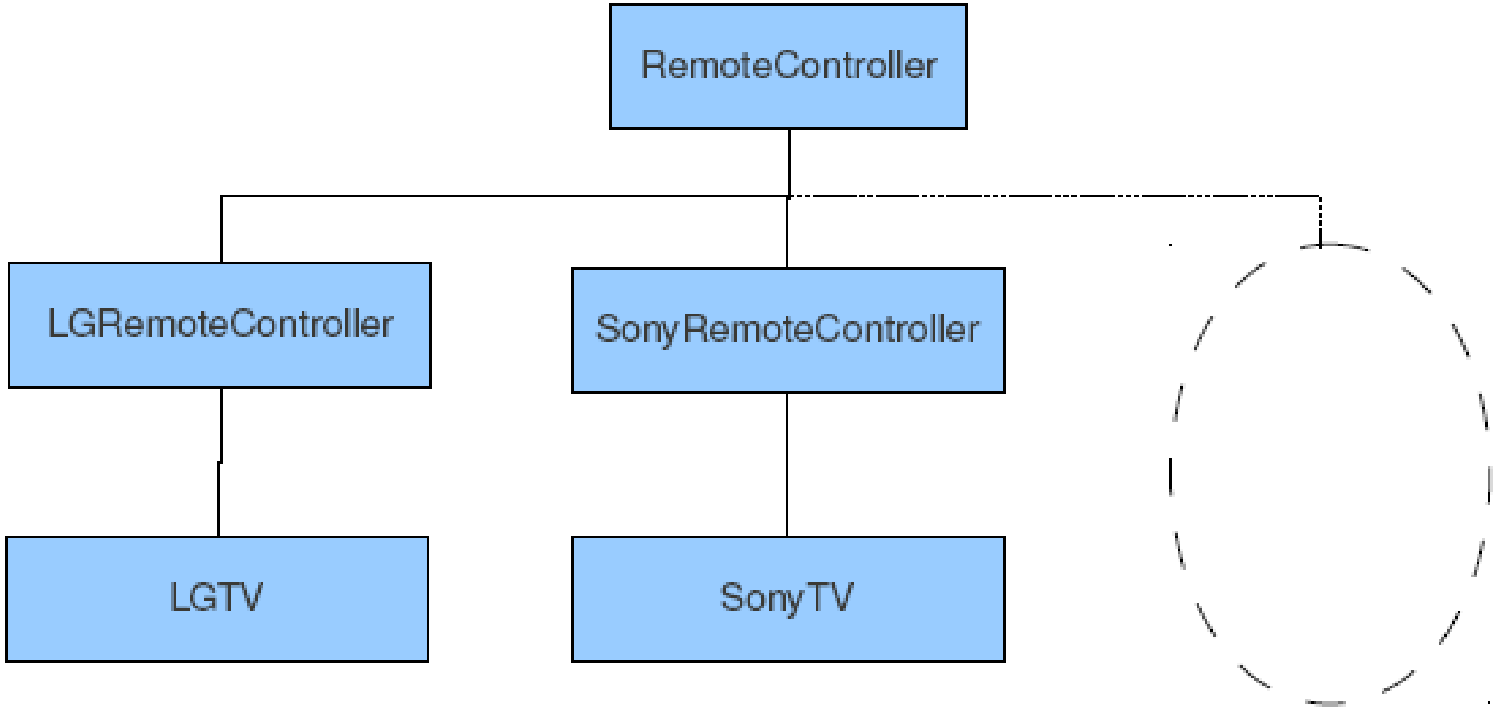
Bridge Pattern

Bridge Pattern

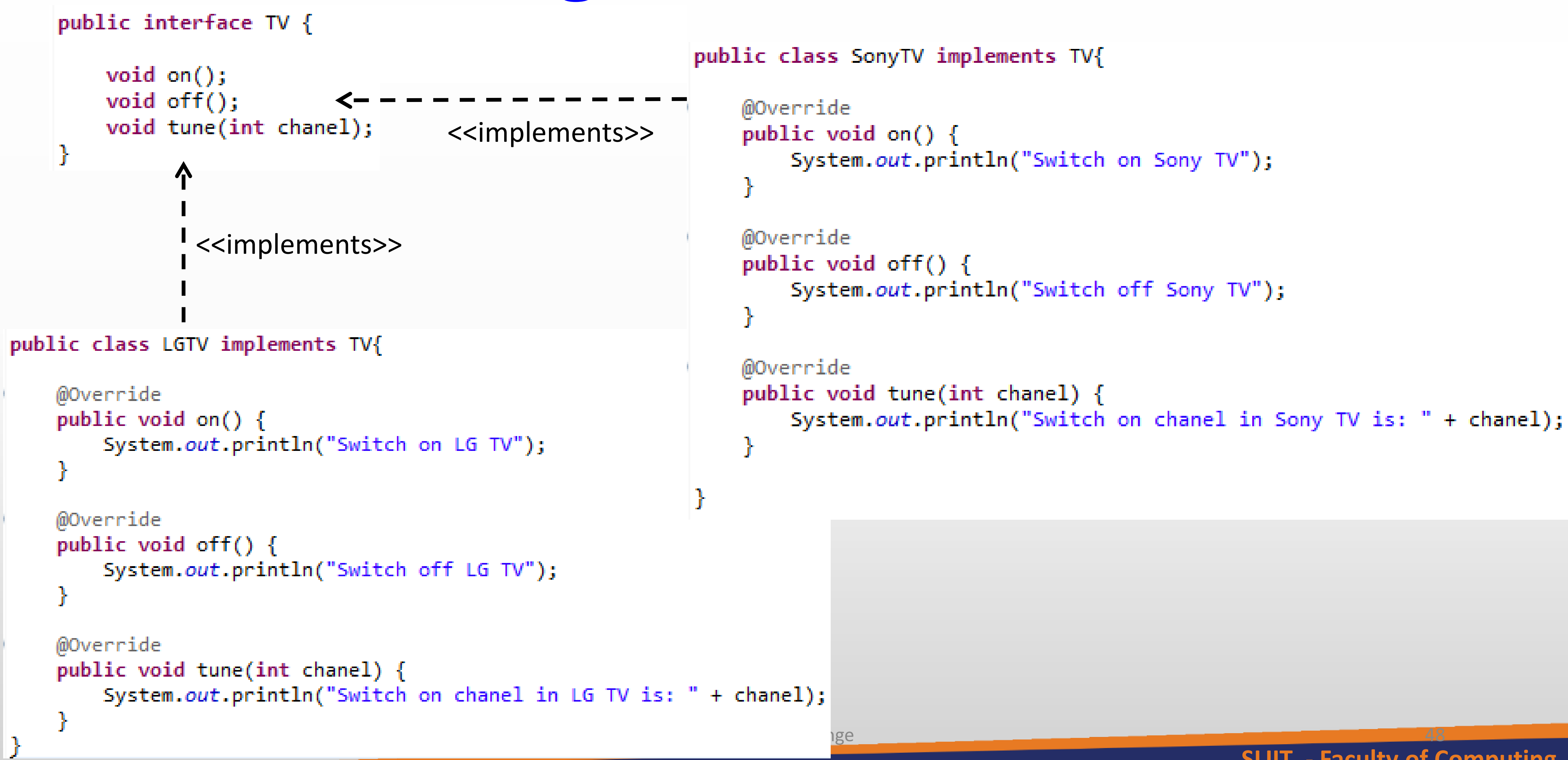


Example for Bridge Pattern

- There are two brands of TVs (Sony and LG) in your living room. So there are two remote controllers for each one. (See below diagram) Just assume a single remote controller can be used to **switch on, switch off** and **tune channels** of both TVs. Think of a design pattern that can solve this.



Bridge Pattern

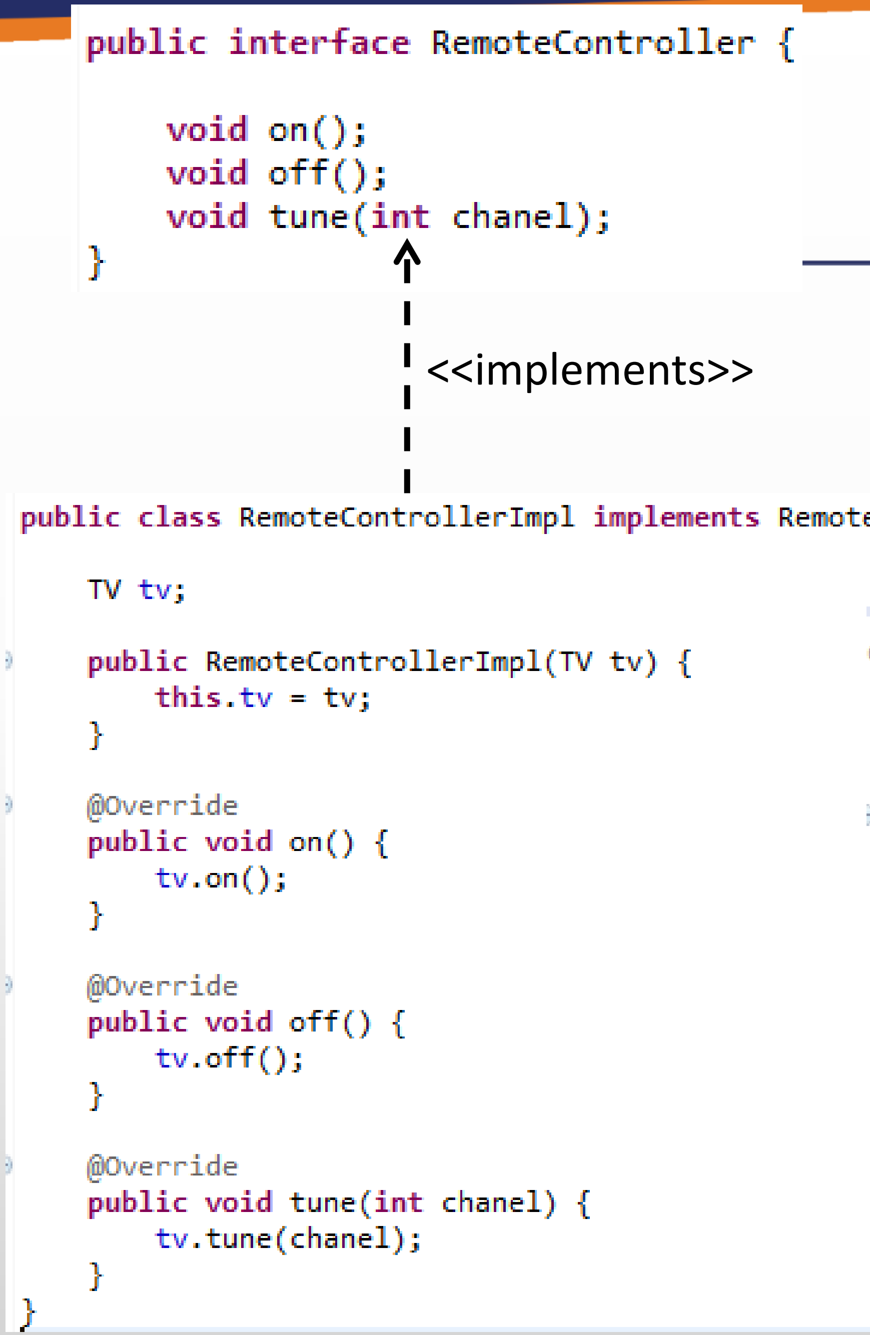


```
public interface TV {

    void on();
    void off();
    void tune(int chanel);
}
```

```
public class SonyTV implements TV{

    @Override
    public void on() {
        System.out.println("Switch on Sony TV");
    }

    @Override
    public void off() {
        System.out.println("Switch off Sony TV");
    }

    @Override
    public void tune(int chanel) {
        System.out.println("Switch on chanel in Sony TV is: " + chanel);
    }

}
```

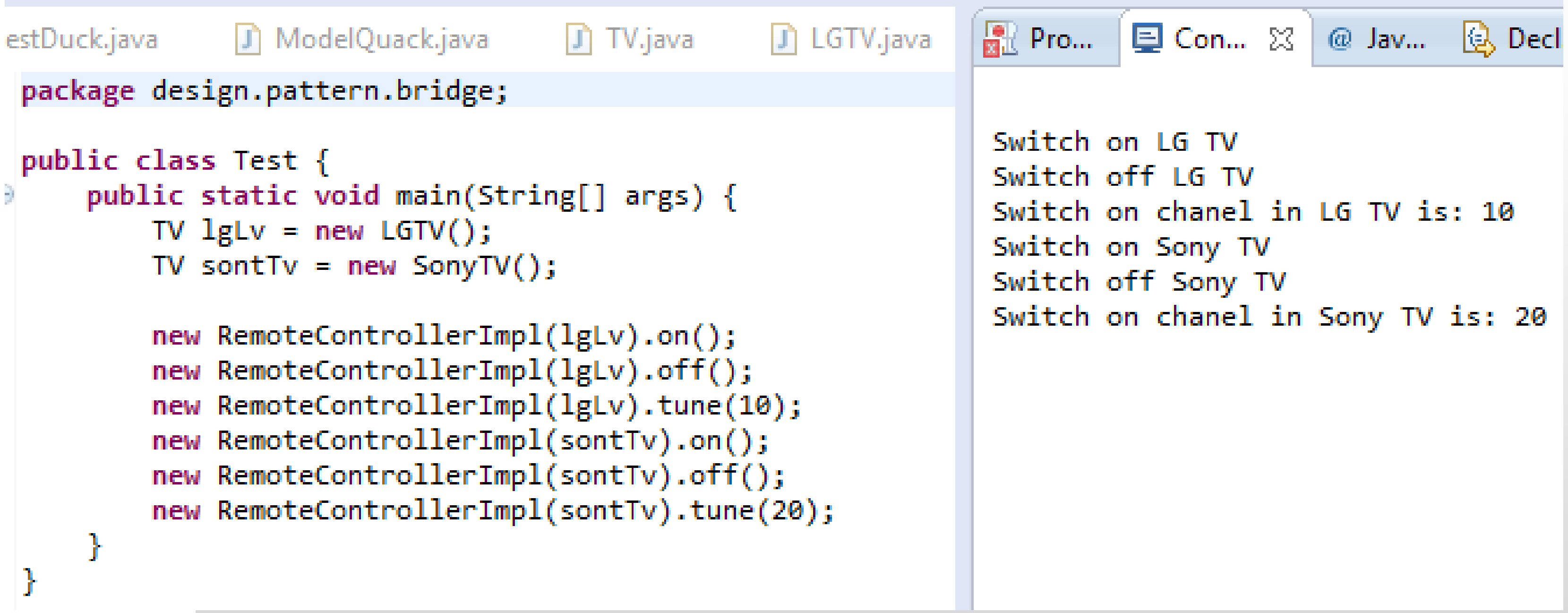
```
public class LGTV implements TV{

    @Override
    public void on() {
        System.out.println("Switch on LG TV");
    }

    @Override
    public void off() {
        System.out.println("Switch off LG TV");
    }

    @Override
    public void tune(int chanel) {
        System.out.println("Switch on chanel in LG TV is: " + chanel);
    }
}
```

Bridge Pattern



```java
public interface RemoteController {

    void on();
    void off();
    void tune(int chanel);
}
```

```java
public class RemoteControllerImpl implements RemoteController{

    TV tv;

    public RemoteControllerImpl(TV tv) {
        this.tv = tv;
    }

    @Override
    public void on() {
        tv.on();
    }

    @Override
    public void off() {
        tv.off();
    }

    @Override
    public void tune(int chanel) {
        tv.tune(chanel);
    }
}
```



```java
package design.pattern.bridge;

public class Test {
    public static void main(String[] args) {
        TV lgLv = new LGTV();
        TV sontTv = new SonyTV();

        new RemoteControllerImpl(lgLv).on();
        new RemoteControllerImpl(lgLv).off();
        new RemoteControllerImpl(lgLv).tune(10);
        new RemoteControllerImpl(sontTv).on();
        new RemoteControllerImpl(sontTv).off();
        new RemoteControllerImpl(sontTv).tune(20);
    }
}
```

```
Switch on LG TV
Switch off LG TV
Switch on chanel in LG TV is: 10
Switch on Sony TV
Switch off Sony TV
Switch on chanel in Sony TV is: 20
```

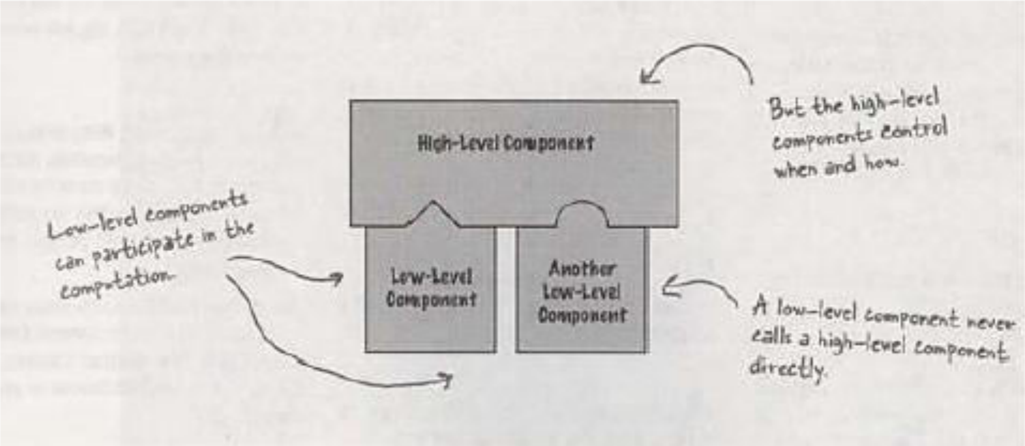
Template method pattern

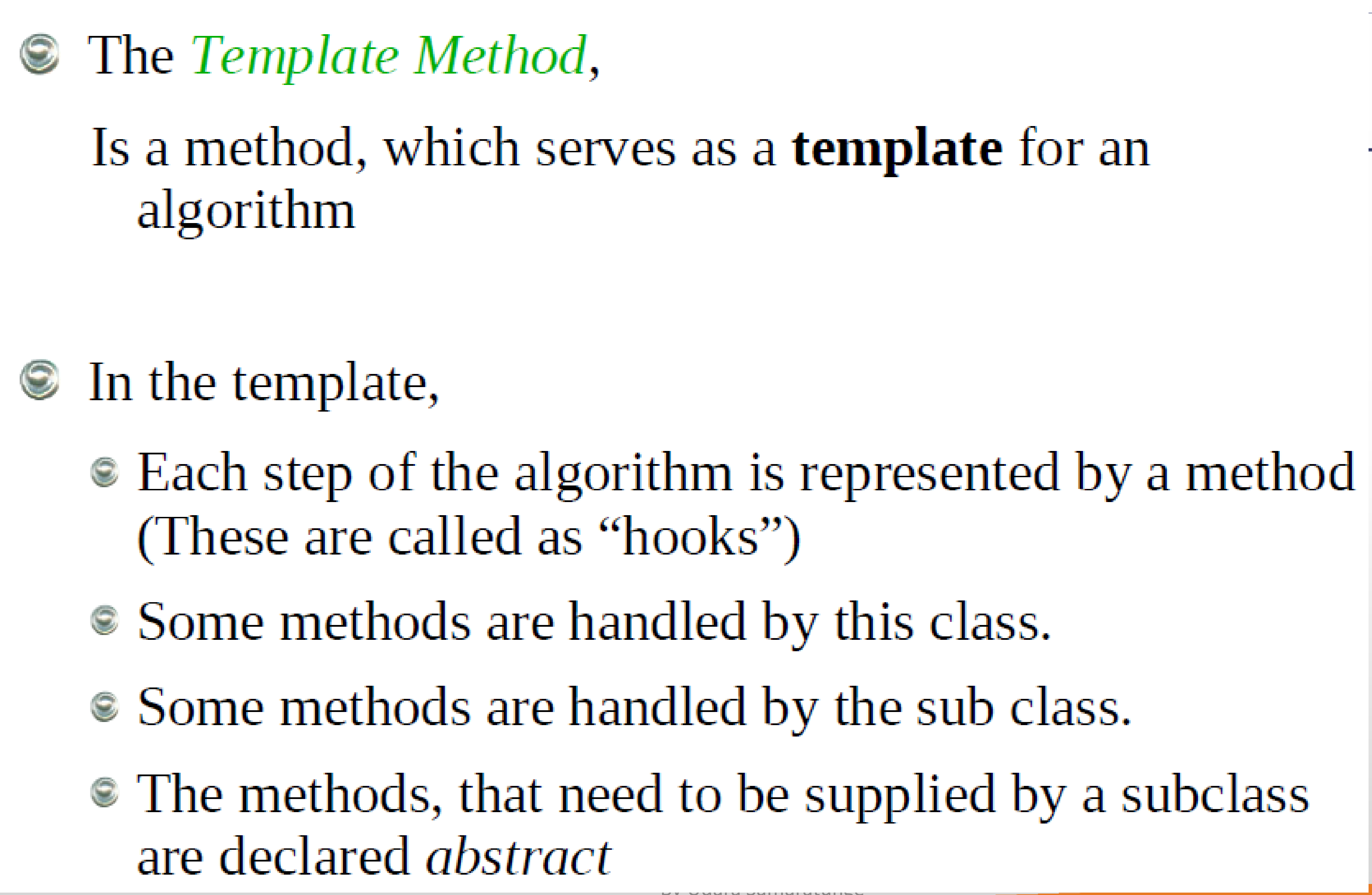
Design Principles covered - (1)

- *Design Principle*

“The Hollywood Principle” - *Don’t call us, we will call you*

Allows low level components to hook themselves into a system. But the high-level components determine when they are needed and how.


High-Level Component
Low-Level Component
Another Low-Level Component
But the high-level components control when and how.
Low-level components can participate in the computation.
A low-level component never calls a high-level component directly.

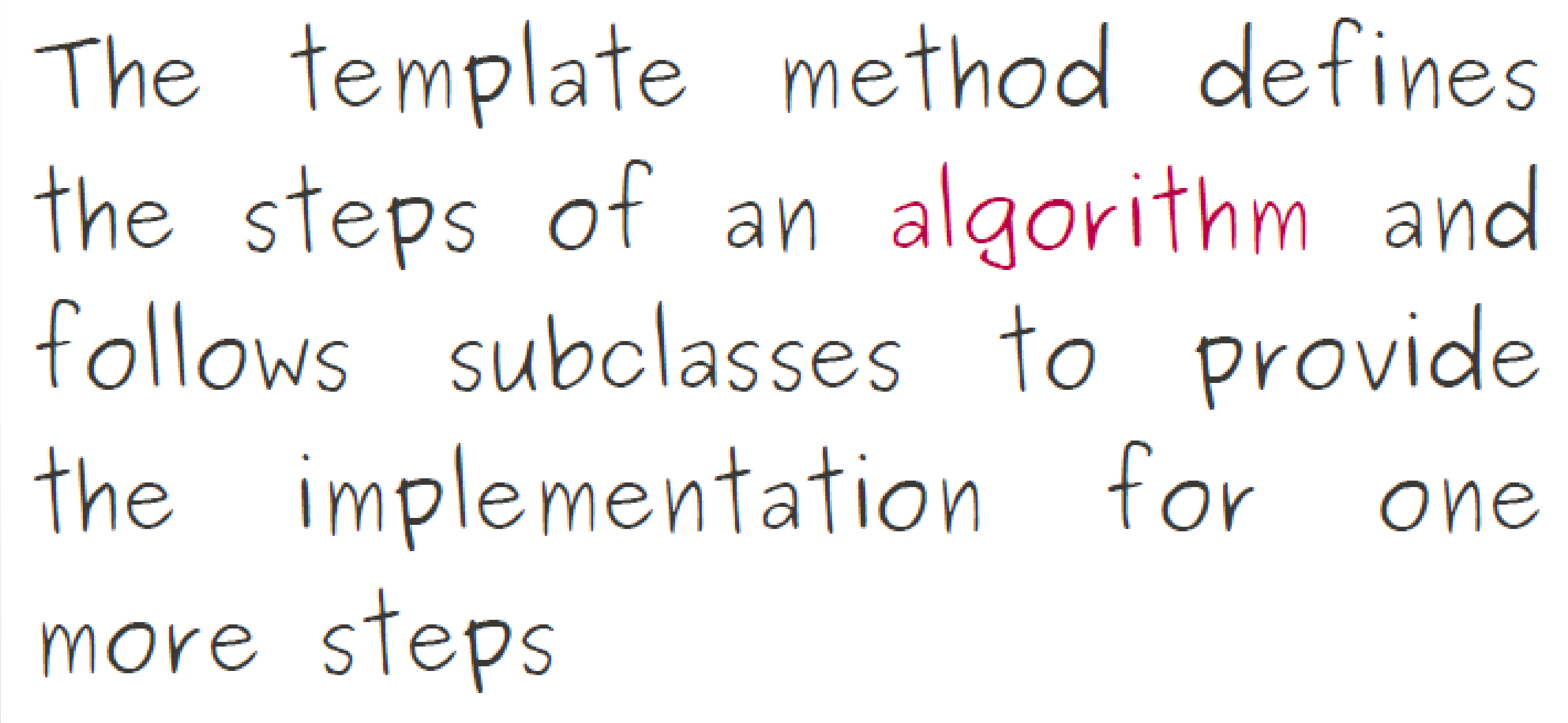


- The *Template Method*,

  Is a method, which serves as a **template** for an algorithm

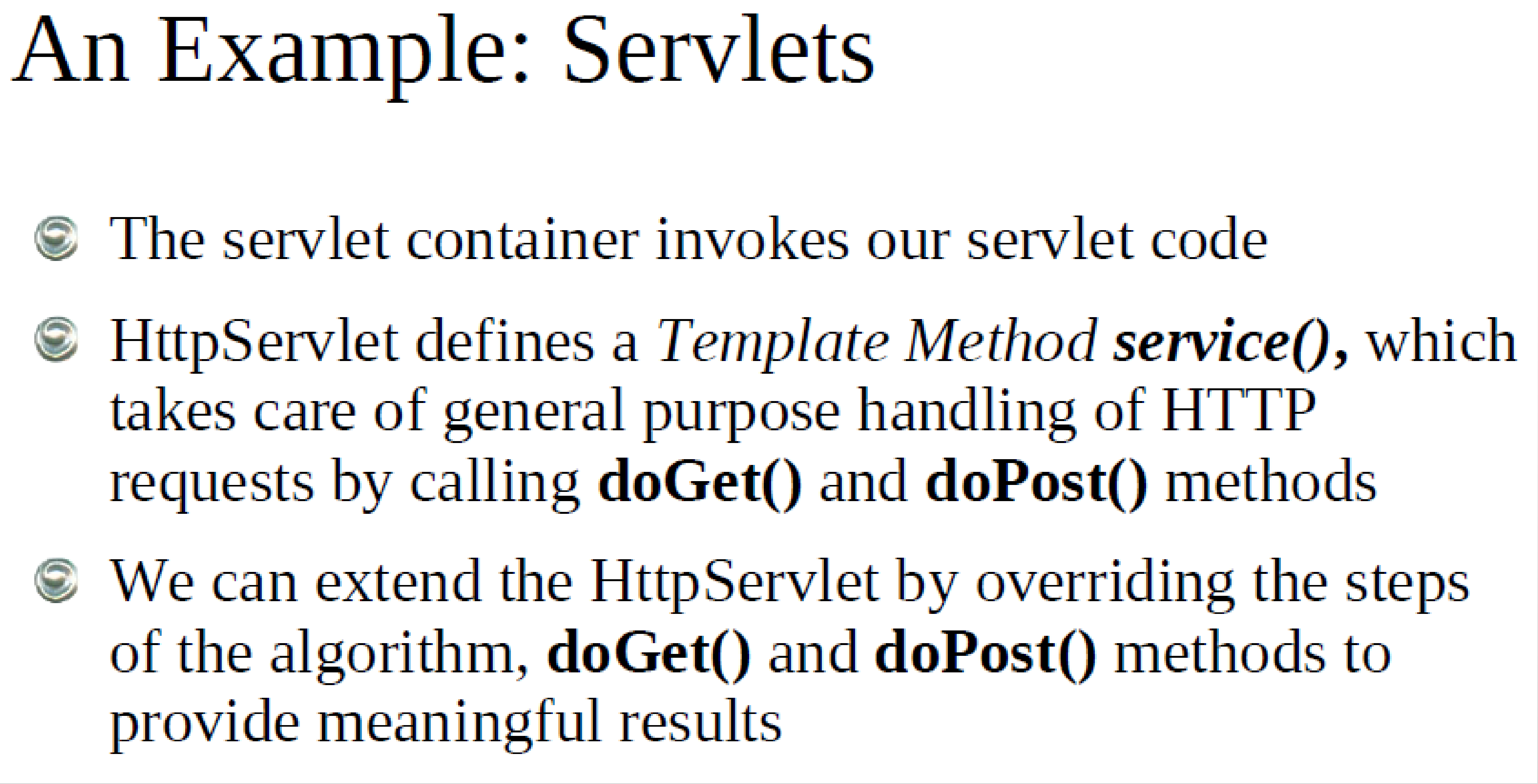
- In the template,
  - Each step of the algorithm is represented by a method (These are called as “hooks”)
  - Some methods are handled by this class.
  - Some methods are handled by the sub class.
  - The methods, that need to be supplied by a subclass are declared *abstract*

Template method pattern



The template method defines the steps of an algorithm and follows subclasses to provide the implementation for one more steps

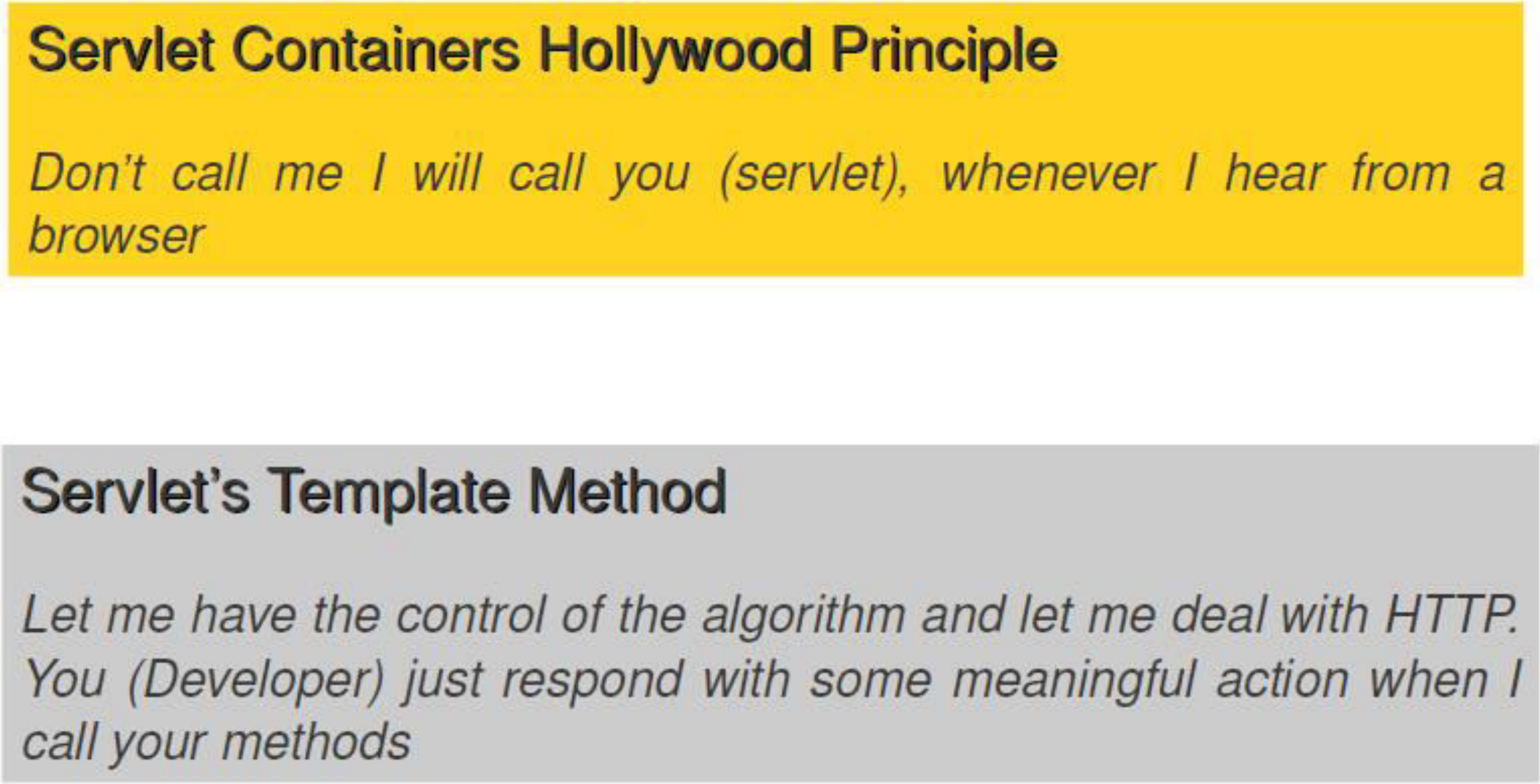
Template method pattern



An Example: Servlets

- The servlet container invokes our servlet code
- HttpServlet defines a *Template Method* ***service()*****,** which takes care of general purpose handling of HTTP requests by calling **doGet()** and **doPost()** methods
- We can extend the HttpServlet by overriding the steps of the algorithm, **doGet()** and **doPost()** methods to provide meaningful results

An Example: Servlets

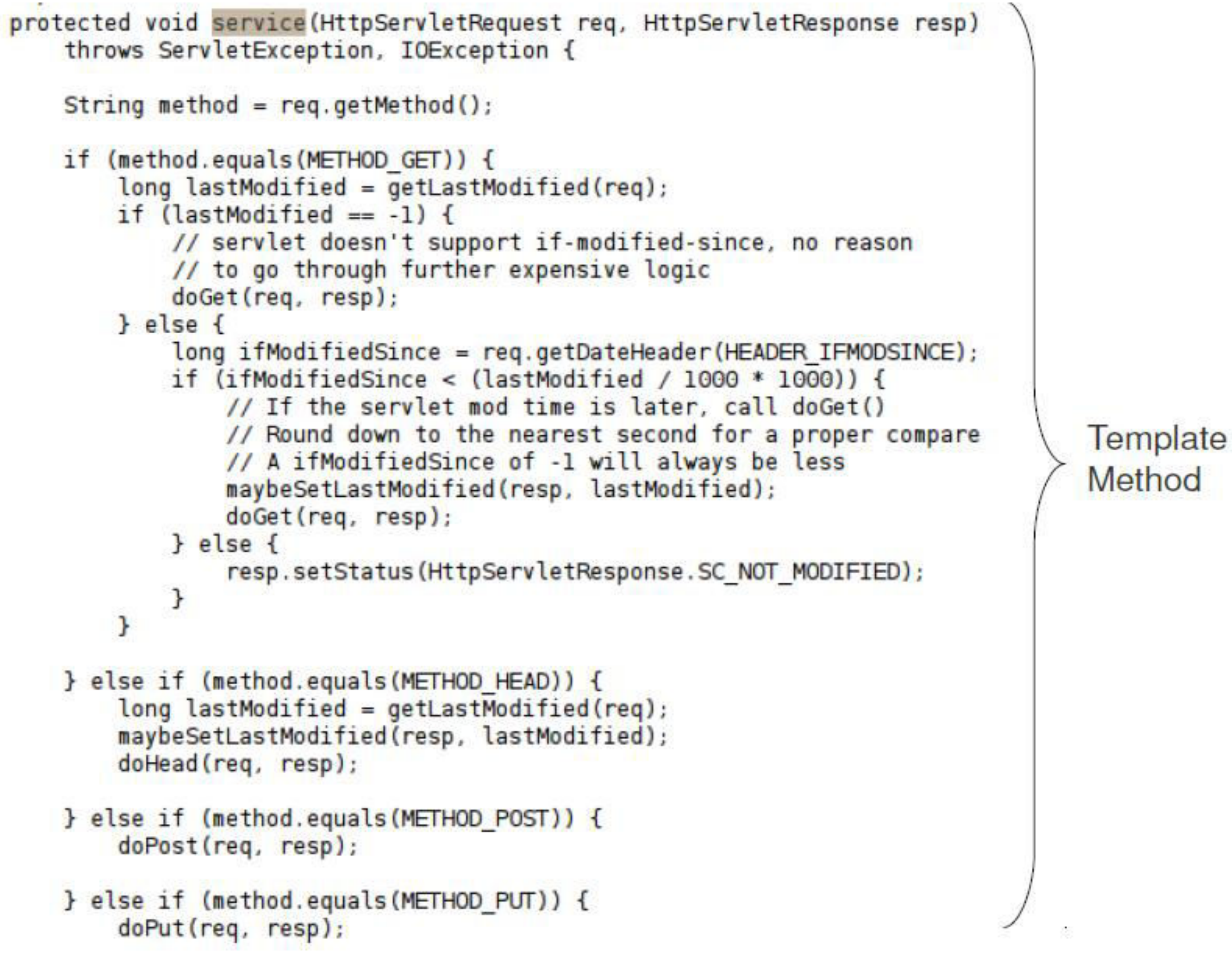


Servlet Containers Hollywood Principle

Don't call me I will call you (servlet), whenever I hear from a browser

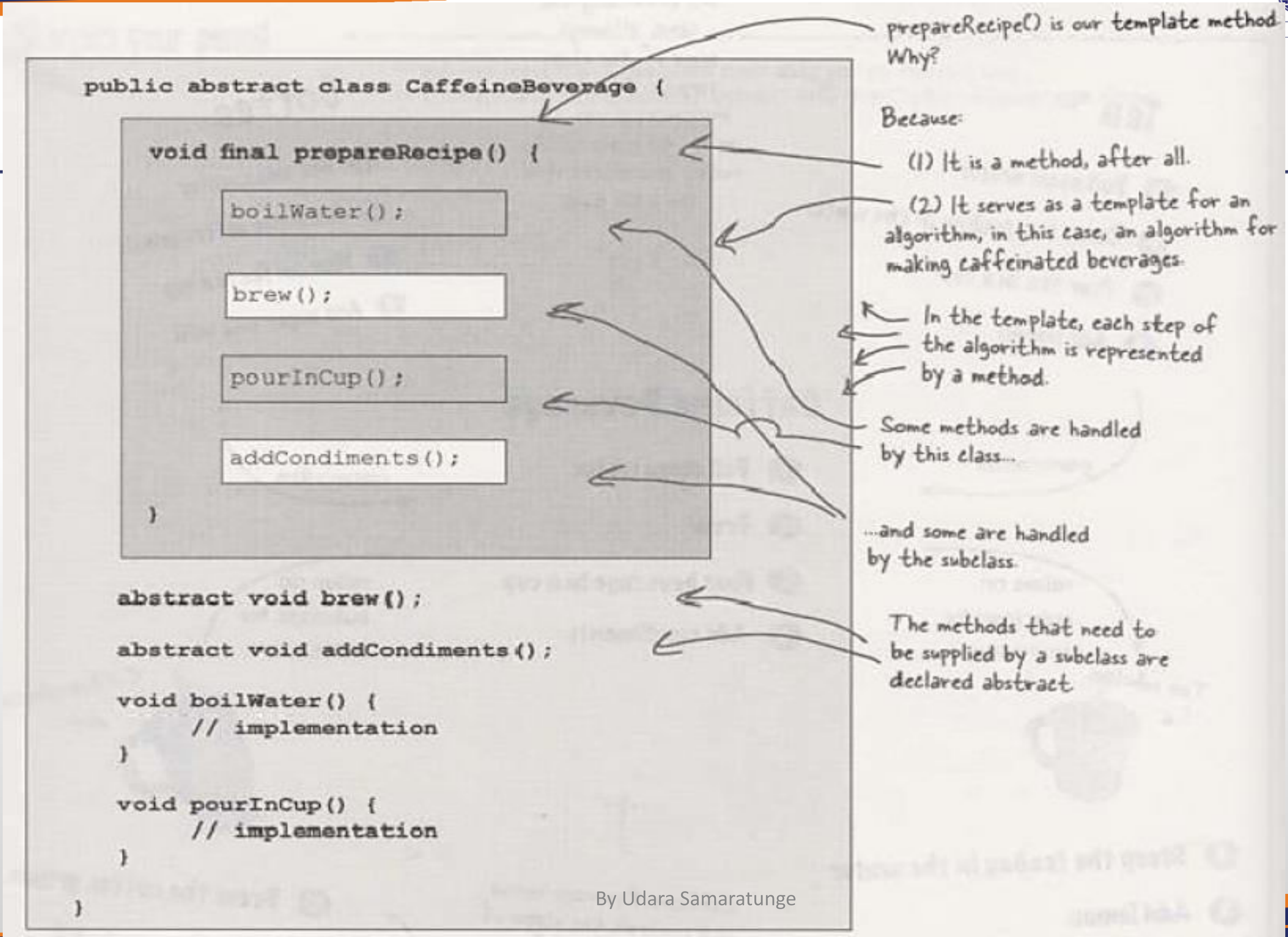
Servlet's Template Method

Let me have the control of the algorithm and let me deal with HTTP. You (Developer) just respond with some meaningful action when I call your methods



```
protected void service(HttpServletRequest req, HttpServletResponse resp)
    throws ServletException, IOException {

    String method = req.getMethod();

    if (method.equals(METHOD_GET)) {
        long lastModified = getLastModified(req);
        if (lastModified == -1) {
            // servlet doesn't support if-modified-since, no reason
            // to go through further expensive logic
            doGet(req, resp);
        } else {
            long ifModifiedSince = req.getDateHeader(HEADER_IFMODSINCE);
            if (ifModifiedSince < (lastModified / 1000 * 1000)) {
                // If the servlet mod time is later, call doGet()
                // Round down to the nearest second for a proper compare
                // A ifModifiedSince of -1 will always be less
                maybeSetLastModified(resp, lastModified);
                doGet(req, resp);
            } else {
                resp.setStatus(HttpServletResponse.SC_NOT_MODIFIED);
            }
        }

    } else if (method.equals(METHOD_HEAD)) {
        long lastModified = getLastModified(req);
        maybeSetLastModified(resp, lastModified);
        doHead(req, resp);

    } else if (method.equals(METHOD_POST)) {
        doPost(req, resp);

    } else if (method.equals(METHOD_PUT)) {
        doPut(req, resp);
```

Template Method



```
public abstract class CaffeineBeverage {

    void final prepareRecipe() {
        boilWater();
        brew();
        pourInCup();
        addCondiments();
    }

    abstract void brew();

    abstract void addCondiments();

    void boilWater() {
        // implementation
    }

    void pourInCup() {
        // implementation
    }
}
```

prepareRecipe() is our template method. Why?

Because:

(1) It is a method, after all.

(2) It serves as a template for an algorithm, in this case, an algorithm for making caffeinated beverages.

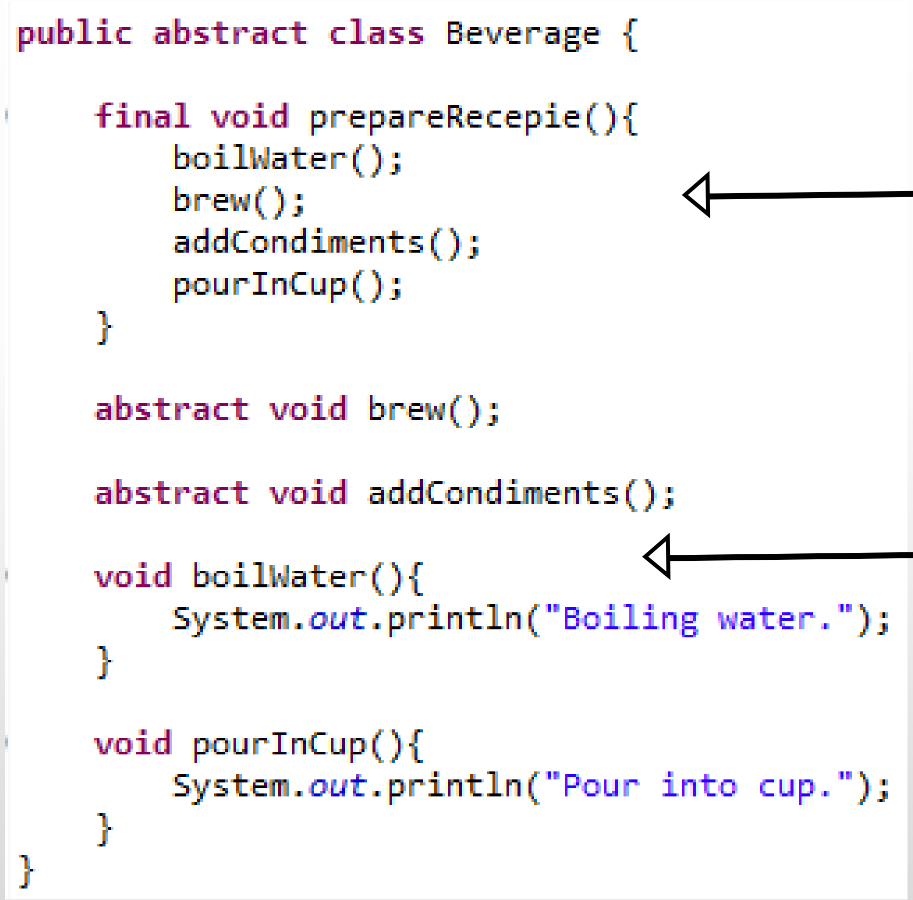
In the template, each step of the algorithm is represented by a method.

Some methods are handled by this class...

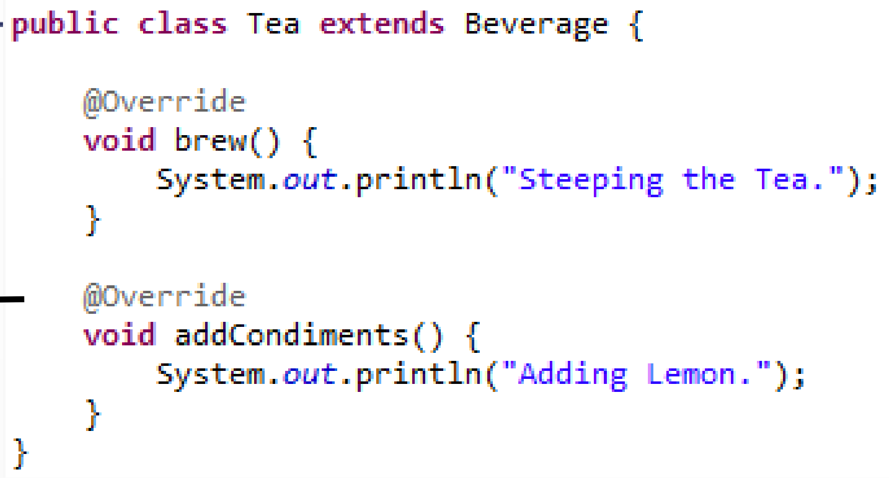
...and some are handled by the subclass.

The methods that need to be supplied by a subclass are declared abstract.

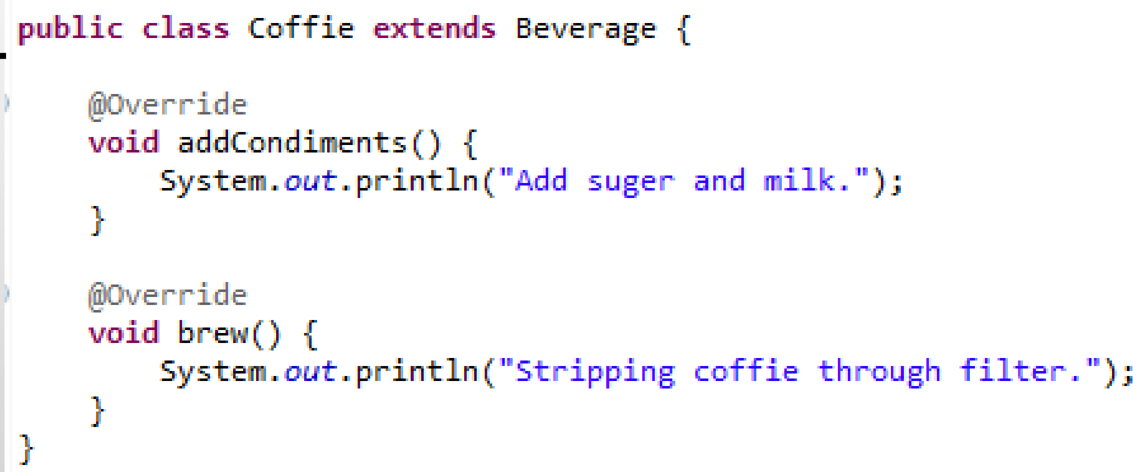
Template method pattern



```java
public abstract class Beverage {

    final void prepareRecepie(){
        boilWater();
        brew();
        addCondiments();
        pourInCup();
    }

    abstract void brew();

    abstract void addCondiments();

    void boilWater(){
        System.out.println("Boiling water.");
    }

    void pourInCup(){
        System.out.println("Pour into cup.");
    }
}
```

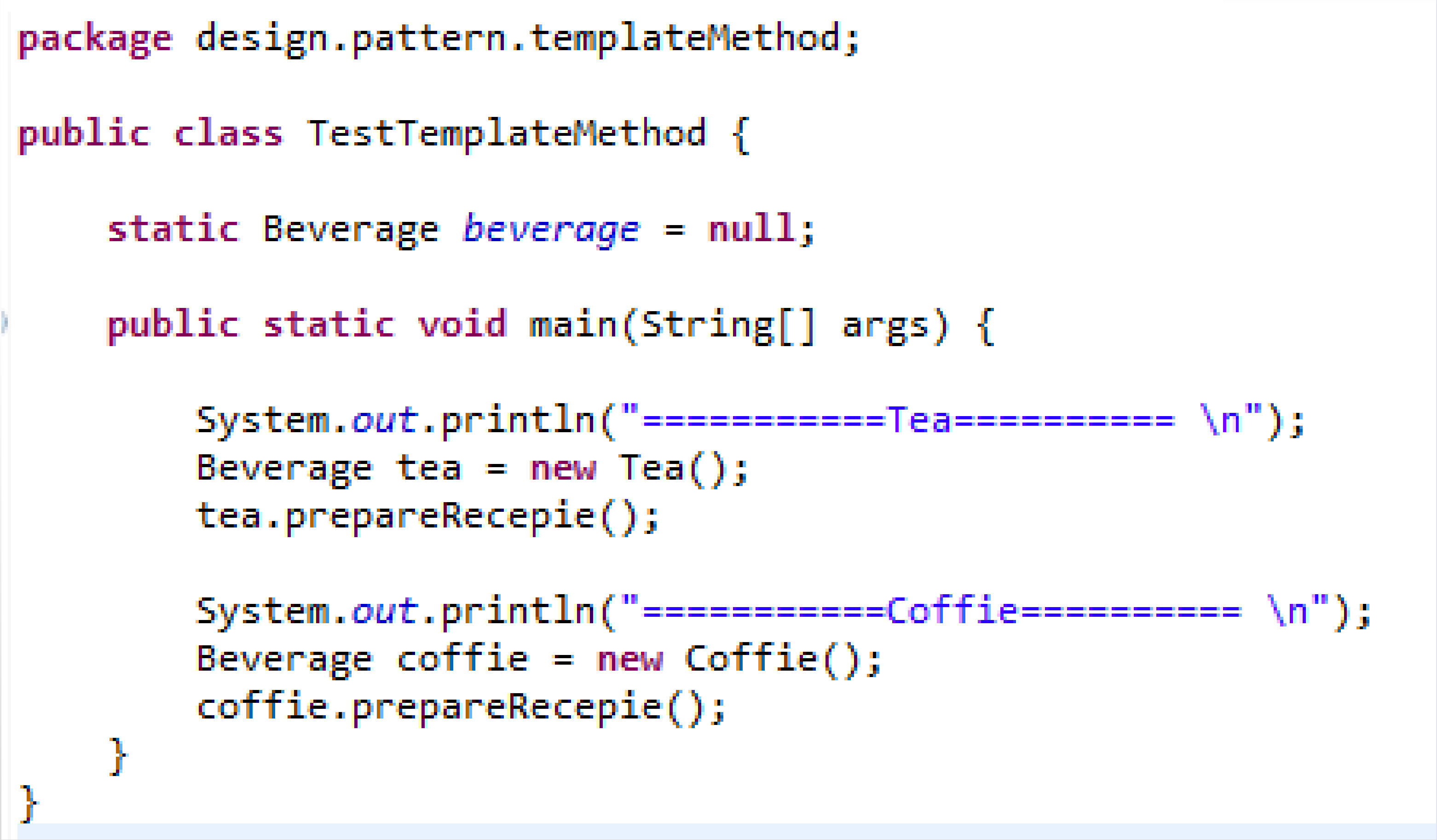


```java
public class Tea extends Beverage {

    @Override
    void brew() {
        System.out.println("Steeping the Tea.");
    }

    @Override
    void addCondiments() {
        System.out.println("Adding Lemon.");
    }
}
```

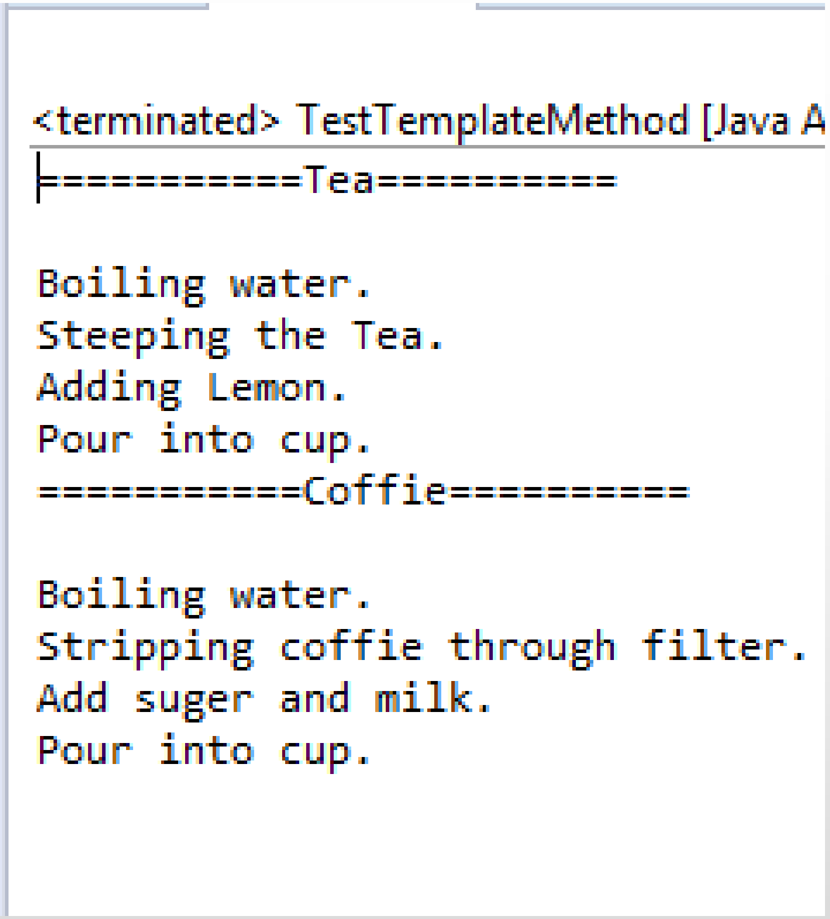


```java
public class Coffie extends Beverage {

    @Override
    void addCondiments() {
        System.out.println("Add suger and milk.");
    }

    @Override
    void brew() {
        System.out.println("Stripping coffie through filter.");
    }
}
```

Template method pattern



```
package design.pattern.templateMethod;

public class TestTemplateMethod {

    static Beverage beverage = null;

    public static void main(String[] args) {

        System.out.println("===========Tea=========== \n");
        Beverage tea = new Tea();
        tea.prepareRecepie();

        System.out.println("===========Coffie========== \n");
        Beverage coffie = new Coffie();
        coffie.prepareRecepie();
    }
}
```



```
<terminated> TestTemplateMethod [Java A
===========Tea===========

Boiling water.
Steeping the Tea.
Adding Lemon.
Pour into cup.
===========Coffie==========

Boiling water.
Stripping coffie through filter.
Add suger and milk.
Pour into cup.
```

References

- Head First Design Patterns: *by Eric Freeman & Elisabeth Freeman*
- Design Patterns: Elements of Reusable Object Oriented Software: *Erich Gamma, Richard Helm, Ralph Johnson & John Vilssides (GOF) (http://www.hillside.net)*
- Core Security Patterns: Best Practices and Strategies J2EE Web Services and Identity Management: *Chris Steel, Ramesh Nagappan, Ray Lai, Sun Microsystems*
- Core J2EE Patterns, Best Practices and Design Strategies: *Deepak Alur, John Crupi, Dan Malks, 2nd Edition, Prentice Hall/ Sun Microsystems, 2003*
- *http://www.javaworld.com/jw-11-1998/jw-11-techniques.html*

The End