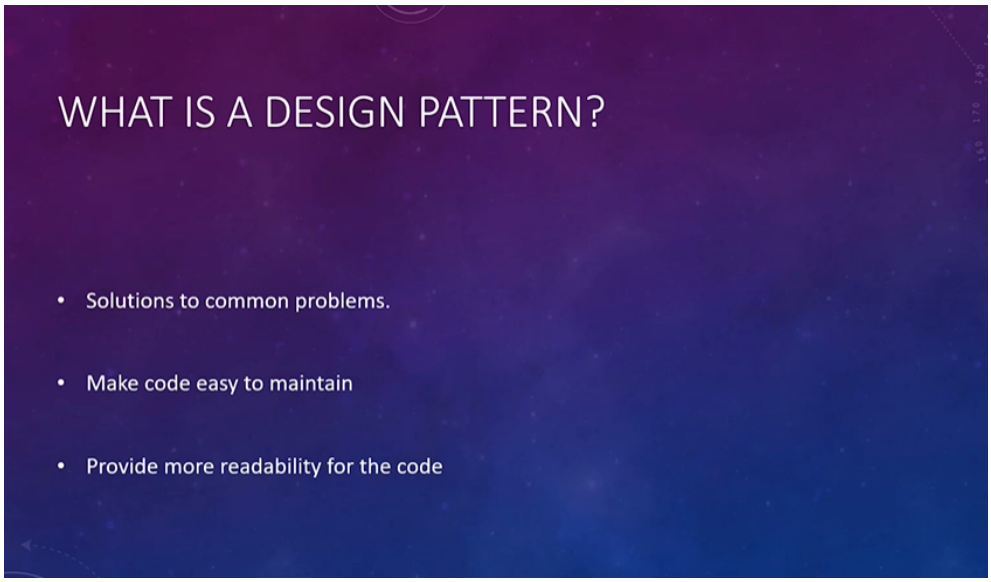
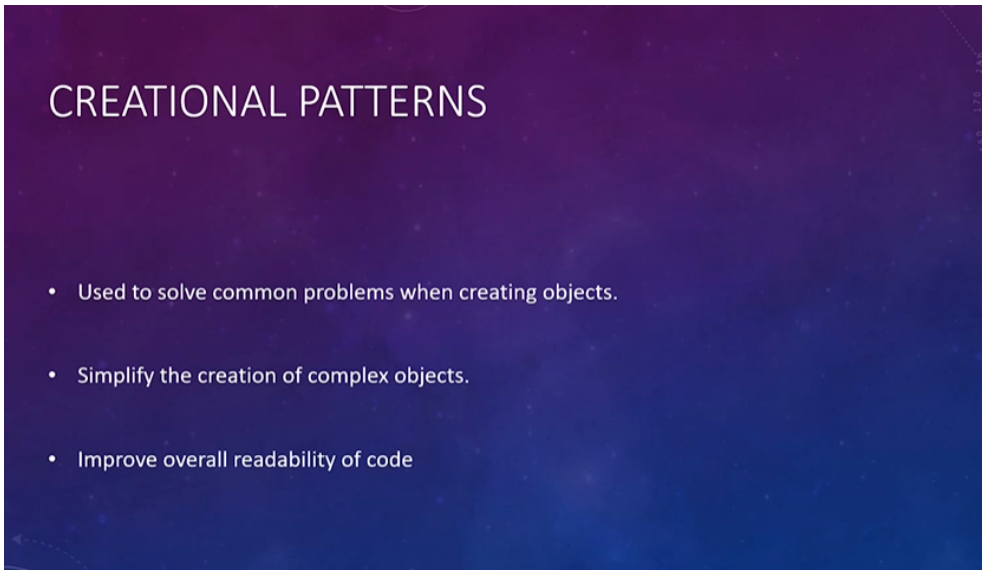
DESIGN PATTERNS IN JAVA



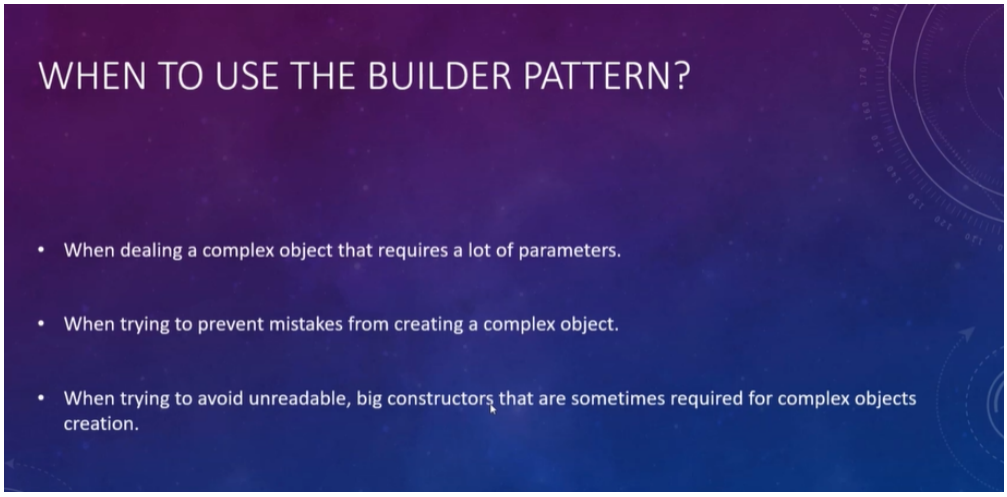
CREATIONAL PATTERNS



TYPES OF CREATIONAL PATTERNS

- Builder
- Factory
- Abstract Factory
- Singleton
- Prototype
- Object Pool

THE BUILDER PATTERN



Problem

We will use OnlineStoreAccount class with 5 attributes. Let's create the OnlineStoreAccount class with a constructor with all variable parameters, and with get-set methods.

OnlineStoreAccount.java

```java
public class OnlineStoreAccount {

    private Long id;
    private String name;
    private String address;
    private Long budget;
    private Long discountRate;

    public OnlineStoreAccount(Long id, String name, String address, Long budget, Long discountRate){
        this.id = id;
        this.name = name;
        this.address = address;
        this.budget = budget;
        this.discountRate = discountRate;
    }

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }
```

```
public class OnlineStoreAccount {

    private Long id;
    private String name;
    private String address;
    private Long budget;
    private Long discountRate;

    public OnlineStoreAccount(Long id, String name, String address, Long budget, Long
discountRate){
        this.id = id;
        this.name = name;
        this.address = address;
        this.budget = budget;
        this.discountRate = discountRate;
    }

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getAddress() {
        return address;
    }

    public void setAddress(String address) {
        this.address = address;
    }

    public Long getBudget() {
        return budget;
    }
```

```
    public void setBudget(Long budget) {
        this.budget = budget;
    }

    public Long getDiscountRate() {
        return discountRate;
    }

    public void setDiscountRate(Long discountRate) {
        this.discountRate = discountRate;
    }
}
```

And let's use this class.

App.java

```
public class App {

    public static void main(String[] args) {
        OnlineStoreAccount johnSmith =
                new OnlineStoreAccount(1L, "John Smith", "Liberty Lane 23", 100L, 2L);
        OnlineStoreAccount JaneTaylor =
                new OnlineStoreAccount(2L, "Jane Taylor", "Goethe Street 55", 2L, 100L);
    }
}
```

```
public class App {

    public static void main(String[] args) {
        OnlineStoreAccount johnSmith =
                new OnlineStoreAccount(1L, "John Smith", "Liberty Lane 23", 100L, 2L);
        OnlineStoreAccount JaneTaylor =
                new OnlineStoreAccount(2L, "Jane Taylor", "Goethe Street 55", 2L, 100L);
    }
}
```

Can you see the problem here with creating objects? Especially the last 2 parameters. Is it easy to understand wcih one is budget, which one is discount rate?

It is very easy to send parameters to the constructor with wrong order.

So, we decided to use Builder pattern here.

We will create the class with variables and get-set methods again.

```
public class OnlineStoreAccount {

    private Long id;
    private String name;
    private String address;
    private Long budget;
    private Long discountRate;
```

Create a private constructor.

Because we don’t want this class’s objects to be created directly.

```
    private OnlineStoreAccount() {
    }
```

Create an inner static Builder class in this class.

If we have a mandatory variable when creating object from the class, put that variable in the Builder constructor.

And create methods for all other variables to set.

Last, put a build method whose return type is OnlineStoreAccount in the inner static class.

```
    private Long id;
    private String name;
    private String address;
    private Long budget;
    private Long discountRate;

    public static class Builder {
        private Long id;
        private String name;
        private String address;
        private Long budget;
        private Long discountRate;

        public Builder(Long id) { this.id = id; }

        public Builder withName(String name) {
            this.name = name;

            return this;
        }

        public Builder withAddress(String address) {
            this.address = address;

            return this;
        }
```

```
    public OnlineStoreAccount build() {
        OnlineStoreAccount onlineStoreAccount = new OnlineStoreAccount();
        onlineStoreAccount.id = this.id;
        onlineStoreAccount.name = this.name;
        onlineStoreAccount.address = this.address;
        onlineStoreAccount.budget = this.budget;
        onlineStoreAccount.discountRate = this.discountRate;

        return onlineStoreAccount;
    }
```

public class OnlineStoreAccount {

```
private Long id;
private String name;
private String address;
private Long budget;
private Long discountRate;

public static class Builder {
    private Long id;
    private String name;
    private String address;
    private Long budget;
    private Long discountRate;

    public Builder(Long id) {
        this.id = id;
    }

    public Builder withName(String name) {
        this.name = name;

        return this;
    }

    public Builder withAddress(String address) {
        this.address = address;

        return this;
    }

    public Builder withBudget(Long budget) {
        this.budget = budget;

        return this;
    }

    public Builder withDiscount(Long discountRate) {
        this.discountRate = discountRate;

        return this;
    }

    public OnlineStoreAccount build() {
        OnlineStoreAccount onlineStoreAccount = new OnlineStoreAccount();
```

```
        onlineStoreAccount.id = this.id;
        onlineStoreAccount.name = this.name;
        onlineStoreAccount.address = this.address;
        onlineStoreAccount.budget = this.budget;
        onlineStoreAccount.discountRate = this.discountRate;

        return onlineStoreAccount;
    }
}

private OnlineStoreAccount() {
}

public Long getId() {
    return id;
}

public void setId(Long id) {
    this.id = id;
}

public String getName() {
    return name;
}

public void setName(String name) {
    this.name = name;
}

public String getAddress() {
    return address;
}

public void setAddress(String address) {
    this.address = address;
}

public Long getBudget() {
    return budget;
}

public void setBudget(Long budget) {
    this.budget = budget;
}
```

```
    public Long getDiscountRate() {
        return discountRate;
    }

    public void setDiscountRate(Long discountRate) {
        this.discountRate = discountRate;
    }
}
```

Now we can create the App class again.

App.java

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

        OnlineStoreAccount johnSmith = new OnlineStoreAccount.Builder(1L)
                .withName("JohnSmith")
                .withAddress("Oxford Lane 35A")
                .withBudget(100L)
                .withDiscount(2L)
                .build();

        System.out.println(johnSmith.getName());

    }
}
```

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

        OnlineStoreAccount johnSmith = new OnlineStoreAccount.Builder(1L)
                .withName("JohnSmith")
                .withAddress("Oxford Lane 35A")
                .withBudget(100L)
                .withDiscount(2L)
                .build();

        System.out.println(johnSmith.getName());

    }
}
```

As a result, it is more difficult to send the parameters with wrong order, through Builder pattern.

FACTORY PATTERN

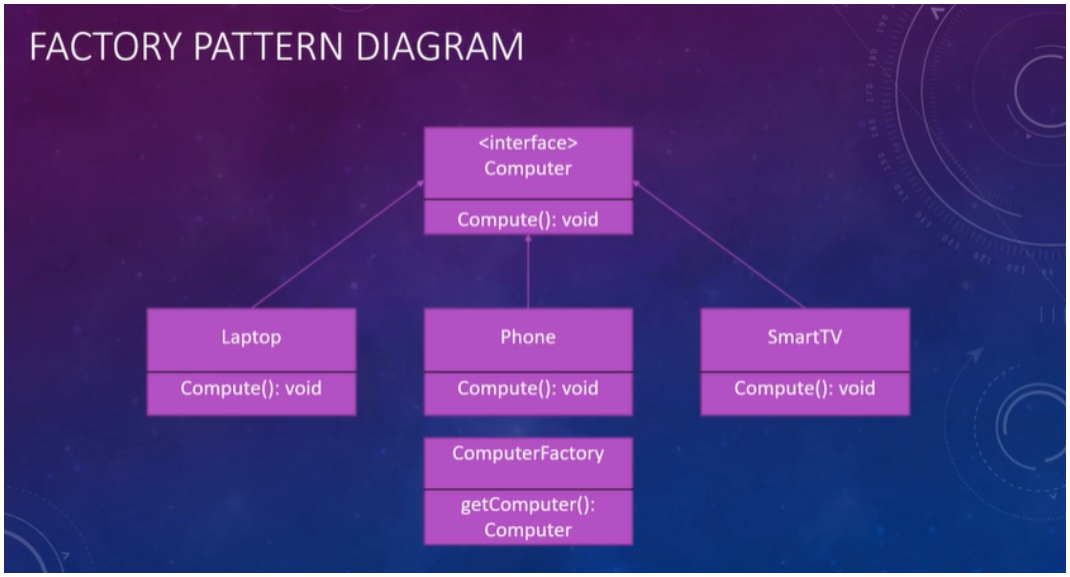
Problem

We have a Computer interface and some methods to implement in it.

3 classes (Laptop, Phone, SmartTV) implement that interface.

Our manager wants from us a method that accepts a parameter (“laptop”,”phone”,”smarttv”) and creates object from that class.

We decided to use Factory Pattern to hide implementation details to use the interface methods.



Let's create our *Computer* interface, and *Laptop, Phone, SmartTV, ComputerFactory* classes.

Computer.java

```
public interface Computer {

    void compute();
}
```


```
public interface Computer {

    void compute();
}
```


Laptop.java

```
public class Laptop implements Computer {
    @Override
    public void compute() { System.out.println("Laptop computes"); }
}
```

```
public class Laptop implements Computer {
    @Override
    public void compute() {
        System.out.println("Laptop computes");
    }
}
```

Phone.java

```
public class Phone implements Computer {
    @Override
    public void compute() { System.out.println("Phone computes"); }
}
```

```
public class Phone implements Computer {
    @Override
    public void compute() {
        System.out.println("Phone computes");
    }
}
```

SmartTV.java

```
public class SmartTv implements Computer {
    @Override
    public void compute() { System.out.println("The Smart TV computes"); }
}
```

```
public class SmartTv implements Computer {
    @Override
    public void compute() {
        System.out.println("The Smart TV computes");
    }
}
```

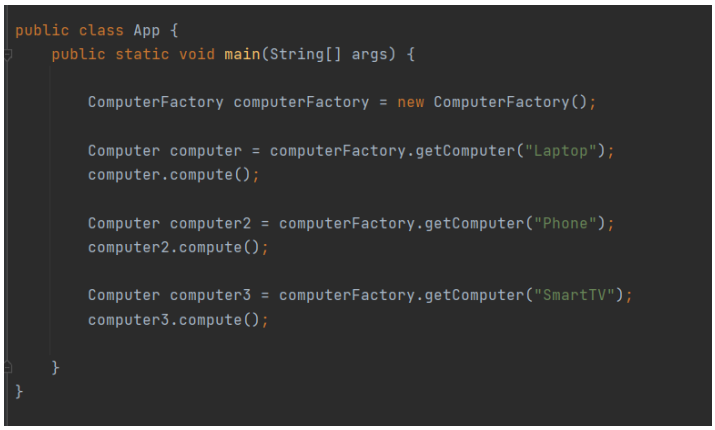
Now we will create a factory class to hide the object instantiation details from the calling function.

ComputerFactory.java


```
public class ComputerFactory {

   public Computer getComputer(String computer){

      if(computer.equalsIgnoreCase("Laptop")){
         return new Laptop();
      }
      else if(computer.equalsIgnoreCase("Phone")){
         return new Phone();
      }
      else if(computer.equalsIgnoreCase("SmartTV")){
         return new SmartTv();
      }

      return null;
   }
}
```

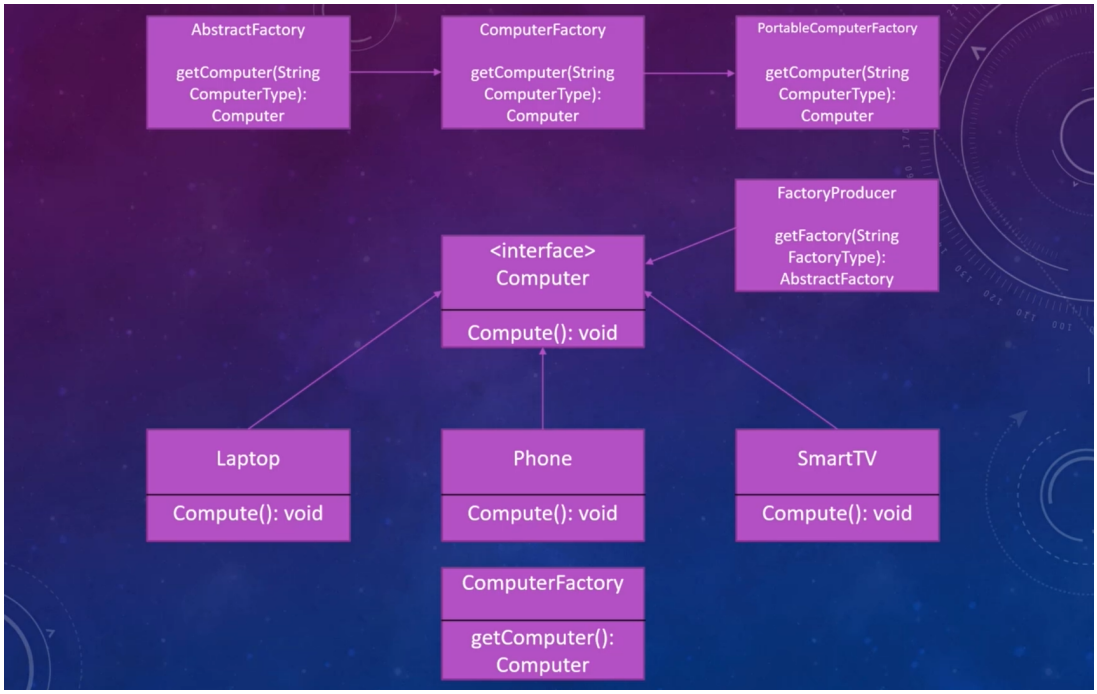
When i am writing my app, i only use factory class.

App.java



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

        ComputerFactory computerFactory = new ComputerFactory();

        Computer computer = computerFactory.getComputer("Laptop");
        computer.compute();

        Computer computer2 = computerFactory.getComputer("Phone");
        computer2.compute();

        Computer computer3 = computerFactory.getComputer("SmartTV");
        computer3.compute();

    }
}
```

THE ABSTRACT FACTORY PATTERN



The difference between Factory and Abstract Factory patterns is, we have more than one factory with Abstract Factory pattern.

For Example, ComputerFactory creates all types of computers. PortableFactory creates only portable computers.

FactoryProducer chooses the Factory type.

Let's create our classes. Phone, Laptop, and SmartTv classes, and the Computer interface are the same with the previous pattern.

First, we'll create AbstractFactory class.

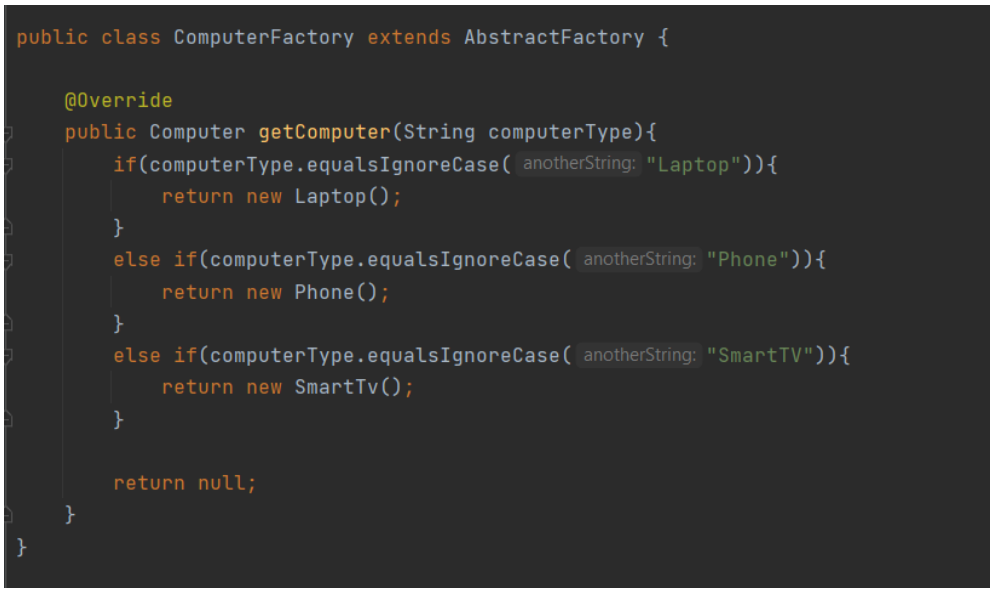
AbstractFactory.java

```
public abstract class AbstractFactory {
    abstract Computer getComputer(String computerType);
}
```

public abstract class AbstractFactory {
   abstract Computer getComputer(String computerType);
}

And then, we will create Factory classes. Both of them will extend from AbstractFactory class.

ComputerFactory.java



```
public class ComputerFactory extends AbstractFactory {

    @Override
    public Computer getComputer(String computerType){
        if(computerType.equalsIgnoreCase( anotherString: "Laptop")){
            return new Laptop();
        }
        else if(computerType.equalsIgnoreCase( anotherString: "Phone")){
            return new Phone();
        }
        else if(computerType.equalsIgnoreCase( anotherString: "SmartTV")){
            return new SmartTv();
        }

        return null;
    }
}
```


```
public class ComputerFactory extends AbstractFactory {

    @Override
    public Computer getComputer(String computerType){
        if(computerType.equalsIgnoreCase("Laptop")){
            return new Laptop();
        }
        else if(computerType.equalsIgnoreCase("Phone")){
            return new Phone();
        }
        else if(computerType.equalsIgnoreCase("SmartTV")){
            return new SmartTv();
        }

        return null;
    }
}
```

PortableComputer.java

```
public class PortableComputer extends AbstractFactory {
    @Override
    Computer getComputer(String computerType) {
        if(computerType.equalsIgnoreCase( anotherString: "Laptop")){
            return new Laptop();
        }
        else if(computerType.equalsIgnoreCase( anotherString: "Phone")){
            return new Phone();
        }

        return null;
    }
}
```


```
public class PortableComputer extends AbstractFactory {
   @Override
   Computer getComputer(String computerType) {
      if(computerType.equalsIgnoreCase("Laptop")){
         return new Laptop();
      }
      else if(computerType.equalsIgnoreCase("Phone")){
         return new Phone();
      }

      return null;
   }
}
```


Now, create the FactoryProducer class that chooses the Factory class to create.

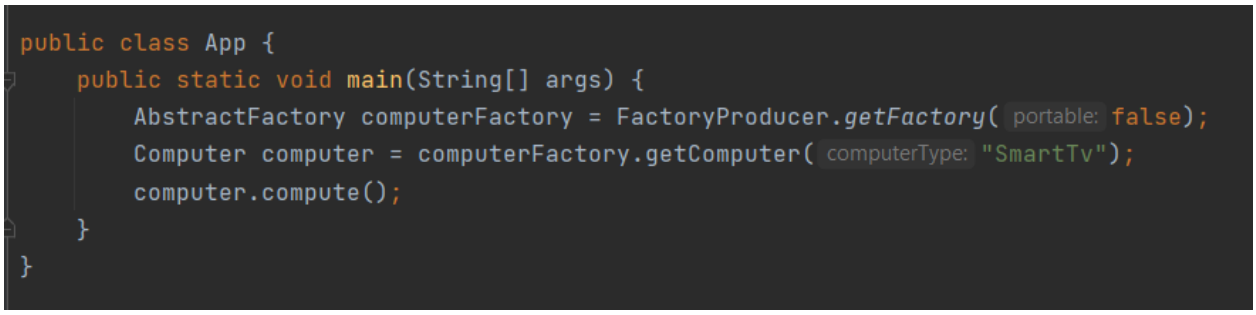
FactoryProducer.java

```
public class FactoryProducer {
    public static AbstractFactory getFactory(boolean portable){
        if(portable){
            return new PortableComputer();
        }
        else{
            return new ComputerFactory();
        }
    }
}
```

```
public class FactoryProducer {
   public static AbstractFactory getFactory(boolean portable){
      if(portable){
         return new PortableComputer();
      }
      else{
         return new ComputerFactory();
      }
   }
}
```

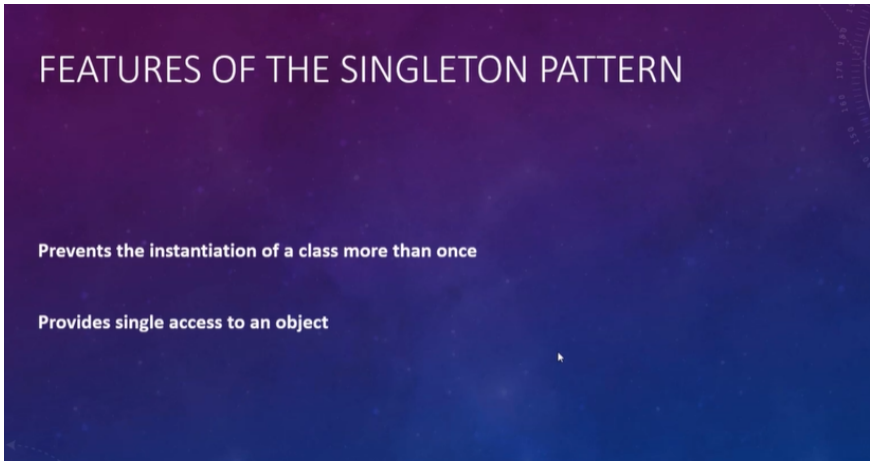
And run the pattern with the App class.

App.java




```
public class App {
   public static void main(String[] args) {
      AbstractFactory computerFactory = FactoryProducer.getFactory(false);
      Computer computer = computerFactory.getComputer("SmartTv");
      computer.compute();
   }
}
```

THE SINGLETON PATTERN



Let’s create our SingletonLogger class.

We will pay attention to the following:

- The constructor will be private. You can’t create an object directly from a singleton class.
- There must be a public static method to get the instance. Method must be lazy loaded. If the object to be returned is null, then create the object.

  Also, this method should be thread safe. More than one thread shouldn’t call this method at the same time, so use synchronized keyword.

SingletonLogger.java

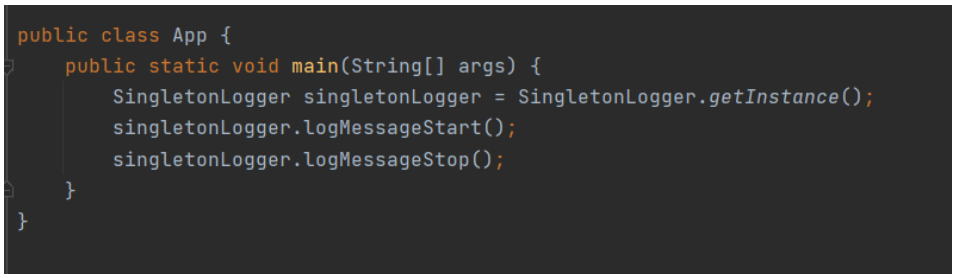
```
public class SingletonLogger {

    private static SingletonLogger instance;

    private SingletonLogger(){}

    //lazy loaded, thread safe
    public static synchronized SingletonLogger getInstance(){
        if(instance == null){
            instance = new SingletonLogger();
        }
        return instance;
    }

    public void logMessageStart() { System.out.println("Start message is logged"); }

    public void logMessageStop() { System.out.println("Stop message is logged"); }
}
```


```
public class SingletonLogger {

    private static SingletonLogger instance;

    private SingletonLogger(){}

    //lazy loaded, thread safe
    public static synchronized SingletonLogger getInstance(){
        if(instance == null){
            instance = new SingletonLogger();
        }
        return instance;
    }

    public void logMessageStart() {
```

```
        System.out.println("Start message is logged");
    }

    public void logMessageStop() {
        System.out.println("Stop message is logged");
    }
}
```

And use the SingletonLogger by calling public static object creation method.

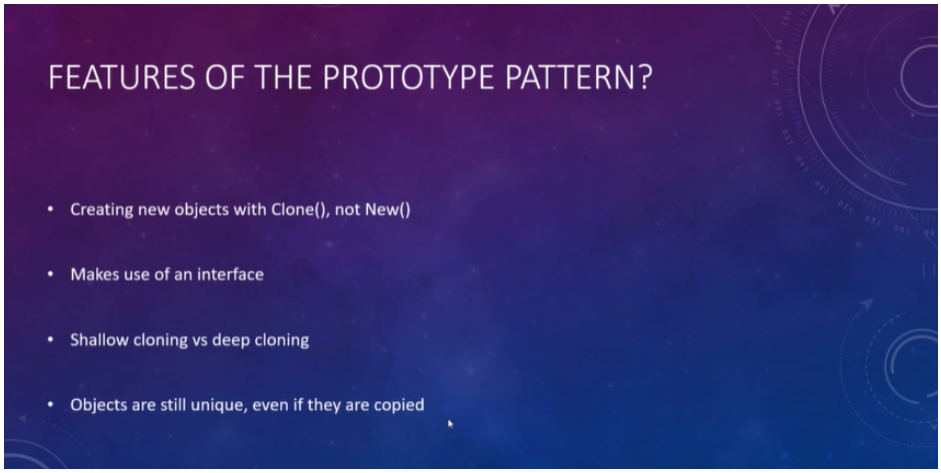
App.java

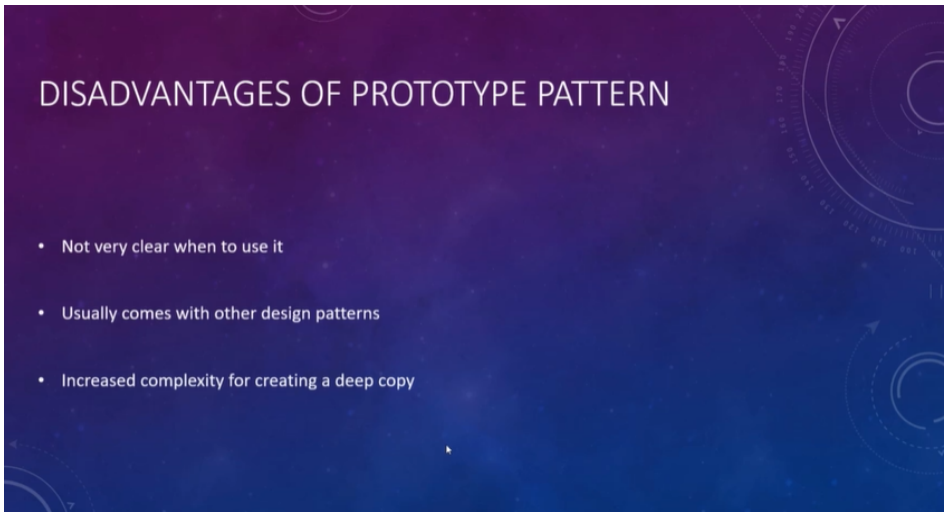


```
public class App {
    public static void main(String[] args) {
        SingletonLogger singletonLogger = SingletonLogger.getInstance();
        singletonLogger.logMessageStart();
        singletonLogger.logMessageStop();
    }
}
```

```
public class App {
    public static void main(String[] args) {
        SingletonLogger singletonLogger = SingletonLogger.getInstance();
        singletonLogger.logMessageStart();
        singletonLogger.logMessageStop();
    }
}
```

THE PROTOTYPE PATTERN





We can use Prototype pattern when it is costly to create an object.

For example, an object can have lots of properties, heavy collections, and other object references in it.

In these situations, we may prefer to clone an object, instead of creating another object.

Pay attention to the following:

- The class of the object to be cloned, should implement Cloneable Interface
- And has a clone method has a return type of the class itself

We will create a Robot class to be cloned, and Components class to referenced by Robot Class.

Components class is a classical POJO.

Components.java

```
public class Components {
    private String name;
    private String functionality;

    public String getName() { return name; }

    public void setName(String name) { this.name = name; }

    public String getFunctionality() { return functionality; }

    public void setFunctionality(String functionality) { this.functionality = functionality; }
}
```

```
public class Components {
    private String name;
    private String functionality;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getFunctionality() {
```

```
        return functionality;
    }

    public void setFunctionality(String functionality) {
        this.functionality = functionality;
    }
}
```

Robot.java

```
import java.util.List;

public class Robot implements Cloneable {

    private int ID;
    private List<String> features;
    private Components components;

    public Robot(int ID, List<String> features, Components components){
        this.ID = ID;
        this.features = features;
        this.components = components;
    }

    public int getID() { return ID; }

    public void setID(int ID) { this.ID = ID; }

    public List<String> getFeatures() { return features; }

    public void setFeatures(List<String> features) { this.features = features; }
```

```
    public Components getComponents() { return components; }

    public void setComponents(Components components) { this.components = components; }

    public Robot clone(){
        try {
            return (Robot) super.clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
            return null;
        }
    }
}
```

```
import java.util.List;

public class Robot implements Cloneable {

    private int ID;
    private List<String> features;
    private Components components;

    public Robot(int ID, List<String> features, Components components){
        this.ID = ID;
        this.features = features;
        this.components = components;
    }

    public int getID() {
        return ID;
    }

    public void setID(int ID) {
        this.ID = ID;
    }

    public List<String> getFeatures() {
        return features;
    }

    public void setFeatures(List<String> features) {
        this.features = features;
    }

    public Components getComponents() {
        return components;
    }

    public void setComponents(Components components) {
        this.components = components;
    }

    public Robot clone(){
        try {
            return (Robot) super.clone();
        } catch (CloneNotSupportedException e) {
            e.printStackTrace();
            return null;
```

```
    }
  }
}
```

Let’s use Prototype Pattern in App.java

Be careful about that, first object will be created with new keyword, and the others will use clone method to copy from first object.

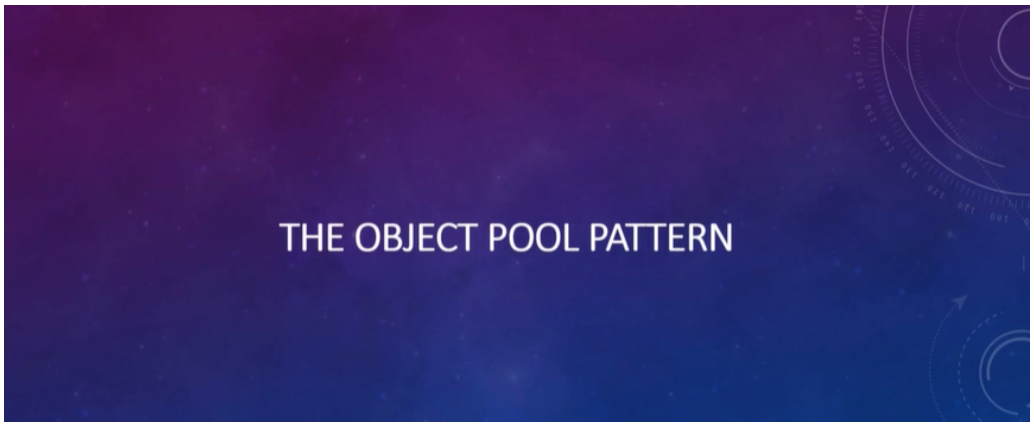
App.java

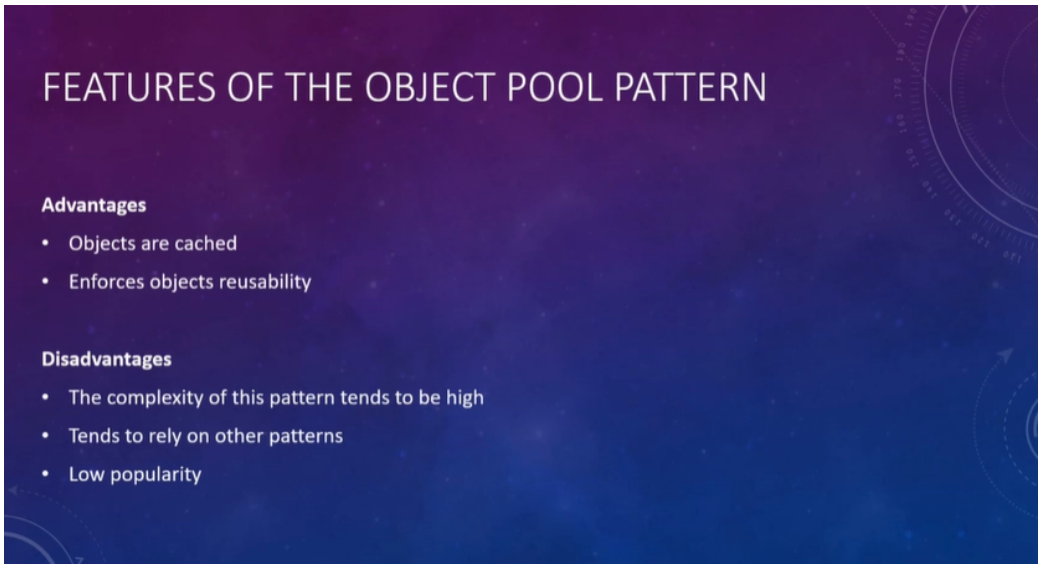
```
import java.util.ArrayList;
import java.util.List;

public class App {
    public static void main(String[] args) {
        List<String> features = new ArrayList<>();
        features.add("Start up");
        features.add("Perform Task");
        features.add("Shut down");
        Components components = new Components();
        components.setName("Infrared Goggles");
        components.setFunctionality("Read temperature of objects");

        Robot robot = new Robot( ID: 1, features, components);
        Robot anotherRobot = robot.clone();

        System.out.println(robot.getID());
        System.out.println(robot.getComponents());
        System.out.println(robot.getFeatures());

        System.out.println(anotherRobot.getID());
        System.out.println(anotherRobot.getComponents());
        System.out.println(anotherRobot.getFeatures());
    }
}
```

```
import java.util.ArrayList;
import java.util.List;

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

```
        List<String> features = new ArrayList<>();
        features.add("Start up");
        features.add("Perform Task");
        features.add("Shut down");
        Components components = new Components();
        components.setName("Infrared Goggles");
        components.setFunctionality("Read temperature of objects");

        Robot robot = new Robot(1, features, components);
        Robot anotherRobot = robot.clone();

        System.out.println(robot.getID());
        System.out.println(robot.getComponents());
        System.out.println(robot.getFeatures());

        System.out.println(anotherRobot.getID());
        System.out.println(anotherRobot.getComponents());
        System.out.println(anotherRobot.getFeatures());
    }
}
```

THE OBJECT POOL PATTERN





We need a class that has an Hashtable or HashMap in it to be used as pool.

Robot and Components classes are the same classes just like in prototype pattern.

We will need an abstract class ObjectPool first.

ObjectPool.java

```
import java.util.Hashtable;

public abstract class ObjectPool<T> {
    private Hashtable<T, Boolean> checkedIn = new Hashtable<>();

    public abstract T create();

    public synchronized void checkOut(T t){
;         checkedIn.put(t, false);
    }

    public synchronized T checkIn(){
        for(T t : checkedIn.keySet()){
            if(checkedIn.get(t)){
                return t;
            }
        }
        return null;
    }
}
```

```
import java.util.Hashtable;

public abstract class ObjectPool<T> {
    private Hashtable<T, Boolean> checkedIn = new Hashtable<>();

    public abstract T create();

    public synchronized void checkOut(T t){
;       checkedIn.put(t, false);
    }

    public synchronized T checkIn(){
        for(T t : checkedIn.keySet()){
            if(checkedIn.get(t)){
                return t;
            }
        }
        return null;
    }
}
```

And a RobotsPool class extending ObjectPool.

RobotsPool.java


```
import java.util.ArrayList;
import java.util.concurrent.ThreadLocalRandom;

public class RobotsPool extends ObjectPool {
```

```
    @Override
    public Object create() {
        int robotId = ThreadLocalRandom.current().nextInt();
        Robot robot = new Robot(robotId, new ArrayList<>(), new Components());

        return robot;
    }
}
```

And let's use the pool.

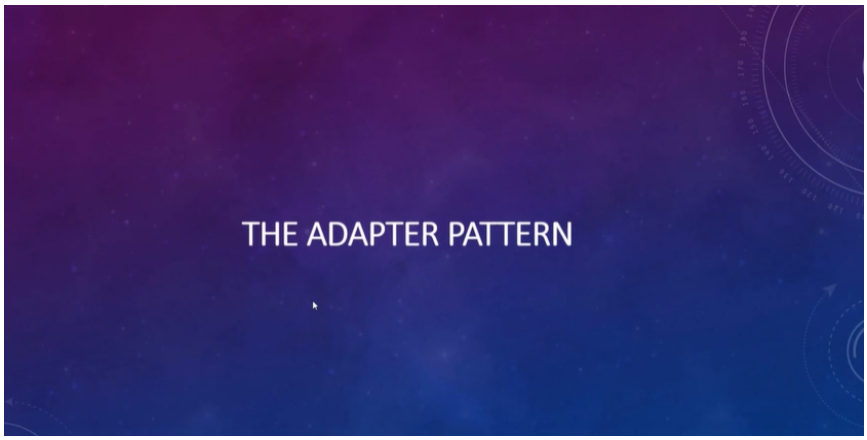
App.java

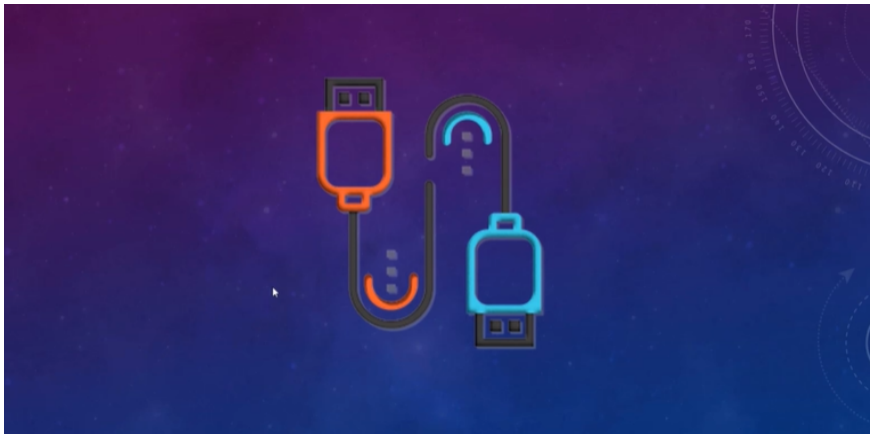
```
public class App {
    public static void main(String[] args) {
        ObjectPool robotsPool = new RobotsPool();
        Robot firstRobot = (Robot) robotsPool.create();
        Robot secondRobot = (Robot) robotsPool.create();

        robotsPool.checkOut(firstRobot);

        Robot thirdRobot = (Robot) robotsPool.checkIn();
    }
}
```


```
public class App {
    public static void main(String[] args) {
        ObjectPool robotsPool = new RobotsPool();
        Robot firstRobot = (Robot) robotsPool.create();
        Robot secondRobot = (Robot) robotsPool.create();

        robotsPool.checkOut(firstRobot);

        Robot thirdRobot = (Robot) robotsPool.checkIn();
    }
}
```

STRUCTURAL PATTERNS

THE ADAPTER PATTERN



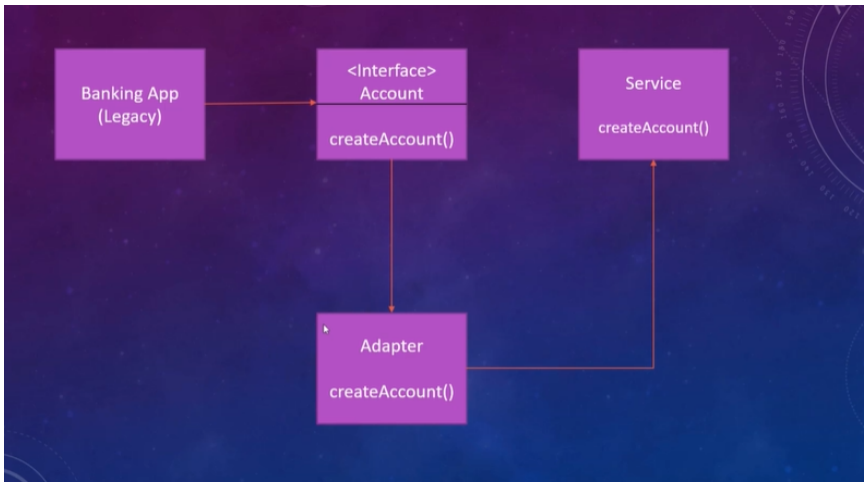


USAGE OF THE ADAPTER PATTERN

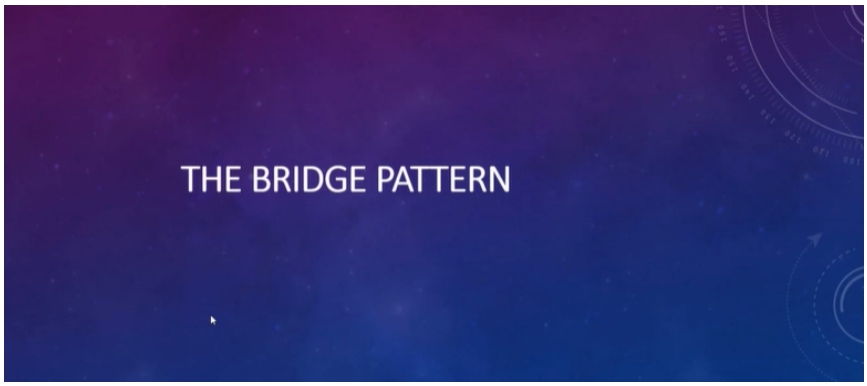
- In case of a legacy application
- In case of converting an interface into another
- In case of translating a client's requests for a webservice

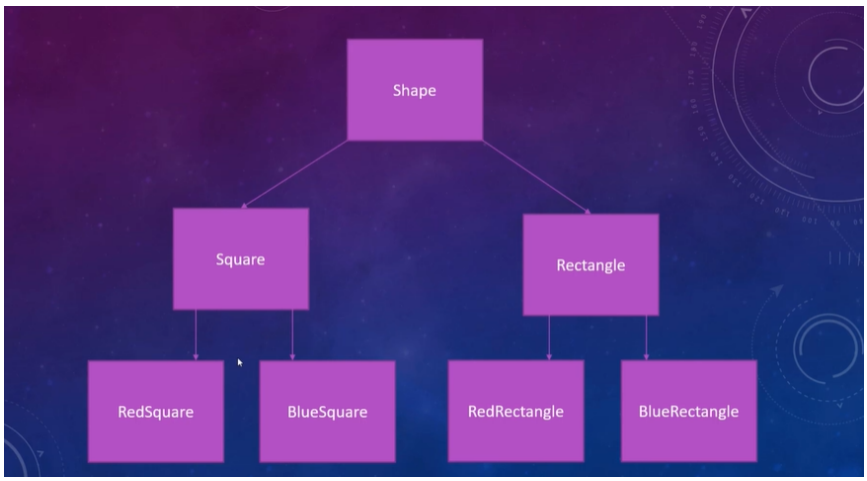
EXAMPLES OF USAGE IN THE JAVA API

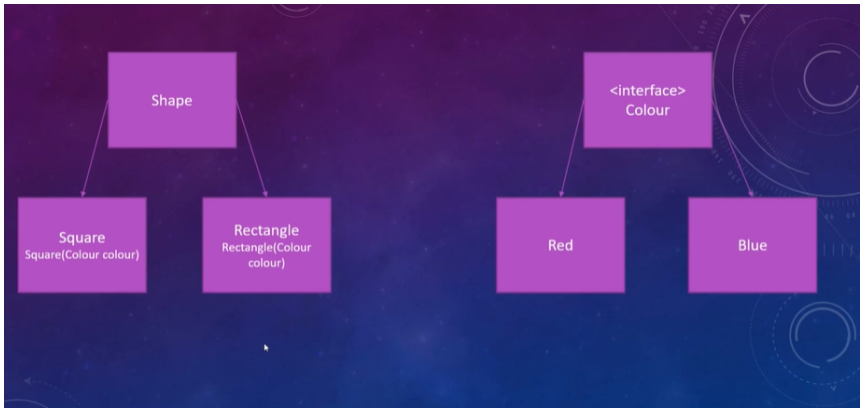
- Arrays.asList()
- I/O Streams



THE BRIDGE PATTERN

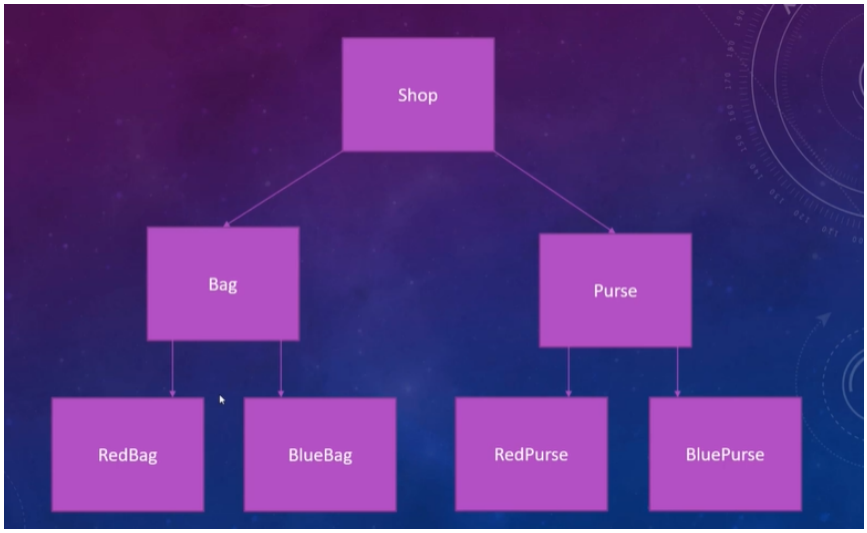


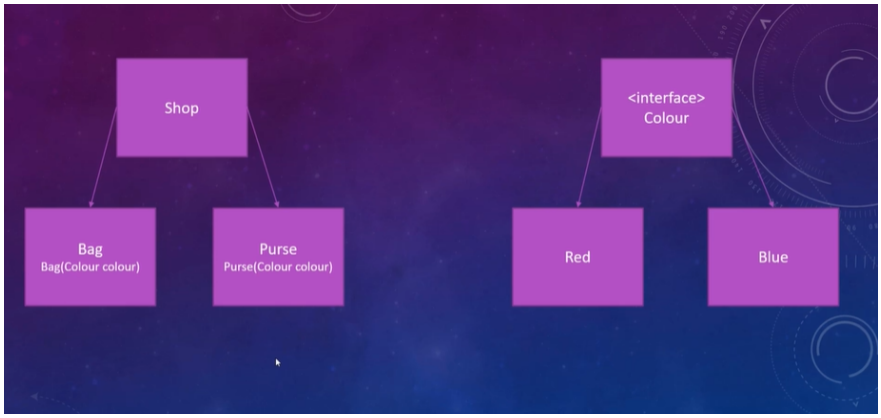




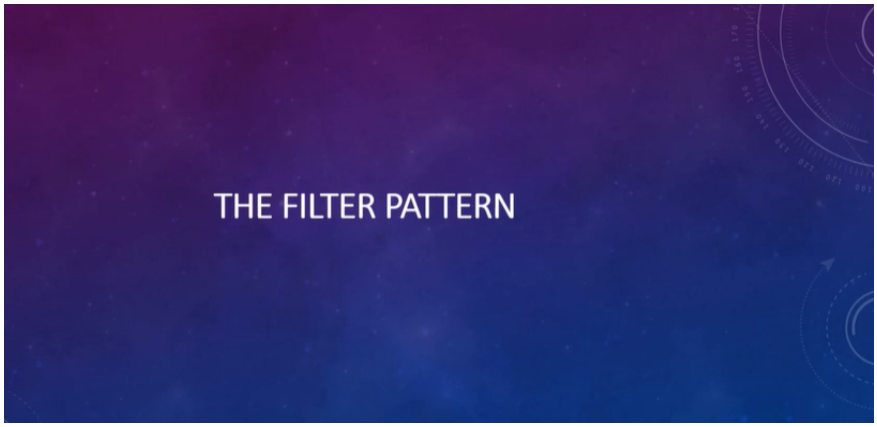
WHEN TO USE THE BRIDGE PATTERN

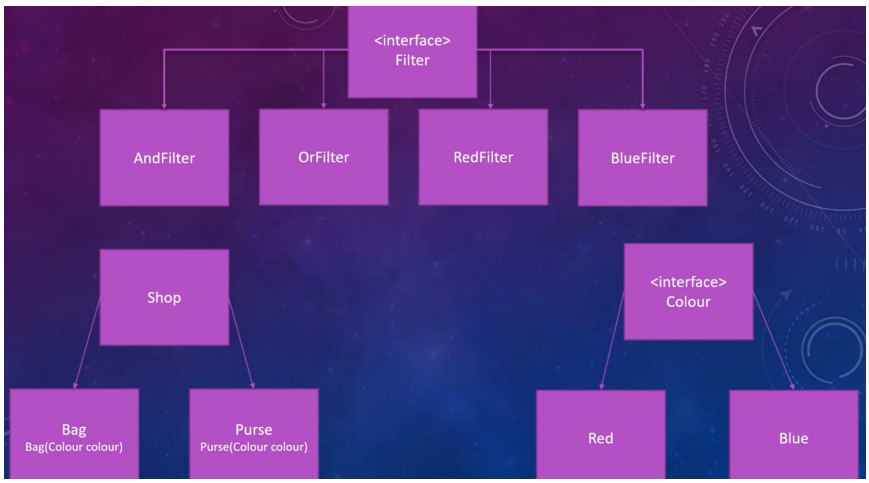
- When having a complex structure of classes with similar implementations
- When want to impose the same implementation for multiple objects
- When providing a map between orthogonal class hierarchies



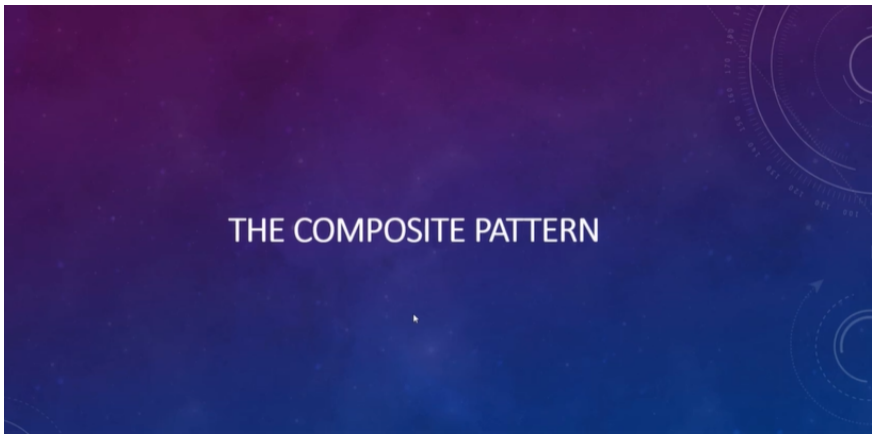


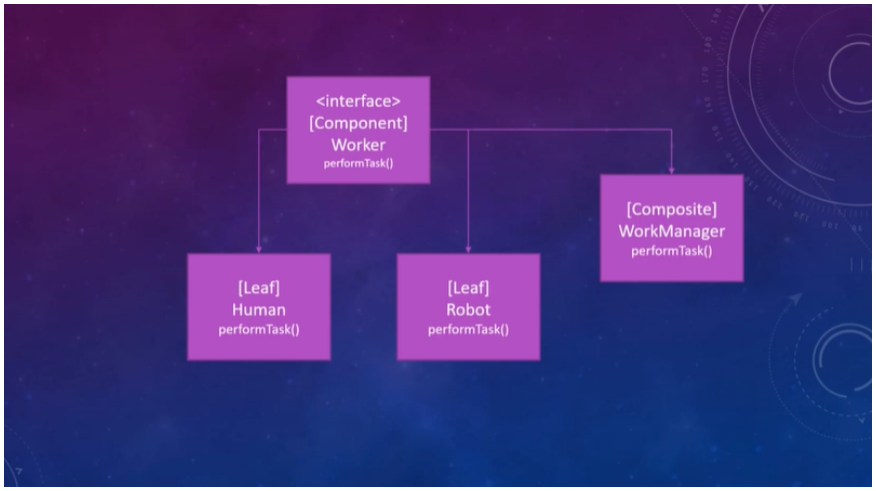
THE FILTER PATTERN

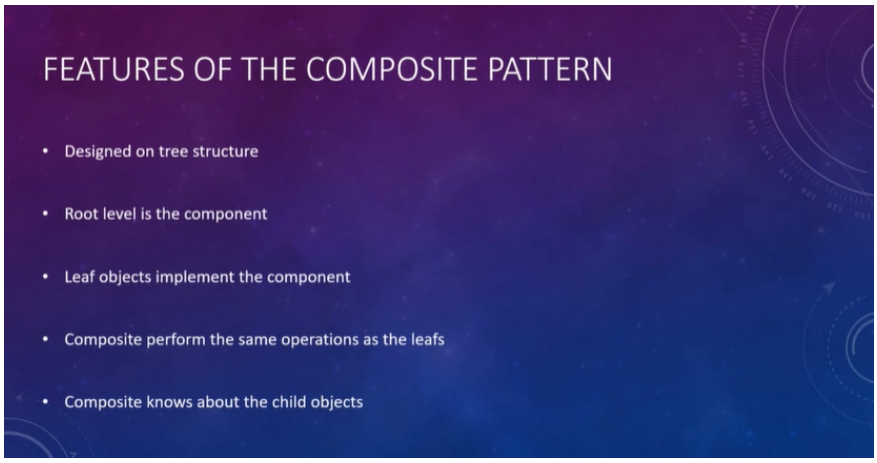




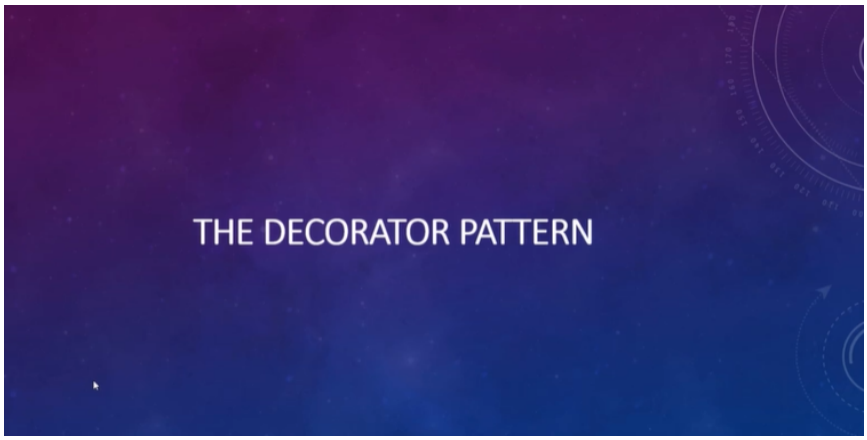
THE COMPOSITE PATTERN

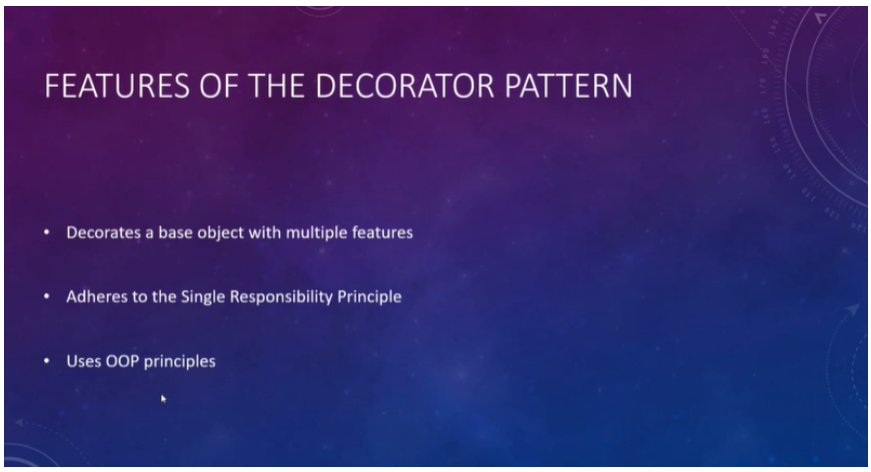


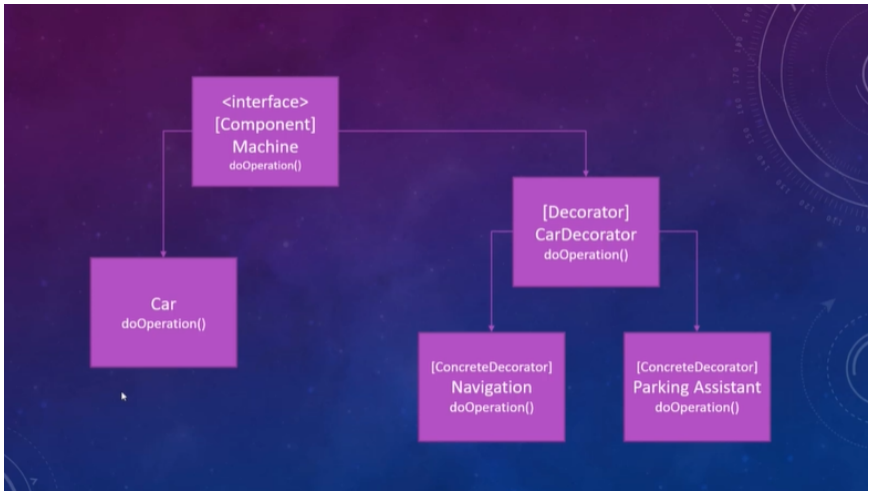


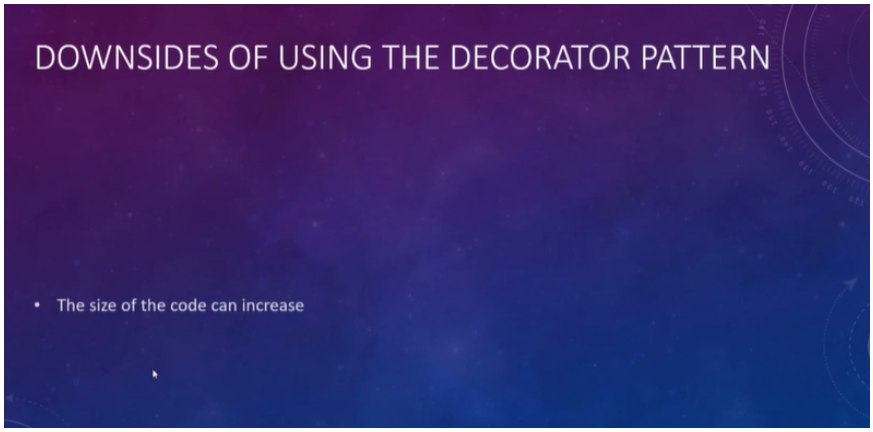


THE DECORATOR PATTERN

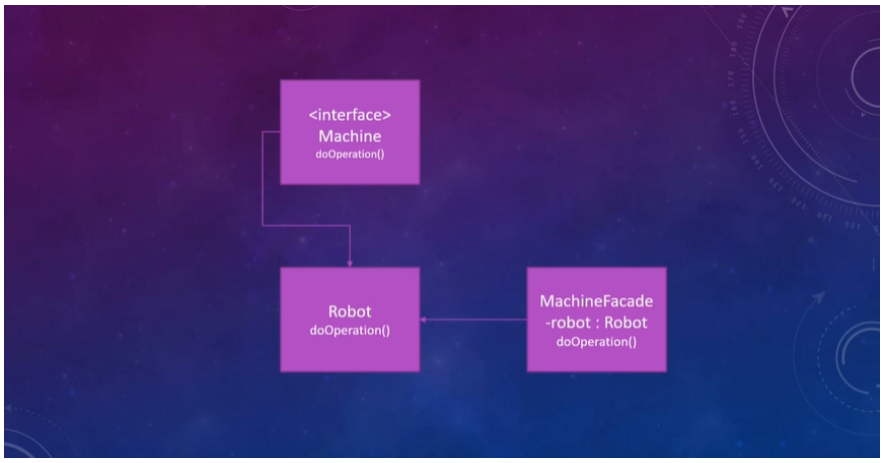




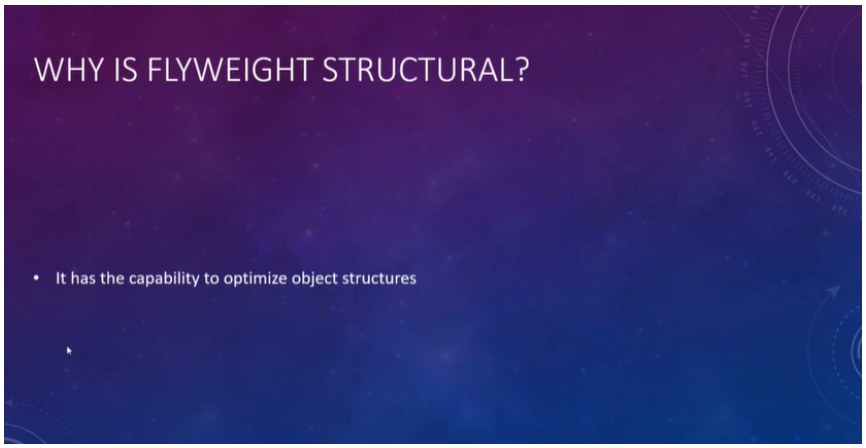


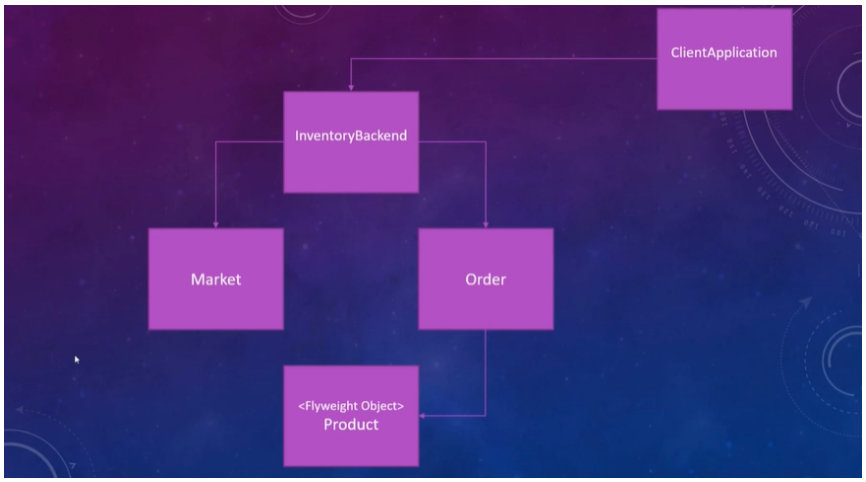


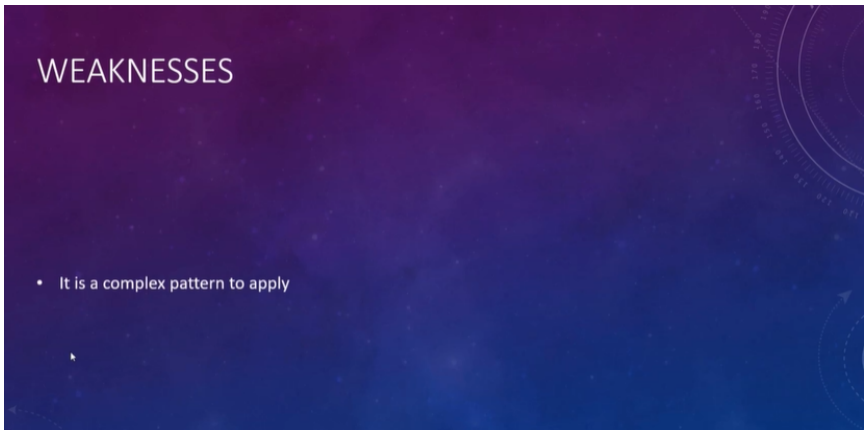
THE FACADE PATTERN



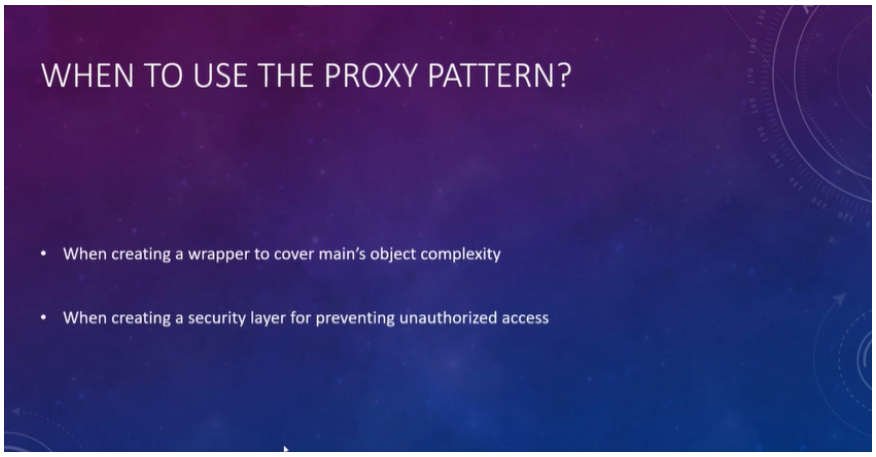
THE FLYWEIGHT PATTERN

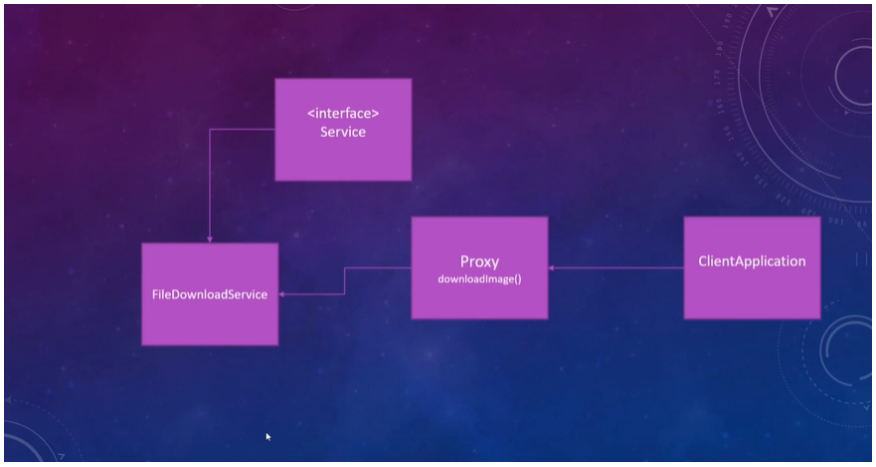


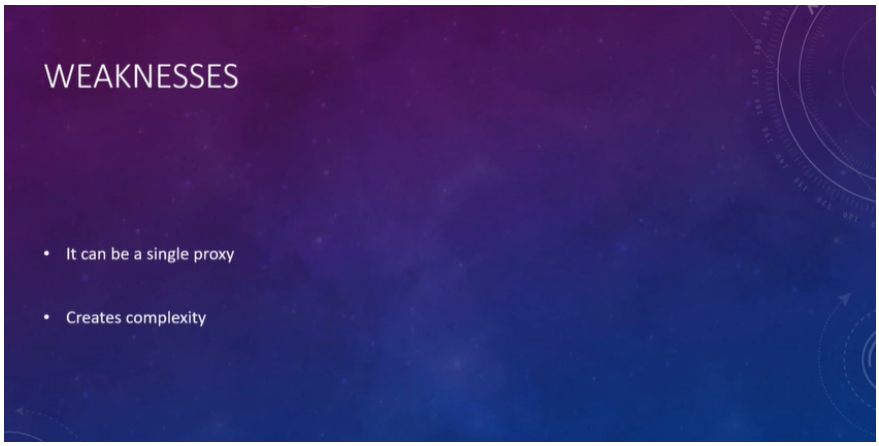




THE PROXY PATTERN







Quiz

Question 1:

You have a very complex application, and your objective is to hide the complexities from the client. Which design pattern would be the most suitable in this situation?

- ◯ Chain of Responsibility
- ◯ Facade pattern
- ◯ Bridge pattern

Question 2:

Which design pattern looks very similar to a Tree data structure?

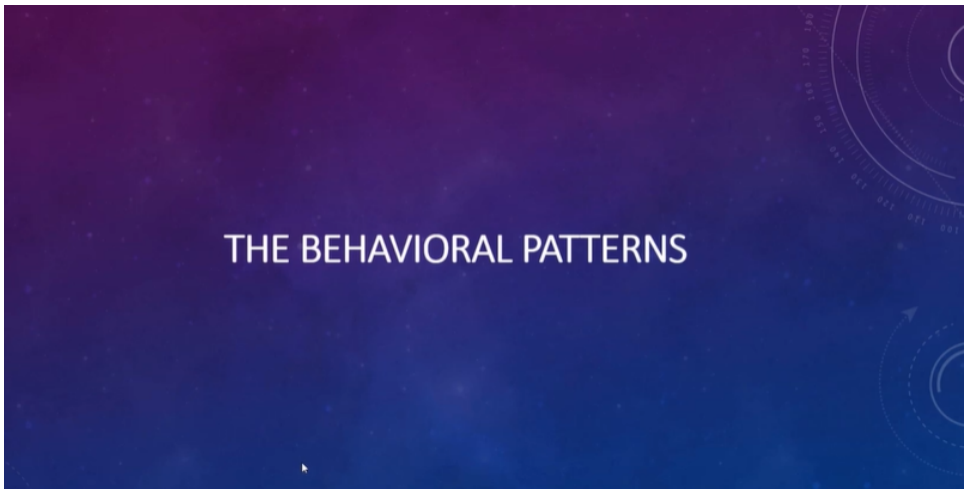
- ◯ Facade pattern
- ◯ The Proxy pattern.
- ◯ The Composite pattern.

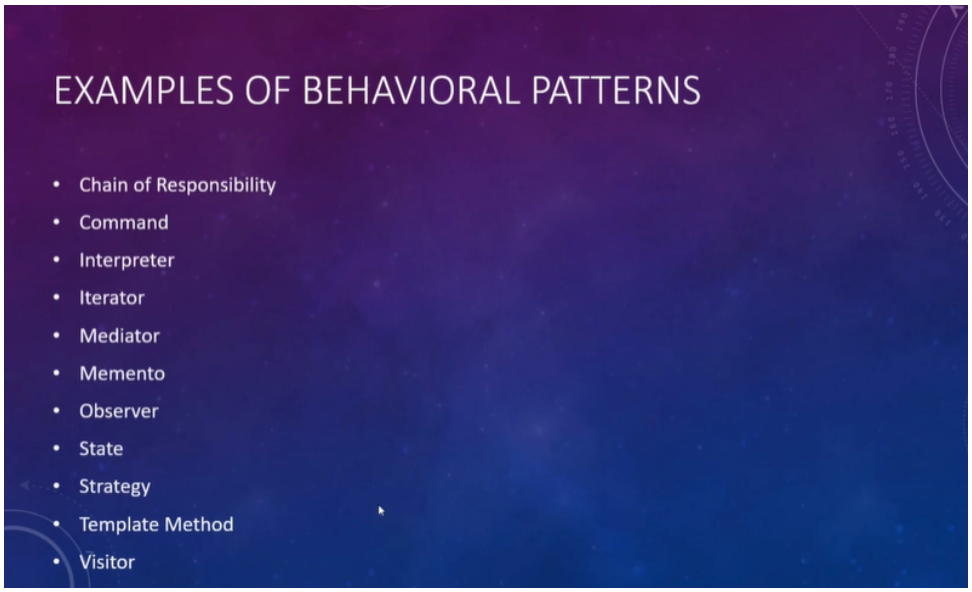
Question 3:

What would be the best pick for adding a security layer to prevent the access of unathorized users to your API?

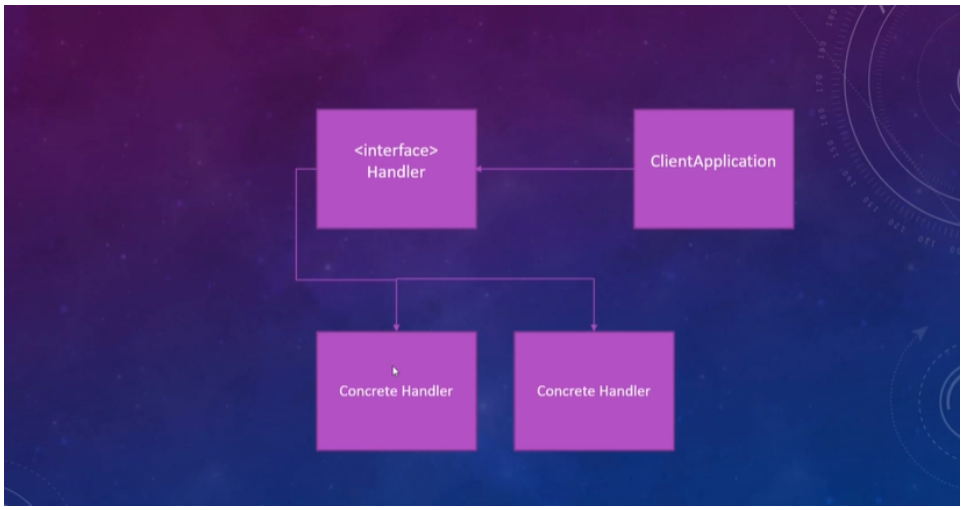
- ◯ The Decorator pattern.
- ◯ The Facade pattern.
- ◯ The proxy pattern.

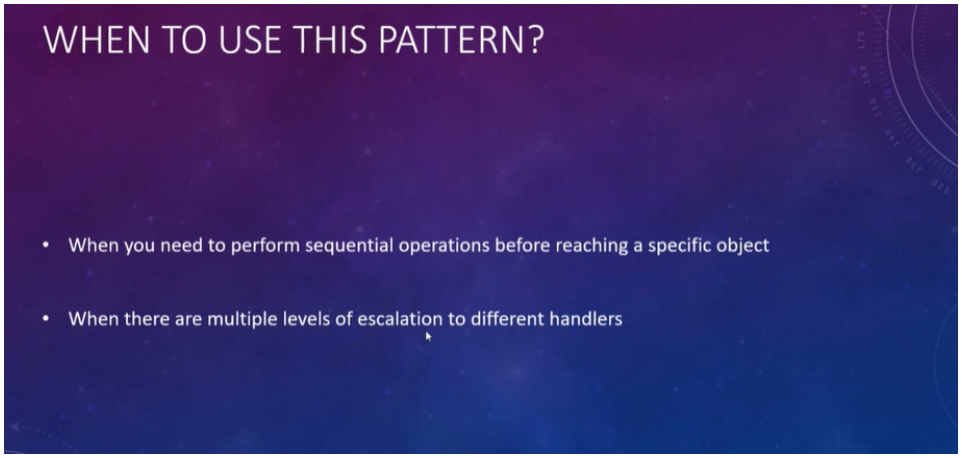
BEHAVIOURAL PATTERNS



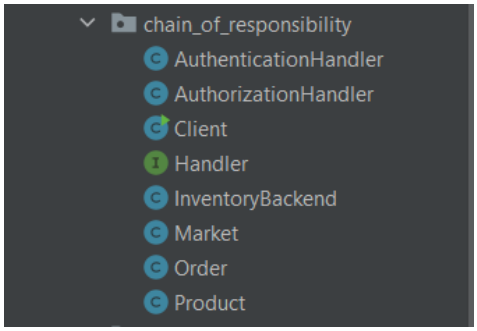


THE CHAIN OF RESPONSIBILITY PATTERN





Bu projede çalışması gereken iki Handler var. Authentication ve Authorization



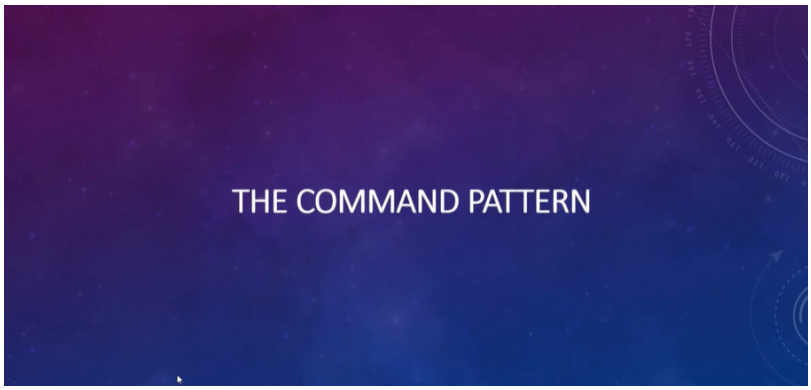
Client sadece bir handler'ı çalıştırıyor.

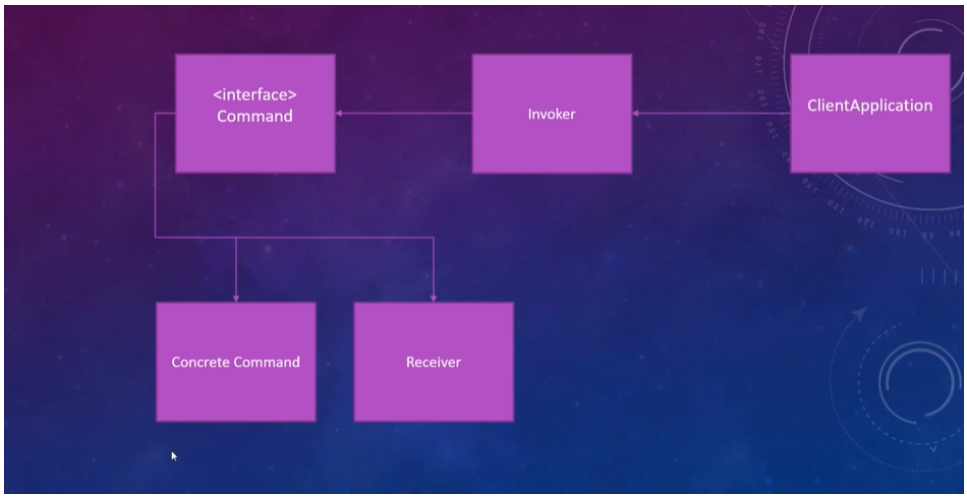
```
public class Client {
    public static void main(String[] args) {
        Handler handler = new AuthenticationHandler();

        handler.handleRequest();

        InventoryBackend inventoryBackend = new InventoryBackend();

        inventoryBackend.takeOrder( productName: "Iphone",  orderNumber: 1);
        inventoryBackend.takeOrder( productName: "Laptop",  orderNumber: 2);
        inventoryBackend.takeOrder( productName: "Smart TV",  orderNumber: 3);
        inventoryBackend.takeOrder( productName: "Iphone",  orderNumber: 4);
        inventoryBackend.takeOrder( productName: "Laptop",  orderNumber: 5);
        inventoryBackend.takeOrder( productName: "Smart TV",  orderNumber: 6);

    }
}
```

Çalıştırılan handler kendi işini yapıp, kendisinden sonraki handleri çağırıyor.

```java
public class AuthenticationHandler implements Handler {
    AuthorizationHandler authorizationHandler = new AuthorizationHandler();

    @Override
    public void handleRequest() {
        System.out.println("Authentication is successful");
        checkAuthorization();
    }

    private void checkAuthorization() { authorizationHandler.handleRequest(); }
}
```

THE COMMAND PATTERN





Command pattern is a behavioral design pattern which is useful to abstract business logic into discrete actions which we call commands.

This command object helps in loose coupling between two classes where one class (invoker) shall call a method on other class (receiver) to perform a business operation.

Design Participants

Participants for command design pattern are:

Command interface – for declaring an operation.

Concrete command classes – which extends the Command interface, and has execute method for invoking business operation methods on receiver. It internally has reference of the receiver of command.

Invoker – which is given the command object to carry out the operation.

Receiver – which execute the operation.

In command pattern, the invoker is decoupled from the action performed by the receiver. The invoker has no knowledge of the receiver. The invoker invokes a command, and the command executes the appropriate action of the receiver. Thus, the invoker can invoke commands without knowing the details of the action to be performed. In addition, this decoupling means that changes to the receiver's action don't directly affect the invocation of the action.

Problem Statement

Suppose we need to build a remote control for home automation system which shall control different lights/electrical units of the home. A single button in remote may be able to perform same operation on similar devices e.g. a TV ON/OFF button can be used to turn ON/OFF different TV set in different rooms.

Here this remote will be a programmable remote and it would be used to turn on and off various lights/fan etc.

Command Pattern Implementation

Let's solve above home automation problem with command design pattern and design each component one at a time.

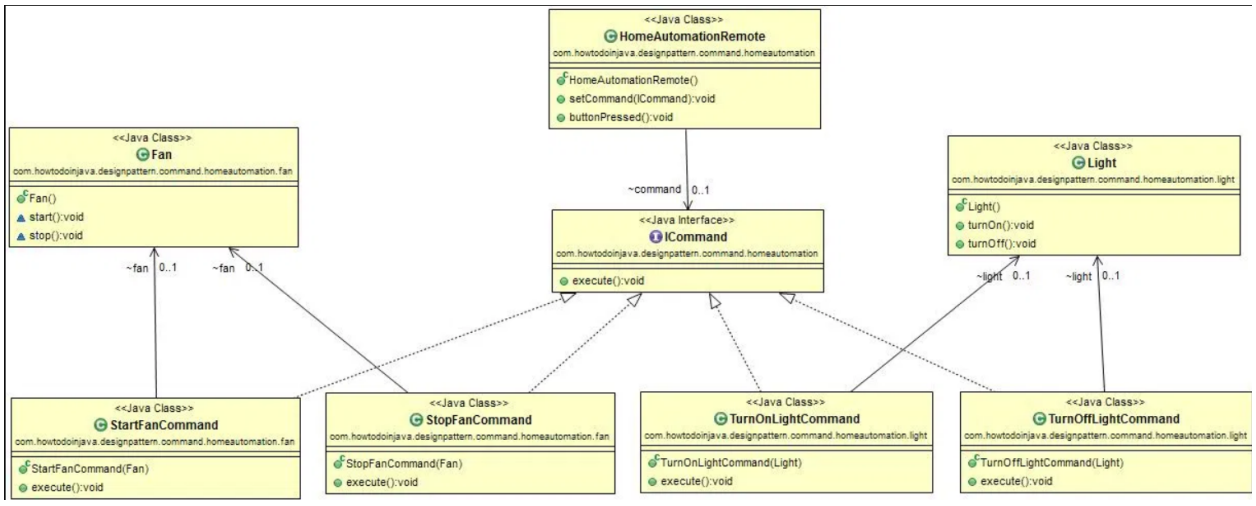
ICommand interface which is the **command interface**

Light is one of a **receiver** component. It can accept multiple commands related to Light like turn on and off

Fan is also another type of a **receiver** component. It can accept multiple commands related to Fan like turn on and off

HomeAutomationRemote is the **invoker** object, which asks the command to carry out the request. Here Fan on/off, Light on/off.

StartFanCommand,StopFanCommand,TurnOffLightCommand,TurnOnLightCommand etc. are different type of **command implementations**.



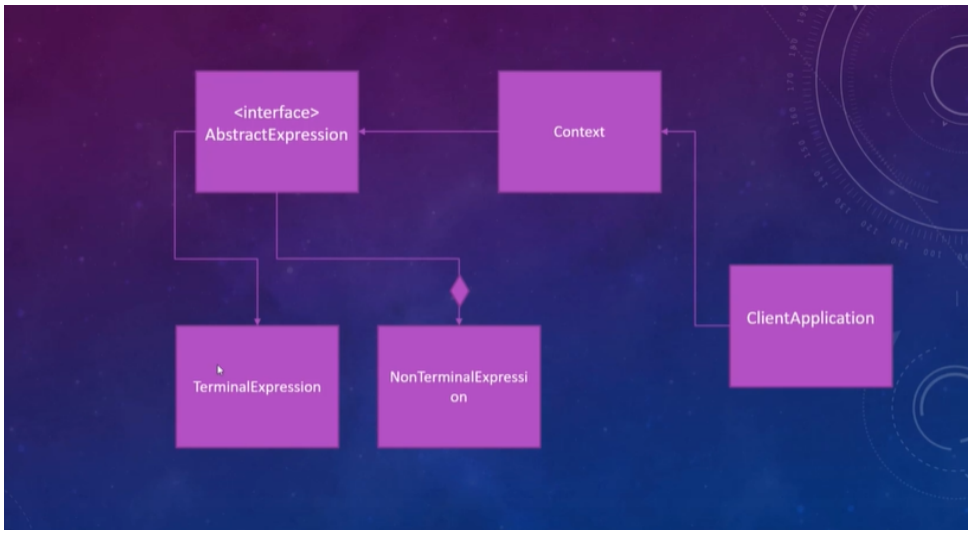
THE INTERPRETER PATTERN

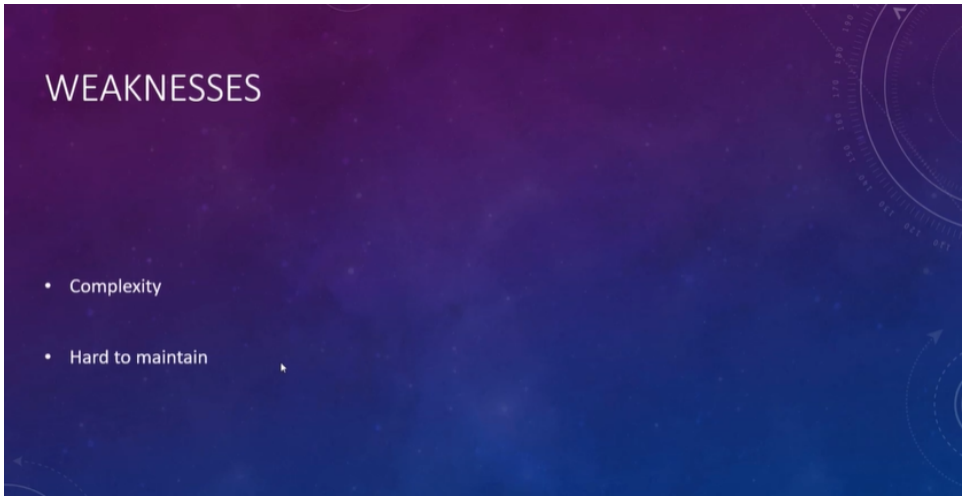
Interpreter pattern is used to defines a grammatical representation for a language and provides an interpreter to deal with this grammar.

To implement interpreter pattern, we need to create Interpreter context engine that will do the interpretation work.

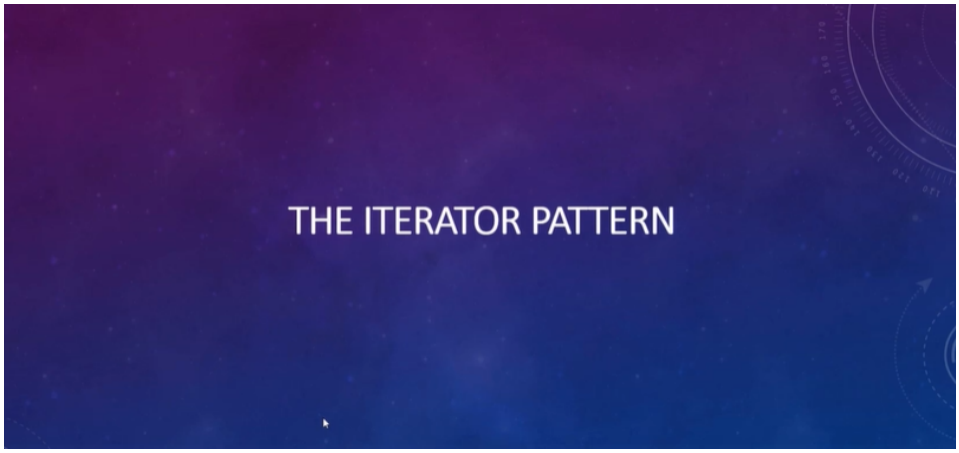
Then we need to create different Expression implementations that will consume the functionalities provided by the interpreter context.

Finally we need to create the client that will take the input from user and decide which Expression to use and then generate output for the user.

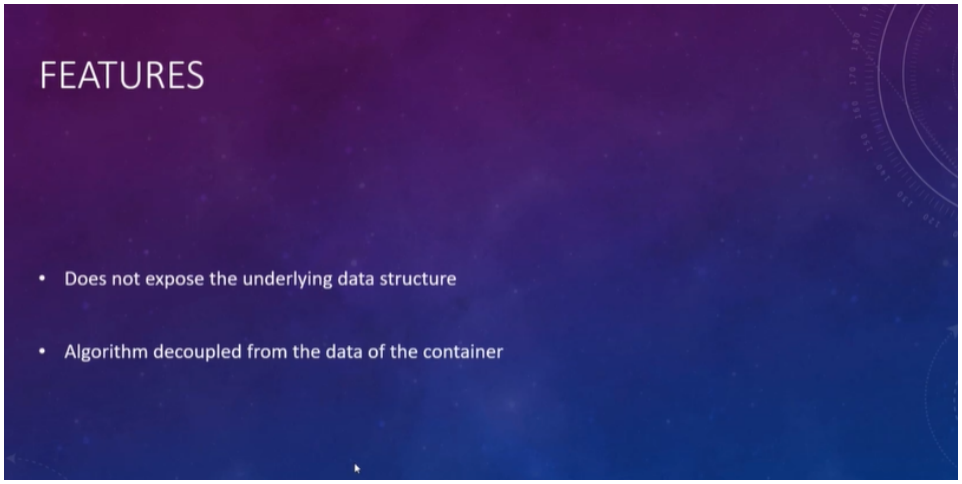




THE ITERATOR PATTERN



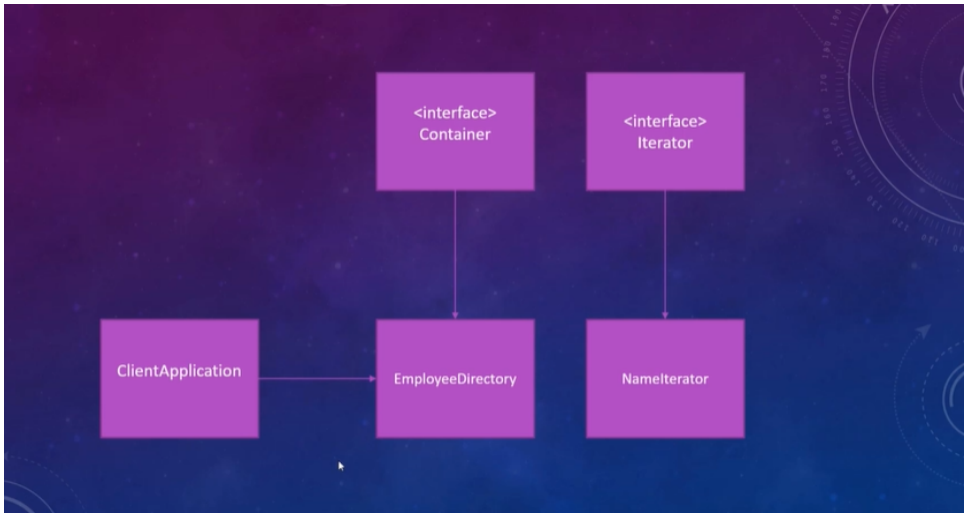
An iterator pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation.



The iterator pattern allow us to design a collection iterator in such a way that;

- we are able to access elements of a collection without exposing the internal structure of elements or collection itself.
- iterator supports multiple simultaneous traversals of a collection from start to end in forward, backward or both directions.
- iterator provide a uniform interface for traversing different collection types transparently.

The key idea is to take the responsibility for access and traversal out of the aggregate object and put it into an Iterator object that defines a standard traversal protocol.



THE MEDIATOR PATTERN

Mediator helps in establishing **loosely coupled communication** between objects and helps in reducing the direct references to each other. This helps in minimizing the complexity of dependency management and communications among participating objects.

Mediator helps to facilitate the interaction between objects in a manner in that **objects are not aware of the existence of other objects**. Objects depend only on a single mediator class instead of being coupled to dozens of other objects.

During designing a solution to a problem, if you encounter a situation where multiple objects need to interact with each other to process the request, but direct communication may create a complex system, you can consider using mediator pattern.

The pattern lets you extract all the relationships between classes into a separate class, isolating any changes to a specific component from the rest of the components.

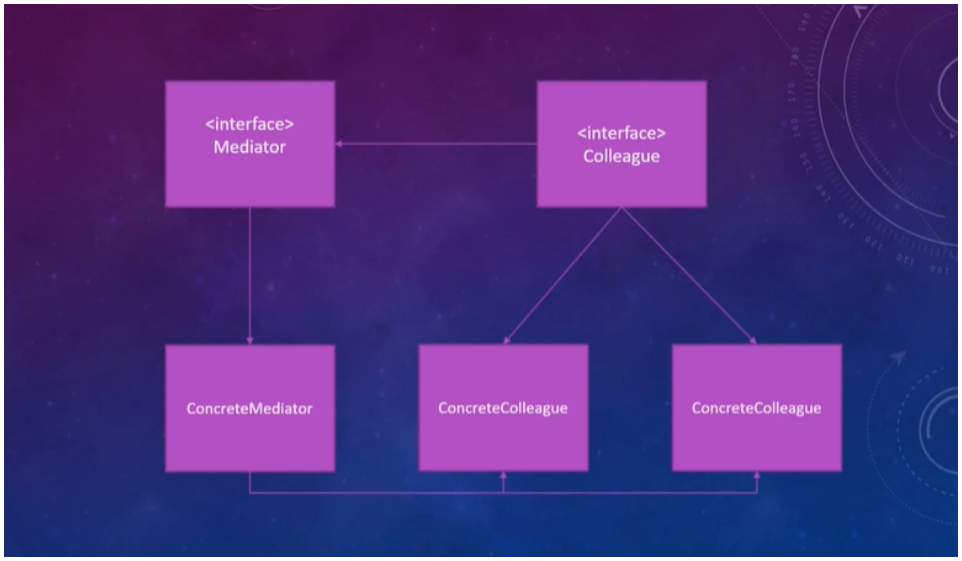
Real world example of mediator pattern

A great real world example of mediator pattern is **traffic control room** at airports. If all flights will have to interact with each other for finding which flight is going to land next, it will create a big mess.

Rather flights only send their status to the tower. These towers in turn send the signals to conform which airplane can take-off or land. We must note that these towers do not control the whole flight. They only enforce constraints in the terminal areas.

WHEN TO USE THE MEDIATOR PATTERN?

- When having too many relationships between objects
- When communication between objects is too complex

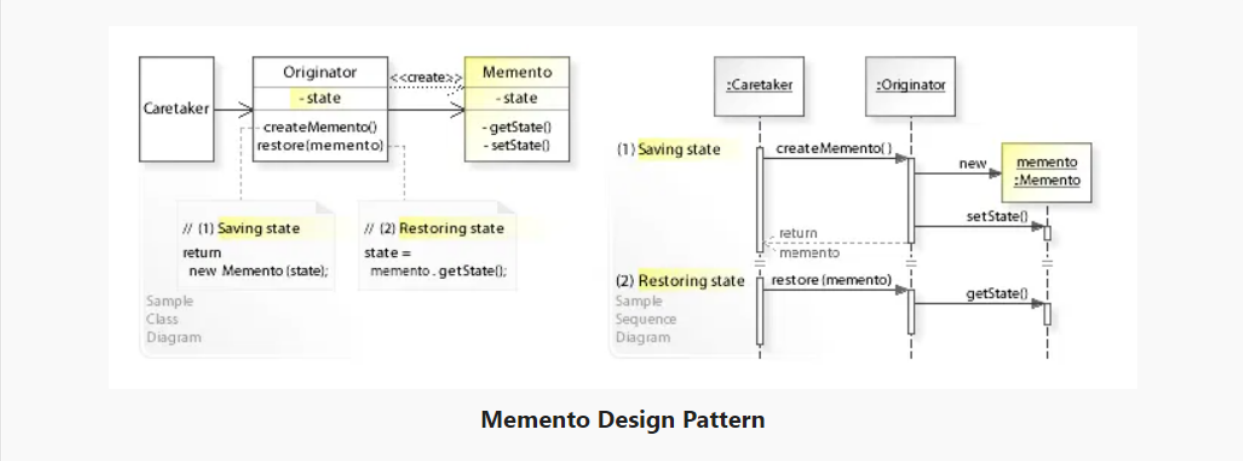


THE MEMENTO PATTERN

Memento pattern is used to **restore state** of an object to a previous state. It is also known as snapshot pattern.

A memento is is like a restore point during the life cycle on the object, which the client application can use to restore the object state to its state. Conceptually, it is very much like we create restore points for operating systems and use to restore the system if something breaks or system crashes.

The intent of memento pattern is to capture the internal state of an object without violating encapsulation and thus providing a mean for restoring the object into initial state when needed.



Memento Design Pattern

Design participants

The memento pattern has three participants.

Originator – is the object that knows how to create and save it's state for future. It provides methods createMemento() and restore(memento).

Caretaker – performs an operation on the Originator while having the possibility to rollback. It keeps track of multiple mementos. Caretaker class refers to the Originator class for saving (createMemento()) and restoring (restore(memento)) originator's internal state.

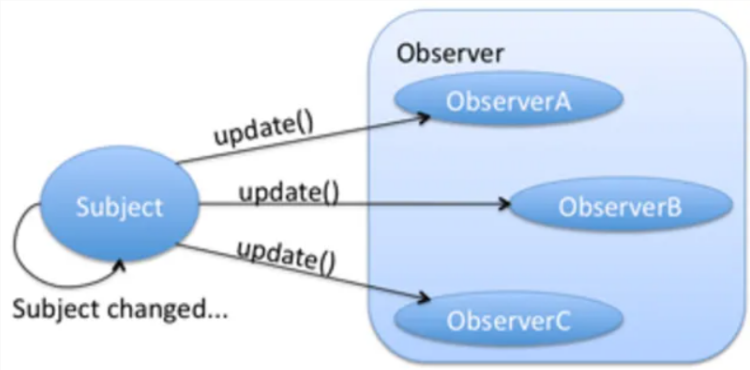
Memento – the lock box that is written and read by the Originator, and shepherded by the Caretaker. In principle, a memento must be in immutable object so that no one can change it's state once created.

THE OBSERVER PATTERN

Observer pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. It is also referred to as the **publish-subscribe pattern**.

In observer pattern, there are many observers (subscriber objects) that are observing a particular subject (publisher object). Observers register themselves to a subject to get a notification when there is a change made inside that subject.

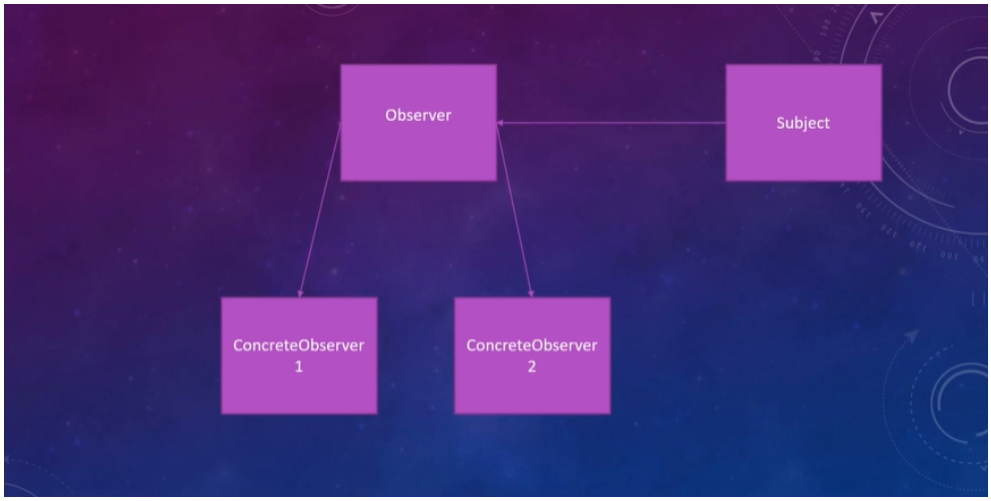
A observer object can register or unregister from subject at any point of time. It helps is making the objects objects **loosely coupled**.

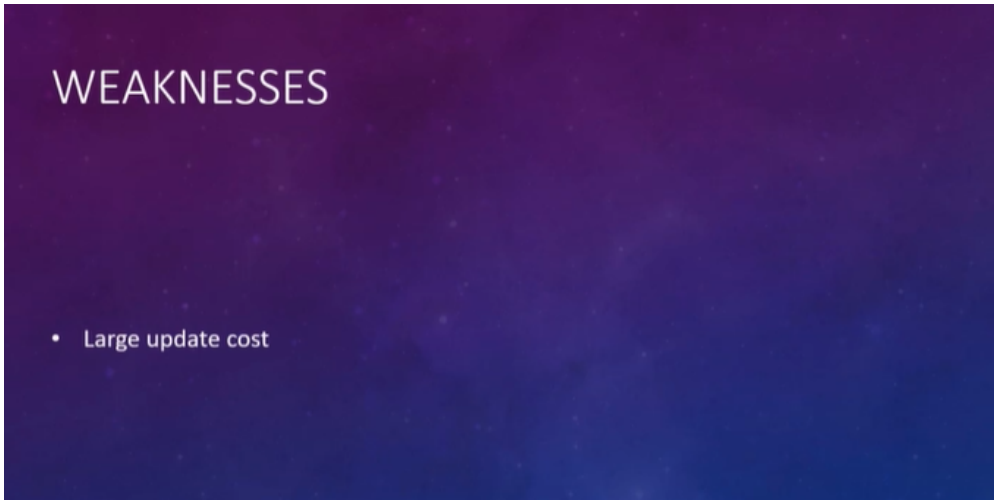


Observer Pattern

FEATURES

- Observers and Subject objects
- Java.awt.Button





THE STATE PATTERN

A state allows an object to alter its behavior when its internal state changes. The object will appear to change its class.

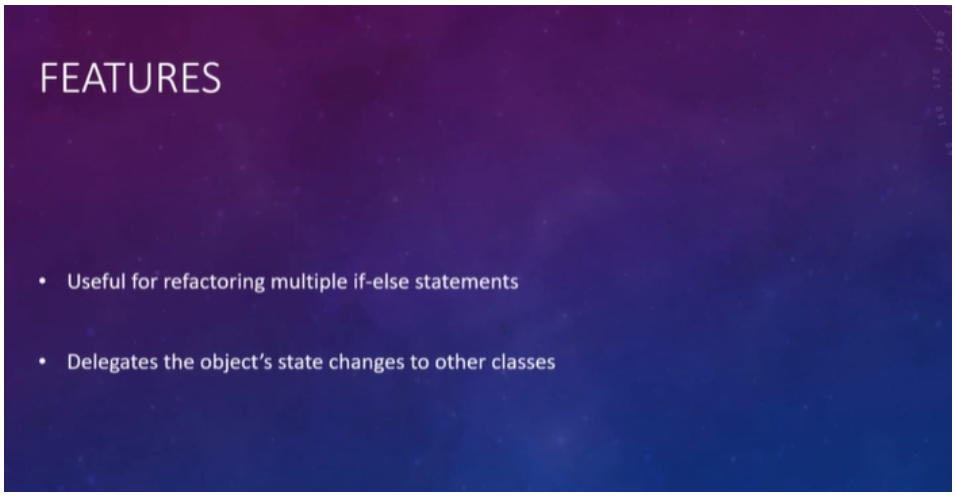
Then, there shall be a separate concrete class per possible state of an object. Each concrete state object will have logic to accept or reject a state transition request based on it’s present state and context information passed to it as method arguments.

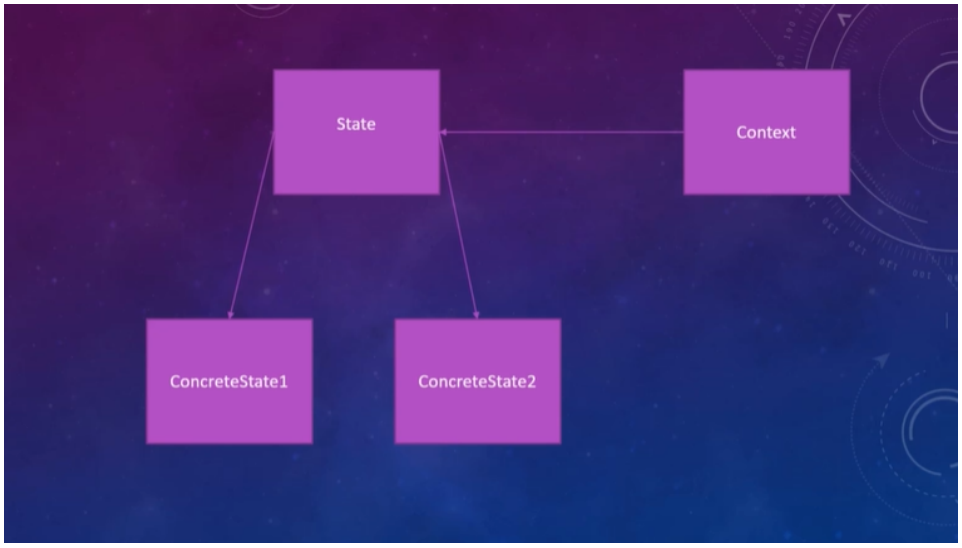
When to use state pattern

In any application, when we are dealing with **an object which can be in different states during it's life-cycle** and how it processes incoming requests (or make state transitions) based on it's present state – we can use the state pattern.

If we do not use the state pattern in such case, we will end up having lots of **if-else** statements which make the code base ugly, unnecessarily complex and hard to maintain. State pattern allows the objects to behave differently based on the current state, and we can define state-specific behaviors within different classes.

The state pattern solves problems where an object should change its behavior when its internal state changes. Also, adding new states should not affect the behavior of existing states.





THE STRATEGY PATTERN

Strategy design pattern is behavioral design pattern where we choose a specific implementation of algorithm or task in run time – out of multiple other implementations for same task.

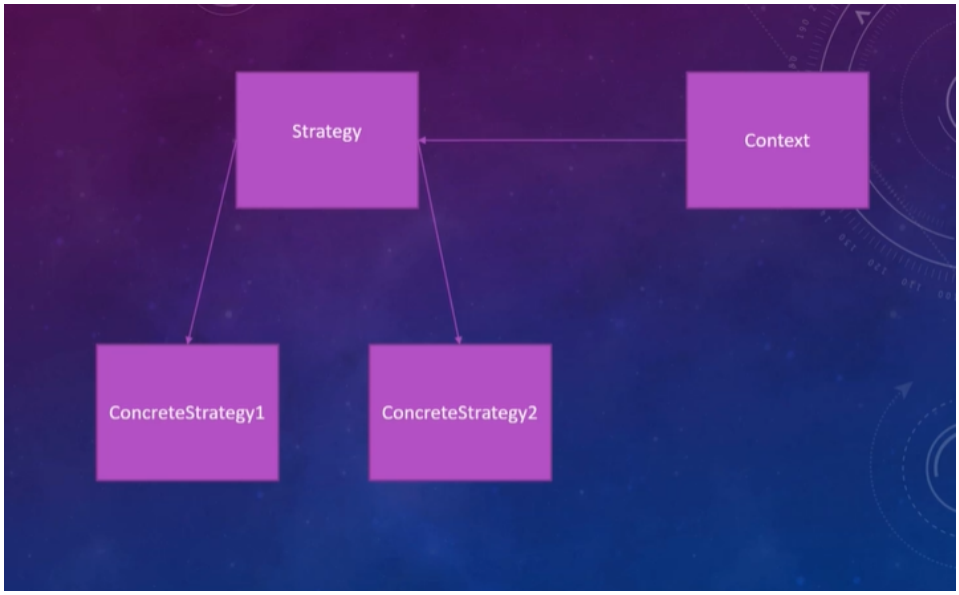
The important point is that these implementations are interchangeable – based on task a implementation may be picked without disturbing the application workflow.

Strategy pattern involves removing an algorithm from its host class and putting it in separate class so that in the same programming context there might be different algorithms (i.e. strategies), which can be selected in runtime.

Strategy pattern enables a client code to choose from a family of related but different algorithms and gives it a simple way to choose any of the algorithm in runtime depending on the client context.

FEATURES

- Allows you to choose algorithms at runtime
- Useful for refactoring multiple if-else statements



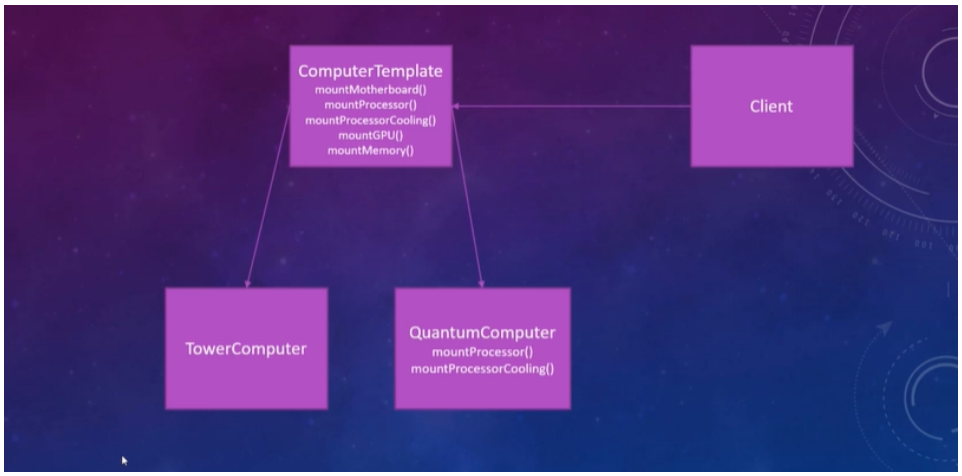
WEAKNESSES

- Client needs to know the strategies
- Big number of classes

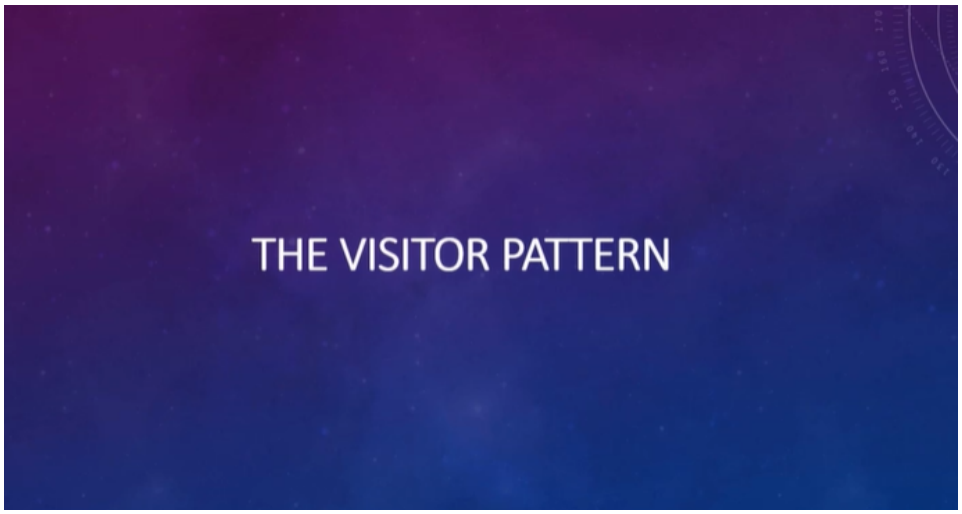
THE TEMPLATE METHOD PATTERN

Template method design pattern is widely accepted behavioral design pattern to enforce some sort of algorithm (fixed set of steps) in the context of programming.

It defines the sequential steps to execute a multi-step algorithm and optionally can provide a default implementation as well (based on requirements).

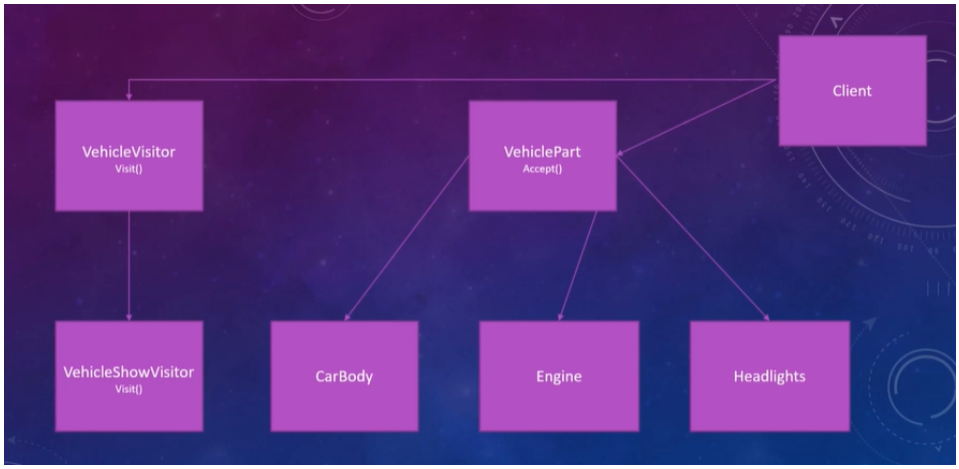


THE VISITOR PATTERN



The visitor design pattern is a way of separating an algorithm from an object structure on which it operates. A practical result of this separation is the ability to add new operations to existing object structures **without modifying** those structures.

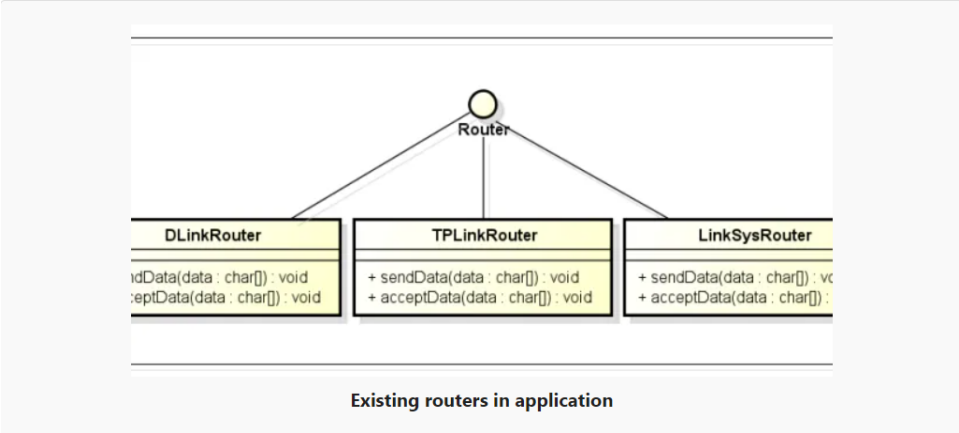
This design flexibility allows to add methods to any object hierarchy without modifying the code written for hierarchy.



Sample problem to solve

Suppose we have an application which manage routers in different environments. Routers should be capable of sending and receiving char data from other nodes and application should be capable of configuring routers in different environment.

Essentially, design should flexible enough to support the changes in way that routers can be configured for additional environments in future, without much modifications in source code.



Existing routers in application

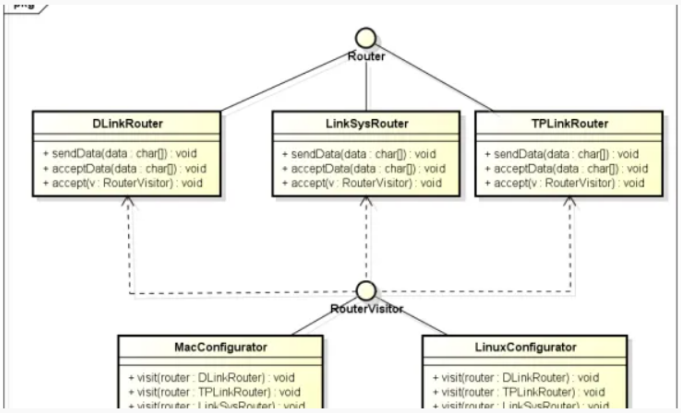
We have above 3 types of routers and we need to write code for them. One way to solve this problem, is to define methods like configureForWinodws() and configureForLinux() in Router.java interface and implement them in different products, cause each will have it's own configuration setting and procedure.

But the problem with above approach is that each time a new environment is introduced, whole router's hierarchy will have to be compiled again.

Solution using Visitor pattern

Visitor pattern is good fit for these types of problems where you want to introduce a new operation to hierarchy of objects, without changing its structure or modifying them. In this solution, we will implement double dispatch technique by introducing two methods i.e. accept() and visit() methods. Method accept(), will be defined in router's hierarchy and and visit methods will be on visitors level.

Whenever a new environment need to be added, a new visitor will be added into visitors hierarchy and that needs to implement visit() method for all the available routers and that's all.



Solution using visitor pattern

In above class diagram, we have routers configured for Mac and Linux operating systems. If we need to add the capability for windows also, then I do not need to change any class, just define a new visitor WindowsConfigurator and implement the visit() methods defined in RouterVisitor interface. It will provide the desired functionality without any further modification.