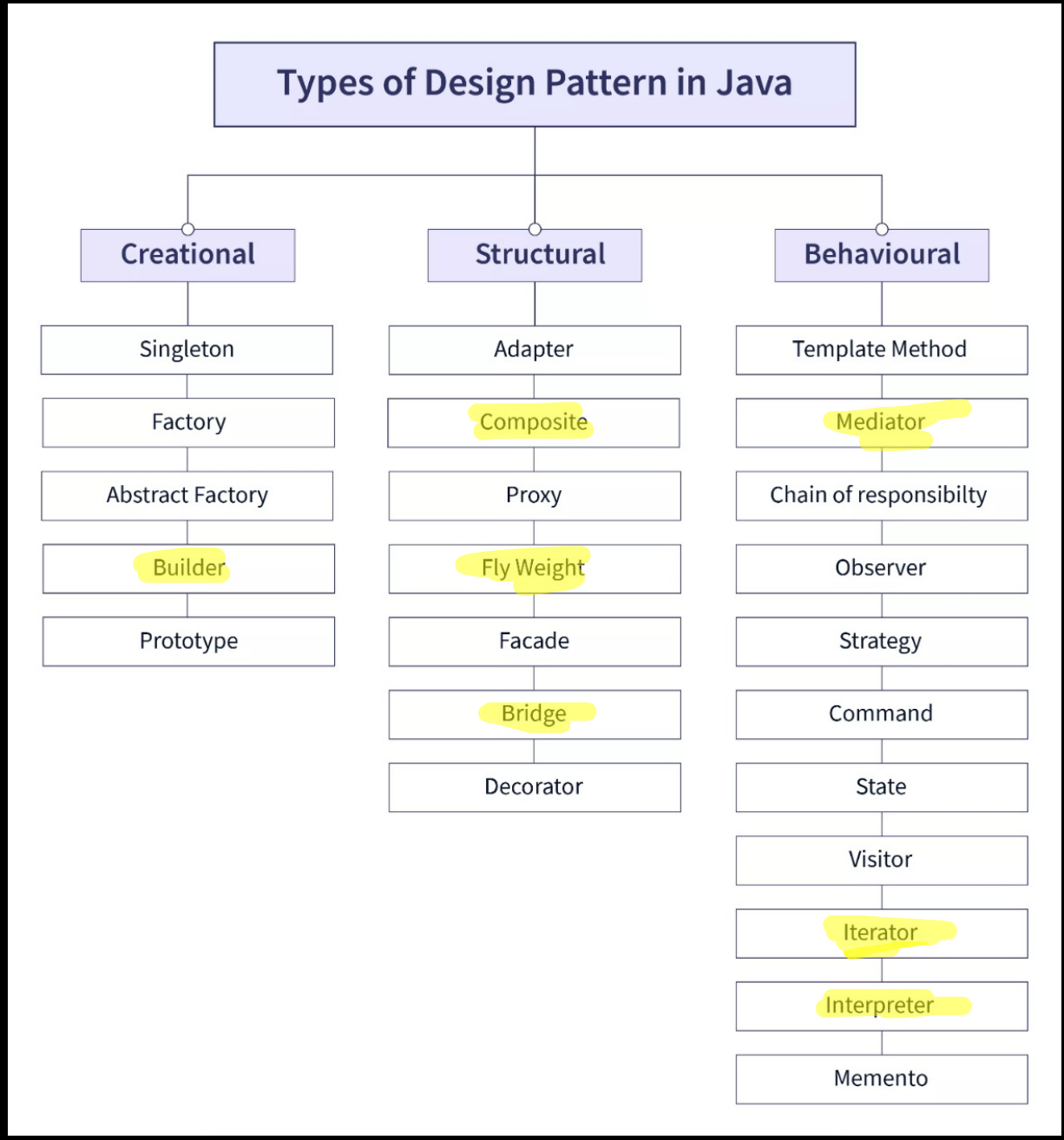
Design Patterns

The Gang of Four authors observed some commonly occurring problems in object-oriented software design and came up with some descriptive solutions to those recurring problems. These solutions to repeating problems in software design are known as Design Patterns. Each design pattern is a template or a description that provides a way to structure the classes and objects in object-oriented software.

These design patterns were a result of well-tested and proven object-oriented design techniques that can be used to solve a particular design problem. As a result, the design patterns are considered a standard in object-oriented software design i.e., if you want to build an object-oriented software using best practices, then you should follow the Software Design Patterns.


Types of Design Pattern in Java
Creational
Structural
Behavioural
Singleton
Factory
Abstract Factory
Builder
Prototype
Adapter
Composite
Proxy
Fly Weight
Facade
Bridge
Decorator
Template Method
Mediator
Chain of responsibilty
Observer
Strategy
Command
State
Visitor
Iterator
Interpreter
Memento

Creational Patterns :- The traditional method of object creation can lead to the instantiation of objects in an uncontrolled fashion. It can scatter objects all over the application which can increase the overall application design complexity. This can make an object-oriented application unstable and can lead to design problems. Creational Design Patterns reduce this design instability and complexity by providing different ways to implement controlled object creation in an object-oriented application.

Structural Patterns :- Structural Design Patterns provide different ways to organize the classes and objects in an object-oriented application such that it leads to simplification of the design of large and complex structures. It describes the process using which we can combine classes and objects to form an organized, flexible, and connected structure.

Behavioral Patterns :- Behavioral design patterns deal with the interaction of objects with each other. These patterns follow the fact that the objects in an object-oriented application should be interconnected in such a way that hard coding can be avoided and the user input can be well handled. These design patterns make use of loose coupling techniques to ensure a flexible and effective flow of information.

⑦ Singleton Design Pattern

Applications :→

Singleton pattern might be useful if the three conditions are met. However, these might not be **exhaustive with complex use** cases and applications.

- You need to **control concurrent access** to a shared resource.
- Only one instance of the **object is sufficient throughout the context** of the application.
- **More than one independent parts** of the application require access to the resource.

There are many objects we only need one of: thread pools, caches, dialog boxes, objects that handle preferences and registry settings, objects used for logging, and objects that act as device drivers to devices like printers and graphics cards. In fact, for many of these types of objects, if we were to instantiate more than one we'd run into all sorts of problems like incorrect program behavior, overuse of resources, or inconsistent results.

Singleton

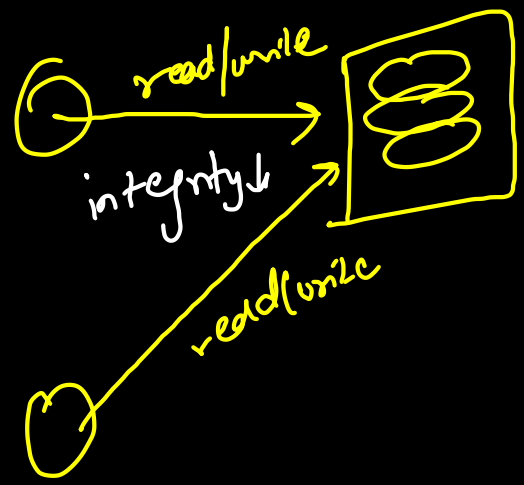
↳ Class → Object Creation (controlled) → single instance/object

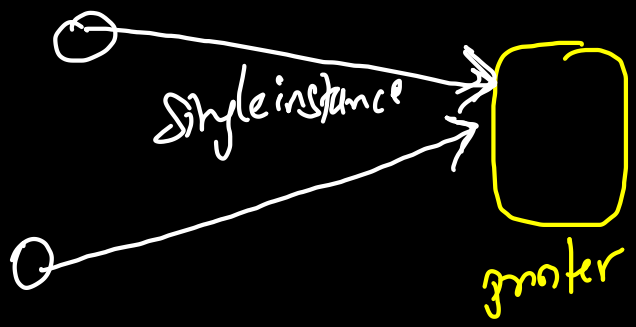
example

→ database connection

→ logging systems {Devops}

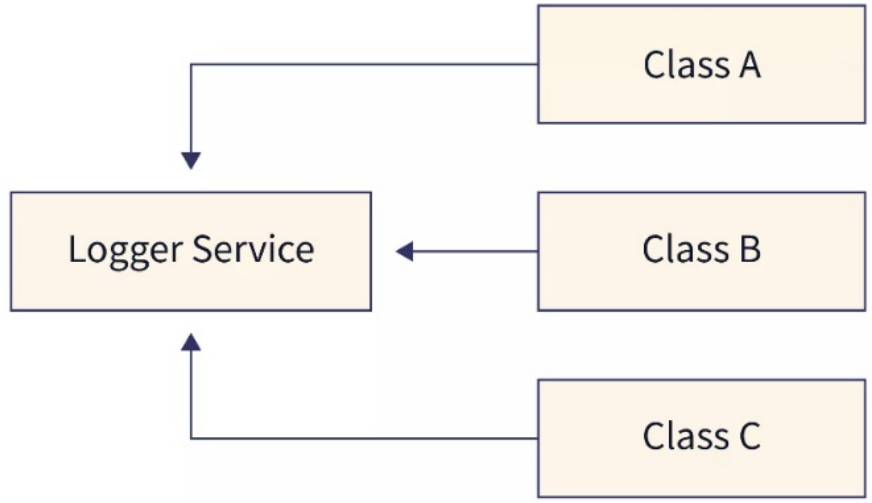
→ device drivers
eg printer





Example :→

Consider we want to create a LoggerService class which we can use across our project to log. Almost all the classes will use the LoggerService to log info, warn, error, etc. If every class creates a new LoggerService instance, it will occupy more memory, leading to an out-of-memory error. Singleton pattern, in this case, ensures that only one instance of this class is created and accessible globally throughout the project context.



Structure :→

Singleton

Singleton is a class with only one instance, and it is globally accessible by other classes across our application. E.g., LoggerService.

How to make a class Singleton?

- It should have a **private constructor** so that no instance of it can be created by other classes.
- It should have a **public static** method so that the it can be called (SingletonClass.getInstance()) to get its singleton instance

Client

The client interacts with the singleton class to get its shared instance and use it to call its methods. E.g., the log() method of the LoggerService singleton is used for logging messages.

Implementation

1. Create the singleton class LoggerService with a private constructor and a public static method to get the instance.
2. Add a public log() method to the LoggerService class used to log messages.
3. Create the client class Client that uses the LoggerService singleton instance to log messages.

Approaches

1. Static Initialization
2. Lazy Initialization
3. synchronized method
4. Synchronized block {double locking}
5. using Enum

```java
class LoggerService {
    private static LoggerService instance = new
    LoggerService();

    private LoggerService() {
    }

    public static LoggerService getInstance() {
        return instance;
    }

    public void log(String str) {
        System.out.println(str);
    }
}

class Client {
    Run | Debug
    public static void main(String[] args) {
        LoggerService log1 = LoggerService.getInstance();
        log1.log(str: "CPU utilization over 50%");

        LoggerService log2 = LoggerService.getInstance();
        log2.log(str: "RAM under 25% left");

        System.out.println(log1 == log2); // true
    }
}
```

Static Initializar

↳ class loading

↳ Resource utilization ↑

Approach (1)
using Static Initializar

```java
// Solution 2: Lazy Propogation
class LoggerService {
    private static LoggerService instance;

    private LoggerService() {

    }

    public static LoggerService getInstance() {
        if (instance == null) {
            instance = new LoggerService();
        }

        return instance;
    }

    public void log(String str) {
        System.out.println(str);
    }
}

class Client {
    Run | Debug
    public static void main(String[] args) {
        LoggerService log1 = LoggerService.getInstance();
        log1.log(str: "CPU utilization over 50%");

        LoggerService log2 = LoggerService.getInstance();
        log2.log(str: "RAM under 25% left");

        System.out.println(log1 == log2); // true
    }
}
```

Approach ② :→ Lazy Initialization

Object Creation only on getInstance

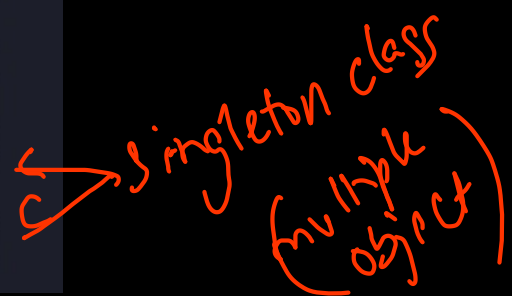
↳ Resource utilization ↓

Synchronization Problem

↳ multithreading

```java
class CustomThread extends Thread {
    @Override
    public void run() {
        LoggerService log = LoggerService.getInstance();
        log.log(str: "New Thread Log Created");
    }
}

class Client {
    public static void multithreaded() {
        CustomThread t1 = new CustomThread();
        t1.start();

        CustomThread t2 = new CustomThread();
        t2.start();
    }
```

```
New Thread Log Created
New Thread Log Created
LoggerService@56ed6baf
LoggerService@1a752056
```



```java
// Solution 3: Synchronized Method
class LoggerService {
    private static LoggerService instance;

    private LoggerService() {

    }

    public synchronized static LoggerService getInstance() {
        if (instance == null) {
            instance = new LoggerService();
        }

        return instance;
    }

    public void log(String str) {
        System.out.println(str);
        System.out.println(this);
    }
}
```

Synchronized Method
(CPU utilization ↓)

```
New Thread Log Created
New Thread Log Created
LoggerService@56ed6baf
LoggerService@56ed6baf
```

⇒ Single Instance

```java
// Solution 4: Double Locking: Synchronized Block
class LoggerService {
    private static LoggerService instance;

    private LoggerService() {

    }

    public static LoggerService getInstance() {
        if (instance == null) {
            synchronized (LoggerService.class) {
                if (instance == null) {
                    instance = new LoggerService();
                }
            }
        }

        return instance;
    }

    public void log(String str) {
        System.out.println(str);
        System.out.println(this);
    }
}
```

Double locking mechanism → Synchronized block

critical section output fast.

//Solution:5 Using Enum

```
enum LoggerServiceEnum {
    INSTANCE;

    public void log(String str) {
        System.out.println(str);
    }
}

class Client {
    Run | Debug
    public static void main(String[] args) {
        LoggerServiceEnum logger = LoggerServiceEnum.INSTANCE;
        logger.log(str: "CPU Utilization Over 75%");
        logger.log(str: "RAM Under 25%");
    }
}
```

equivalent

```
public final class LoggerEnum {
    public final static LoggerEnum LOGGER = new LoggerEnum();
    private LoggerEnum() {}
}
```

1. Lazy Initializam
2. multithreading safe
   Synchronized
3. short implementam

Pros and Cons of Singleton Design Pattern

Pros

- Singleton design pattern guarantees that only one instance of the class will be available **throughout the application context**. This ensures that you do not waste memory for a new object instance when you don't need one.
- Singleton design pattern might be handy when dealing with concurrent access to resources. Singletons can also provide thready safety.
- Singleton design pattern eliminates the **unnecessary** instantiation overhead, i.e., the extra memory required to create an object.

Cons

↗ scope & lifetime ⇒ static

- **Singleton design pattern** introduces a global state in the application. This makes unit testing difficult. The global state does not align well with unit testing because it increases coupling, and control over more than one unit is needed while writing the unit tests.
- Due to the resource being locked in a parallel **processing environment,** multi-threading in some cases can not be used to its full potential with singelton design pattern.
- Singleton design pattern solves less than it causes. This means this pattern has minimal practical use cases and, if used otherwise, can generate more problems. Some of these are also direct violations of other design patterns. **(e.g., Single Responsibility)**

② Adapter Design Pattern

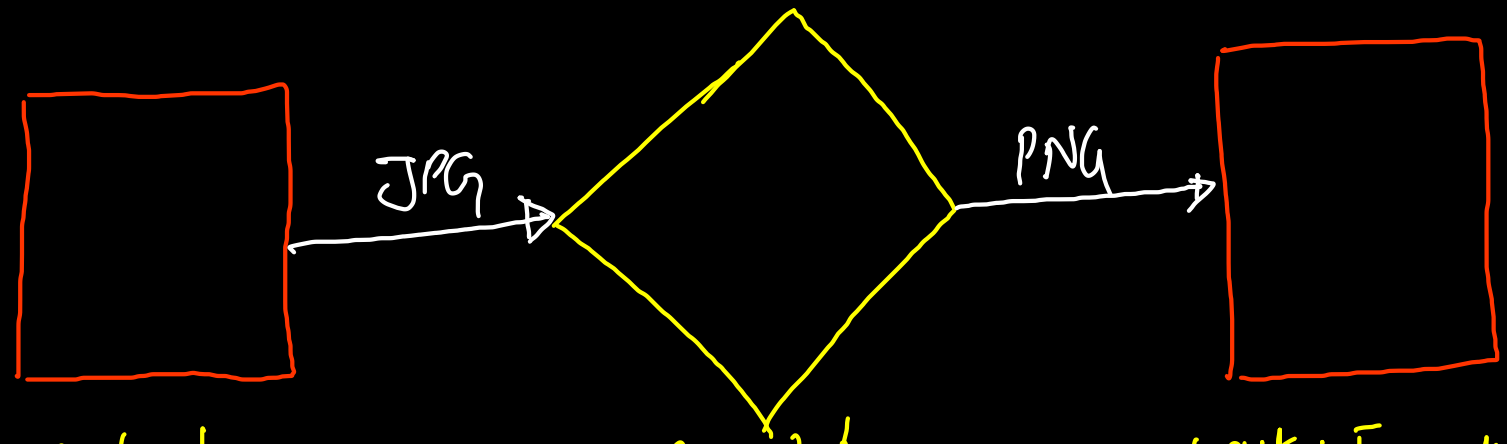
What ?

The **Adapter Design Pattern** is a structural design pattern and is also known as the **Wrapper Design Pattern**. This design pattern acts as a bridge between two different interfaces. It can convert the interface of a class, to make it compatible with a client who is expecting a different interface, without changing the source code of the class.

Examples :→

- → card reader for Computer & memory card
- → charging adapter { micro-USB to USB-Type C }
- → Convert JPG to PNG / JPEG
- → Convert XML to JSON.

Problem



Output Format 1: JPG image

Networking

Adapter

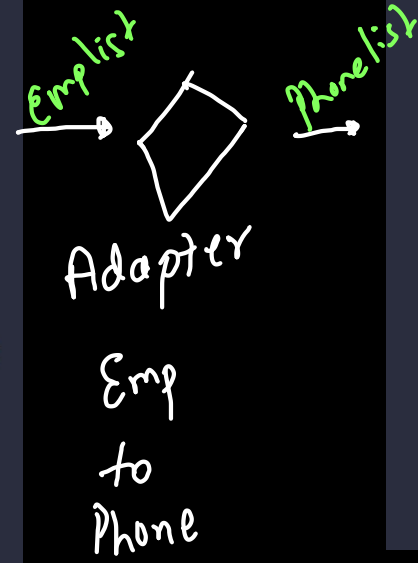
loosely coupling

Input : Format 2 ↳ PNG image

Graphic Illustration

Producer List of Employees

```java
package Adapter;

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

public class EmployeeDetails {
    public List<String> GetEmployees() {
        List<String> emps = new ArrayList<>();

        emps.add(e: "1-ABC-SDE1-9999786543");
        emps.add(e: "2-DEF-SDE2-9876523456");
        emps.add(e: "3-GHI-TL-9321499876");
        emps.add(e: "4-JKL-Architect-9985432121");
        emps.add(e: "5-MNO-HR-9211345667");

        return emps;
    }
}
```



Consumer List of PhoneNos

```java
// Data consumer
public class Intranet {
    List<String> source;

    public Intranet(List<String> source) {
        this.source = source;
    }

    public void printPhoneNumbers() {
        List<String> phones = this.source;
        System.out.println(phones);
    }
}
```

```java
public class EmptoPhoneNoAdapter {
    List<String> employees;

    public EmptoPhoneNoAdapter(List<String> employees) {
        this.employees = employees;
    }

    public List<String> convert() {
        List<String> phones = new ArrayList<>();

        for (String emp : employees) {
            String[] data = emp.split(regex: "-");
            phones.add(data[3]);
        }

        return phones;
    }
}
```

```java
package Adapter;

import java.util.*;

public class Client {
    Run | Debug
    public static void main(String[] args) {
        EmployeeDetails producer = new EmployeeDetails();
        List<String> employees = producer.GetEmployees();

        EmptoPhoneNoAdapter adapter = new EmptoPhoneNoAdapter
        (employees);

        List<String> phones = adapter.convert();

        Intranet consumer = new Intranet(phones);
        consumer.printPhoneNumbers();
    }
}
```

```
[9999786543, 9876523456, 9321499876, 9985432121, 9211345667]
```

Solution violating Dependency Inversion Principle (D in SOLID)

```java
package Adapter;

import java.util.*;

public interface IPhoneList {
    public List<String> getPhoneList();
}
```

Stud to PhoneNo Adapter

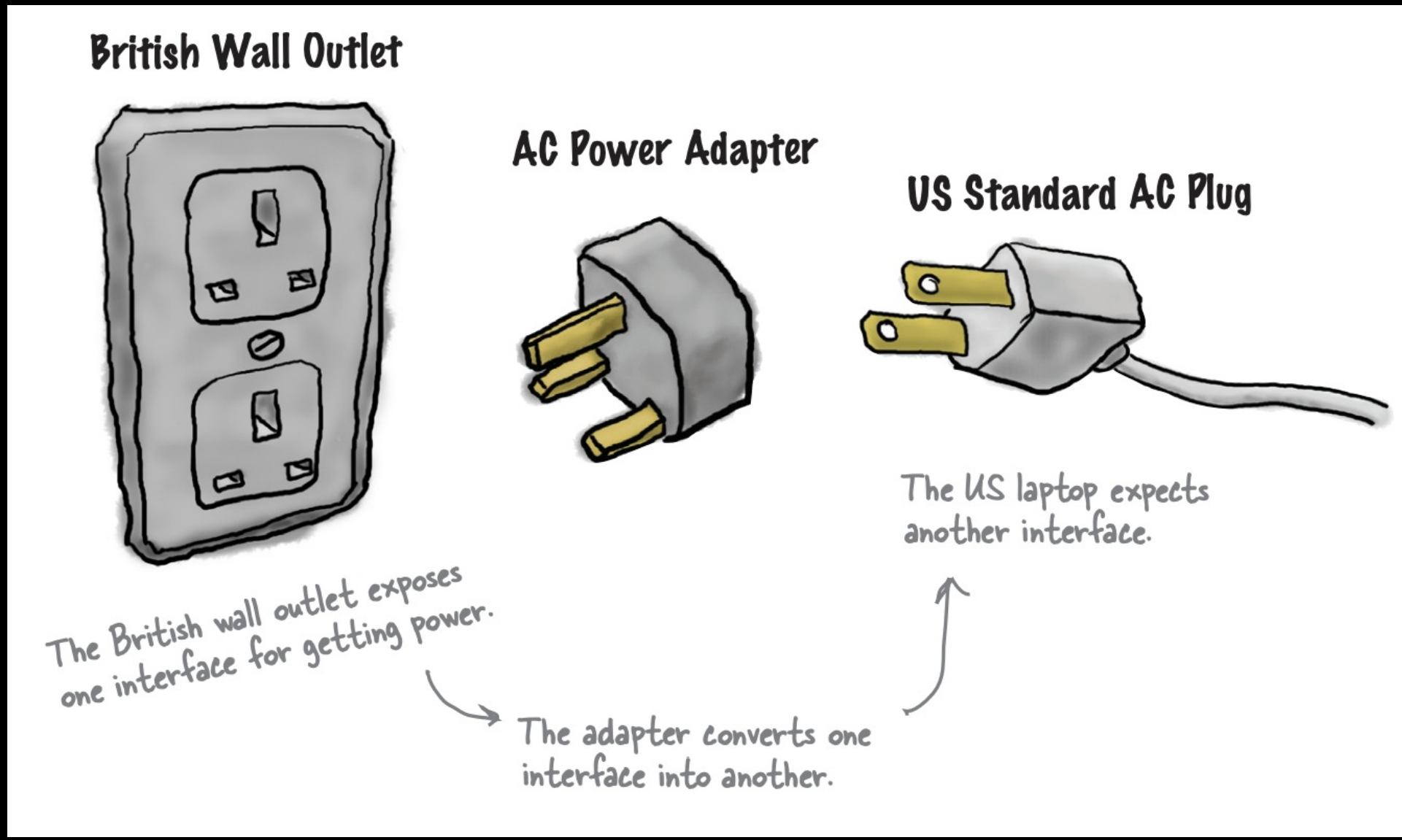
```java
package Adapter;

import java.util.*;

public class EmptoPhoneNoAdapter implements IPhoneList {
    List<String> employees;

    public EmptoPhoneNoAdapter(List<String> employees) {
        this.employees = employees;
    }
    @Override
    public List<String> getPhoneList() {
        List<String> phones = new ArrayList<>();

        for (String emp : employees) {
            String[] data = emp.split(regex: "-");
            phones.add(data[3]);
        }

        return phones;
    }
}
```

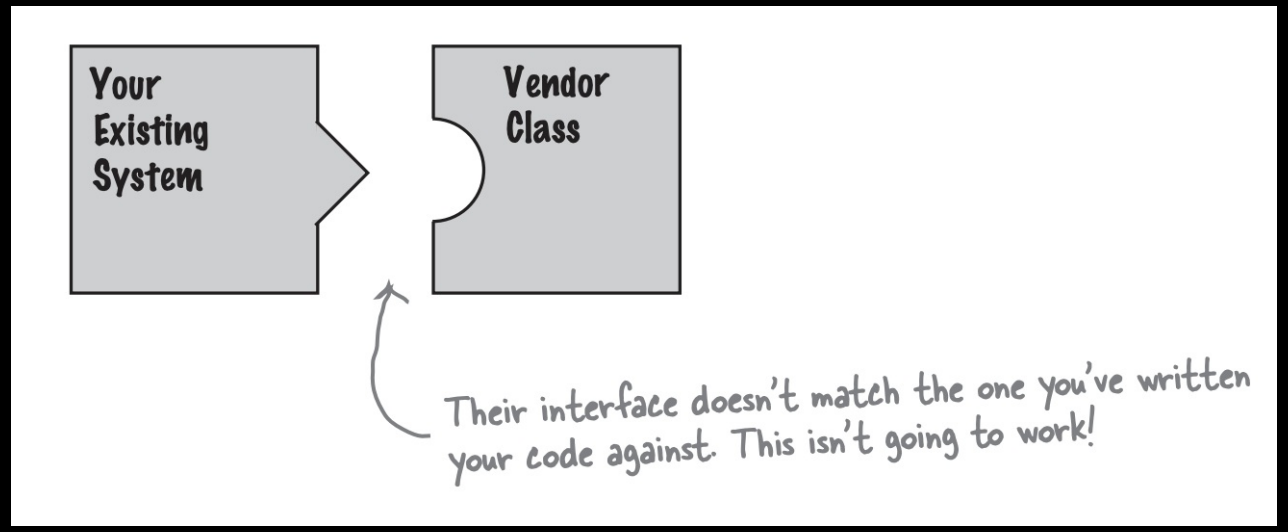
```java
package Adapter;

import java.util.List;

// Data consumer
public class Intranet {
    IPhoneList source;

    public Intranet(IPhoneList source) {
        this.source = source;
    }

    public void printPhoneNumbers() {
        List<String> phones = this.source.getPhoneList();
        System.out.println(phones);
    }
}
```

It is loosely coupled with Producer

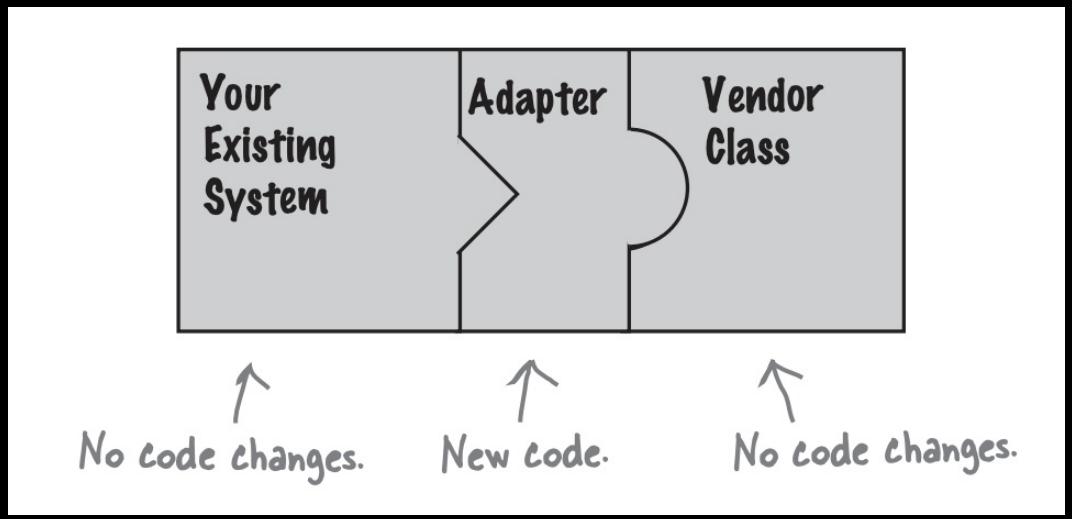
```java
package Adapter;

import java.util.*;

public class Client {
    Run | Debug
    public static void main(String[] args) {
        EmployeeDetails producer = new EmployeeDetails();
        List<String> employees = producer.GetEmployees();

        EmptoPhoneNoAdapter adapter = new EmptoPhoneNoAdapter
        (employees);

        Intranet consumer = new Intranet(adapter);
        consumer.printPhoneNumbers();
    }
}
```

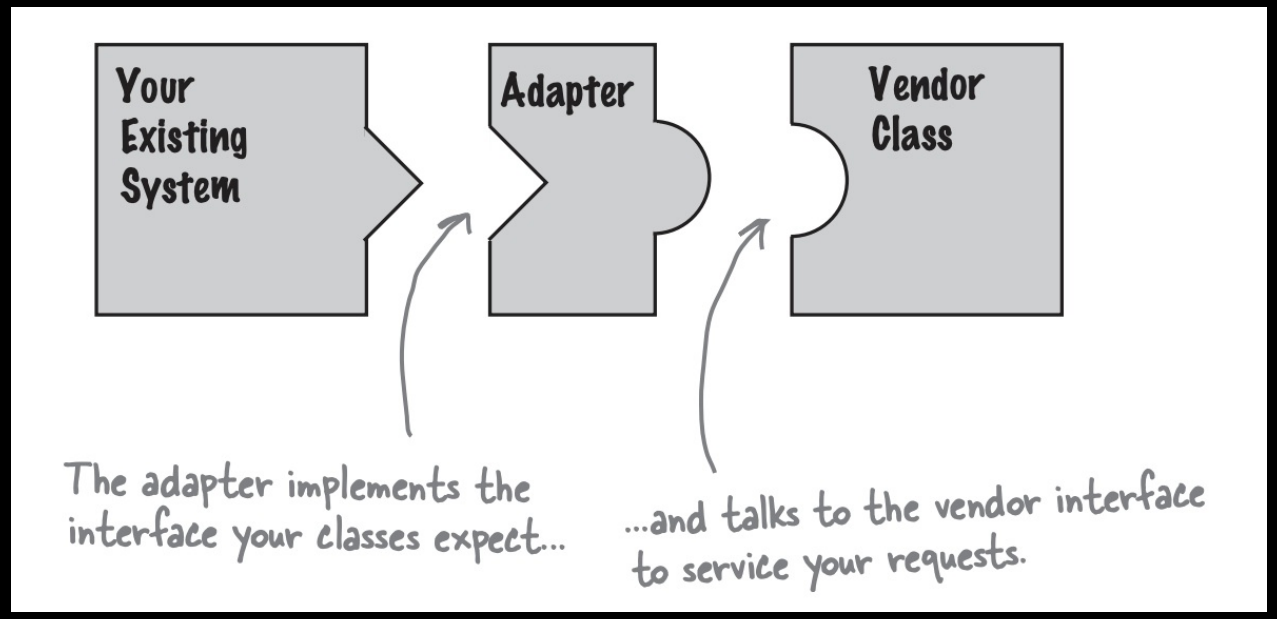
Example :→


British Wall Outlet
AC Power Adapter
US Standard AC Plug
The British wall outlet exposes one interface for getting power.
The adapter converts one interface into another.
The US laptop expects another interface.


Your Existing System
Vendor Class
Their interface doesn't match the one you've written your code against. This isn't going to work!

Approach i—


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


Your Existing System
Adapter
Vendor Class
The adapter implements the interface your classes expect...
...and talks to the vendor interface to service your requests.

Example Problem :→

How Does the Adapter Design Pattern Work?

We have an ImageViewer interface and a concrete class named GalleryApp which is implementing the ImageViewer interface. GalleryApp can show jpeg files by default.

Let us say we have another interface named AdvancedImageViewer and concrete classes implementing this interface. These classes can show png and jpg files by default.

Now we want our GalleryApp class to be able to show png and jpg files without changing the source code of the ImageViewer interface.

So here we will use an adapter design pattern to solve this problem.

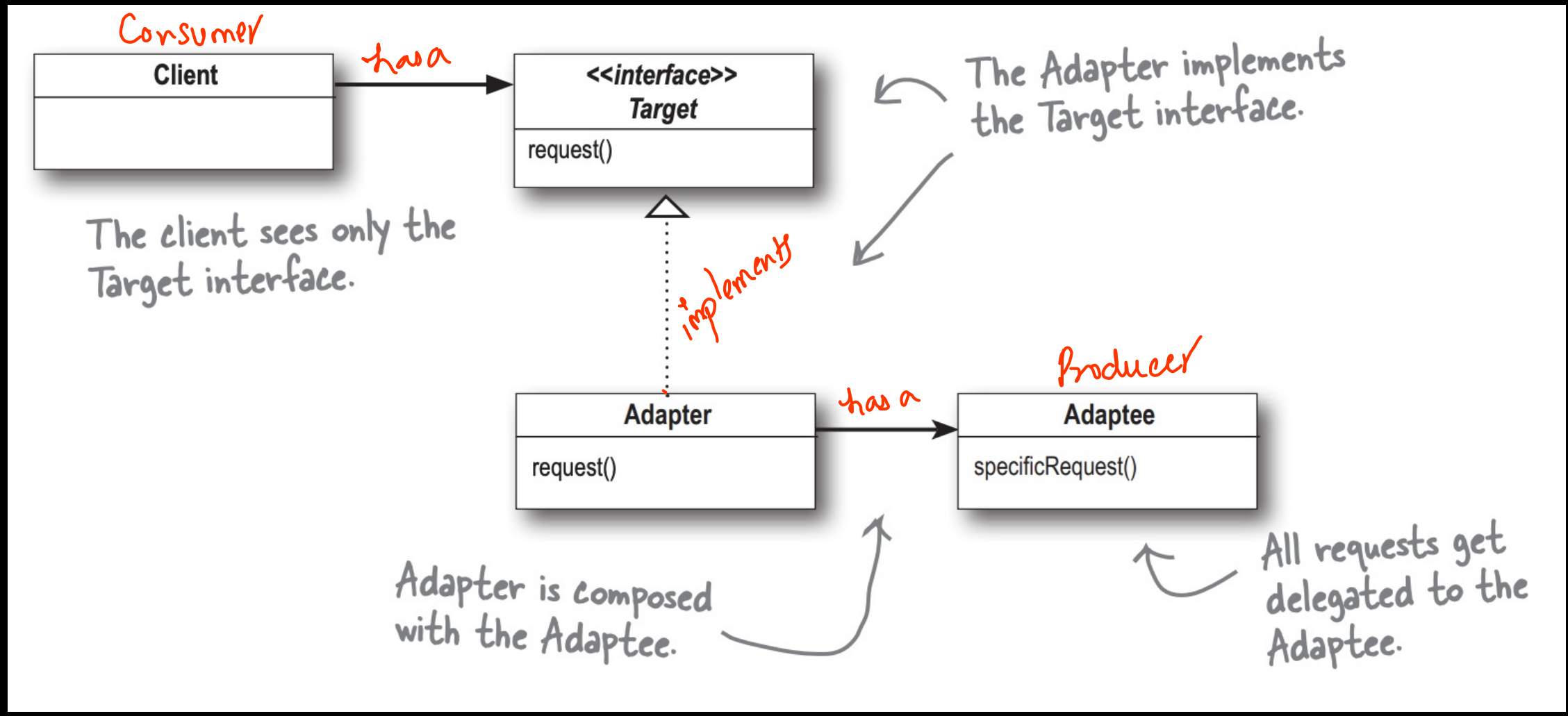
Implementation of Adapter Design Pattern

So we can use an Adapter design pattern to achieve this. We will create a ImageAdapter which **implements** ImageViewer and uses a AdvancedImageViewer object to show the desired image format.

Our Gallery application will use the ImageAdapter and simply pass the desired image format without worrying about which class can show what image format.

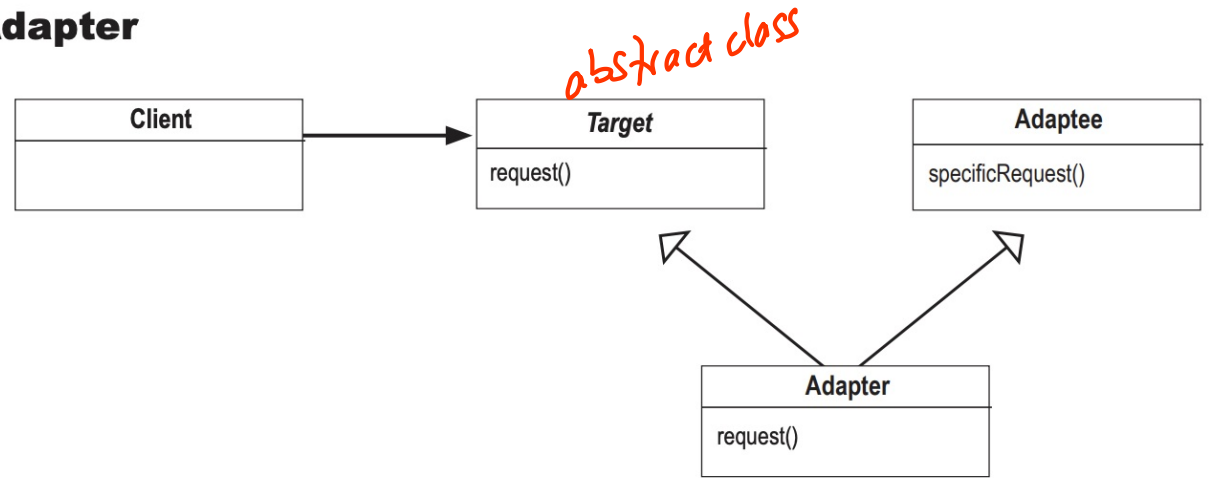
Implementation :→

Class Diagram



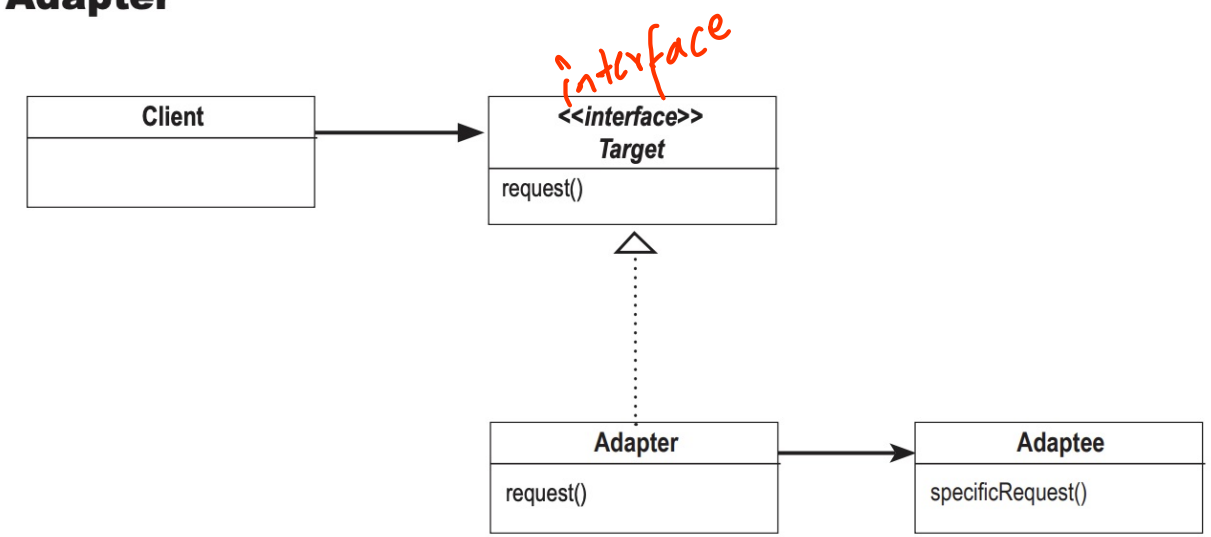
Note:- Types of Adapters

Class Adapter



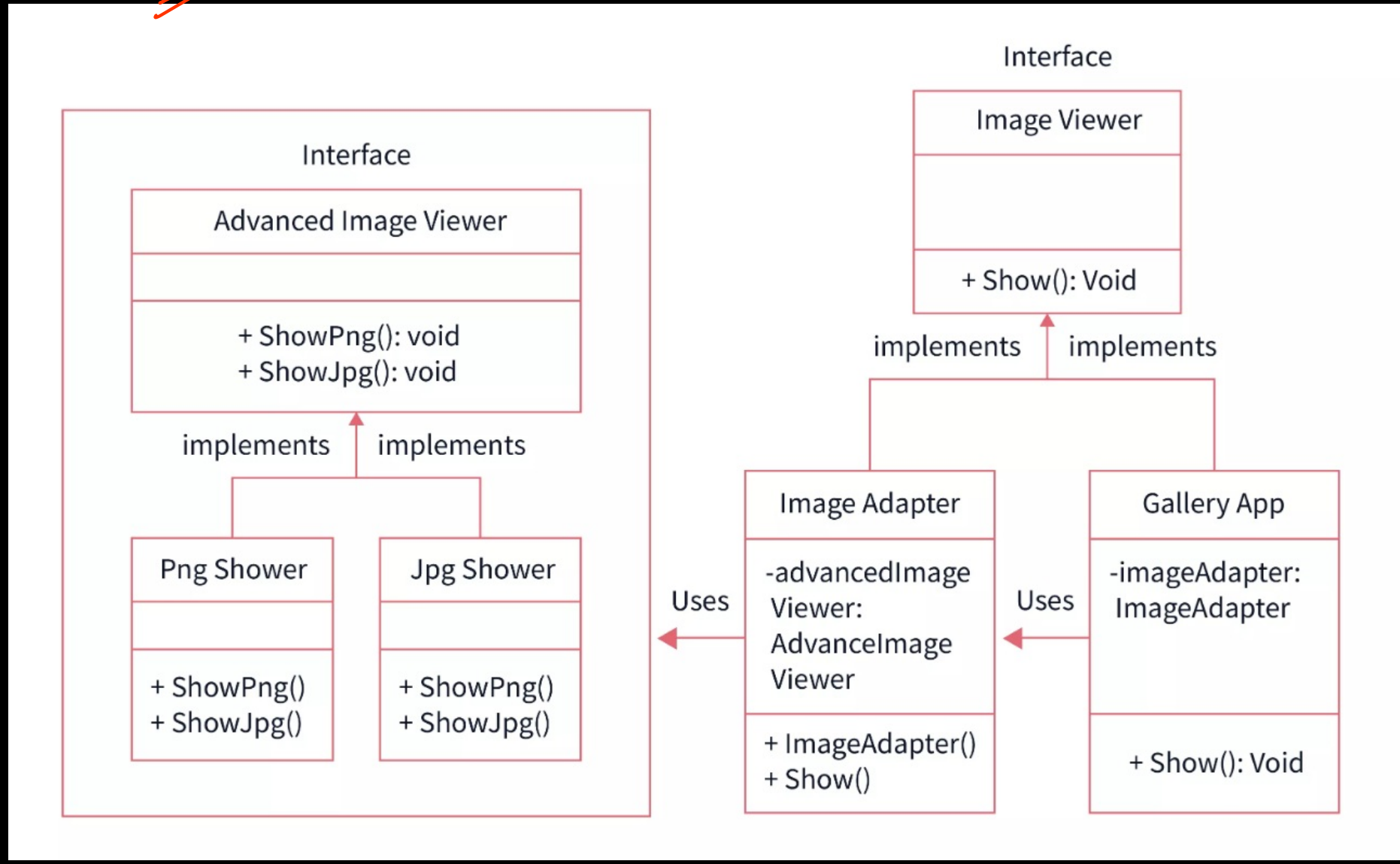
not preferred
It will not support multiple inheritance!

Object Adapter



Preferred
we can implement as many interfaces as we want!

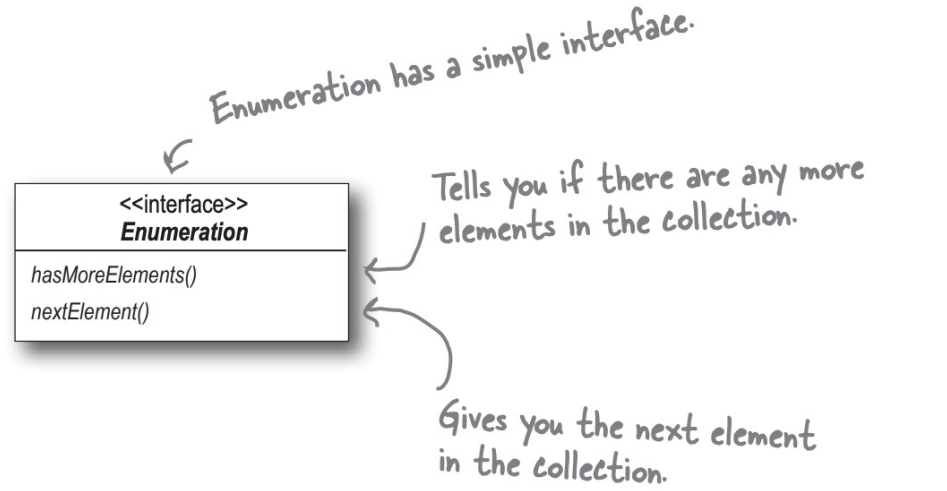
Example Solution



Java Example

Enumerators

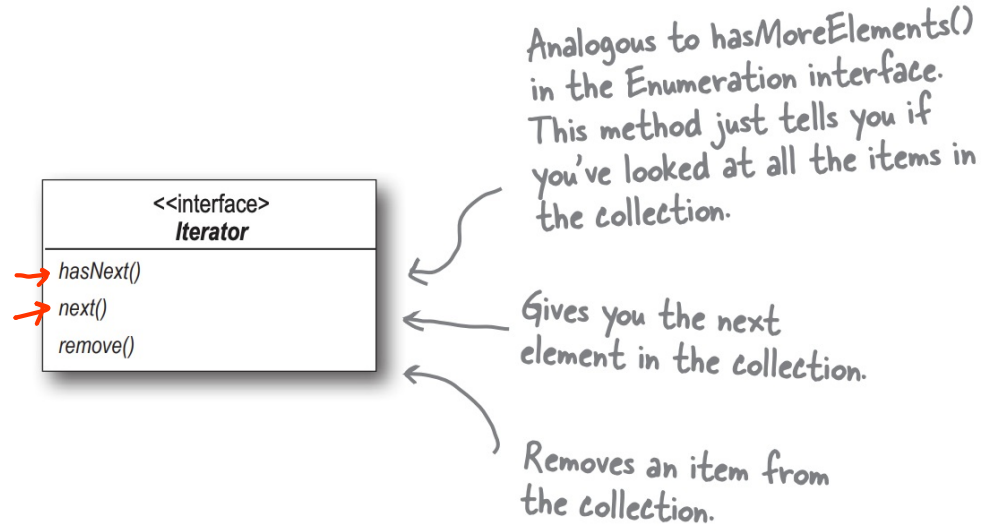
If you've been around Java for a while, you probably remember that the early collection types (Vector, Stack, Hashtable, and a few others) implement a method, elements(), which returns an Enumeration. The Enumeration interface allows you to step through the elements of a collection without knowing the specifics of how they are managed in the collection.



adapt

Iterators

The more recent Collection classes use an Iterator interface that, like the Enumeration interface, allows you to iterate through a set of items in a collection, and adds the ability to remove items.



Pros and Cons of Adapter Design Pattern

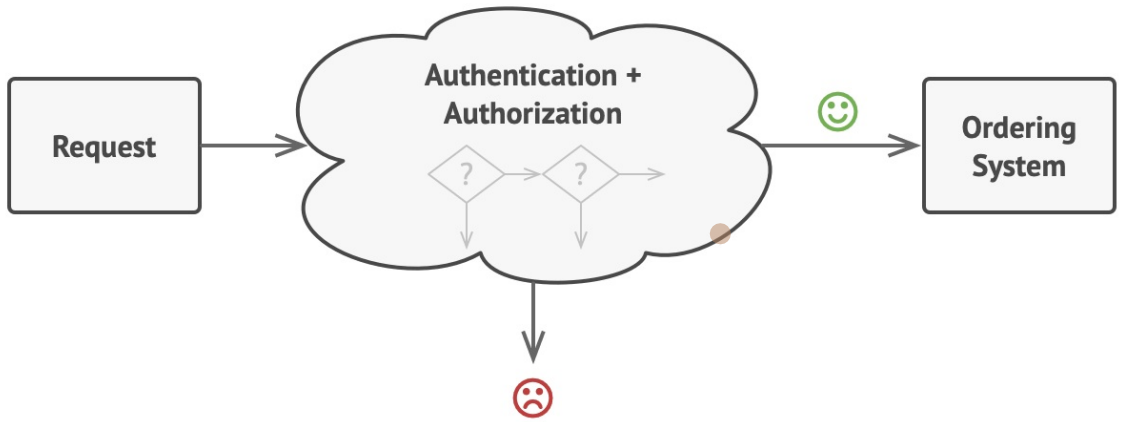
Pros

1. **Separation of Concern:** We can separate the interface or data conversion code from the main business logic part of the code.
2. **Independence of Code:** We can implement and use the various adapters without breaking the existing client or main code.

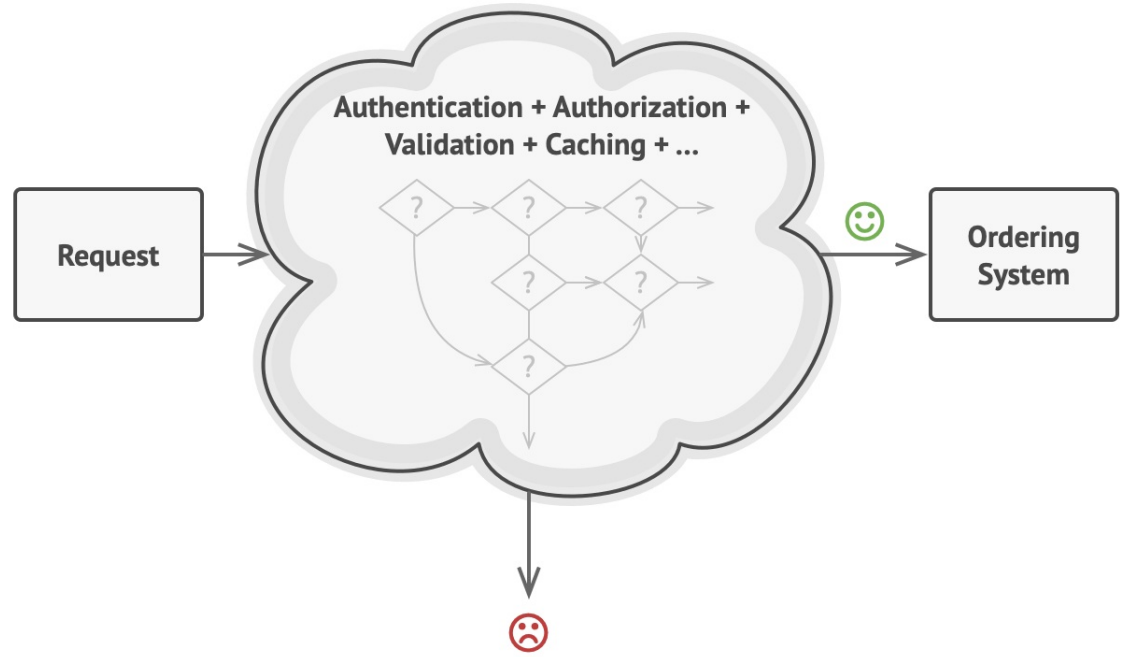
Cons

1. We have to write a lot of code and it can decrease the efficiency. Sometimes it will be simpler to just change the code of a particular interface.

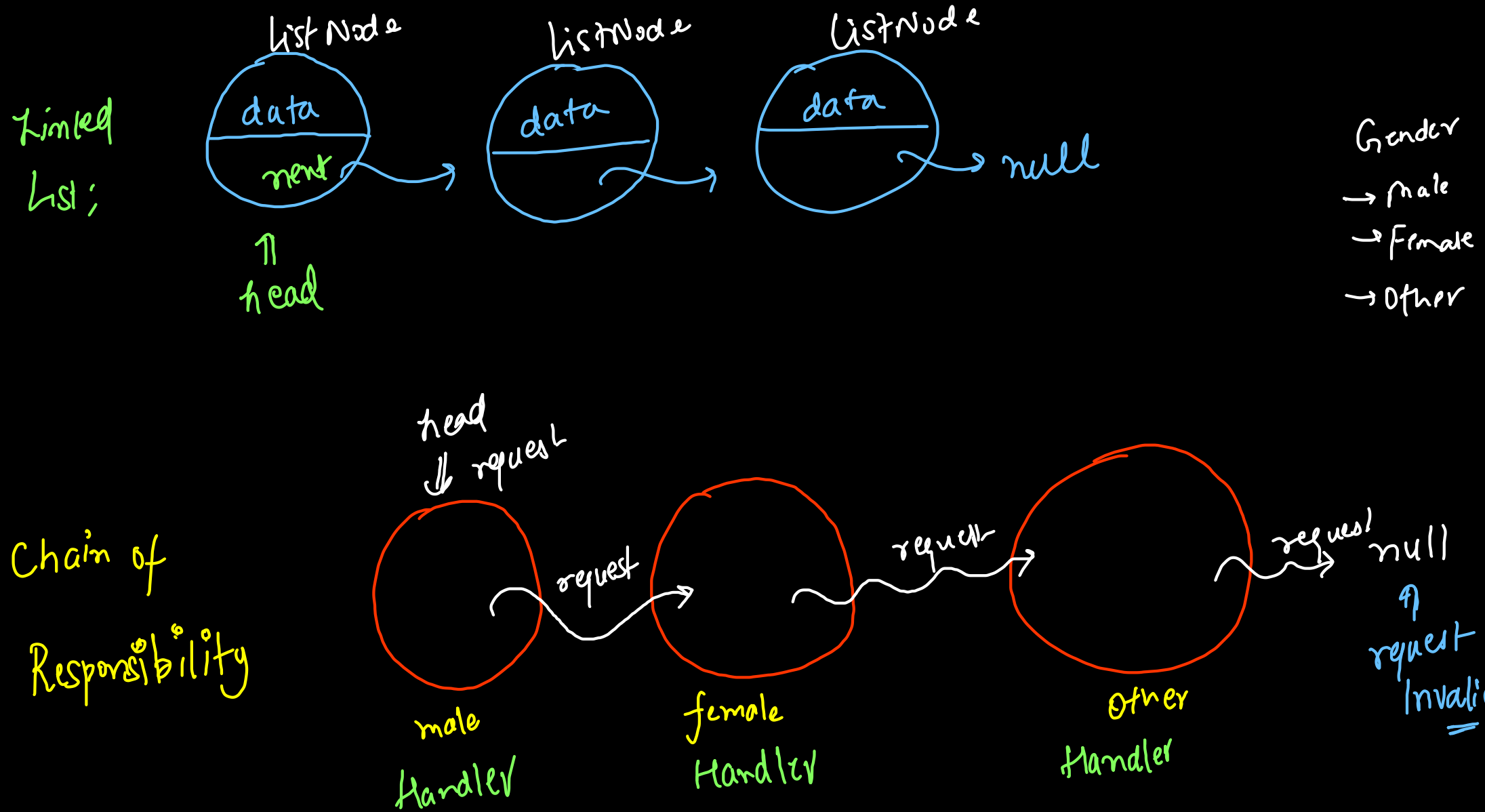
③ Chain of Responsibility Design Pattern



The request must pass a series of checks before the ordering system itself can handle it.



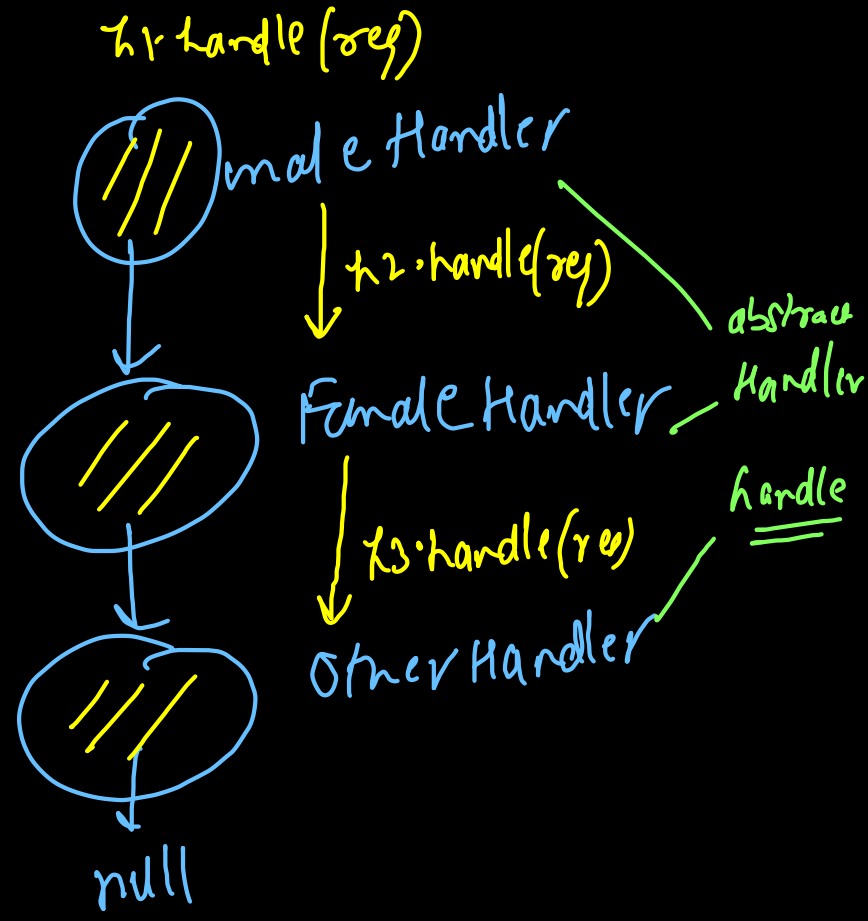
The bigger the code grew, the messier it became.


Linked List;
ListNode
data
next
ListNode
data
ListNode
data
null
head
Gender
→ Male
→ Female
→ Other
Chain of Responsibility
head
request
request
request
request
null
request Invalid
male Handler
female Handler
Other Handler

```java
package ChainOfResponsibility;

public class Client {
    Run | Debug
    public static void main(String[] args) {
        Request r1 = new Request(state: "Beard");
        Request r2 = new Request(state: "Manicure");
        Request r3 = new Request(state: "Haircut");
        Request r4 = new Request(state: "");

        Handler h3 = new OtherHandler(next: null);
        Handler h2 = new FemaleHandler(h3);
        Handler h1 = new MaleHandler(h2);

        h1.handle(r1); // Male: h1
        h1.handle(r2); // Female: h2
        h1.handle(r3); // Other: h3
        h1.handle(r4); // Invalid: h3
    }
}
```

next

```
Male Barber cuts beard
Female Parlour does manicure
Unisex Barber
Invalid Request
```



```java
public abstract class Handler {
    Handler next;

    public Handler() {
        this.next = null;
    }

    public Handler(Handler next) {
        this.next = next;
    }

    public abstract void handle(Request request);
}
```

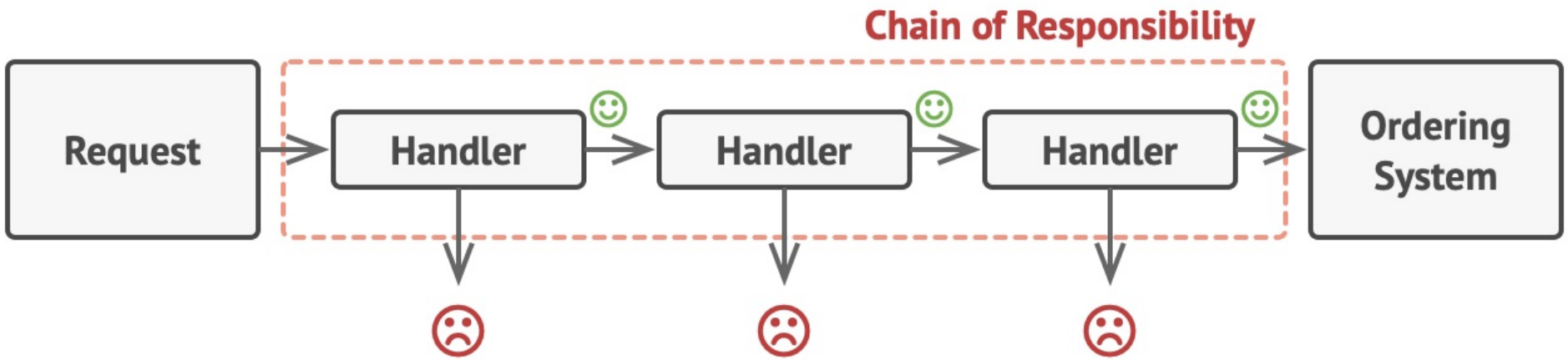
```java
public class MaleHandler extends Handler {
    public MaleHandler(Handler next) {
        super(next);
    }

    @Override
    public void handle(Request request) {
        if (request.state.equals(anObject: "Beard") == true) {
            System.out.println(x: "Male Barber cuts beard");
        } else if (request.state.equals(anObject: "Moustache") ==
        true) {
            System.out.println(x: "Male Barber cuts moustache");
        } else {
            this.next.handle(request);
        }
    }
}
```

What ? Solution !

Like many other behavioral design patterns, the **Chain of Responsibility** relies on transforming particular behaviors into stand-alone objects called *handlers*. In our case, each check should be extracted to its own class with a single method that performs the check. The request, along with its data, is passed to this method as an argument.

The pattern suggests that you link these handlers into a chain. Each linked handler has a field for storing a reference to the next handler in the chain. In addition to processing a request, handlers pass the request further along the chain. The request travels along the chain until all handlers have had a chance to process it.

In our example with ordering systems, a handler performs the processing and then decides whether to pass the request further down the chain. Assuming the request contains the right data, all the handlers can execute their primary behavior, whether it's authentication checks or caching.



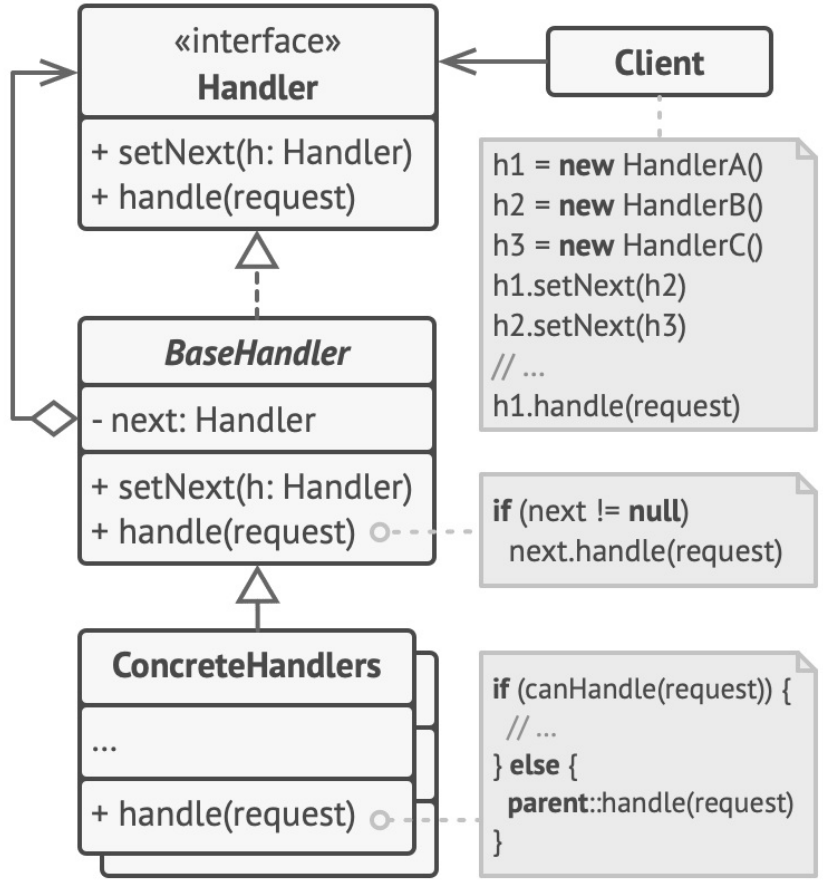
Handlers are lined up one by one, forming a chain.

Structure

1 The **Handler** declares the interface, common for all concrete handlers. It usually contains just a single method for handling requests, but sometimes it may also have another method for setting the next handler on the chain.

2 The **Base Handler** is an optional class where you can put the boilerplate code that's common to all handler classes.

Usually, this class defines a field for storing a reference to the next handler. The clients can build a chain by passing a handler to the constructor or setter of the previous handler. The class may also implement the default handling behavior: it can pass execution to the next handler after checking for its existence.



4 The **Client** may compose chains just once or compose them dynamically, depending on the application's logic. Note that a request can be sent to any handler in the chain—it doesn't have to be the first one.

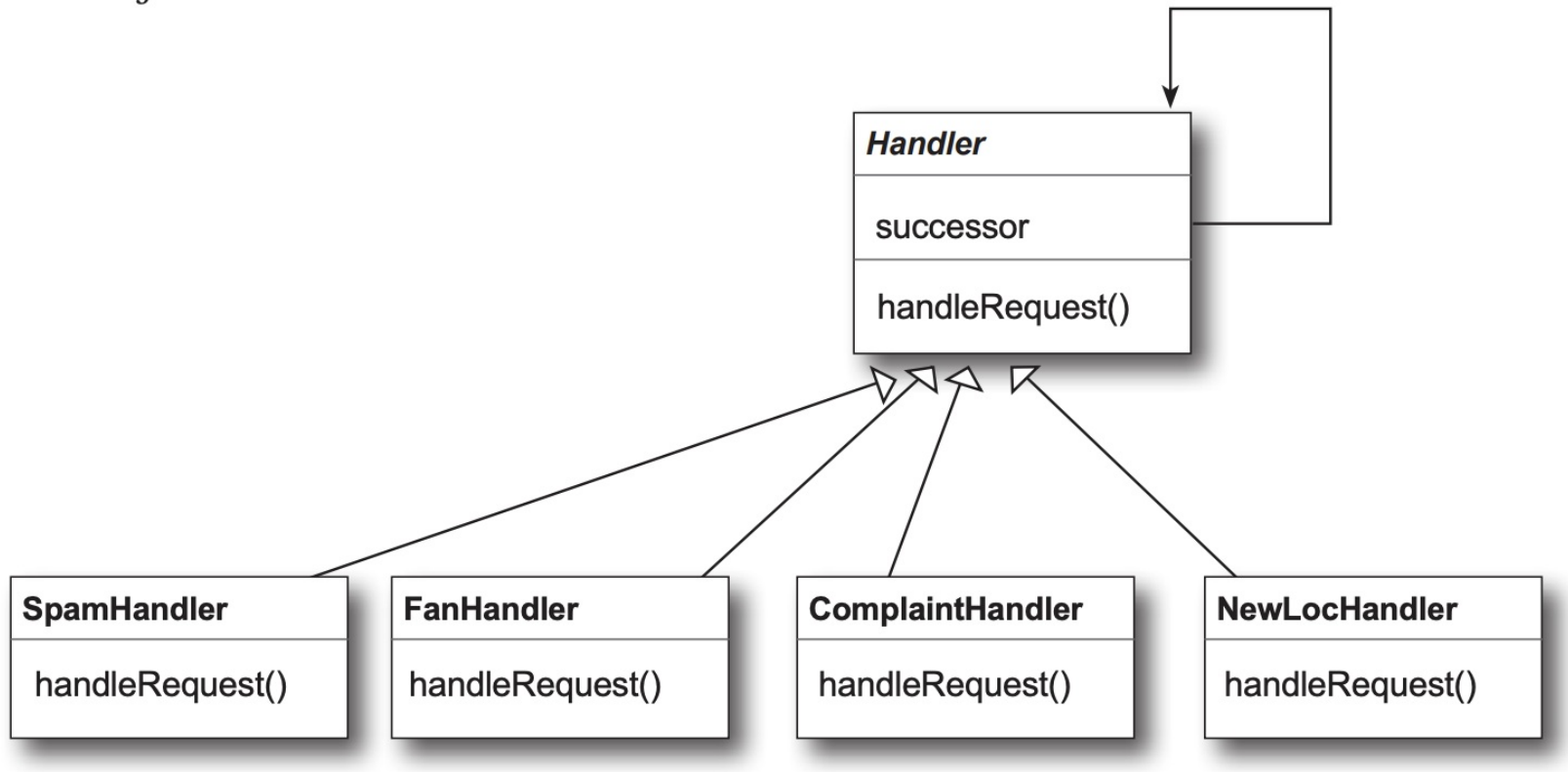
3 **Concrete Handlers** contain the actual code for processing requests. Upon receiving a request, each handler must decide whether to process it and, additionally, whether to pass it along the chain.

Handlers are usually self-contained and immutable, accepting all necessary data just once via the constructor.

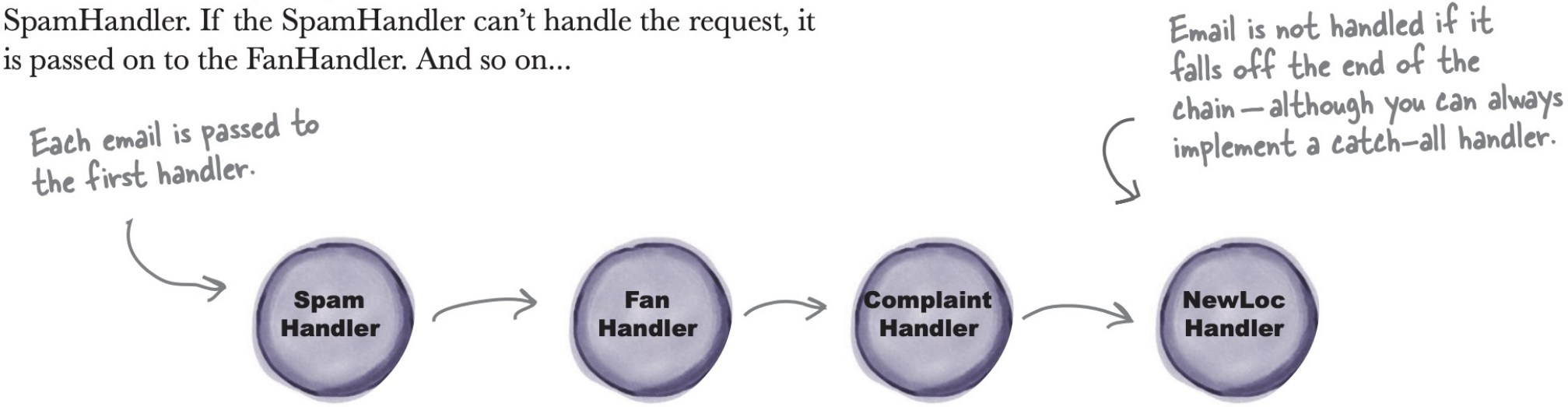
How to use the Chain of Responsibility Pattern

With the Chain of Responsibility Pattern, you create a chain of objects to examine requests. Each object in turn examines a request and either handles it or passes it on to the next object in the chain.

Each object in the chain acts as a handler and has a successor object. If it can handle the request, it does; otherwise, it forwards the request to its successor.



As email is received, it is passed to the first handler: SpamHandler. If the SpamHandler can't handle the request, it is passed on to the FanHandler. And so on...



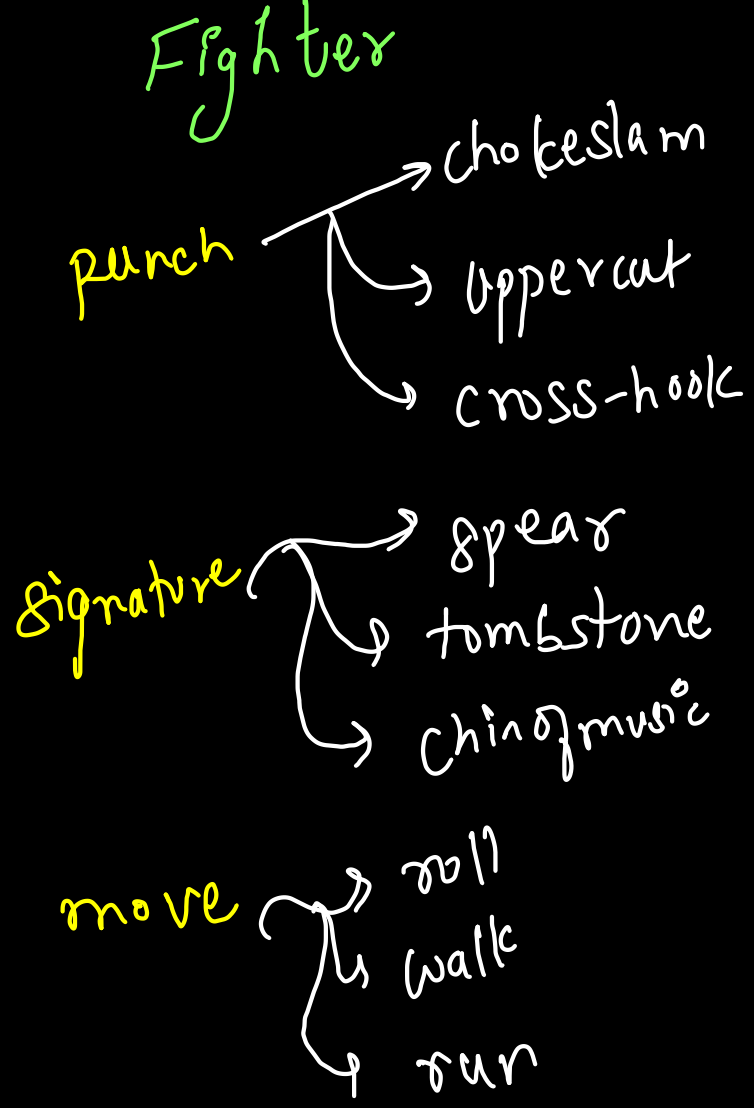
Chain of Responsibility Benefits

- Decouples the sender of the request and its receivers.
- Simplifies your object because it doesn't have to know the chain's structure and keep direct references to its members.
- Allows you to add or remove responsibilities dynamically by changing the members or order of the chain.

Chain of Responsibility Uses and Drawbacks

- Commonly used in Windows systems to handle events like mouse clicks and keyboard events.
- Execution of the request isn't guaranteed; it may fall off the end of the chain if no object handles it (this can be an advantage or a disadvantage).
- Can be hard to observe and debug at runtime.

④ Strategy Design Pattern

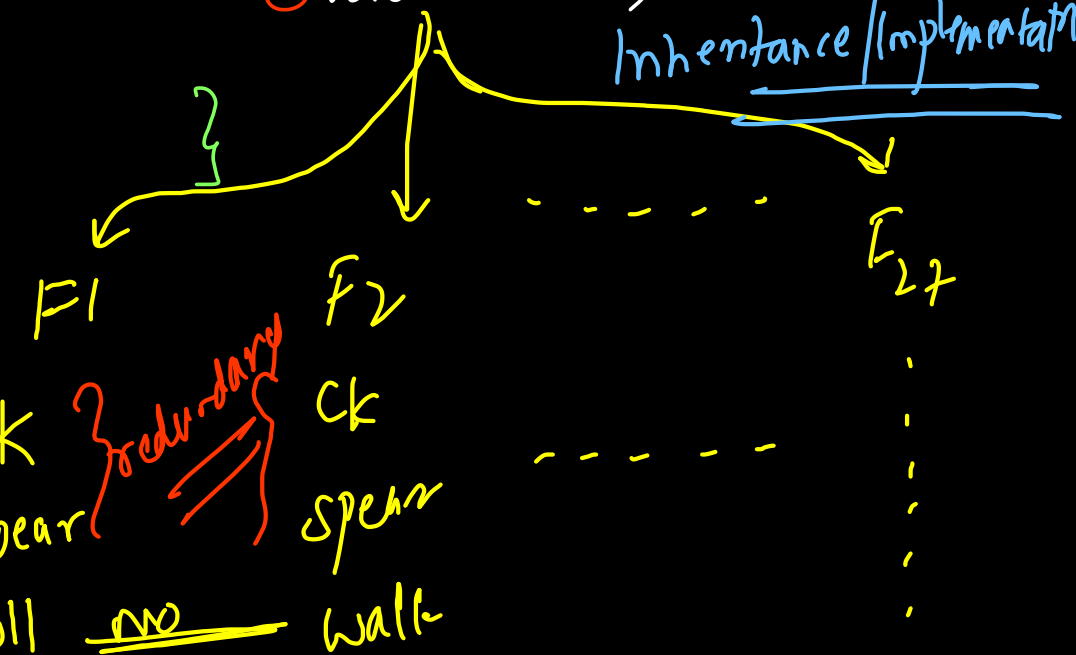


Problem :-

```
abstract clas Fighter{
    void punch();
    void kick();
    void move();
}
```

abstract

violating LSP



Solution

① Interface Segregation
② Composition over Inheritance
③ DRY

PunchInterface — implements → chokeslam, uppercut, cross-hook (3) +

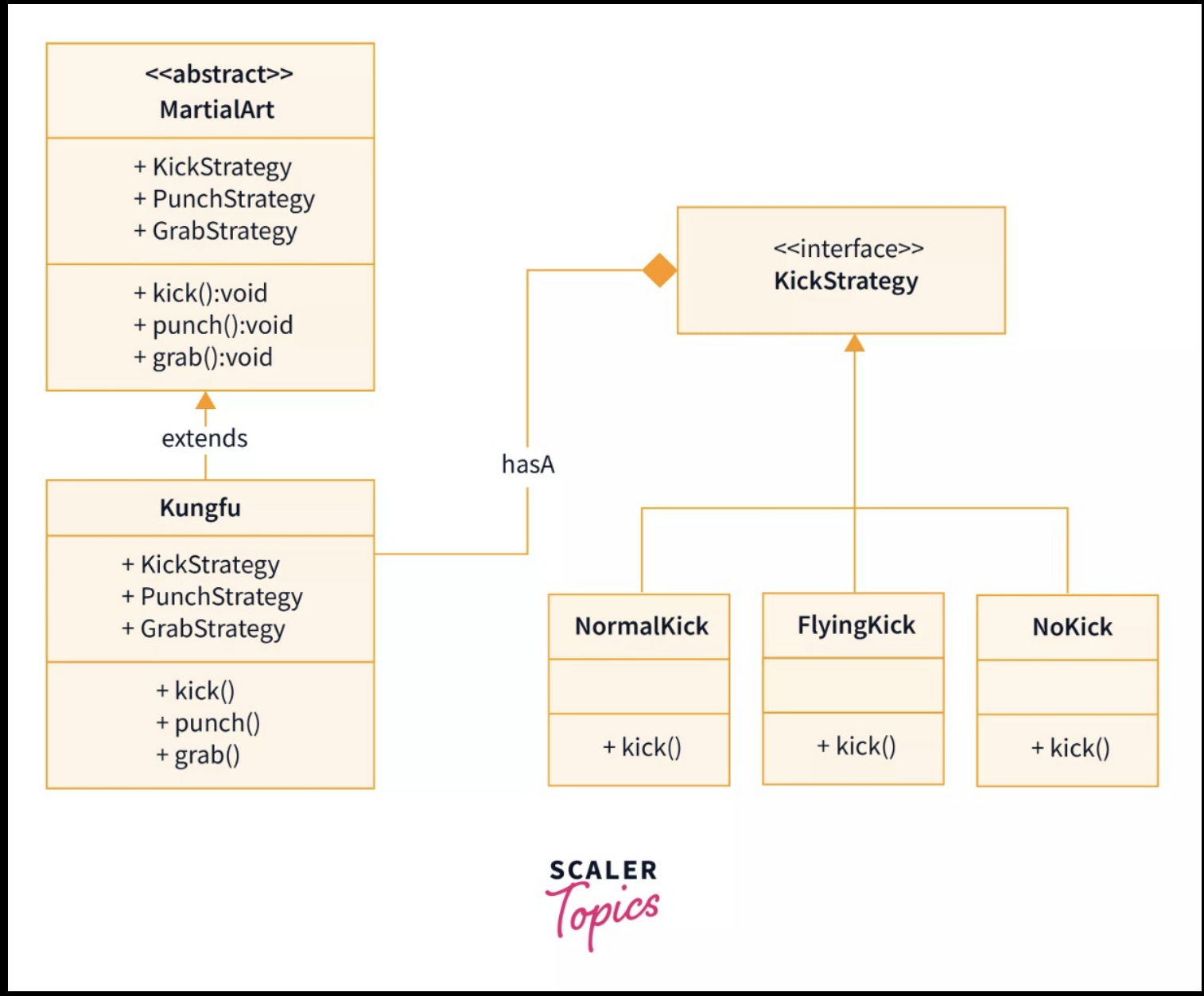
SignatureInterface — implements → spear, tombstone, chingmusic (3) +

moveInterface — implements → roll, walk, run (3) +

```
class Fighter{
    punchInterface pmove;
    SignatureInterface Sigmove;
    moveInterface move;
}

Fighter obj = new Fighter(
    new Chokeslam(),
    new Tombstone(),
    new Wall()
);
```

```java
public class Fighter {
    private IMove move;
    private IPunch punch;
    private ISignature signature;

    public IMove getMove() {
        return move;
    }

    public void setMove(IMove move) {
        this.move = move;
    }

    public IPunch getPunch() {
        return punch;
    }

    public void setPunch(IPunch punch) {
        this.punch = punch;
    }

    public ISignature getSignature() {
        return signature;
    }

    public void setSignature(ISignature signature) {
        this.signature = signature;
    }

    public void move() {
        move.move();
    }

    public void punch() {
        punch.punch();
    }

    public void signature() {
        signature.signatureMove();
    }
}
```

```java
package Strategy;

import Strategy.Move.Walk;
import Strategy.Punch.ChokeSlam;
import Strategy.Signature.Spear;
import Strategy.Signature.Tombstone;

public class Client {
    Run | Debug
    public static void main(String[] args) {
        Fighter undertaker = new Fighter();
        undertaker.setMove(new Walk());
        undertaker.setPunch(new ChokeSlam());
        undertaker.setSignature(new Tombstone());

        undertaker.punch();
        undertaker.move();
        undertaker.signature();

        Fighter kane = new Fighter();
        kane.setMove(new Walk());
        kane.setPunch(new ChokeSlam());
        kane.setSignature(new Spear());

        kane.punch();
        kane.move();
        kane.signature();

    }
}
```


<<abstract>>
MartialArt
+ KickStrategy
+ PunchStrategy
+ GrabStrategy
+ kick():void
+ punch():void
+ grab():void
extends
Kungfu
+ KickStrategy
+ PunchStrategy
+ GrabStrategy
+ kick()
+ punch()
+ grab()
hasA
<<interface>>
KickStrategy
NormalKick
+ kick()
FlyingKick
+ kick()
NoKick
+ kick()
SCALER
Topics

Pros and Cons of Strategy Design pattern

Pros:

1. We can avoid using conditionals like switch if-else statements to decide the behaviour of entity classes at run time
2. The main behavioural algorithms are abstracted out from the main context entities These advantages make our code **easy to maintain** and **extensible**

Cons:

1. Since we have abstracted out the behaviour of context entities from their classes, the onus is on the client to understand various strategies to implement the behaviours, and pass it accordingly
2. We will end up creating more objects in our application, demanding more memory footprint

(5) Observer Design Pattern

or Publisher - Subscriber Model

Publisher → Subject

Subscriber → observers

one ⟶ many

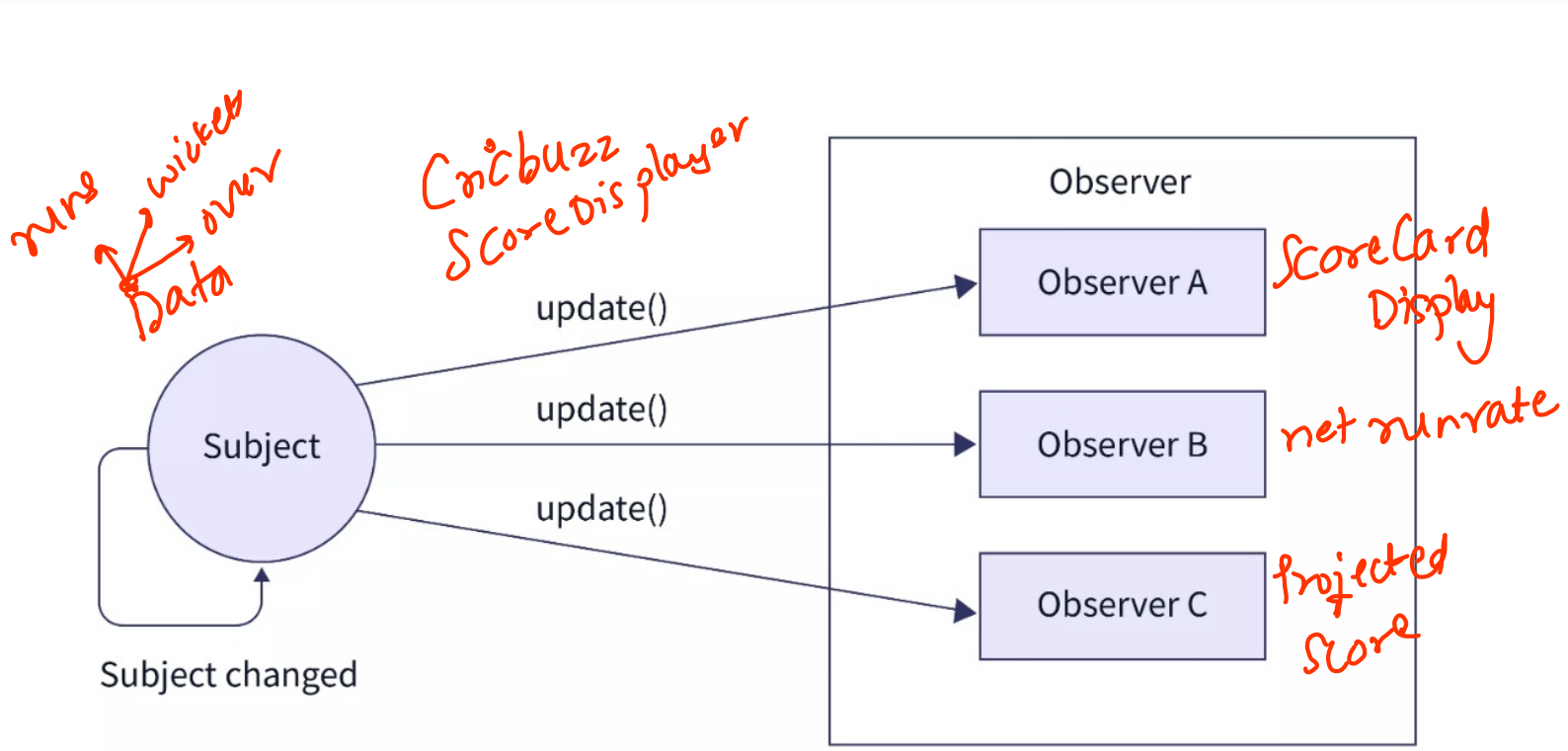
Overview

Observer design pattern falls under the category of behavioral design patterns. The Observer Pattern maintains a one-to-many relationship among objects, ensuring that when the state of one object is changed, all of its dependent objects are simultaneously informed and updated. This design pattern is also referred to as Dependents.

A subject and observer(many) exist in a one-to-many relationship. Here the observers do not have any access to data, so they are dependent on the subject to feed them with information.

When Will We Need Observer Design Pattern?

One could implement the observer design patterns when designing a system where several objects are interested in any possible modification to a specific object. In other words, the observer design pattern is employed when there is a one-to-many relationship between objects, such as when one object is updated, its dependent objects must be automatically notified.



(1) Subject has a list of observers

(2) Observer has a Subject

Javascript

button click
- next screen
- form open
- sound
- camera

```java
public abstract class Subject {
    private List<Observer> observers;

    public Subject() {
        observers = new ArrayList<>();
    }

    public void addObserver(Observer observer) {
        observers.add(observer);
    }

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

    public void notifyObservers() {
        for (Observer observer : observers) {
            observer.update(this);
        }
    }
}
```

① data member

② Constructor

③ Concrete operations

```java
public interface Observer {
    public abstract void update(Subject s);
}
```

One subject : many observers

data.add(6 mars, 0 wicket);

button.addListener('click', next

'double-click', close

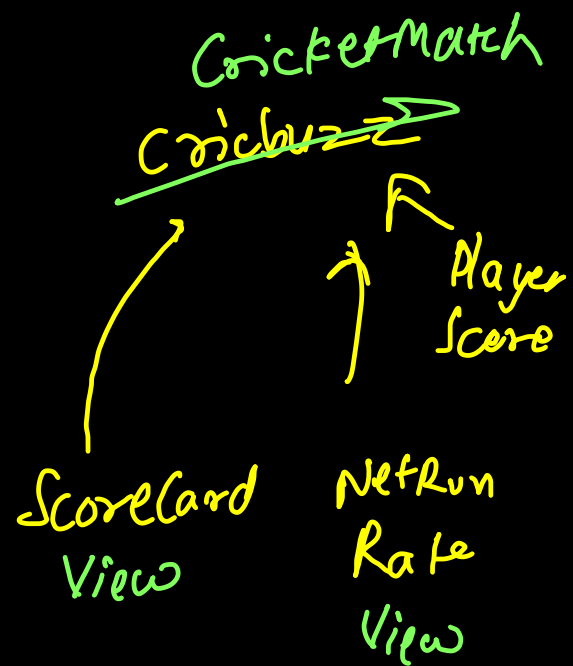
'hover', zoom)

Hashmap<String, list<Observer>>

↳ random sequence

```java
public static void main(String[] args) {
    CricketData data = new CricketData();
    data.setBalls(balls: 1);
    // no observers

    ScoreCardDisplay score = new ScoreCardDisplay();
    NetRunRateDisplay runrate = new NetRunRateDisplay();
    PlayerScoreDisplay player = new PlayerScoreDisplay();

    // register
    data.addObserver(score);
    data.addObserver(runrate);
    data.addObserver(player);

    data.setBalls(balls: 2);
    data.setRuns(runs: 4);

    data.setBalls(balls: 3);
    data.setWickets(wickets: 1);

    // unregister
    data.removeObserver(player);

    data.setBalls(balls: 3);
    data.setRuns(runs: 2);
}
```

```
Score Card Display: --
4 / 0
2
Net Run Rate Display: --
12
Player Score Display: --
4 / 0
Score Card Display: --
4 / 1
3
Net Run Rate Display: --
6
Player Score Display: --
4 / 1
Score Card Display: --
2 / 1
3
Net Run Rate Display: --
0
```



Examples

Real-World Example: If a bus gets delayed, then all the passengers who were supposed to travel in it get notified about the delay, to minimize inconvenience. Here, the bus agency is the subject and all the passengers are the observers. All the passengers are dependent on the agency to provide them with information about any changes regarding their travel journey. Hence, there is a one-to-many relationship between the travel agency and the passengers.

Other **Examples** of observer design patterns:

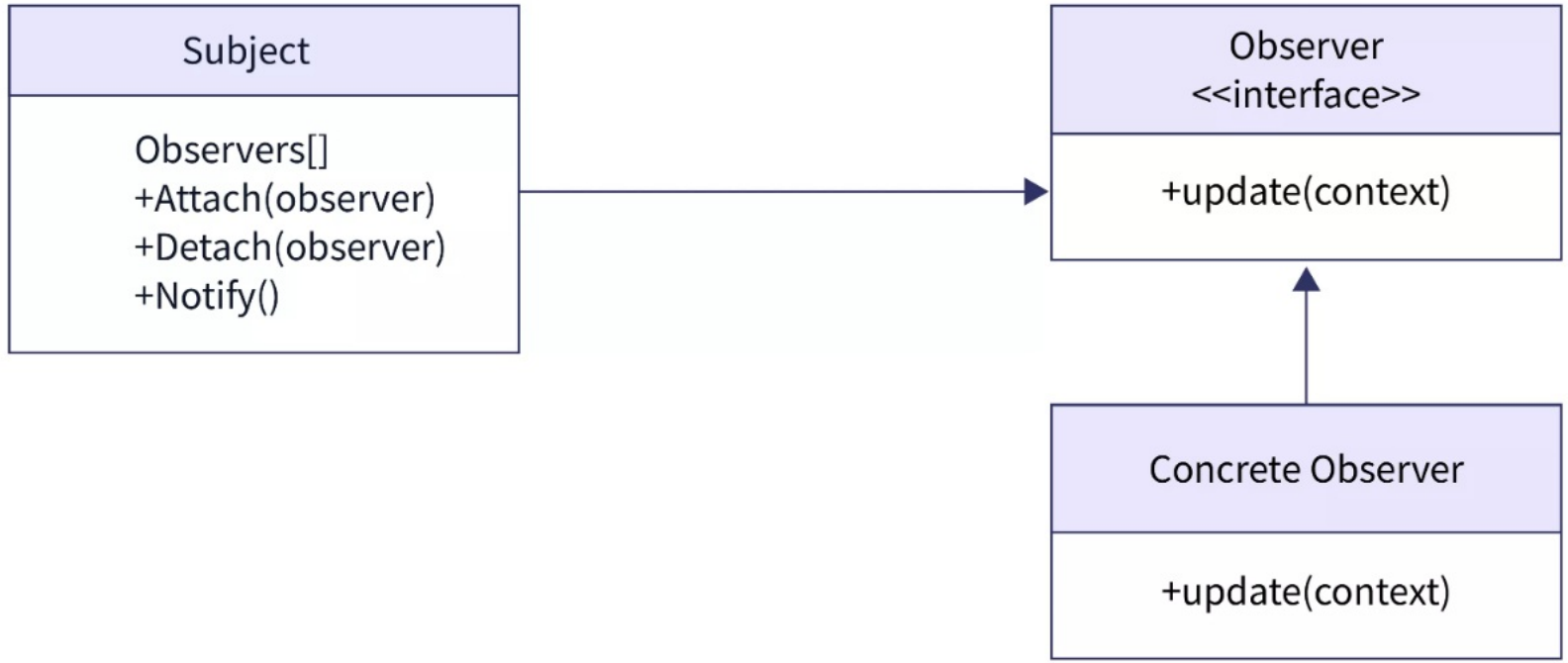
- Social Media Platforms for example, many users can follow a particular person(subject) on a social media platform. All the followers will be updated if the subject updates his/her profile. The users can follow and unfollow the subject anytime they want.
- Newsletters or magazine subscription
- Journalists providing news to the media
- E-commerce websites notify customers of the availability of specific products

- Monitoring: Log / Ticket (PagerDuty) → on call engineers ← high severity
  Slack ← mid severity
  Email ← low severity

Structure

How does Observer Design Patterns work?

Structure



Implementation

1. Declare an Observer interface with an update method at the least.

2. Declare the subject interface and a couple of methods for attaching and detaching observer objects from the list of observers. (Remember that publishers must only interact with subscribers through the subscriber interface.)

3. Create a concrete subject class if needed and add the subscribers' list to this class. Since this concrete class extends the Subject interface, it inherits all its properties including adding and removing observers. Every time something significant happens inside the subject class, it must inform to all the observers about it.

4. In concrete observer classes, add the update notification methods. Some observers might want basic background information about the occurrence. The notification method accepts it as an input.

Pros

1. This design pattern allows information or data transfer to multiple objects without any change in the observer or subject classes.
2. It adheres to the loose coupling concept among objects that communicate with each other.
3. This design pattern follows the Open/Closed Principle, which says that entities should be open for extension but closed for modification. Here the observers can easily be added or removed anytime without any change in the code.

Cons

1. The Observer pattern can increase complexity and potentially cause efficiency issues if it's not executed properly.
2. The fundamental shortcoming of this design pattern is that the subscribers/observers are updated in a random sequence.

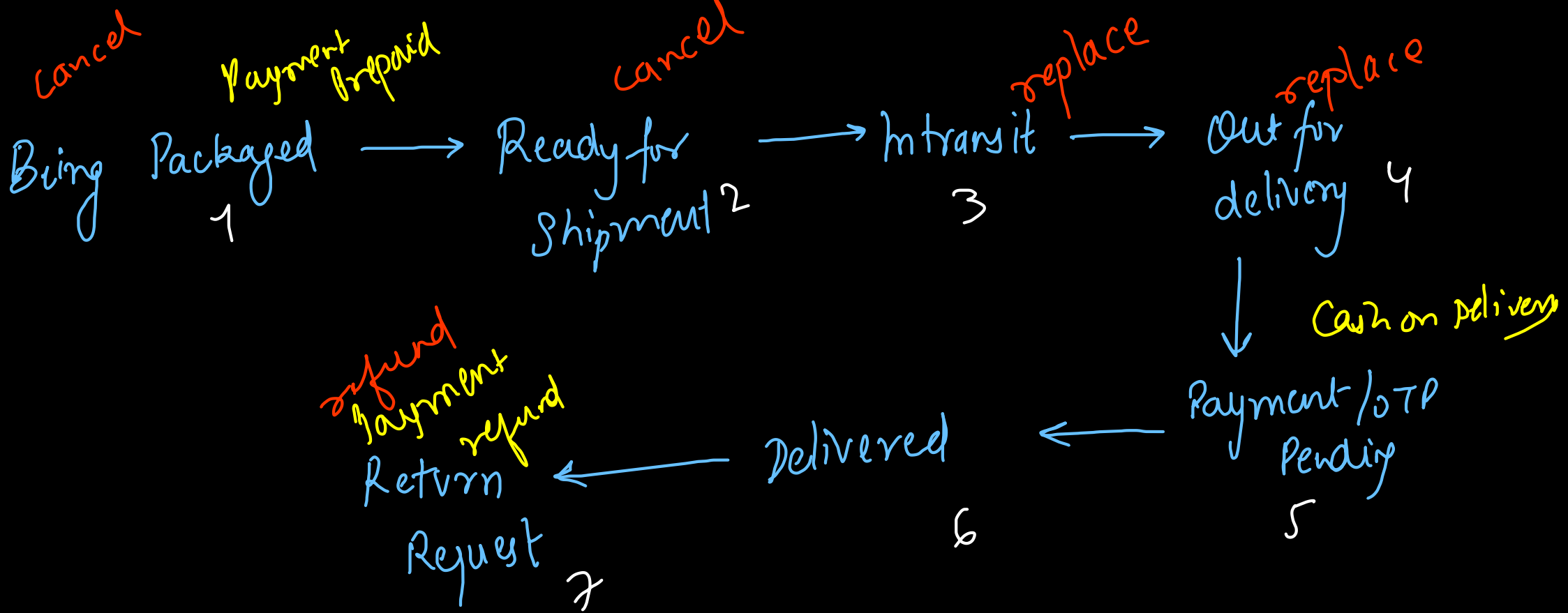
6 State Design Pattern

Overview

State Design Patterns belongs to the category of Behavioral design pattern. It is used to allow an object to modify its behavior based on the changes in its internal state. It makes the object behave similarly to finite state machines. It makes the object flexible to alter its state without handling a lot of **if / else conditions**. State pattern ensures loose coupling between the performance of existing states versus the addition of new states.

Note: A Finite State Machine is a model of computation which is based on a hypothetical machine made of one or more states. Only one single state of this machine can be active at a particular time. It means the machine has to do the transition from one state to another in order to reflect different behaviors.

Example # Delhivery / DHL / Amazon Delivery / Fed Ex



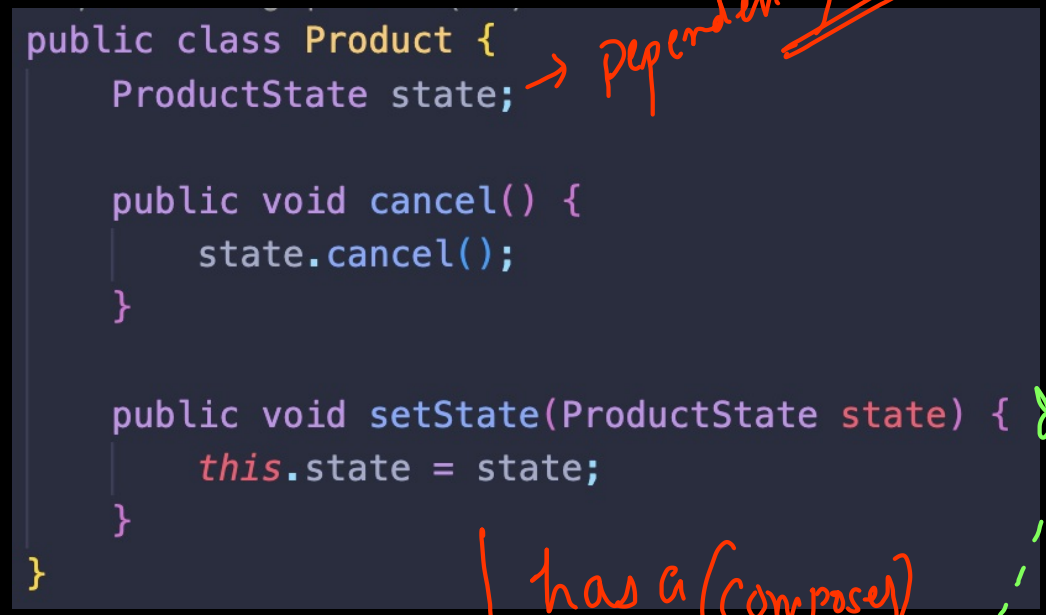
Problem code

```java
public class Product {
    int state;

    public int getState() {
        return state;
    }

    public void setState(int state) {
        this.state = state;
    }

    public void cancel() {
        if (this.state < 5) {
            // Packaging, Transit, Out for Delivery
            System.out.println(x: "Cancellation Not Allowed");
        } else if (this.state == 5) {
            // Wrong Product or Damaged at Payment
            System.out.println(x: "Replace Product");
        } else {
            // After payment
            System.out.println(x: "Return & Refund Product");
        }
    }
}
```

→ violates
→ Single Responsibility
→ Open-closed

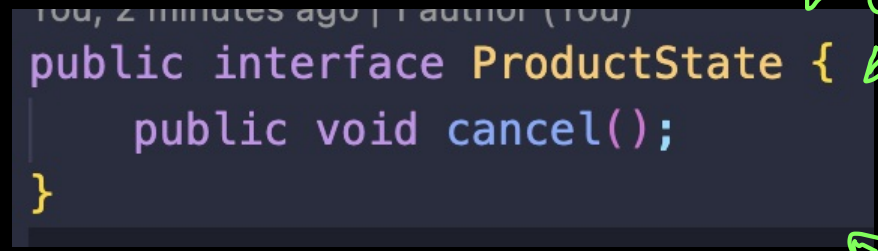
```java
package State.Problem;

public class Client {
    Run | Debug
    public static void main(String[] args) {
        Product iphone = new Product();
        iphone.setState(state: 1); // Being
        Packaged
        iphone.cancel(); // Cancellation Not
        Allowed

        iphone.setState(state: 4); // Out For
        Delivery
        iphone.setState(state: 5); // Payment
        Pending
        iphone.cancel(); // Replace Allowed

        iphone.setState(state: 6); // Payment
        Done & Recieved
        iphone.cancel(); // Return & Refund
    }
}
```

Solution State



Dependency Inversion

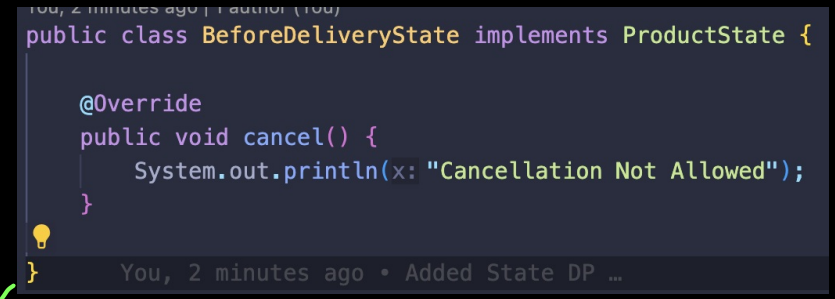
has a (composes)



depends

depends

depends



```java
package State.Solution;

You, 6 minutes ago | 1 author (You)
public class Client {
    Run | Debug
    public static void main(String[] args) {
        Product iphone = new Product();
        iphone.setState(new BeforeDeliveryState()); // Being
        Packaged
        iphone.cancel(); // Cancellation Not Allowed

        iphone.setState(new OnDoorState()); // Payment Pending
        iphone.cancel(); // Replace Allowed

        iphone.setState(new DeliveredState()); // Payment Done &
        Recieved
        iphone.cancel(); // Return & Refund
    }
}
```

Pros and Cons of State Design Pattern

Pros:

1. Whenever we want to alter the behavior of an object by changing the internal state, and we do not want to handle a lot of if / else complex conditional blocks, then in that **state design pattern** can be highly advantageous.

2. Whenever we want to add new states so that new behaviors can be generated in the application and we do not want to impact the existing behaviors pertaining to the previously defined states, in that case also State design pattern is the best bet.

3. State design pattern is beneficial in implementing a **polymorphic behavior**, as the same function call is performing different tasks each time.

Cons:

1. State Design pattern requires you to write a lot of code. The addition of new states requires more code, which makes code management a little difficult.

2. When there are only a few states, then applying the state pattern just increases the unnecessary maintenance of a lot of code.

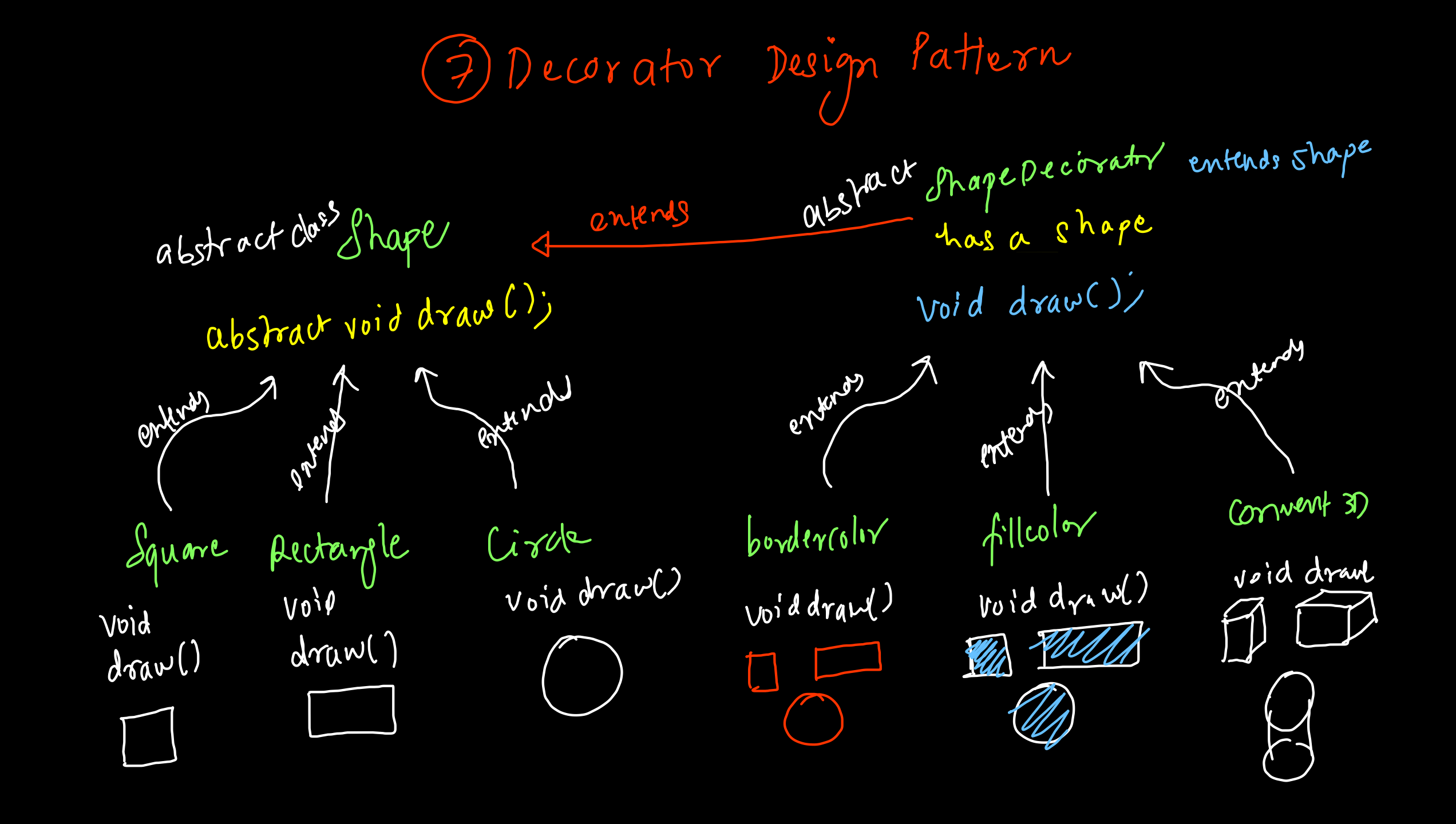
SRP?

OCP?

Complexity ↑
states ↑ ⇒ classes ↑

→ KISS

⑦ Decorator Design Pattern



```java
public static void main(String[] args) {
    Shape square = new Square();

    ShapeDecorator borderSquare = new BorderDecorator(square);
    borderSquare.draw();

    Shape rectangle = new Rectangle();

    ShapeDecorator filledRectangle = new ColorDecorator
    (rectangle);
    filledRectangle.draw();

    Shape circle = new Circle();

    ShapeDecorator cylinder = new Convert3DDecorator(circle);
    cylinder.draw();
}
```

```
Unfilled Unbordered 2D Square
Is Now Bordered
Unfilled Unbordered 2D Rectangle
Is Now Filled With Color
Unfilled Unbordered 2D Circle
Is Now 3D
```

```java
public abstract class Shape {
    public abstract void draw();
}
```

```java
public abstract class ShapeDecorator extends Shape {
    Shape shape;

    public ShapeDecorator(Shape shape) {
        this.shape = shape;
    }

    public abstract void draw();
}
```

new BorderRectangle(
new filledRectangle(
new Rectangle()));

```java
Shape rectangle = new Rectangle();

ShapeDecorator filledRectangle = new ColorDecorator(rectangle);
// filledRectangle.draw();

Shape circle = new Circle();

ShapeDecorator cylinder = new Convert3DDecorator(circle);
// cylinder.draw();

// We can decorate a already decorated shape
ShapeDecorator coloredCylinder = new ColorDecorator(cylinder);
coloredCylinder.draw();

ShapeDecorator borderedFilledRect = new BorderDecorator(filledRectangle);

borderedFilledRect.draw();
```

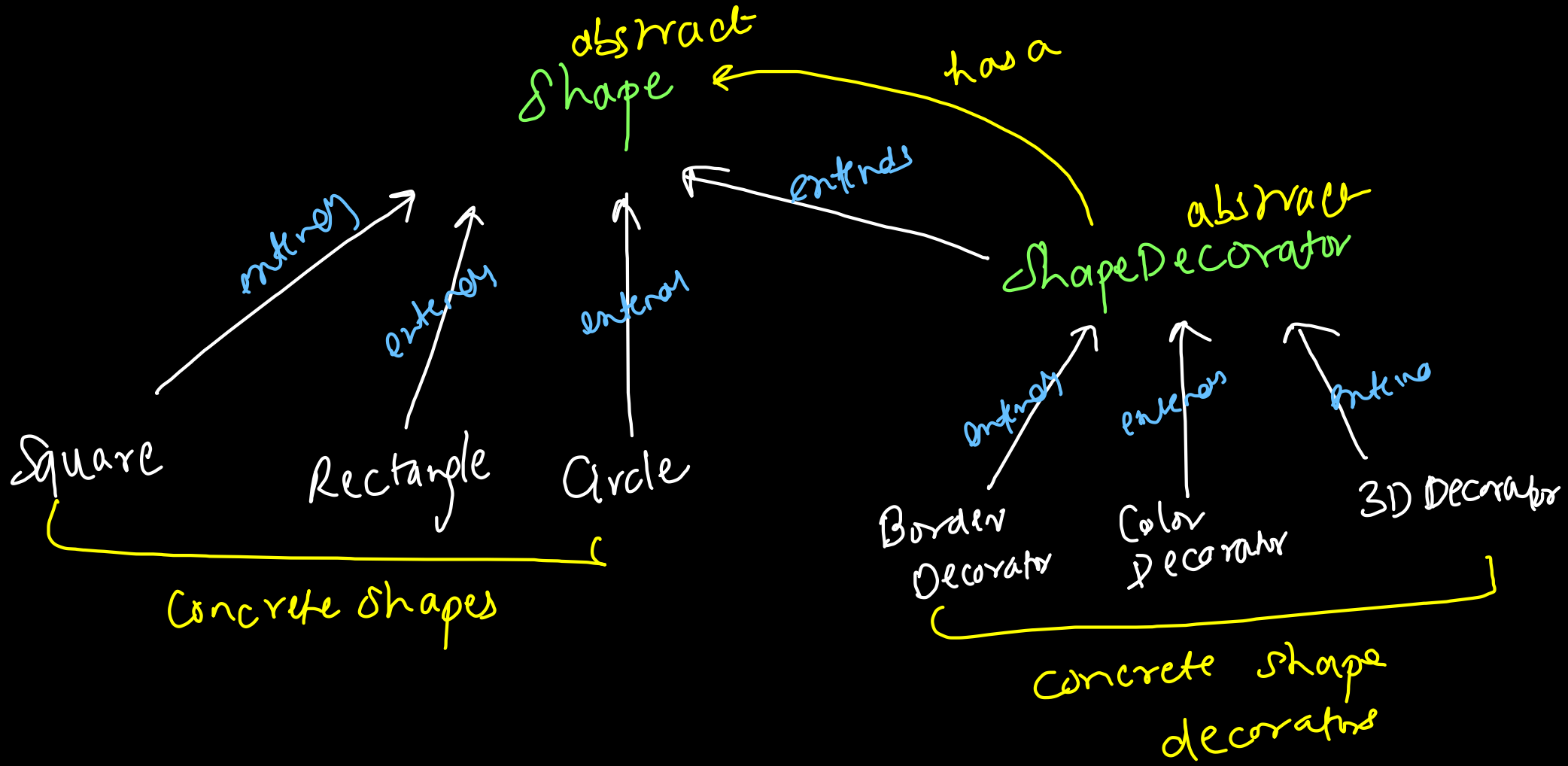
When Will We Need Decorator Design Pattern?

The following are the indicators which show the requirement of the decorator design pattern:

- When one has an object that requires functionality extension. For extending **functionality, decorators are a versatile alternative to subclassing**.
- When one has to **recursively rewrap** or change the functionalities of an object according to the requirements, dynamically without affecting other objects of the class.

A real-life **example** of a decorator design pattern would be a pizza, pizza base here would be the original class, and the variety of different toppings would act as the added functionalities. The customer can add toppings (**functionalities**) as per their choice and the pizza base (**original class**) will remain intact.

Structure


abstract
Shape
has a
extends
abstract
ShapeDecorator
extends
extends
extends
extends
extends
extends
Square
Rectangle
Circle
Border
Decorator
Color
Decorator
3D Decorator
Concrete Shapes
concrete shape
decorators

Pros:

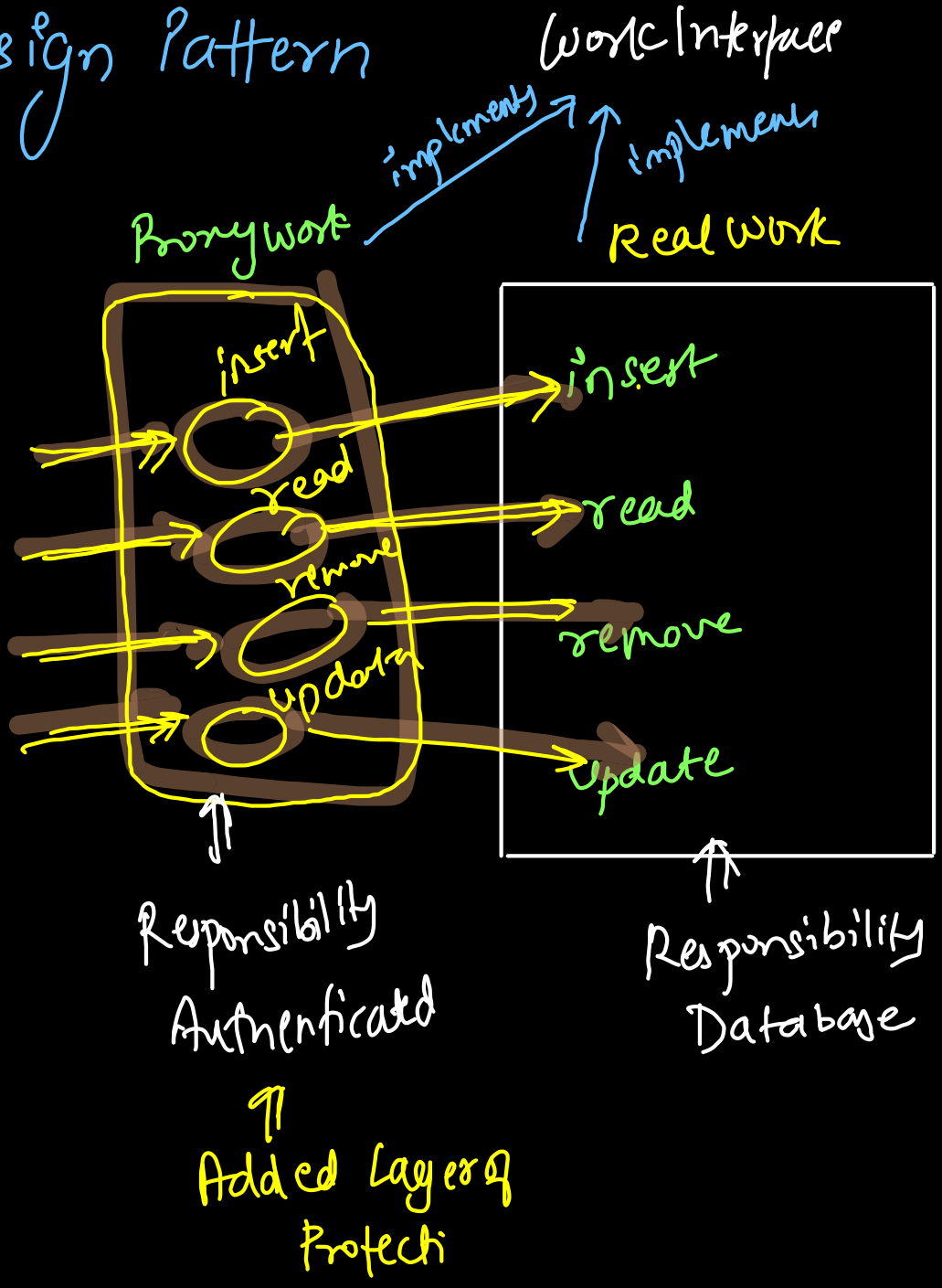
1. A decorator design pattern provides a **high degree of flexibility** as an alternative to subclassing for functionality extension.
2. Instead of editing the existing code, decorators allow behaviour modification at **runtime**.
3. Problems with permutation are resolved. Wrapping an object in numerous decorators allows you to **mix multiple behaviours**.
4. The decorator design pattern adheres to the **concept that classes** should be extensible but not modifiable.
5. Decorator pattern follows the Single **responsibility principle** which states that a monolithic class with multiple tasks can be broken down into various classes, each with a **particular responsibility** or task.

Cons:

1. Decorators might result in a **lot of small elements** in our design, which can be difficult to manage.
2. This design pattern is not **beginner-friendly**.
3. Debugging is quite difficult because of the **extended decorator components**.
4. The architecture may have **very high complexity**, especially due to the decorator interface.

⑧ Proxy Design Pattern

if in the same class
which is responsible
for interacting with database,
if I add responsibility of
user authorizatn/authencatn,
Single Responsbility Principle
is violated!



Client

```
WorkInterface dao2 = new AuthProxy();
((AuthProxy) dao2).password = "1234";

dao2.insert(val: 6, key: 2021);
dao2.insert(val: 5, key: 2022);
System.out.println(dao2.read(key: 2022));
```

```
public class AuthProxy implements WorkInterface {
    String password;
    WorkInterface realWork = Client.factory();

    @Override
    public void insert(int val, int key) {
        if (password.equals(anObject: "1234")) {
            realWork.insert(val, key);
        }
    }

    @Override
    public int read(int key) {
        if (password.equals(anObject: "1234")) {
            return realWork.read(key);
        }
        return -1;
    }

    @Override
    public void remove(int key) {
        if (password.equals(anObject: "1234")) {
            realWork.remove(key);
        }
    }

    @Override
    public void update(int val, int key) {
        if (password.equals(anObject: "1234")) {
            realWork.update(val, key);
        }
    }
}
```

delegar

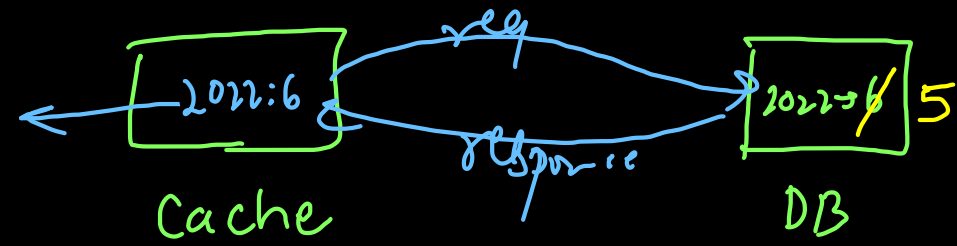
```
public class DatabaseWork implements WorkInterface {
    HashMap<Integer, Integer> hm = new HashMap<>();

    @Override
    public void insert(int val, int key) {
        hm.put(key, val);
    }

    @Override
    public int read(int key) {
        if (hm.containsKey(key) == false) {
            System.out.println(x: "Data Not Existing");
        }
        return hm.get(key);
    }

    @Override
    public void remove(int key) {
        if (hm.containsKey(key) == false) {
            System.out.println(x: "Data Not Existing");
        }
        hm.remove(key);
    }

    @Override
    public void update(int val, int key) {
        if (hm.containsKey(key) == false) {
            System.out.println(x: "Data Not Existing");
        }
        hm.put(key, val);
    }
}
```

```java
public class CacheProxy implements WorkInterface {
    HashMap<Integer, Integer> cache = new HashMap<>();
    // key: request, value: response
    WorkInterface nextWork = new AuthProxy();

    @Override
    public void insert(int val, int key) {
        nextWork.insert(val, key);
    }

    @Override
    public int read(int key) {
        if (cache.containsKey(key) == true) {
            return cache.get(key);
        }

        return nextWork.read(key);
    }

    @Override
    public void remove(int key) {
        nextWork.remove(key);
    }

    @Override
    public void update(int val, int key) {
        nextWork.update(val, key);
    }
}
```

→ data copy

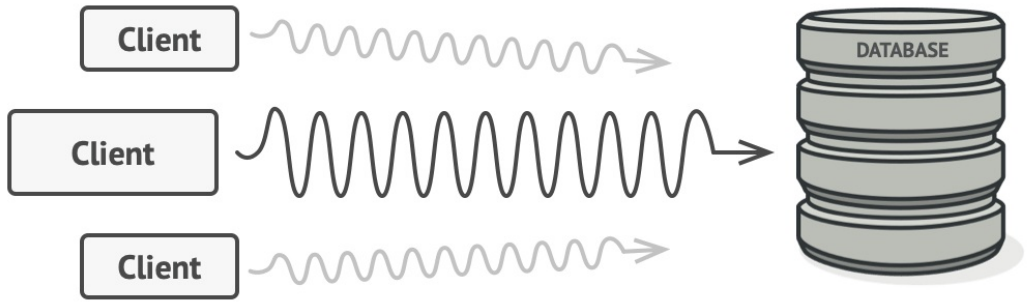
data inconsistency problem may arise

insert (2022, 6);
read (2022); : ⑥ Req-Res
update (2022, 5)
read (2022); ⑥ : Cache



☹ Problem

Why would you want to control access to an object? Here is an example: you have a massive object that consumes a vast amount of system resources. You need it from time to time, but not always.



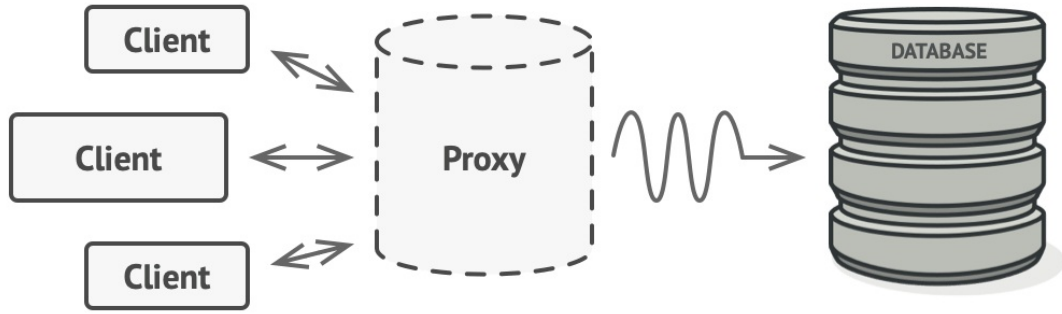
Database queries can be really slow.

You could implement lazy initialization: create this object only when it's actually needed. All of the object's clients would need to execute some deferred initialization code. Unfortunately, this would probably cause a lot of code duplication.

In an ideal world, we'd want to put this code directly into our object's class, but that isn't always possible. For instance, the class may be part of a closed 3rd-party library.

😊 Solution

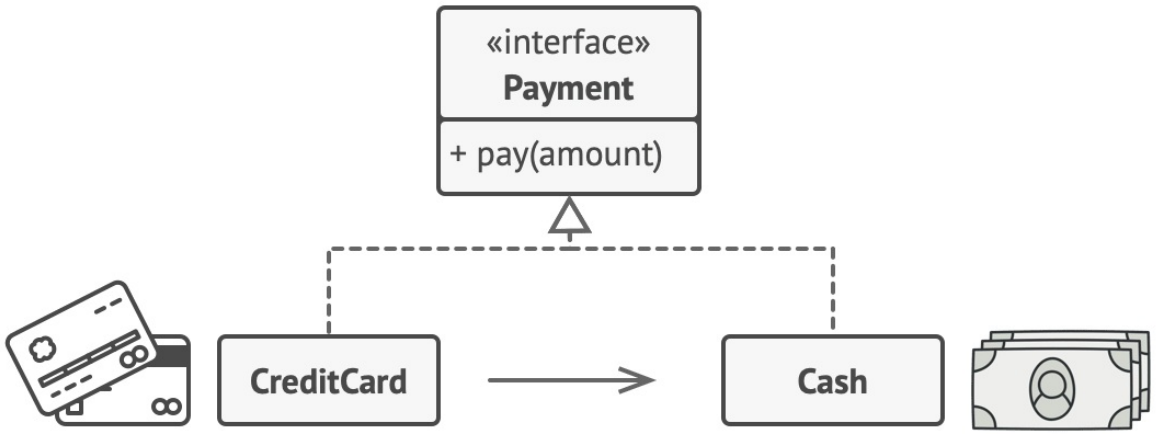
The Proxy pattern suggests that you create a new proxy class with the same interface as an original service object. Then you update your app so that it passes the proxy object to all of the original object's clients. Upon receiving a request from a client, the proxy creates a real service object and delegates all the work to it.



The proxy disguises itself as a database object. It can handle lazy initialization and result caching without the client or the real database object even knowing.

But what's the benefit? If you need to execute something either before or after the primary logic of the class, the proxy lets you do this without changing that class. Since the proxy implements the same interface as the original class, it can be passed to any client that expects a real service object.

Real-World Analogy

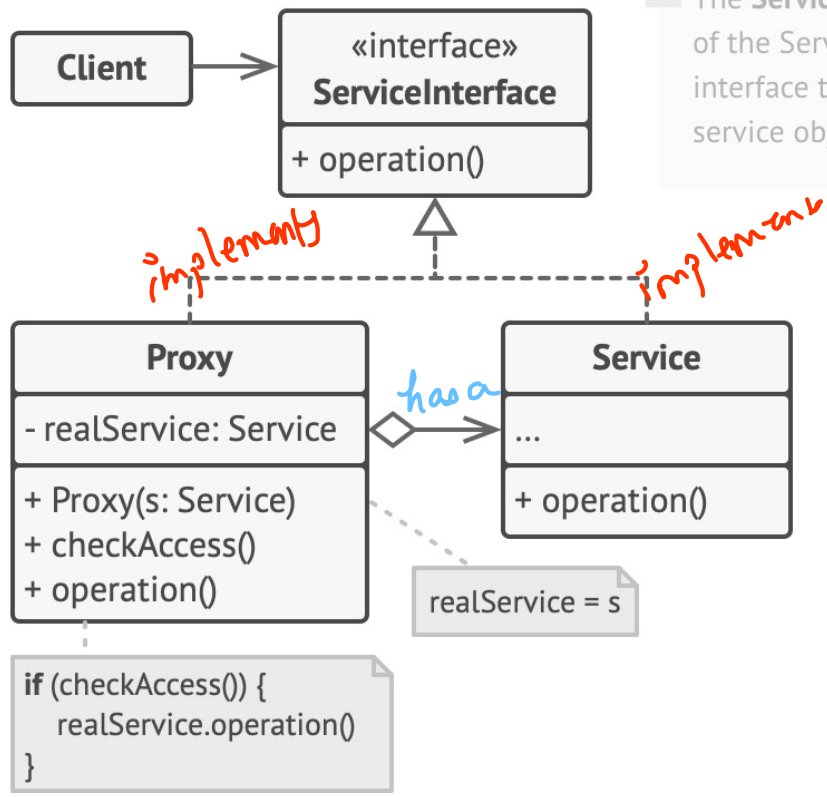


Credit cards can be used for payments just the same as cash.

A credit card is a proxy for a bank account, which is a proxy for a bundle of cash. Both implement the same interface: they can be used for making a payment. A consumer feels great because there's no need to carry loads of cash around. A shop owner is also happy since the income from a transaction gets added electronically to the shop's bank account without the risk of losing the deposit or getting robbed on the way to the bank.

Structure

4 The **Client** should work with both services and proxies via the same interface. This way you can pass a proxy into any code that expects a service object.



1 The **Service Interface** declares the interface of the Service. The proxy must follow this interface to be able to disguise itself as a service object.

3 The **Proxy** class has a reference field that points to a service object. After the proxy finishes its processing (e.g., lazy initialization, logging, access control, caching, etc.), it passes the request to the service object.

Usually, proxies manage the full lifecycle of their service objects.

2 The **Service** is a class that provides some useful business logic.

Pros and Cons

- ✓ You can control the service object without clients knowing about it.
- ✓ You can manage the lifecycle of the service object when clients don't care about it.
- ✓ The proxy works even if the service object isn't ready or is not available.
- ✓ *Open/Closed Principle*. You can introduce new proxies without changing the service or clients.

- ✗ The code may become more complicated since you need to introduce a lot of new classes.
- ✗ The response from the service might get delayed.

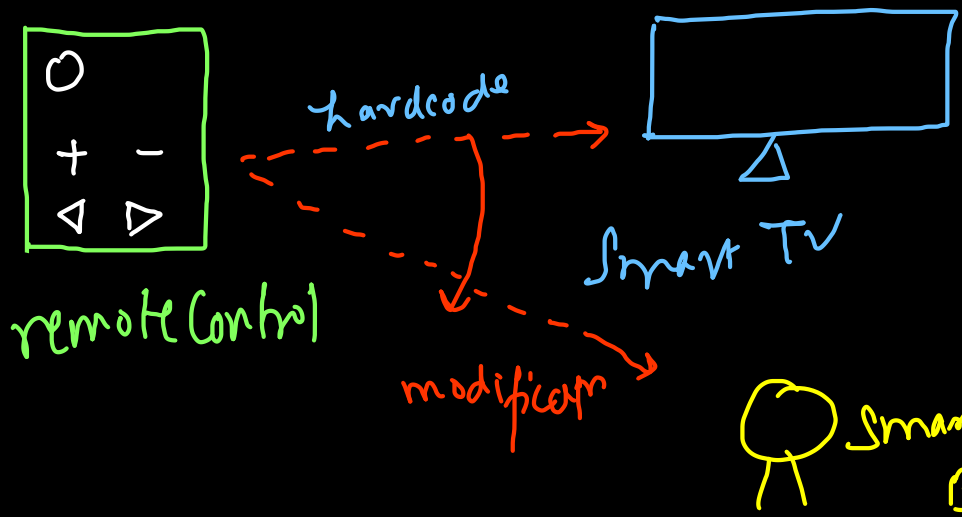
9 Command Design Pattern

Invoker

giving command

Receiver

performing command



```java
public class RemoteControl {
    Television receiver = new Television(

    public void increaseButton() {
        receiver.increaseButton();
    }

    public void decreaseButton() {
        receiver.decreaseButton();
    }

    public void powerButton() {
        receiver.powerButton();
    }

    public void nextButton() {
        receiver.nextButton();
    }

    public void previousButton() {
        receiver.previousButton();
    }
}
```

tightly coupling

```java
public class Television {
    public void increaseButton() {
        System.out.println(x: "Volume Increase");
    }

    public void decreaseButton() {
        System.out.println(x: "Volume Decrease");
    }

    public void powerButton() {
        System.out.println(x: "Power On/Off");
    }

    public void nextButton() {
        System.out.println(x: "Next Channel");
    }

    public void previousButton() {
        System.out.println(x: "Previous Channel");
    }
}
```

```java
    public static void main(String[] args) {
        RemoteControl remote = new
        RemoteControl();

        remote.powerButton();
        remote.increaseButton();
        remote.nextButton();
    }
}
```

```java
public class RemoteControl {
    ICommand receiver;

    public void setReceiver(ICommand receiver) {
        this.receiver = receiver;
    }

    public void increaseButton() {
        receiver.increaseButton();
    }

    public void decreaseButton() {
        receiver.decreaseButton();
    }

    public void powerButton() {
        receiver.powerButton();
    }

    public void nextButton() {
        receiver.nextButton();
    }

    public void previousButton() {
        receiver.previousButton();
    }
}
```

Depending Inversion

```java
public class Client {
    Run | Debug
    public static void main(String[] args) {
        RemoteControl remote = new RemoteControl();
        remote.setReceiver(new Television());
        remote.powerButton();
        remote.increaseButton();
        remote.nextButton();

        remote.setReceiver(new LED());
        remote.powerButton();
        remote.increaseButton();
        remote.nextButton();
    }
}
```

any receiver

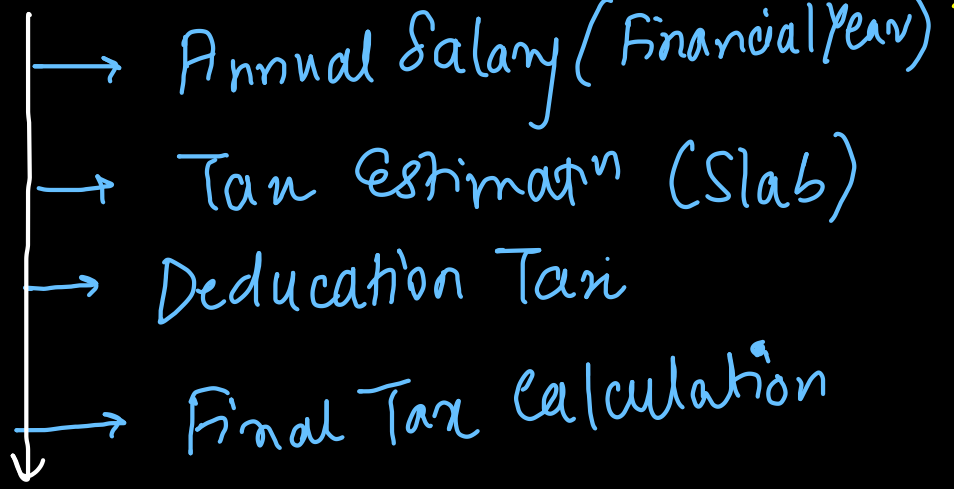
```
Power On/Off
Volume Increase
Next Channel
Power On/Off
Brightness Increase
Next Color
```

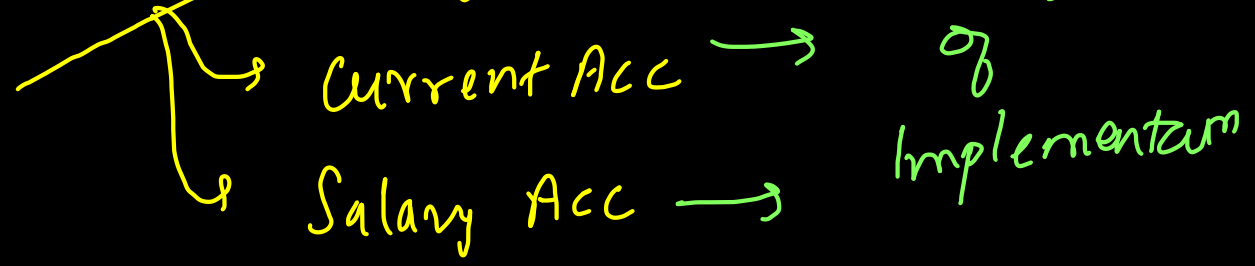
```java
public class LED implements ICommand {
    public void increaseButton() {
        System.out.println(x: "Brightness Increase");
    }

    public void decreaseButton() {
        System.out.println(x: "Brightness Decrease");
    }

    public void powerButton() {
        System.out.println(x: "Power On/Off");
    }

    public void nextButton() {
        System.out.println(x: "Next Color");
    }

    public void previousButton() {
        System.out.println(x: "Previous Color");
    }
}
```

```java
public class Television implements ICommand {
    public void increaseButton() {
        System.out.println(x: "Volume Increase");
    }

    public void decreaseButton() {
        System.out.println(x: "Volume Decrease");
    }

    public void powerButton() {
        System.out.println(x: "Power On/Off");
    }

    public void nextButton() {
        System.out.println(x: "Next Channel");
    }

    public void previousButton() {
        System.out.println(x: "Previous Channel");
    }
}
```

(10) Template Method Design Pattern

eg Tax/TDS Calculation





Set of submodules → Same

```java
public abstract class Account {
    public double taxCalculator() {
        double annualIncome = incomeCalculator();
        double estimatedTax = estimatedTaxCalculator(annualIncome);
        double taxDeduction = taxDeductionCalculator(estimatedTax);
        return taxDeduction;
    }

    public abstract double incomeCalculator();

    public abstract double estimatedTaxCalculator(double
    annualIncome);

    public abstract double taxDeductionCalculator(double
    estimatedTax);
}
```

Template method

extends

```java
public class CurrentAccount extends Account {

    @Override
    public double estimatedTaxCalculator(double annualIncome) {
        return (annualIncome * 0.05);
    }

    @Override
    public double incomeCalculator() {
        double revenue = 1000000000.0;
        return revenue;
    }

    @Override
    public double taxDeductionCalculator(double estimatedTax) {
        return estimatedTax * 0.98;
    }

}
```

extends

```java
public class SavingsAccount extends Account {
    @Override
    public double estimatedTaxCalculator(double annualIncome) {
        return (annualIncome * 0.10);
    }

    @Override
    public double incomeCalculator() {
        double credit = 1000000;
        return credit;
    }

    @Override
    public double taxDeductionCalculator(double estimatedTax) {
        return estimatedTax * 1.00;
    }
}
```

extends

```java
public class SalaryAccount extends Account {
    @Override
    public double estimatedTaxCalculator(double annualIncome) {
        return (annualIncome * 0.15);
    }

    @Override
    public double incomeCalculator() {
        double salary = 100000000;
        return salary;
    }

    @Override
    public double taxDeductionCalculator(double estimatedTax) {
        return estimatedTax * 0.90;
    }
}
```

Abstract (Summary)

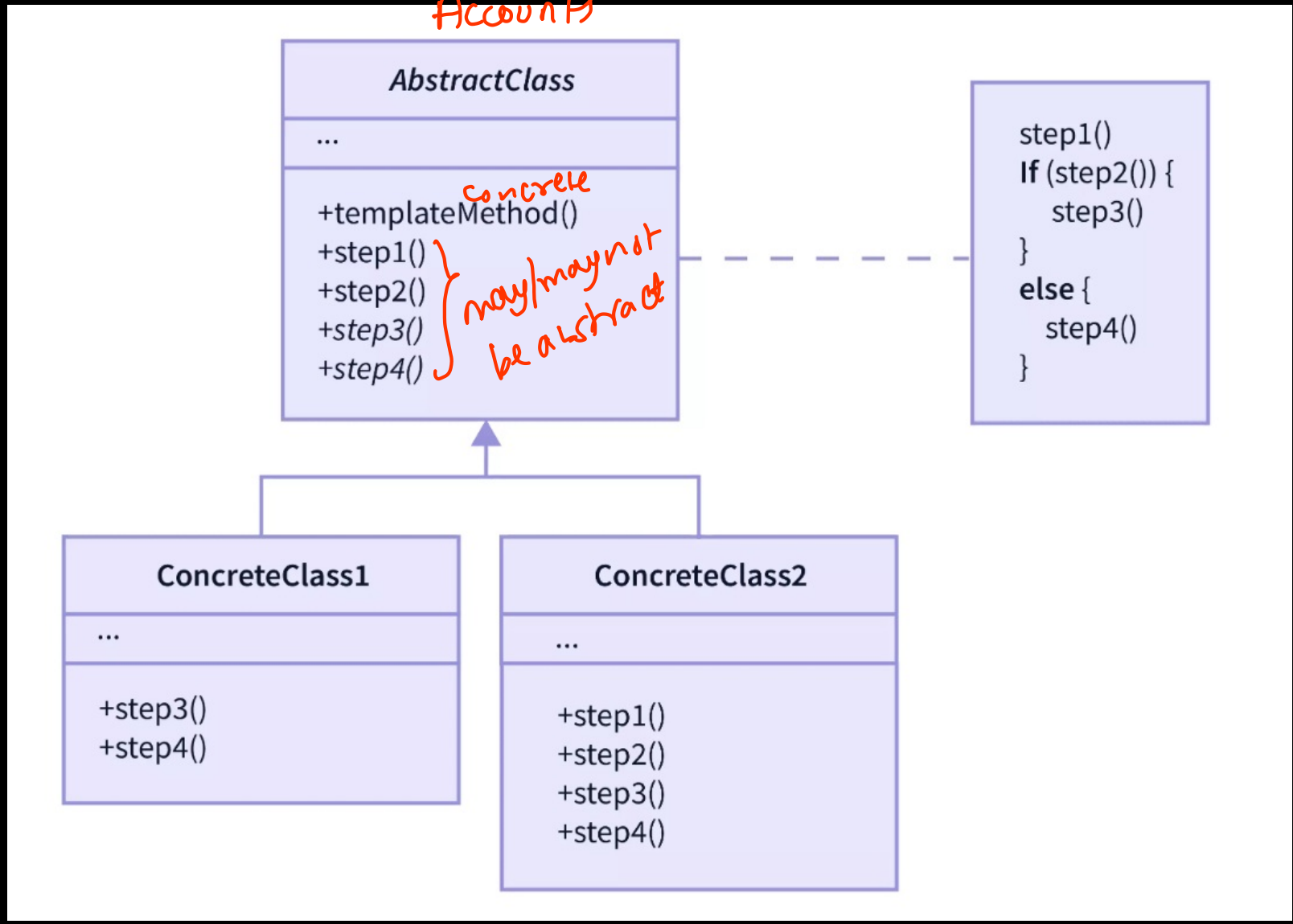
The template method design pattern falls under the category of behavioral design pattern. It describes a technique (also known as an algorithm) that is in the form of a skeleton of functions. The child classes are responsible for implementing the specifics for this method. The parent class maintains the algorithm's basic framework as well as sequencing.

When will we need the template method design pattern?

Frameworks make extensive application of the Template Method Design Pattern. Every framework executes the immutable components of a domain's design as well as it creates placeholders for any essential or relevant user modification choices. As a result, the framework becomes the centre of the world, while the client modifications can be said as just the fourth rock from the sun in the sky.

Structure

Account



Pros and Cons of Template Method Design Pattern

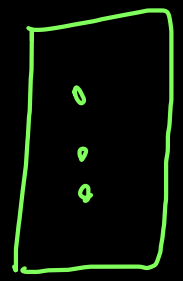
Pros of template method design pattern-

- This behavioural design pattern is among the simplest to grasp as well as apply.
- The design pattern is generally utilized in the development of frameworks.
- The code becomes much cleaner since it also aims to alleviate the duplication of code.

Cons of template method design pattern-

- At times, it might be difficult to debug as well as comprehend the flow sequence in the Template method pattern design. There might be cases where the programmer might find themselves applying a function that should not be implemented or you might not implement a given abstract method at all. The programmer is responsible for documentation as well as stringent error handling.
- It might be a difficult task to maintain the template framework since any kind of modifications in the lower or higher level might conflict with the implementation.

(14) Visitor Design Pattern



Class
Production

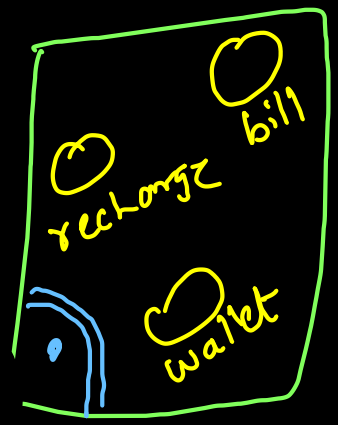
add functionality

⇒ open for modification { Tested, Deployment }

⇒ open for extension.

Wrong Solution

1. Modification ✗
2. Inherit the parent class ✗ { violate Liskov Substitution
   ↳ UPI is not Payim
3. Decouple { write code of UPI in new class }
   { coupling → low }
   Cohesion ↓



Paytm

V1.0

V1.1

V1.2

V1.3

V1.4

```java
public class Paytm {
    public void wallet() {
        System.out.println(x: "Wallet Service");
    }

    public void recharge() {
        System.out.println(x: "Recharge Mobile");
    }

    public void billPayment() {
        System.out.println(x: "Electricity Bill Payment");
    }

    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
}
```

```java
public interface Visitor {
    public void visit(Paytm app);
}
```

// to accept new functionality w/o modifying the existing code.

```java
public class Paytm {
    public void wallet() {
        System.out.println(x: "Wallet Service");
    }

    public void recharge() {
        System.out.println(x: "Recharge Mobile");
    }

    public void billPayment() {
        System.out.println(x: "Electricity Bill
        Payment");
    }

    public void accept(Visitor visitor) {
        visitor.visit(this);
    }
}
```

```java
public class Client {
    Run | Debug
    public static void main(String[] args) {
        Paytm app = new Paytm();
        app.wallet();
        app.recharge();
        app.billPayment();

        Visitor upi = new UPIVisitor();
        app.accept(upi);
    }
}
```

visitor = upi

high cohesion (strongly)

have

have

```java
public interface Visitor {
    public void visit(Paytm app);
}
```

implements

```java
public class.UPIVisitor implements Visitor {
    @Override
    public void visit(Paytm app) {
        System.out.println(x: "Added UPI Functionality
        in Paytm");
    }
}
```

Overview

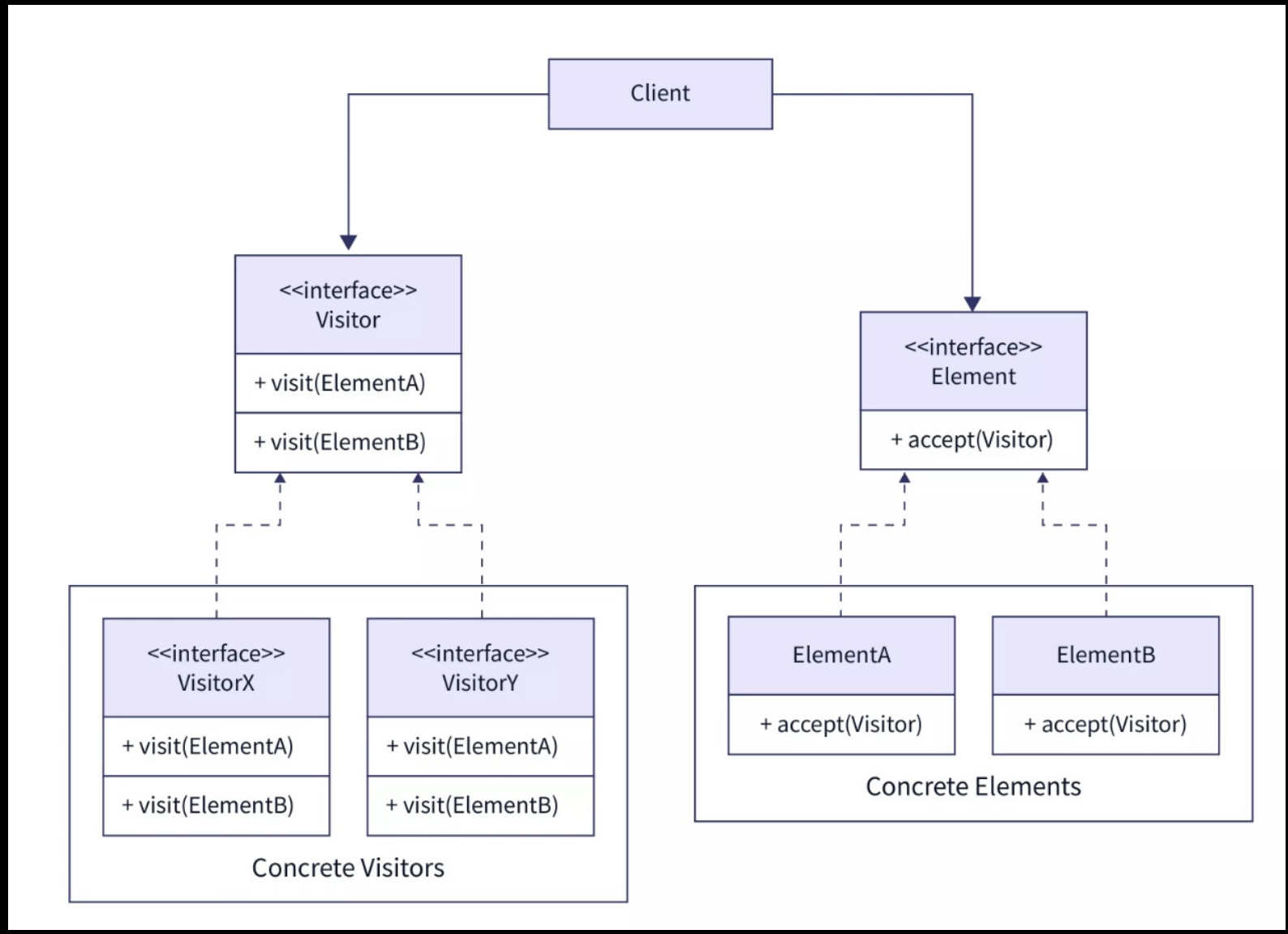
A visitor design pattern is a behavioral design pattern used to decouple the **logic/algorithm** from the objects on which they operate. The logic is moved to separate classes called **visitors**. Each visitor is responsible for performing a specific operation.

When Will We Need the Visitor Design Patterns?

We have a list consisting of many shapes **(square, circle, rectangle, etc.)**, and we want to calculate values like **area**, **perimeter** of those shapes. The straightforward way is to have area() and perimeter() methods in the respective shape classes. But in this method, we are **tightly coupling the algorithm (calculate area, perimeter, etc.) from the objects (square, circle, rectangle, etc.) on which they operate**.

With the visitor design patterns, we can remove this tight coupling by separating the algorithms and the objects on which they operate. In the future, if any new algorithms are to be added to the objects, they can be added separately without touching the underlying objects. This leverages the **Open/Closed principle** (i.e.) **classes should be open for extension but closed for modification**.

Structure


Client
<<interface>>
Visitor
+ visit(ElementA)
+ visit(ElementB)
<<interface>>
Element
+ accept(Visitor)
<<interface>>
VisitorX
+ visit(ElementA)
+ visit(ElementB)
<<interface>>
VisitorY
+ visit(ElementA)
+ visit(ElementB)
Concrete Visitors
ElementA
+ accept(Visitor)
ElementB
+ accept(Visitor)
Concrete Elements

Pros and Cons of Visitor Design Pattern

Pros

1. Adding a new operation to all the Elements can be done easily by **adding a new Concrete** Visitor that implements the Visitor interface. The Element classes are **left untouched**.
2. Visitors can maintain the state when they visit each element and are **encapsulated.** The data can be **retrieved through a separate public method** (get()) provided by the visitors.

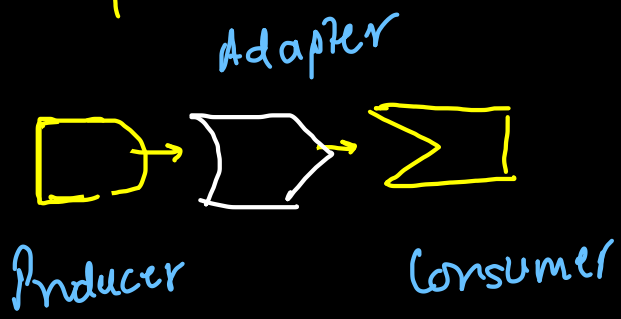
Cons

1. A new class is added for every **new operation** performed on the Elements.
2. All the visitor classes have to be changed for every **new element introduced**.

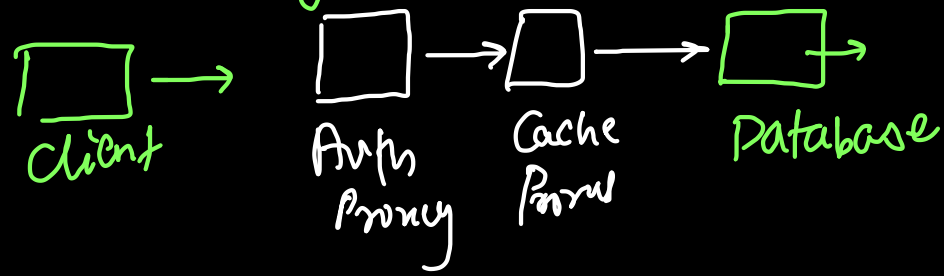
(42) Facade Design Pattern

↳ mask

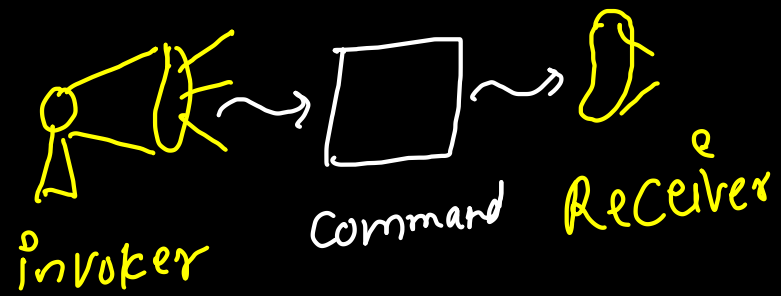
Adapter Pattern



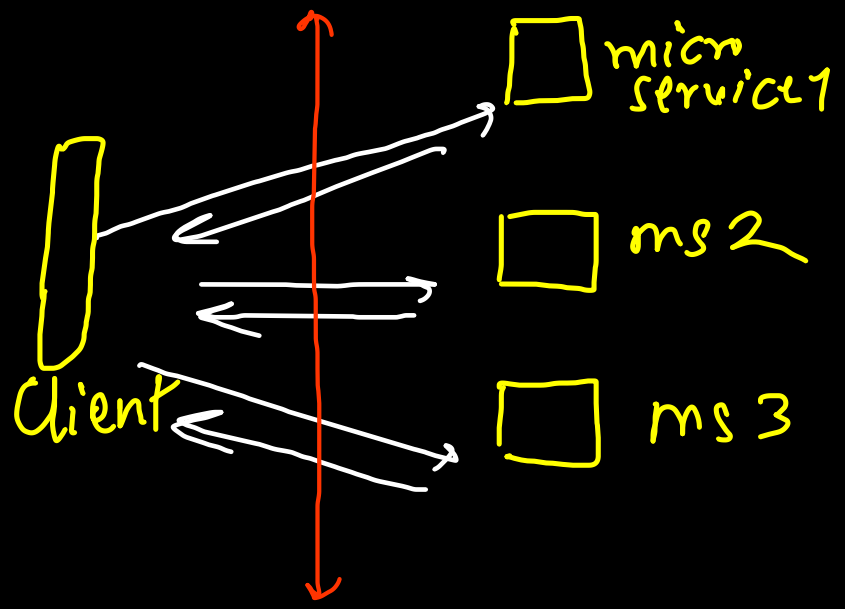
Proxy Pattern



Command Pattern

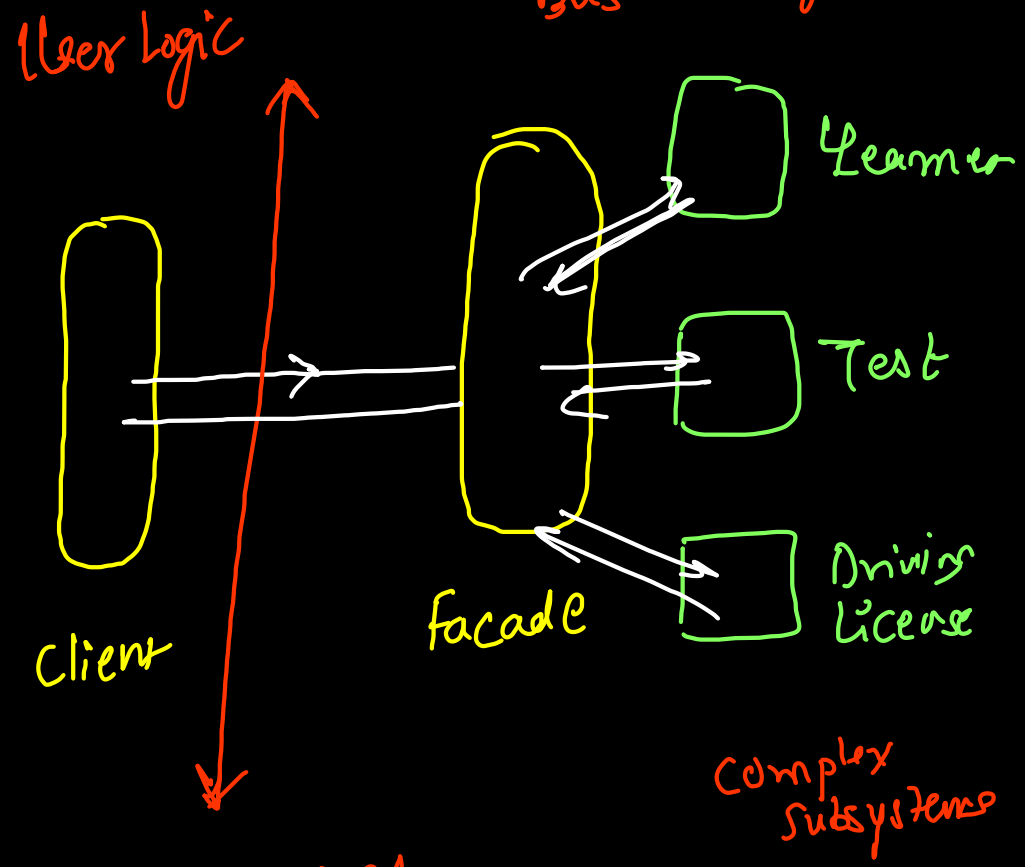


Problem

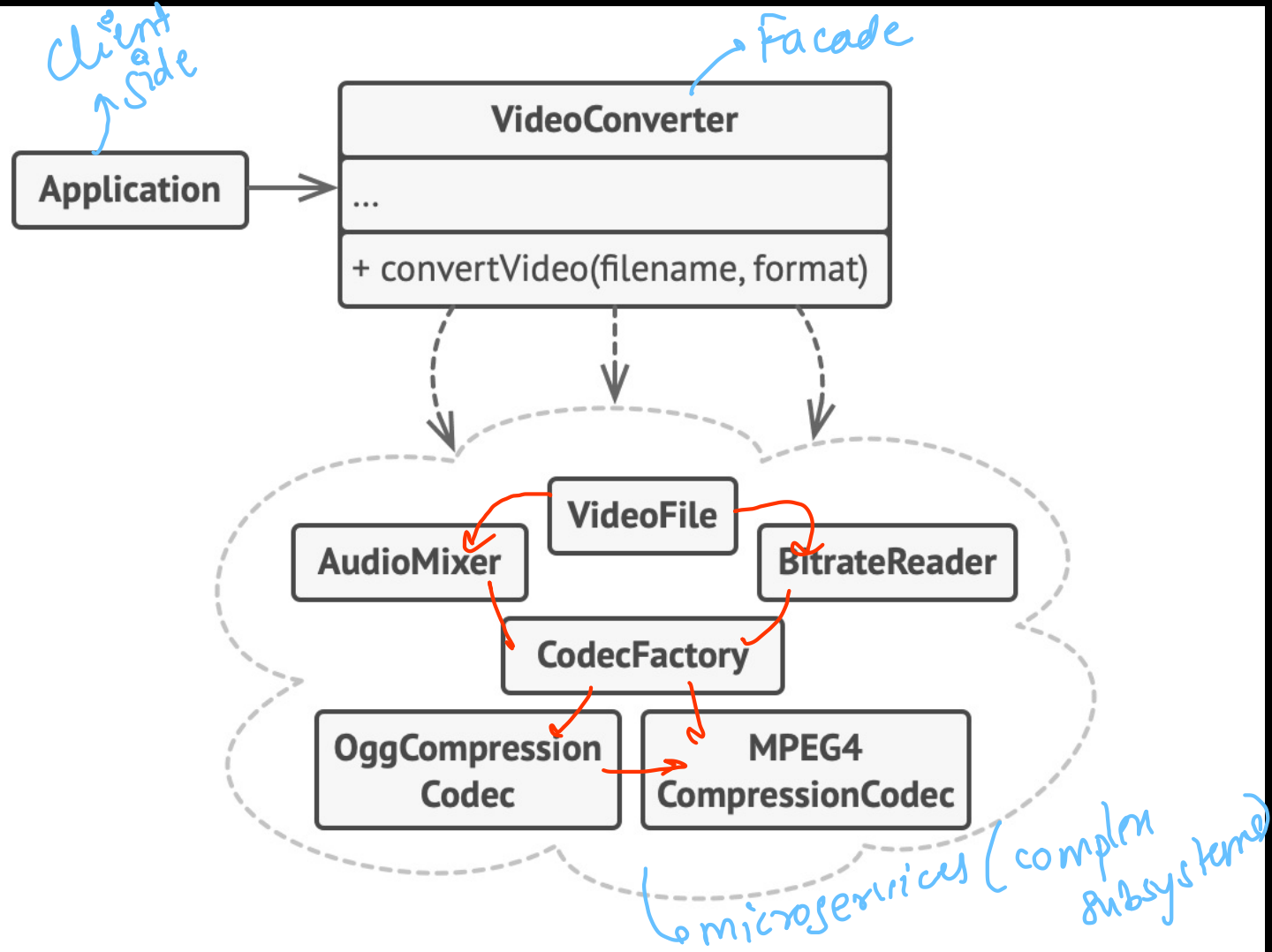


Solution

Business Logic



Client → abstract → Microservices

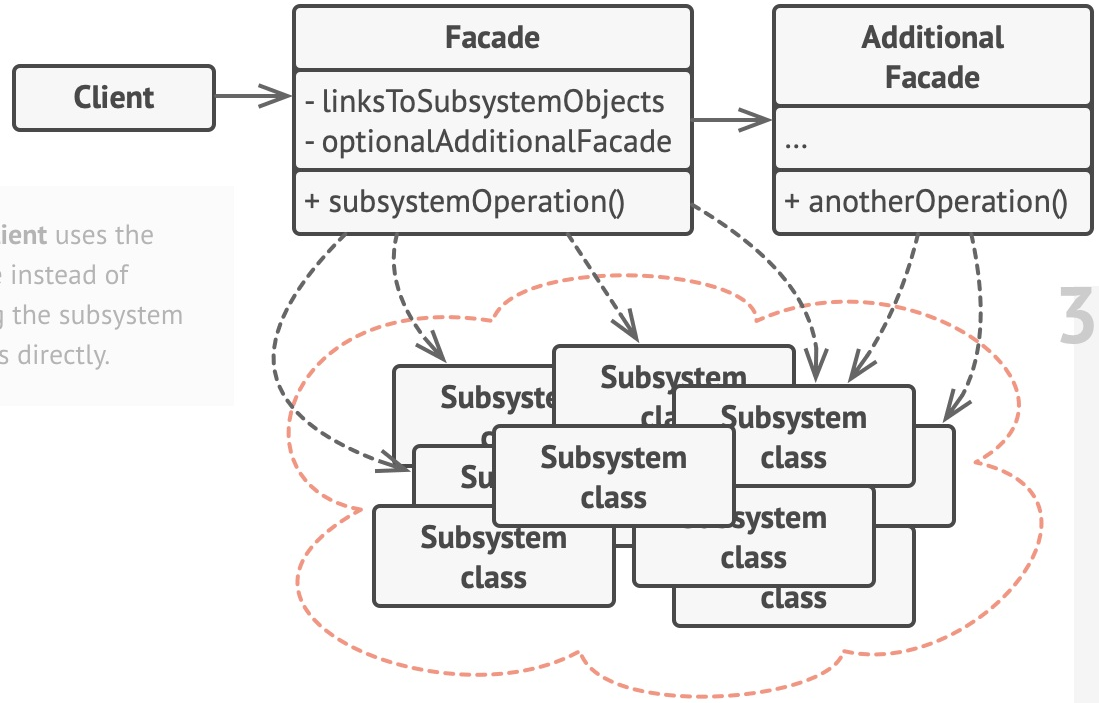


An example of isolating multiple dependencies within a single facade class.

Structure

1 The **Facade** provides convenient access to a particular part of the subsystem's functionality. It knows where to direct the client's request and how to operate all the moving parts.

2 An **Additional Facade** class can be created to prevent polluting a single facade with unrelated features that might make it yet another complex structure. Additional facades can be used by both clients and other facades.



4 The **Client** uses the facade instead of calling the subsystem objects directly.

3 The **Complex Subsystem** consists of dozens of various objects. To make them all do something meaningful, you have to dive deep into the subsystem's implementation details, such as initializing objects in the correct order and supplying them with data in the proper format.

Subsystem classes aren't aware of the facade's existence. They operate within the system and work with each other directly.

```java
package Facade;

class Uploader {
    public void upload(String s) {
        System.out.println(x: "File uploaded");
    }
}

class Downloader {
    public String download() {
        System.out.println(x: "File Downloaded");
        return "video.ma4";
    }
}

class Adapter {
    public void convertor() {
        System.out.println(x: "File Conversion");
    }
}

public class VideoConvertorFacade {
    public static String work(String str) {
        Uploader o1 = new Uploader();
        o1.upload(str);

        Adapter o2 = new Adapter();
        o2.convertor();

        Downloader o3 = new Downloader();
        return o3.download();
    }
}
```

```java
package Facade;

public class Client {
    Run | Debug
    public static void main(String[] args) {
        VideoConvertorFacade video = new VideoConvertorFacade();
        System.out.println(video.work(str: "video.mp4"));
    }
}
```

```
File uploaded
File Conversion
File Downloaded
video.ma4
```

How to Implement

1. Check whether it's possible to provide a simpler interface than what an existing subsystem already provides. You're on the right track if this interface makes the client code independent from many of the subsystem's classes.

2. Declare and implement this interface in a new facade class. The facade should redirect the calls from the client code to appropriate objects of the subsystem. The facade should be responsible for initializing the subsystem and managing its further life cycle unless the client code already does this.

3. To get the full benefit from the pattern, make all the client code communicate with the subsystem only via the facade. Now the client code is protected from any changes in the subsystem code. For example, when a subsystem gets upgraded to a new version, you will only need to modify the code in the facade.

4. If the facade becomes **too big**, consider extracting part of its behavior to a new, refined facade class.

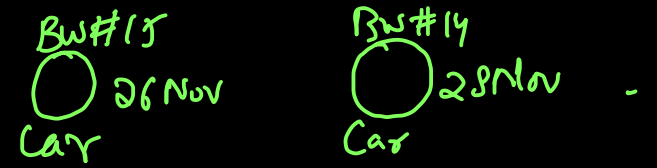
Pros and Cons

- ✓ You can isolate your code from the complexity of a subsystem.
- ✗ A facade can become **a god object** coupled to all classes of an app.

Client ⊄ Microservices

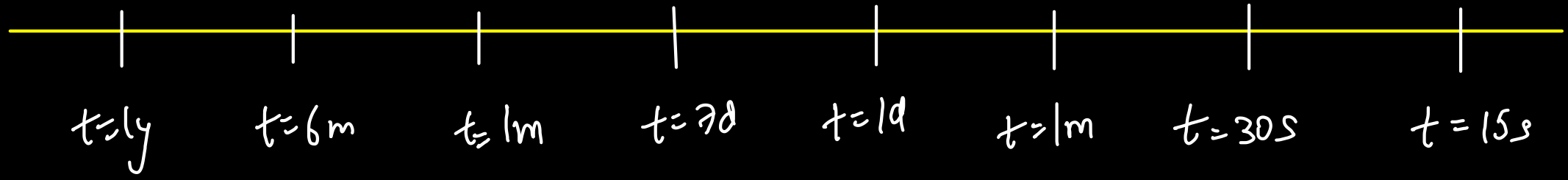
(snapshots)

13 Memento Design Pattern



eg1) timeseries database → Account statement

Hashmap

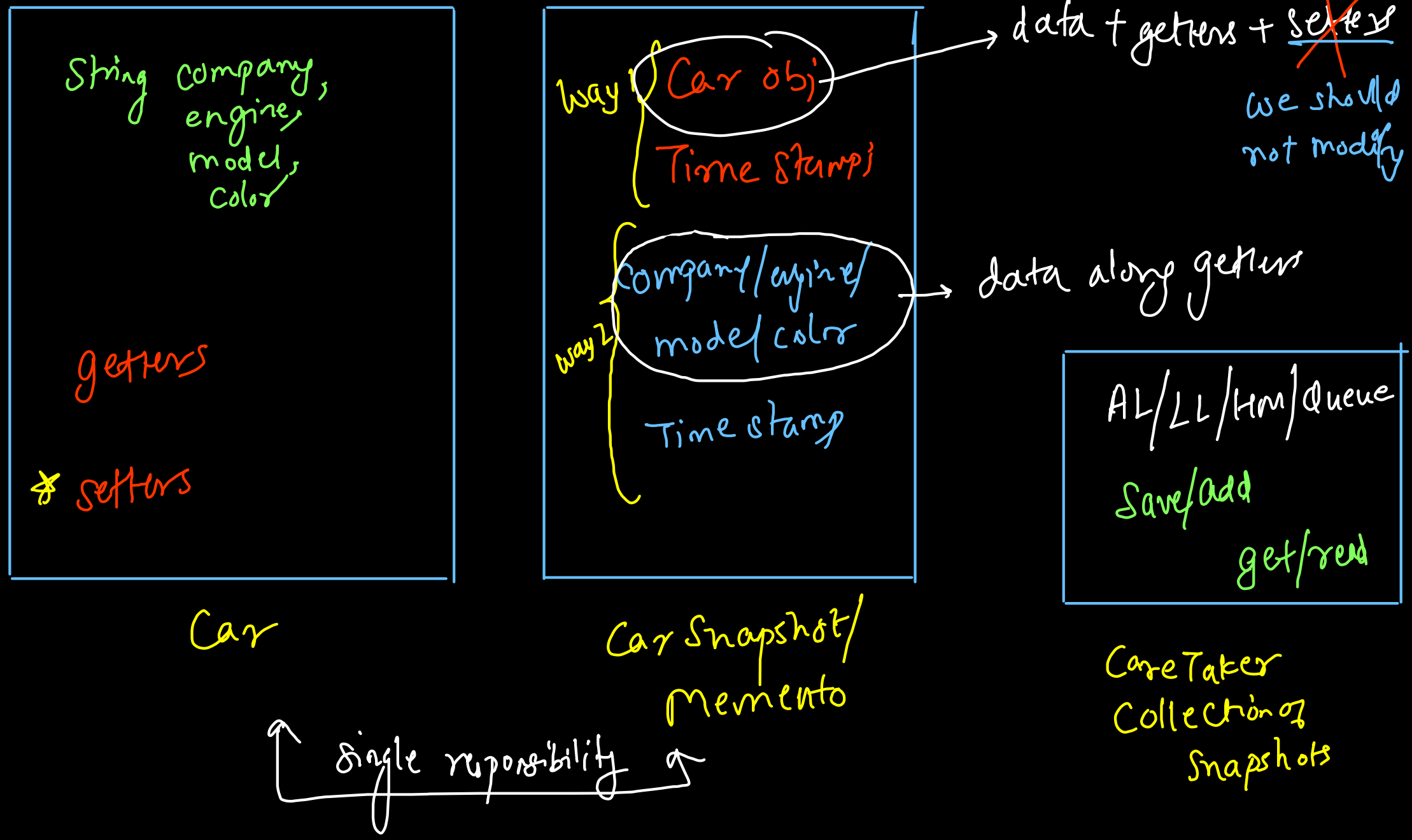


eg2 Text editor ↔ cut/copy paste ↺ undo/redo ⇒ Stack Data Structure

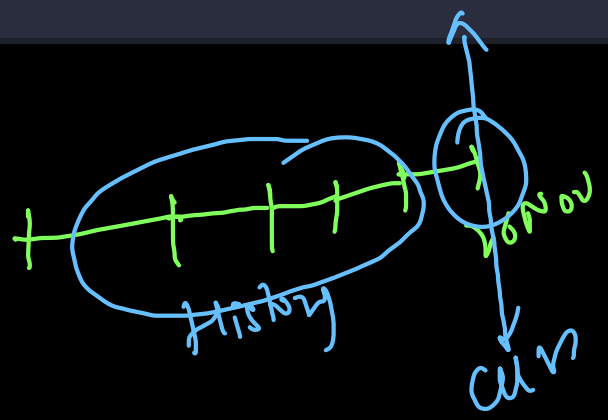
eg3 Games NFS mostwanted → BW#13 → checkpoint, BW#14 → checkpoint

timestamp =

ArrayList / LinkedList


String company, engine, model, color
getters
* setters
Car
way 1
Car obj
Time stamp
data + getters + setters
we should not modify
way 2
company/engine/ model/color
data along getters
Time stamp
Car Snapshot/ Memento
AL/LL/HM/Queue
Save/add
get/read
CareTaker
Collection of Snapshots
single responsibility

```java
public class Car {
    private String color, name;

    public String getColor() {
        return color;
    }

    public void setColor(String color) {
        this.color = color;
    }

    public String getName() {
        return name;
    }

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




```java
public class CarSnapshot {
    private String color, name;
    LocalTime timestamp;

    public LocalTime getTimestamp() {
        return timestamp;
    }

    public String getColor() {
        return color;
    }

    public String getName() {
        return name;
    }

    CarSnapshot(Car car) {
        this.color = car.getColor();
        this.name = car.getName();
        timestamp = LocalTime.now();
    }

    public String toString() {
        return ("Color : " + color + " , Name : "
        + name);
    }
}
```

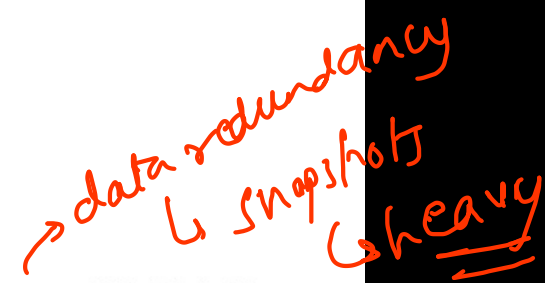
```java
// CareTaker
public class TimeSeriesDB {
    static HashMap<LocalTime, CarSnapshot> db;

    TimeSeriesDB() {
        db = new HashMap<>();
    }

    public void save(CarSnapshot car) {
        db.put(car.getTimestamp(), car);
    }

    public CarSnapshot getSnapshot(LocalTime time)
    {
        return db.get(time);
    }
}
```

```java
public class Client {      You, 25 seconds ago • Ad
    Run | Debug
    public static void main(String[] args) {
        Car blacklistEasy = new Car();
        blacklistEasy.setColor(color: "red");
        blacklistEasy.setName(name: "BMW");

        Car blacklistMedium = new Car();
        blacklistMedium.setColor(color: "green");
        blacklistMedium.setName(name: "Ferrari");

        Car blacklistHard = new Car();
        blacklistHard.setColor(color: "black");
        blacklistHard.setName(name: "Mustang");

        CarSnapshot s1 = new CarSnapshot
        (blacklistEasy);
        CarSnapshot s2 = new CarSnapshot
        (blacklistMedium);
        CarSnapshot s3 = new CarSnapshot
        (blacklistHard);

        TimeSeriesDB db = new TimeSeriesDB();

        db.save(s1);
        db.save(s2);
        db.save(s3);

        LocalTime time = s2.getTimestamp();
        System.out.println(db.getSnapshot(time));
    }
}
```

color: green
name: ferrari

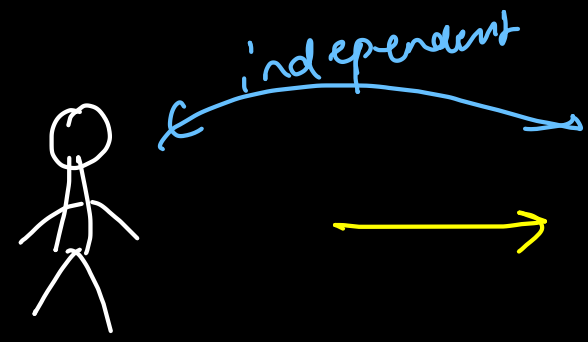
⚖ Pros and Cons

- ✓ You can produce snapshots of the object's state without violating its encapsulation.
- ✓ You can simplify the originator's code by letting the caretaker maintain the history of the originator's state.
- ✗ The app might consume lots of RAM if clients create mementos too often.
- ✗ Caretakers should track the originator's lifecycle to be able to destroy obsolete mementos.
- ✗ Most dynamic programming languages, such as PHP, Python and JavaScript, can't guarantee that the state within the memento stays untouched.

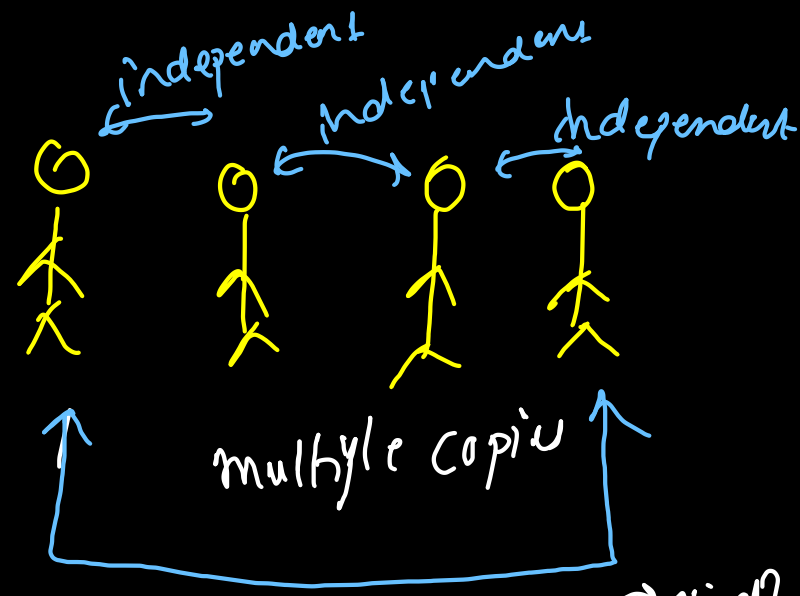


(14) Prototype Design Pattern {Creational Design Pattern}

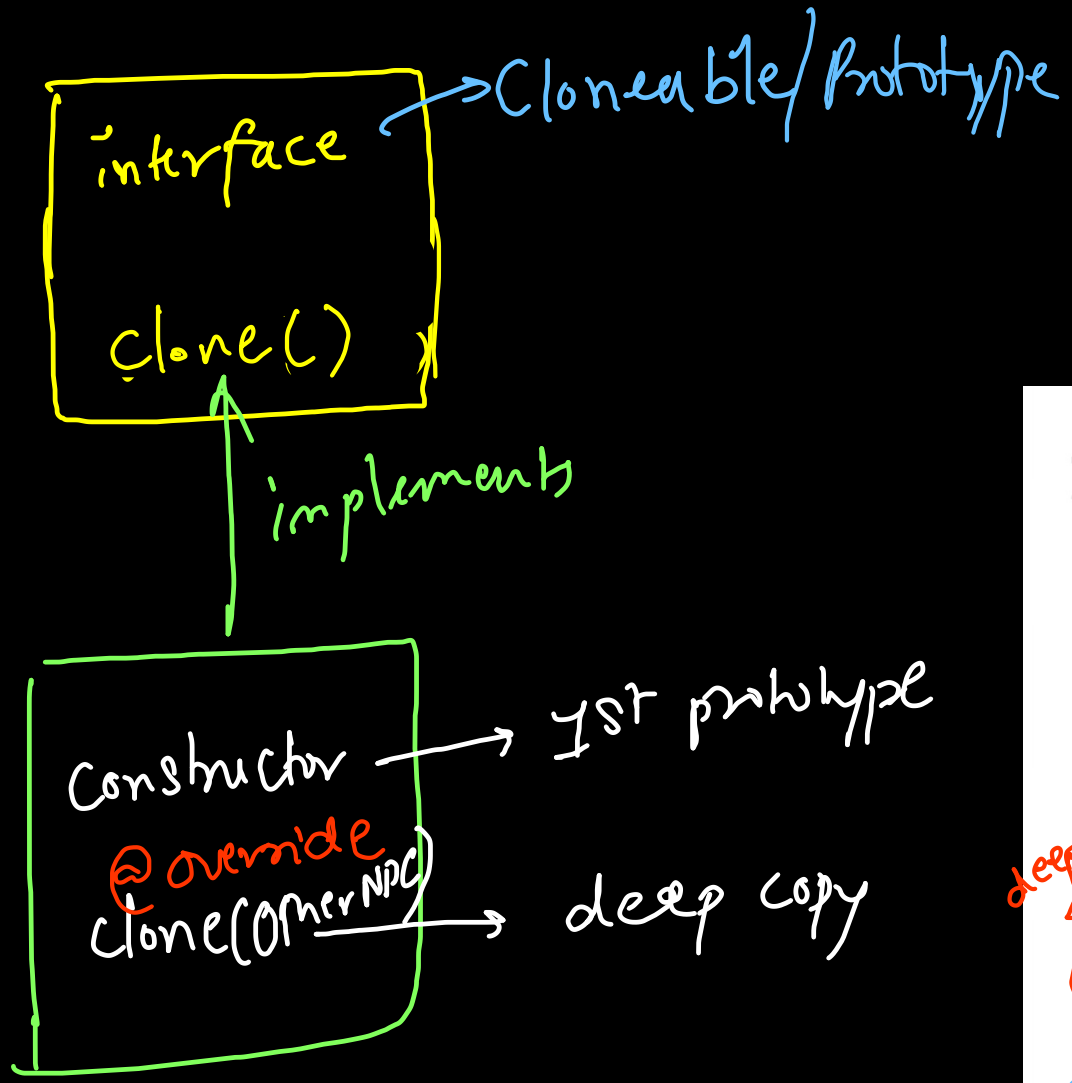
Games



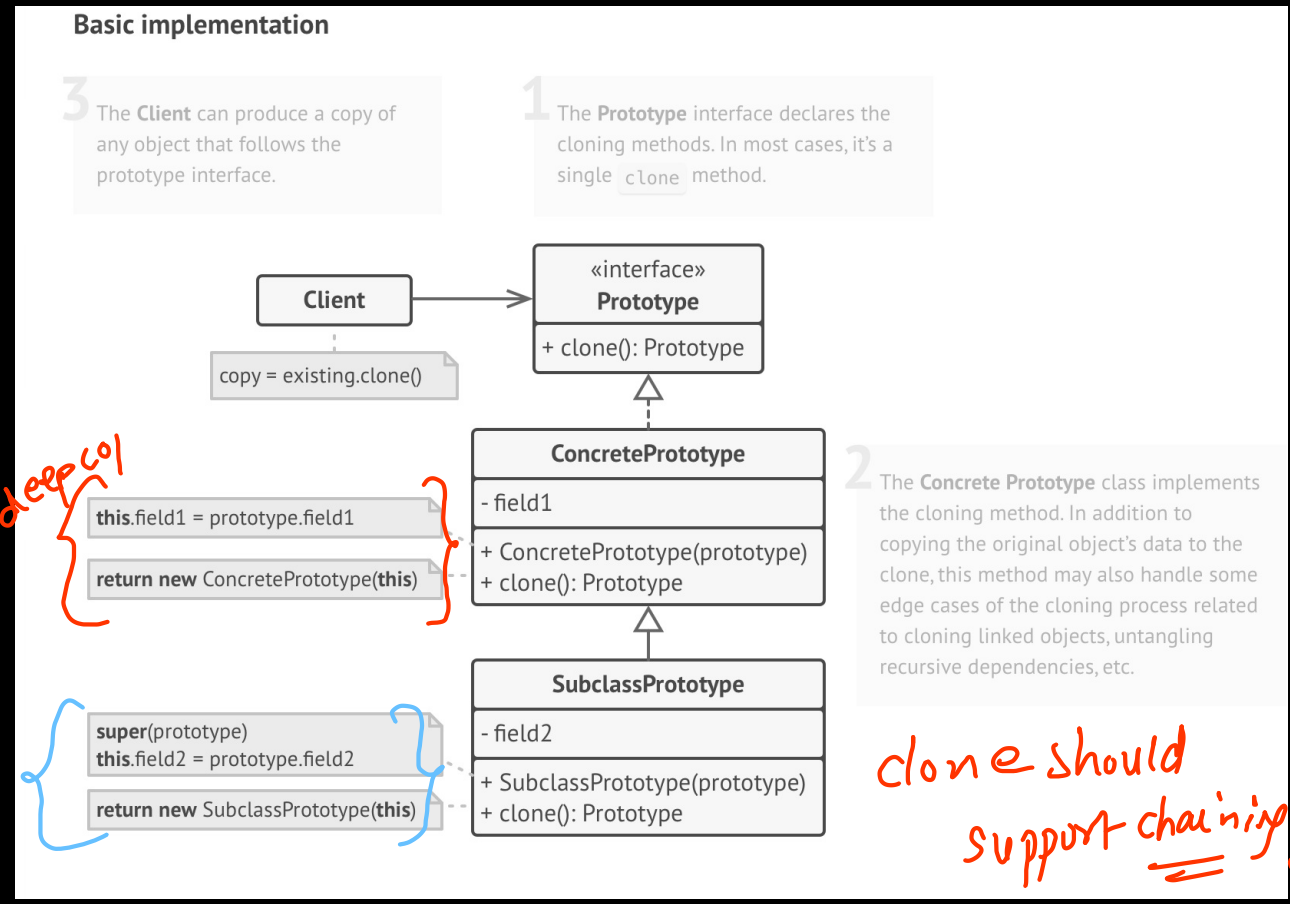
NPC
height
weight
gender
clone



clone() {Java Object}
copy constructor
↳ Deep copy


interface
clone()
Cloneable/Prototype
implements
Constructor
@override
clone(OtherNPC)
1st prototype
deep copy
NPC

Structure


Basic implementation
3 The Client can produce a copy of any object that follows the prototype interface.
1 The Prototype interface declares the cloning methods. In most cases, it's a single clone method.
Client
copy = existing.clone()
«interface»
Prototype
+ clone(): Prototype
ConcretePrototype
- field1
+ ConcretePrototype(prototype)
+ clone(): Prototype
this.field1 = prototype.field1
return new ConcretePrototype(this)
deep col
SubclassPrototype
- field2
+ SubclassPrototype(prototype)
+ clone(): Prototype
super(prototype)
this.field2 = prototype.field2
return new SubclassPrototype(this)
2 The Concrete Prototype class implements the cloning method. In addition to copying the original object's data to the clone, this method may also handle some edge cases of the cloning process related to cloning linked objects, untangling recursive dependencies, etc.
clone should support chaining!

```java
public class NPC implements Cloneable {
    String height, weight, gender;

    // for 1st prototype
    NPC(String height, String weight, String
    gender) {
        this.height = height;
        this.weight = weight;
        this.gender = gender;
    }

    @Override
    protected Object clone() throws
    CloneNotSupportedException {
        return super.clone();
    }

    public String toString() {
        return (height + ", " + weight + ", " +
        gender);
    }
}
```

this is faster implementation

```java
public class Client {
    Run | Debug
    public static void main(String[] args) throws
    Exception {
        NPC firstNPC = new NPC(height: "100.0",
        weight: "75.0", gender: "Male");

        System.out.println(firstNPC);

        NPC secondNPC = (NPC) (firstNPC.clone());
        System.out.println(secondNPC);

        NPC thirdNpc = (NPC) (firstNPC.clone());
        System.out.println(thirdNpc);

        // Address Comparison: False
        System.out.println(firstNPC == secondNPC);
        System.out.println(thirdNpc == secondNPC);
    }
}
```

Equal Data

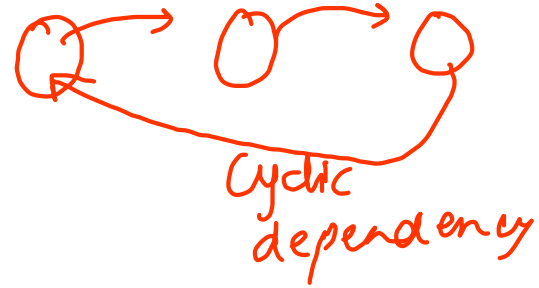
false

⚖ Pros and Cons

- ✓ You can clone objects without coupling to their concrete classes. (w/o new)
- ✓ You can get rid of repeated initialization code in favor of cloning pre-built prototypes.
- ✓ You can produce complex objects more conveniently.
- ✓ You get an alternative to inheritance when dealing with configuration presets for complex objects. Reflect^n API

} dependency file ".yaml, .xml

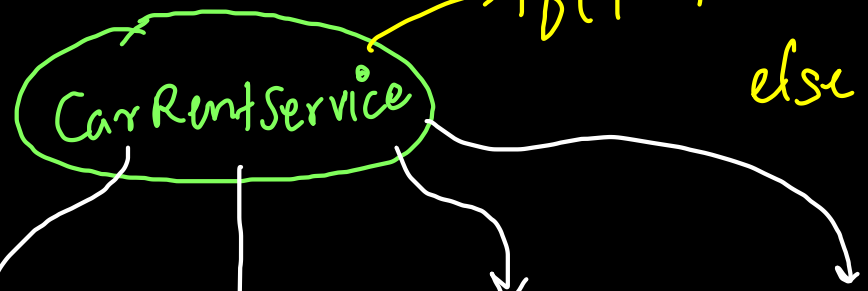
- ✗ Cloning complex objects that have circular references might be very tricky.



(15) Factory Method Design Pattern

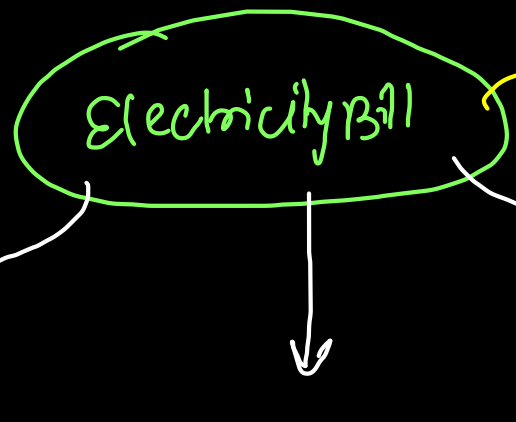
↳ object creation

Problem 1




```
if(people ≤ 1) return new Bike(-,-,->
else if (people ≤ 3) return
        new Auto(---;)
else if(people ≤ 4)
        return new
            Sedan(---)
else new SUV(---);
```

example 2)



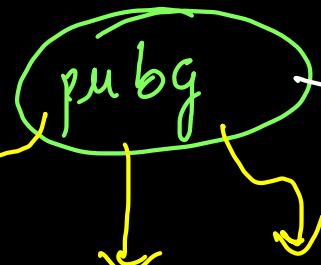
Household

Industrial

Corporate

```
if (area == "residential")
    return new Household();
else if (area == "industrial")
    return new Industry();
else
    return new Corporate();
```

example 3)
Games

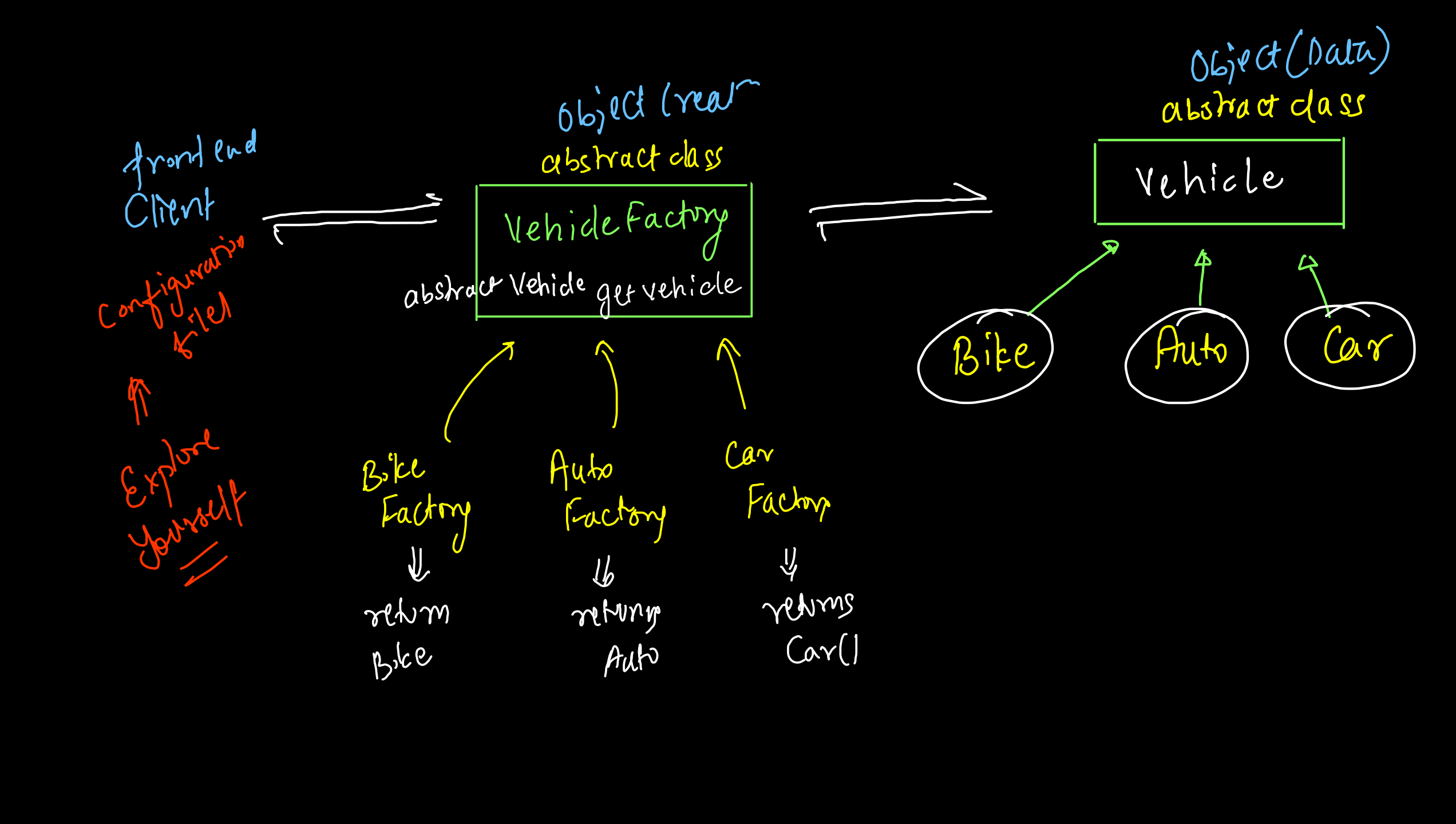


if (100 enemies)

Erangel map

Miramar Map

Deathmatch map

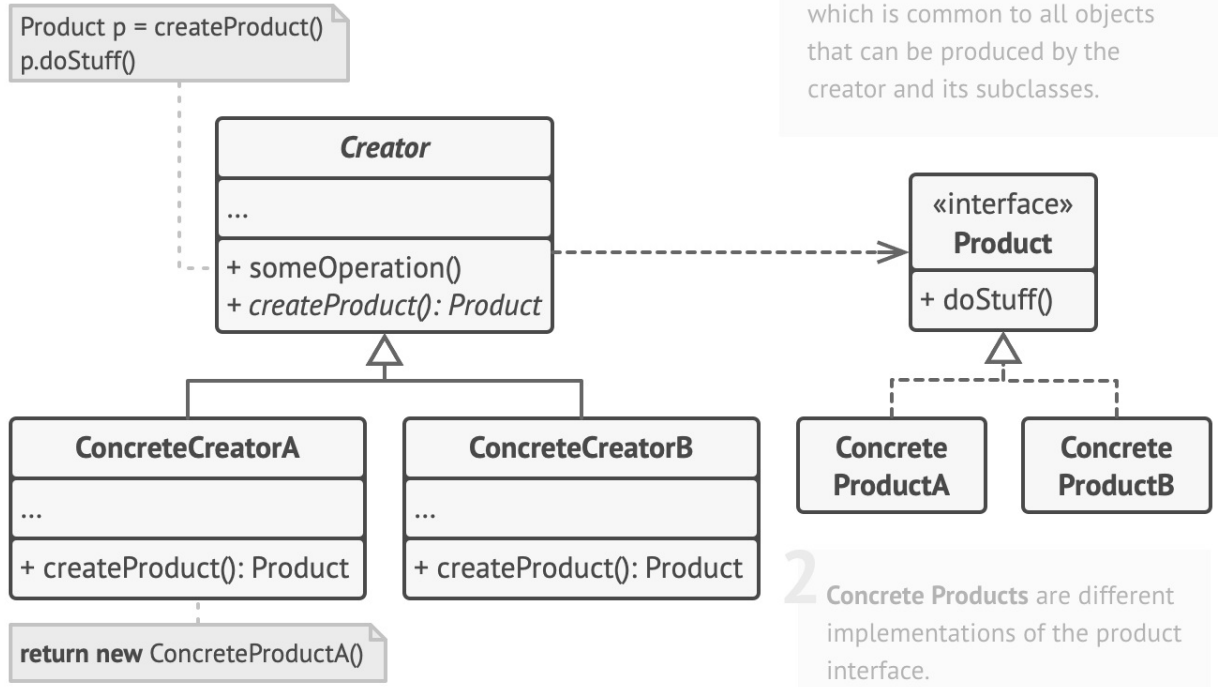

Object (Data)
abstract class
Vehicle
Object (real)
abstract class
VehicleFactory
abstract Vehicle getVehicle
Front end
Client
configuration file
Explore Yourself
Bike Factory
return Bike
Auto Factory
returns Auto
Car Factory
returns Car()
Bike
Auto
Car

Structure

3 The **Creator** class declares the factory method that returns new product objects. It's important that the return type of this method matches the product interface.

You can declare the factory method as `abstract` to force all subclasses to implement their own versions of the method. As an alternative, the base factory method can return some default product type.

Note, despite its name, product creation is **not** the primary responsibility of the creator. Usually, the creator class already has some core business logic related to products. The factory method helps to decouple this logic from the concrete product classes. Here is an analogy: a large software development company can have a training department for programmers. However, the primary function of the company as a whole is still writing code, not producing programmers.



1 The **Product** declares the interface, which is common to all objects that can be produced by the creator and its subclasses.

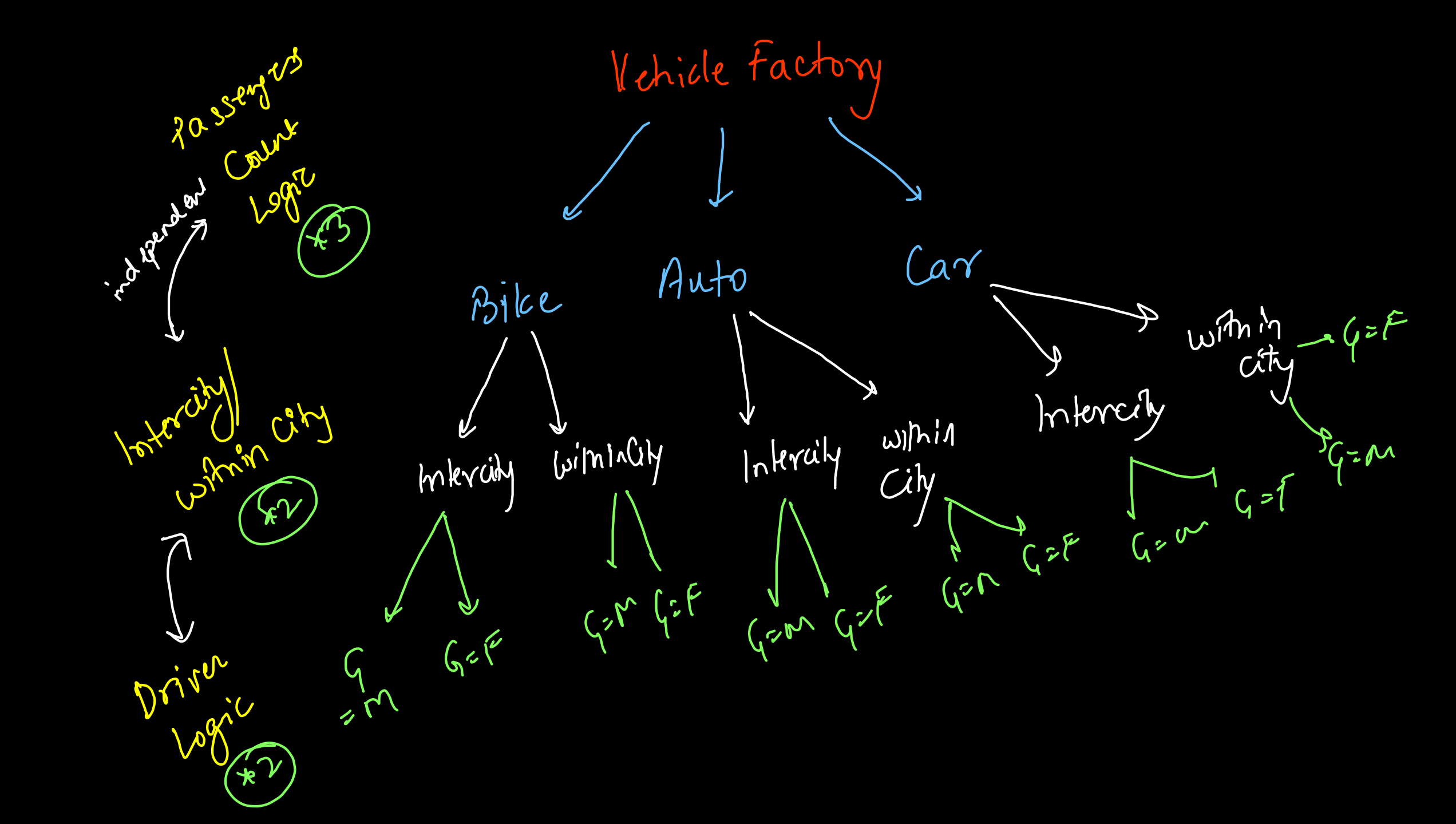
2 **Concrete Products** are different implementations of the product interface.

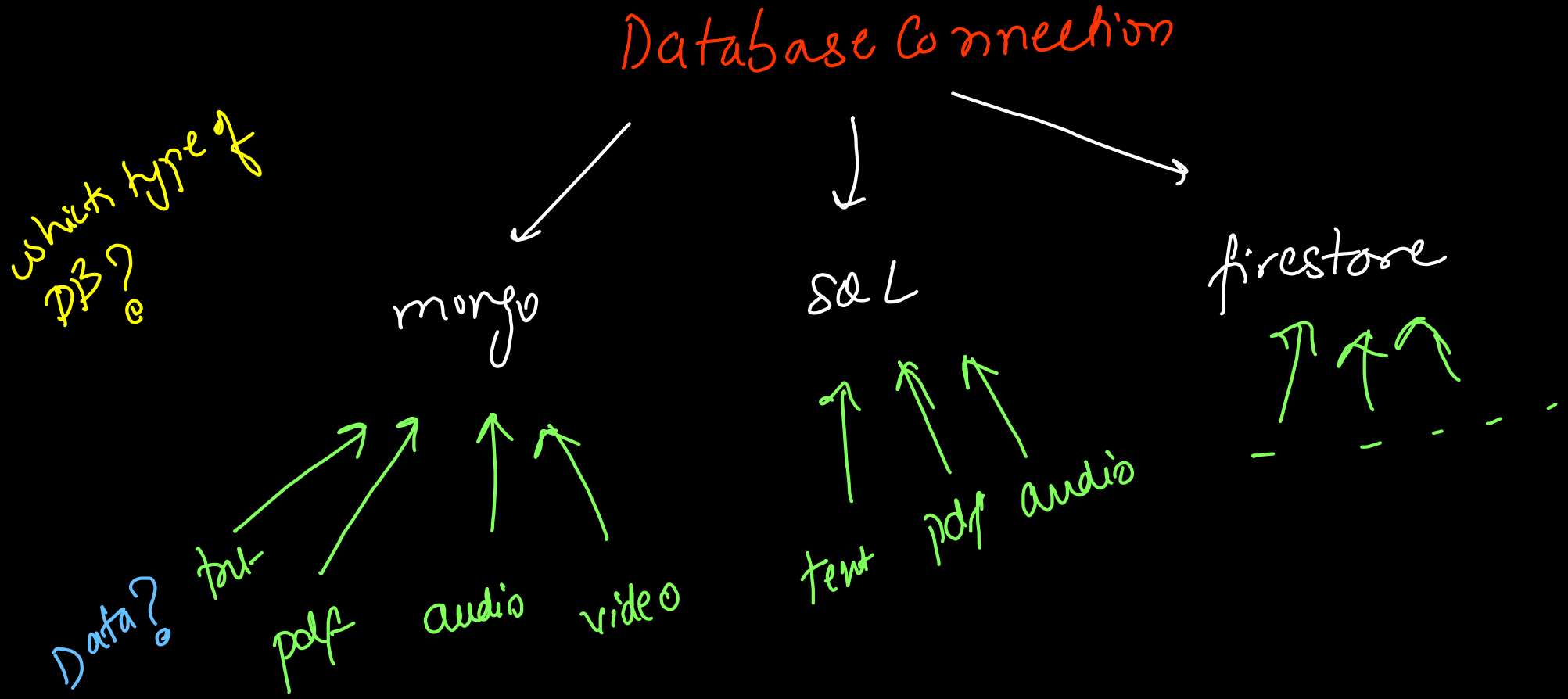
4 **Concrete Creators** override the base factory method so it returns a different type of product.

Note that the factory method doesn't have to **create** new instances all the time. It can also return existing objects from a cache, an object pool, or another source.

⚖ Pros and Cons

- ✓ You avoid tight coupling between the creator and the concrete products.
- ✓ *Single Responsibility Principle*. You can move the product creation code into one place in the program, making the code easier to support.
- ✓ *Open/Closed Principle*. You can introduce new types of products into the program without breaking existing client code.
- ✗ The code may become more complicated since you need to introduce a lot of new subclasses to implement the pattern. The best case scenario is when you're introducing the pattern into an existing hierarchy of creator classes.


Vehicle Factory
Bike
Auto
Car
Passengers Count Logic
*3
Intercity/ within city
*2
Driver Logic
*2
Intercity
WithinCity
Intercity
within city
Intercity
within city
G=M
G=F
G=M
G=F
G=M
G=F
G=M
G=F
G=M
G=F
G=F




```
public abstract class DatabaseFactory {
    public abstract DatabaseConnection getDatabase();

    public abstract DataFormat getDataFormat();
}
```

```
class PDFSQL extends DatabaseFactory {

    @Override
    public DataFormat getDataFormat() {
        return new PDF();
    }

    @Override
    public DatabaseConnection getDatabase() {
        return new SQL();
    }
}

class PDFMongo extends DatabaseFactory {

    @Override
    public DatabaseConnection getDatabase() {
        return new Mongo();
    }

    @Override
    public DataFormat getDataFormat() {
        // TODO Auto-generated method stub
        return new PDF();
    }
}
```

```
public class DataFormat {

}

class PDF extends DataFormat {

}

class Video extends DataFormat {

}

class Audio extends DataFormat {

}
```

```
public class DatabaseConnection {

}

class SQL extends DatabaseConnection {

}

class Mongo extends DatabaseConnection {

}

class Firestore extends DatabaseConnection {

}
```

Inheritance is my problem here!

total 3x3=9 combinations

Factory design Pattern

Factory method

- Cons
  - multiple clause
- uses inheritance
- ⇒ single class factory

Abstract Factory

- uses composition
- ⇒ multiple class factory


```
public abstract class DatabaseFactory {
    DatabaseConnectionFactory dbConnection;
    DataFormatFactory dataformat;

    public DatabaseFactory(DatabaseConnectionFactory
    dbConnection, DataFormatFactory dataformat) {
        this.dbConnection = dbConnection;
        this.dataformat = dataformat;
    }

    public void setupDatabase(){
        // dbConnection.getDB();
        // dataformat.getDataFormat();
    }
}
```

```
public abstract class DataFormatFactory {

}

class PDFFactory extends DataFormatFactory {

}

class AudioFactory extends DataFormatFactory {

}

class VideoFactory extends DataFormatFactory {

}
```

```
public class DataFormat {

}

class PDF extends DataFormat {

}

class Video extends DataFormat {

}

class Audio extends DataFormat {

}
```

1 + 3 = 3

```
public class DatabaseConnectionFactory {

}

class SQLFactory extends DatabaseConnectionFactory {

}

class MongoFactory extends DatabaseConnectionFactory {

}

class FirestoreFactory extends DatabaseConnectionFactory {

}
```

return SQL

return mongo

return firestore

```
public class DatabaseConnection {

}

class SQL extends DatabaseConnection {

}

class Mongo extends DatabaseConnection {

}

class Firestore extends DatabaseConnection {

}
```

⚖ Pros and Cons

- ✓ You can be sure that the products you're getting from a factory are compatible with each other.
- ✓ You avoid tight coupling between concrete products and client code.
- ✓ *Single Responsibility Principle*. You can extract the product creation code into one place, making the code easier to support.
- ✓ *Open/Closed Principle*. You can introduce new variants of products without breaking existing client code.
- ✗ The code may become more complicated than it should be, since a lot of new interfaces and classes are introduced along with the pattern.