Adapter
Design pattern

Intent Adapter is a structural design pattern that allows object with incompatible interface to collaborate.

Applicability

1. Use Adapter class when you want to use some existing class, but its interface isn't compatible with rest of your code

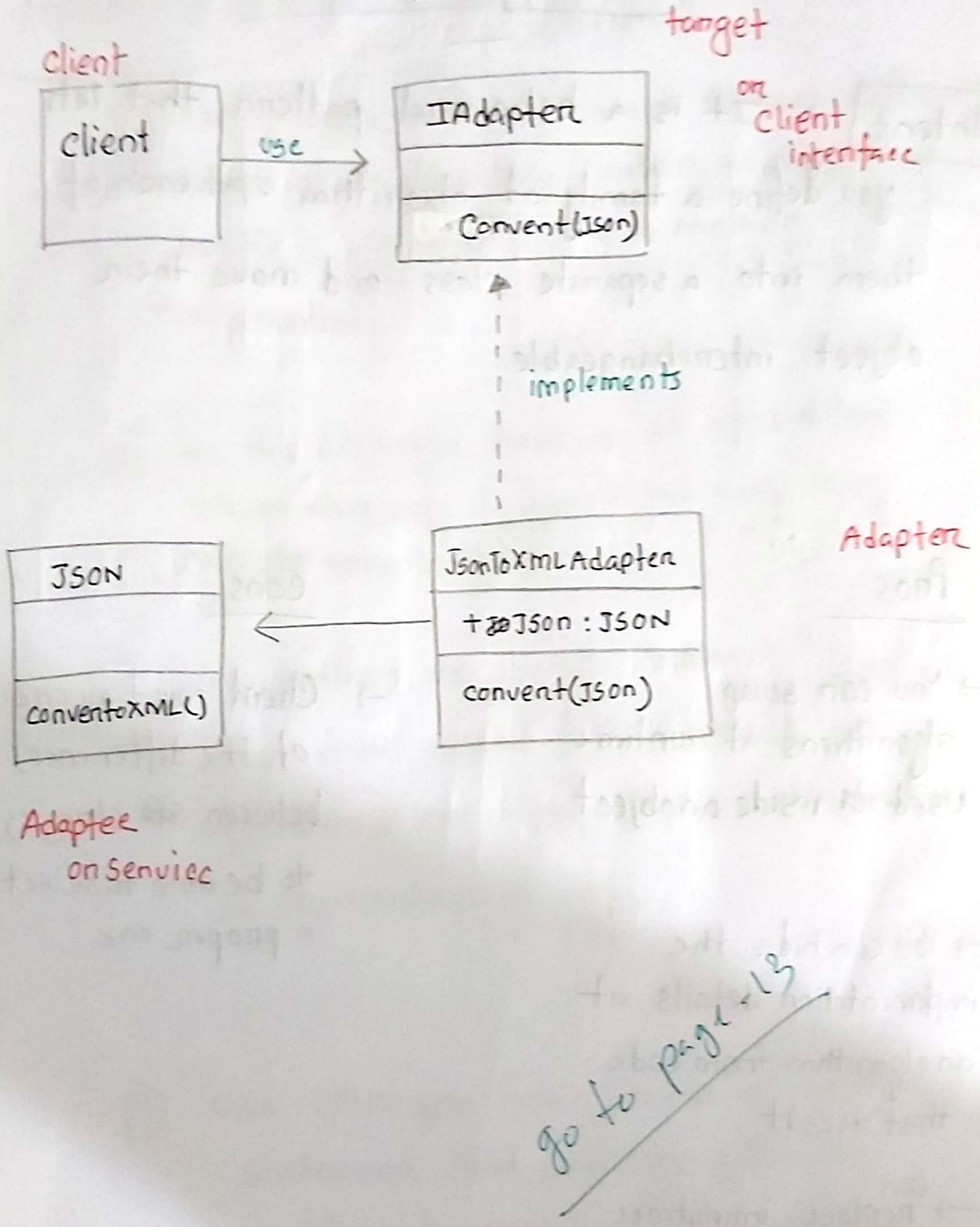
2. Use this pattern when you want to reuse several existing subclasses that lack some common functionality that can't be added to the super class

Motivation → motivation behind is to enable the collaboration between incompatiable interface or classes.

It is used when you have existing code or components that cannot be directly used by other parts of the system due to interface mismatch or incompatible behaviors.

1. Interface Conversion
2. ~~Res~~ Reusability
3. Separation of concern
4. Legacy system integration

UML Diagram



go to page-13

Explaining the following code:

1. The `Client` class creates an instance of the `JSON` class, which represents some JSON data.
2. It also creates an instance of the `JsonToXmlAdapter` class, which implements the `IAdapter` interface.
3. The `convert` method in the `JsonToXmlAdapter` class takes a `JSON` object as a parameter and converts it to an `XML` object.
4. The client code calls the `convert` method on the `iAdapter` object, passing the `json` object as the argument: `XML xml = iAdapter.convert(json);`
5. Inside the `JsonToXmlAdapter` class, the `convert` method receives the `JSON` object and calls the `convertToXML` method on it.
6. The `convertToXML` method in the `JSON` class performs the logic to convert the JSON data to XML format and returns an `XML` object representing the converted data.
7. The `convert` method in the `JsonToXmlAdapter` class receives the `XML` object from the `convertToXML` method and returns it.
8. The `XML` object returned from the `convert` method of the adapter is assigned to the `xml` variable in the client code: `XML xml = iAdapter.convert(json);`

(1)
```
public class XML {
    public XML(){}
    public XML(String data){}
}
```

(2)
```
public class JSON {

    public JSON(){};
    public JSON(String data){}
    XML convertToXML(){
        // logic to convert the data to XML
        return new XML("Stringified JSON data");
    }
}
```

(3)
```
public interface IAdapter {
    XML convert(JSON json);
}
```

(4)
```
public class JsonToXmlAdapter implements IAdapter {
    private JSON json;

    public JsonToXmlAdapter(JSON json){
        this.json = json;
    }

    @Override
    public XML convert(JSON json) {
        return this.json.convertToXML();
    }
}
```

(5)
```
public class Client {
    JSON json = new JSON("json data");
    IAdapter iAdapter = new JsonToXmlAdapter(json);
    XML xml = iAdapter.convert(json);
}
```

(x)
```
public interface IAdapter<TypeA , TypeB> {
    TypeB convert(TypeA source);
}

public class PROTOBUFFER {
    public PROTOBUFFER(){};
    public PROTOBUFFER(String data){}
    XML convertToXML(){
        // logic to convert the data to XML
        return new XML("Stringified PROTOBUFFER data");
    }
}
```

(x)
```
public class ProtobufferToXmlAdapter implements IAdapter<PROTOBUFFER, XML>{
    private PROTOBUFFER Protobuffer;

    public ProtobufferToXmlAdapter(PROTOBUFFER Protobuffer){
        this.Protobuffer = Protobuffer;
    }

    @Override
    public XML convert(PROTOBUFFER PROTOBUFFER) {
        return this.Protobuffer.convertToXML();
    }
}
```

(x) suppose if want to introduce new type protbuffer

Participants of the following code:

1. Client: The `Client` class
2. Client Interface: The `IAdapter` interface
3. Adapter: The `JsonToXmlAdapter` class
4. Service: The `JSON` and `XML` classes

Explaining the following code:

1. The `Client` class creates an instance of the `JSON` class, which represents some JSON data.
2. It also creates an instance of the `JsonToXmlAdapter` class, which implements the `IAdapter` interface.
3. The `convert` method in the `JsonToXmlAdapter` class takes a `JSON` object as a parameter and converts it to an `XML` object.
4. The client code calls the `convert` method on the `iAdapter` object, passing the `json` object as the argument: `XML xml = iAdapter.convert(json);`
5. Inside the `JsonToXmlAdapter` class, the `convert` method receives the `JSON` object and calls the `convertToXML` method on it.
6. The `convertToXML` method in the `JSON` class performs the logic to convert the JSON data to XML format and returns an `XML` object representing the converted data.
7. The `convert` method in the `JsonToXmlAdapter` class receives the `XML` object from the `convertToXML` method and returns it.
8. The `XML` object returned from the `convert` method of the adapter is assigned to the `xml` variable in the client code: `XML xml = iAdapter.convert(json);`

```
public class XML {
    public XML(){}
    public XML(String data){}
}
```

```
public class JSON {

    public JSON(){};
    public JSON(String data){}
    XML convertToXML(){
        // logic to convert the data to XML
        return new XML("Stringified JSON data");
    }
}
```

```
public interface IAdapter {
    XML convert(JSON json);
}
```

```
public class JsonToXmlAdapter implements IAdapter {
    private JSON json;

    public JsonToXmlAdapter(JSON json){
        this.json = json;
    }

    @Override
    public XML convert(JSON json) {
        return this.json.convertToXML();
    }
}
```

```
public class Client {
    JSON json = new JSON("json data");
    IAdapter iAdapter = new JsonToXmlAdapter(json);
    XML xml = iAdapter.convert(json);
}
```

```
public interface IAdapter<TypeA , TypeB> {
    TypeB convert(TypeA source);
}
```

```
public class PROTOBUFFER {
    public PROTOBUFFER(){};
    public PROTOBUFFER(String data){}
    XML convertToXML(){
        // logic to convert the data to XML
        return new XML("Stringified PROTOBUFFER data");
    }
}
```

```
public class ProtobufferToXmlAdapter implements IAdapter<PROTOBUFFER, XML>{
    private PROTOBUFFER Protobuffer;

    public ProtobufferToXmlAdapter(PROTOBUFFER Protobuffer){
        this.Protobuffer =  Protobuffer;
    }

    @Override
    public XML convert(PROTOBUFFER PROTOBUFFER) {
        return this. Protobuffer.convertToXML();
    }
}
```

Participants of the following code:

1. Client: The `Client` class
2. Client Interface: The `IAdapter` interface
3. Adapter: The `JsonToXmlAdapter` class
4. Service: The `JSON` and `XML` classes

Strategy Pattern

Intent It is a behavioral pattern that lets you define a family of algorithm, put each of them into a separate class and make their object interchangeable.

Pros

→ You can swap algorithms at runtime used inside an object

→ You can hide the implementation details of an algorithm from code that uses it

→ can replace inheritance with composition

→ OCP principle

cons

→ Client must aware of the differences between strategies to be able to select a proper one

Applicability

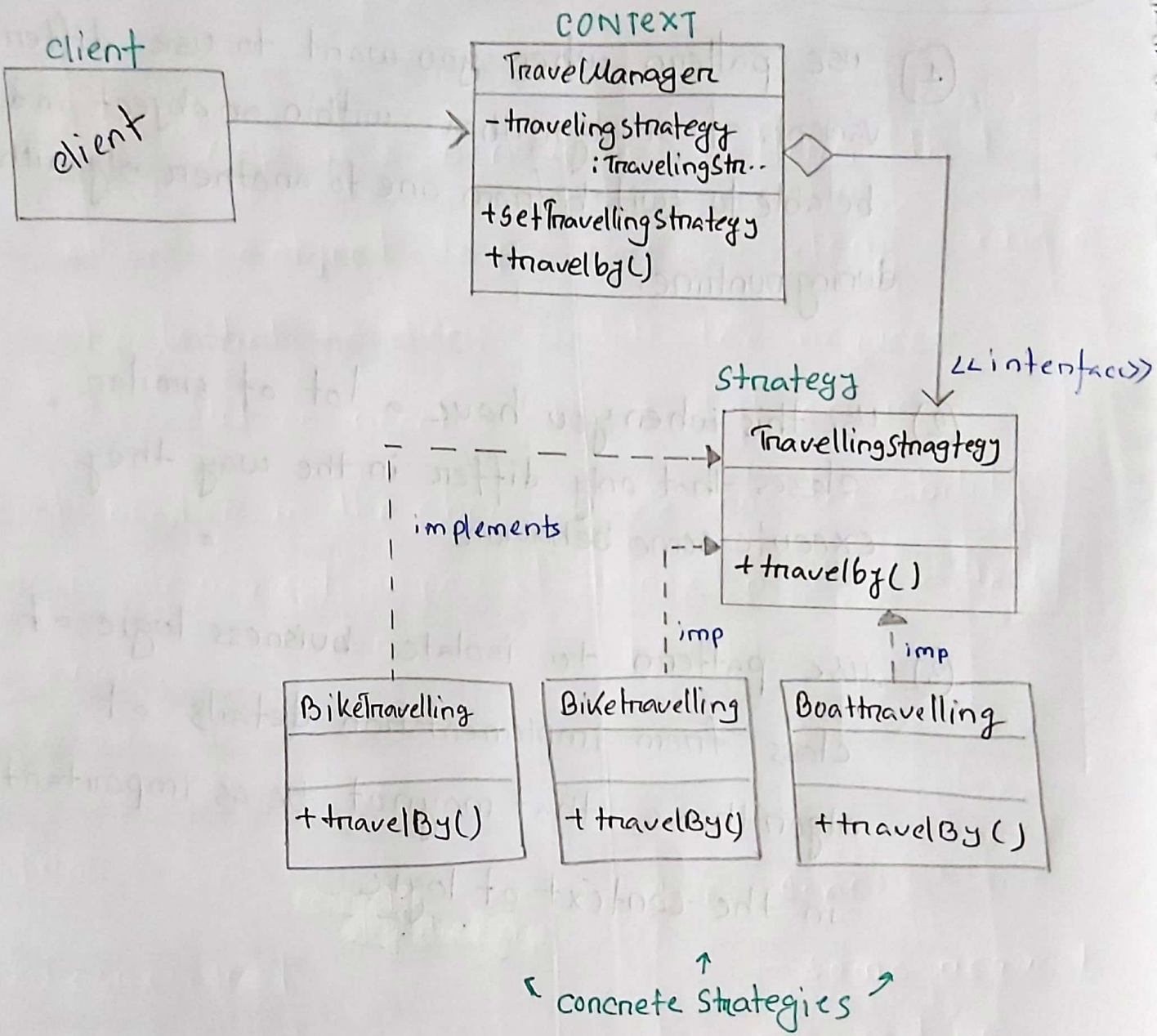
(1) use pattern when you want to use different variants of an algorithm within an object and be able to switch from one to another algorithm during runtime.

(2) Use this when you have a lot of similar classes that only differ in the way they execute some behavior

(3) use pattern to isolate buisness logic of class from implementation details of algorithms that maynot be as important in the context of logic

(4) use when you have massive conditional statement that switches between different variant of same algorithm

UML Diagram


client
client
CONTEXT
TravelManager
-travelingstrategy
:TravelingStr...
+setTravellingstrategy
+travelby()
<<interface>>
Strategy
TravellingStragtegy
+travelby()
implements
imp
imp
BikeTravelling
+travelBy()
BikeTravelling
+travelBy()
BoatTravelling
+travelBy()
concrete Strategies

what are the participant of strategy pattern?

1. client
2. Context :- Travel manager
3. strategy :- Travelling strategy
4. concrete strategies :-
   - TravellingBus
   - TravellingBoat
   - TravellingBike

Motivation → strategy pattern is to enable interchangeable algorithms or strategies with an object

→ encapsulates different algorithm/behaviour making them interchangeable at runtime

1. Flexibility and extensibility
2. Encapsulation
3. Dynamic selection of strategies
4. code reuse

(1)

```
public interface TravellingStrategy {
    void travelBy(String location);
}
```

(2)

```
public class BikeTravelling implements TravellingStrategy{
    public void travelBy(String location) {
        System.out.println("Travelling to "+location+" by Bike");
        // System holds bike travelling operations
    }
}
```

(3)

```
public class BusTravelling implements TravellingStrategy{
    @Override
    public void travelBy(String location) {
        System.out.println("Travelling to "+location+" by Bus");
        // System holds bus travelling operations
    }
}
```

(4)

```
public class BoatTravelling implements TravellingStrategy{
    public void travelBy(String location) {
        System.out.println("Travelling to "+location+" by Boat");
        // System holds boat travelling operations
    }
}
```

(5)

```
public class TravelManager {
    private TravellingStrategy travellingStrategy;

    public void setTravellingStrategy(TravellingStrategy travellingStrategy){
        this.travellingStrategy = travellingStrategy;
    }

    public void travelBy(String Location) {
        travellingStrategy.travelBy(Location);
    }
}
```

(6)

```
public class Client {
    public static void main(String[] args) {
        // Suppose I am planning a trip
        // Sylhet - Shreemangal - MadhabpurLake - Sylhet
        // Want to travel to sylhet with bus
        // when I reach sylhet
        // I want to switch to bike to go to shreemangal
        // To travel MadhabpurLake, I switch to boat
        // Then I again switch to bike to come back to sylhet

        // Strategy pattern provide the facility
        // to interchange between object (even in runtime)

        TravelManager travelManager = new TravelManager();
        travelManager.setTravellingStrategy(new BusTravelling());
        travelManager.travelBy("Sylhet");

        travelManager.setTravellingStrategy(new BikeTravelling());
        travelManager.travelBy("Shreemangal");

        travelManager.setTravellingStrategy(new BoatTravelling());
        travelManager.travelBy("MadhabpurLake");

        travelManager.setTravellingStrategy(new BikeTravelling());
        travelManager.travelBy("Sylhet");

    }
}
```

Advantage :

1. Switch between algorithms in runtime
2. Interchangeable objects strategy

```java
public interface TravellingStrategy {
    void travelBy(String location);
}
```

```java
public class BusTravelling implements TravellingStrategy{
    @Override
    public void travelBy(String location) {
        System.out.println("Travelling to "+location+" by Bus");
        // System holds bus travelling operations
    }
}
```

```java
public class BoatTravelling implements TravellingStrategy{
    public void travelBy(String location) {
        System.out.println("Travelling to "+location+" by Boat");
        // System holds boat travelling operations
    }
}
```

```java
public class BikeTravelling implements TravellingStrategy{
    public void travelBy(String location) {
        System.out.println("Travelling to "+location+" by Bike");
        // System holds bike travelling operations
    }
}
```

```java
public class TravelManager {
    private TravellingStrategy travellingStrategy;

    public void setTravellingStrategy(TravellingStrategy travellingStrategy){
        this.travellingStrategy = travellingStrategy;
    }

    public void travelBy(String Location) {
        travellingStrategy.travelBy(Location);
    }
}
```

```java
public class Client {
    public static void main(String[] args) {
        // Suppose I am planning a trip
        // Sylhet - Shreemangal - MadhabpurLake - Sylhet
        // Want to travel to sylhet with bus
        // when I reach sylhet
        // I want to switch to bike to go to shreemangal
        // To travel MadhabpurLake, I switch to boat
        // Then I again switch to bike to come back to sylhet

        // Strategy pattern provide the facility
        // to interchange between object (even in runtime)

        TravelManager travelManager = new TravelManager();
        travelManager.setTravellingStrategy(new BusTravelling());
        travelManager.travelBy("Sylhet");

        travelManager.setTravellingStrategy(new BikeTravelling());
        travelManager.travelBy("Shreemangal");

        travelManager.setTravellingStrategy(new BoatTravelling());
        travelManager.travelBy("MadhabpurLake");

        travelManager.setTravellingStrategy(new BikeTravelling());
        travelManager.travelBy("Sylhet");

    }
}
```

Advantage :

1. Switch between algorithms in runtime
2. Interchangeable objects strategy

Chain of Responsibility pattern

COR

Intent :- COR is a behavonial pattern that allows an object to pass request along a chain of potential handlers until the request is handled on reaches the end of chain.

Each handler in the chain ~~is able~~ has the ability to handle request on pass it to next handler in the chain.

Pros

- You can control the order of request handling
- can follow SRP & OCP principle

cons

- some request may end up unhandled.

Applicability

→ ① use when youn program is expect to process different kindof request in various ways. but the exact type of request and thein sequence maynot known before

② use when it's essential to execute several handlens in panticulan onden

③ use when setof ~~onden~~ handlen and thein onden supased to change at nun time

Motivation

Pattenn allows flexible and dynamic handling of request without tightly coupling the senden of request with is its neciven

① Decoupling senden and neciven

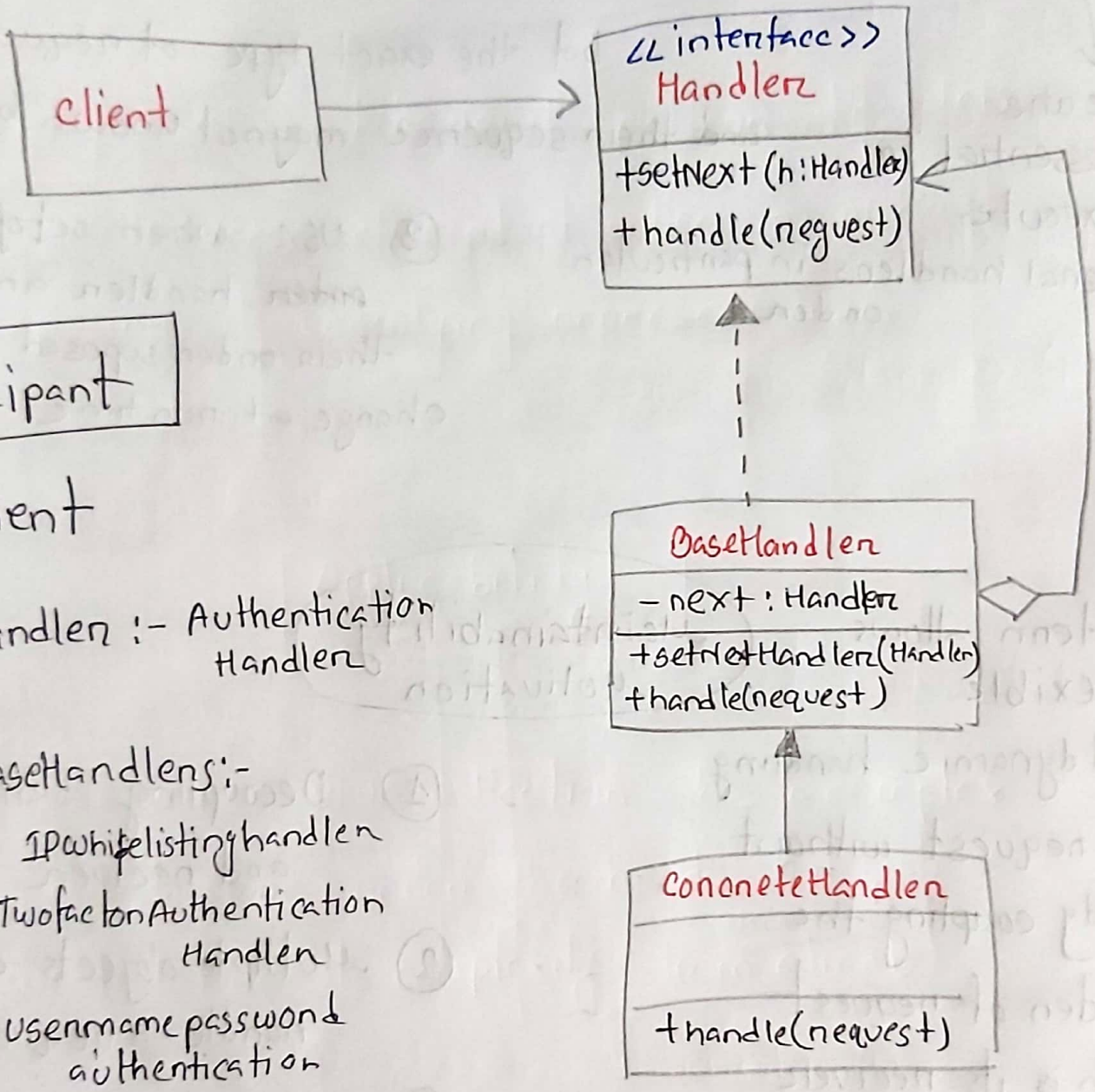
② Multiple objects can handle a nequest

③ Avoiding hand-coded dependency

④ Adding on Modifying handlens dynamically

⑤ sepanting Responsibilities

UML Diagram

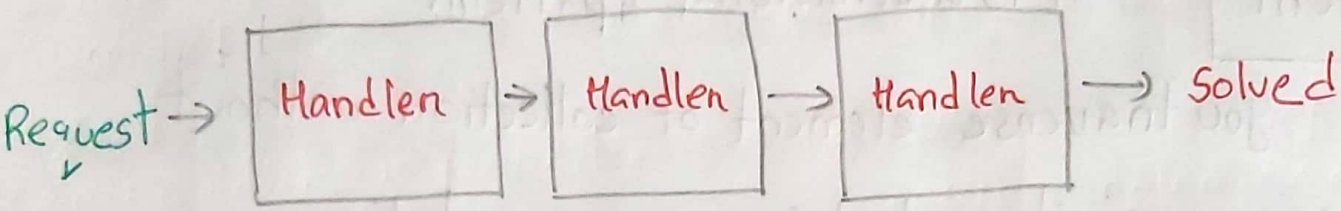


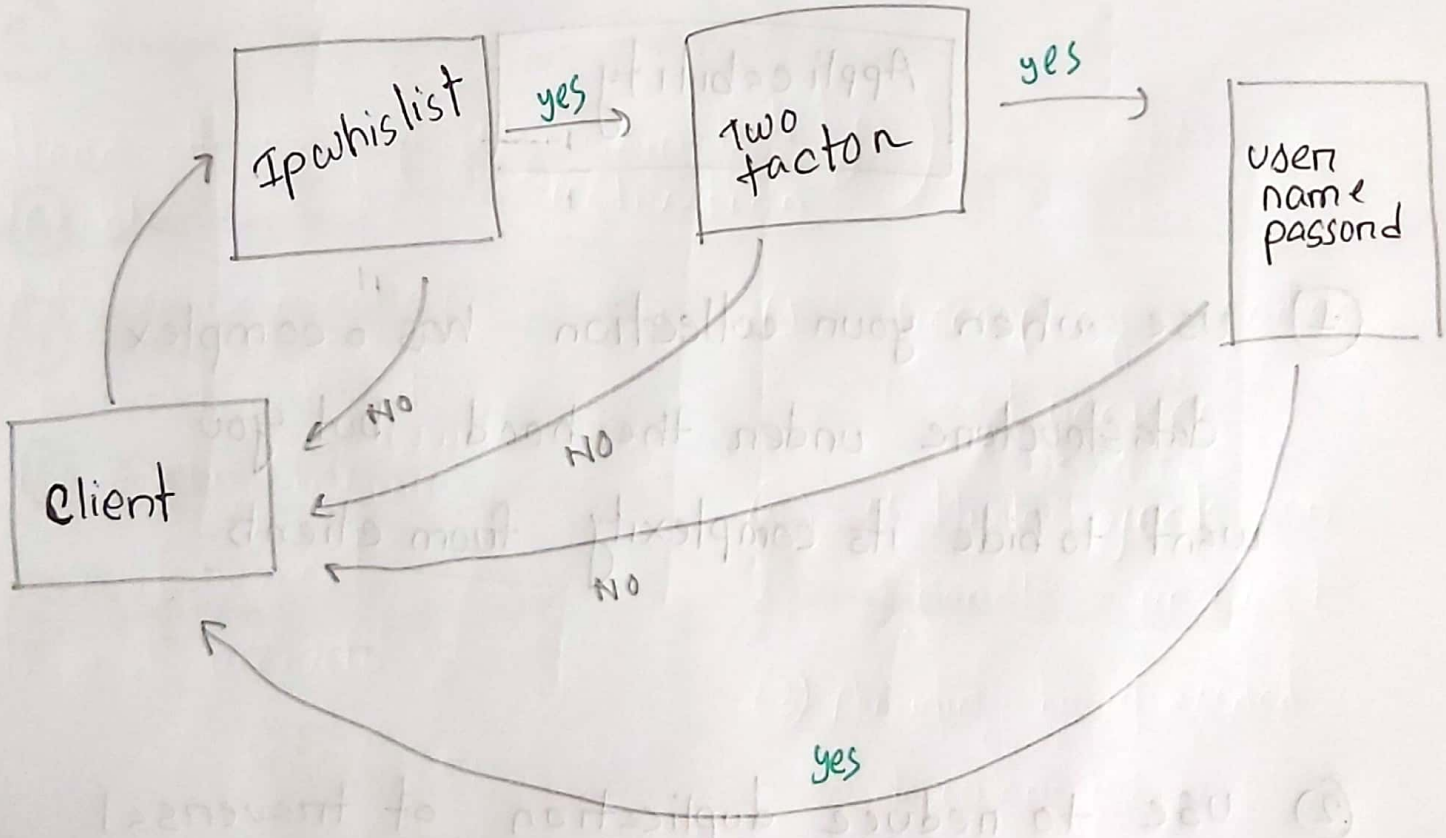
Participant

1. client
2. Handler :- Authentication Handler
3. BaseHandlers:-
   - IpWhitelistingHandler
   - TwofactorAuthentication Handler
   - usernamepassword authentication handler
4. Concrete Handler

Suppose →

Chain of Responsibility





How chainof Responsibility works

Suppose you have a Three step authentication system.

1. First Ipwhitlist authentication: If successful, then move to next authentication(Two Factor Authentication), Otherwise authenticationA fails
2. (If successful) Two factor authentication, if successful, then move to next authentication (Usernname Password Authentication), Otherwise authentication fails
3. (If successful) Username Password authentication ,If successful, then move whole authentication Process successful .Otherwise authentication fails

1

```java
public interface AuthenticationHandler {
    void setNextHandler(AuthenticationHandler authenticationHandler);
    boolean authenticate(String userName, String password);
}
```

2

```java
public class IPWhitelistingHandler implements AuthenticationHandler{
    private AuthenticationHandler authenticationHandler;

    @Override
    public void setNextHandler(AuthenticationHandler authenticationHandler) {
        this.authenticationHandler = authenticationHandler;
    }

    @Override
    public boolean authenticate(String userName, String password) {
        // Simulating IP whitelisting
        String clientIP = getClientIP();
        if (!clientIP.contains("192.168.192.")) {
            System.out.println("IPWhitelistingHandler: Authentication failed.");
            return false;
        } else if (authenticationHandler != null) {
            System.out.println("IPWhitelistingHandler: Authentication successful.");
            return authenticationHandler.authenticate(userName, password);
        } else {
            System.out.println("IPWhitelistingHandler: Authentication failed.");
            return false;
        }
    }

    private String getClientIP() {
        // Simulated method to get client IP address
        return "192.168.192.";
    }
}
```

3

```java
public class TwoFactorAuthenticationHandler implements AuthenticationHandler{
    private AuthenticationHandler authenticationHandler;
    @Override
    public void setNextHandler(AuthenticationHandler authenticationHandler) {
        this.authenticationHandler = authenticationHandler;
    }

    @Override
    public boolean authenticate(String userName, String password) {
        // Simulating two-factor authentication
        if (userName.equals("user") && password.equals("user123") && verifyOTP("123456")) {
            System.out.println("TwoFactorAuthenticationHandler: Authentication successful.");
            return true;
        } else if (authenticationHandler != null) {
            return authenticationHandler.authenticate(userName, password);
        } else {
            System.out.println("TwoFactorAuthenticationHandler: Authentication failed.");
            return false;
        }
    }

    private boolean verifyOTP(String otp) {
        // Simulated OTP verification logic
        return otp.equals("123456");
    }
```

```
public class UsernamePasswordAuthenticationHandler implements AuthenticationHandler{
    private AuthenticationHandler authenticationHandler;

    @Override
    public void setNextHandler(AuthenticationHandler authenticationHandler) {
        this.authenticationHandler = authenticationHandler;
    }

    @Override
    public boolean authenticate(String userName, String password) {
        if (userName.equals("admin") && password.equals("admin123")) {
            System.out.println("UsernamePasswordAuthenticationHandler: Authentication successful.");
            return true;
        } else if (authenticationHandler != null) {
            return authenticationHandler.authenticate(userName, password);
        } else {
            System.out.println("UsernamePasswordAuthenticationHandler: Authentication failed.");
            return false;
        }
    }
}
```

```
public class Client {
    public static void main(String[] args) {
        AuthenticationHandler upHandler = new UsernamePasswordAuthenticationHandler();
        AuthenticationHandler tfaHandler = new TwoFactorAuthenticationHandler();
        AuthenticationHandler ipHandler = new IPWhitelistingHandler();

        ipHandler.setNextHandler(upHandler);
        upHandler.setNextHandler(tfaHandler);

        boolean isAuthenticated = ipHandler.authenticate("user", "user123");
        if (isAuthenticated) {
            // Proceed with server access
            System.out.println("Access granted.");
        } else {
            // Handle authentication failure
            System.out.println("Access denied.");
        }
    }
}
```

1. We start by defining the `AuthenticationHandler` interface, which represents the base handler in the chain. It has two methods: `setNextHandler()` to set the next handler in the chain and `authenticate()` to perform authentication.
2. Next, we have three concrete implementations of the `AuthenticationHandler` interface: `IPWhitelistingHandler`, `TwoFactorAuthenticationHandler`, and `UsernamePasswordAuthenticationHandler`. Each handler implements the `setNextHandler()` and `authenticate()` methods according to its specif authentication logic.
3. In the `Client` class, we create instances of the authentication handler `upHandler` for username/password authentication, `tfaHandler` for two factor authentication, and `ipHandler` for IP whitelisting.
4. We then set up the chain of responsibility by calling `setNextHandler()` on each handler, in the desired order. In this case, the request will flow from `ipHandler` to `upHandler`, and then to `tfaHandler`.
5. Finally, we call the `authenticate()` method on the `ipHandler` and pass the username and password for authentication. The request will propagate through the chain of handlers until it is handled or reaches the end of the chain.
6. Each handler performs its specific authentication logic and decides whether to handle the request or pass it to the next handler in the chain. If a handler can handle the request, it returns `true`. Otherwise, it delegates the request to the next handler.
7. If the request is handled successfully by any of the handlers, the `isAuthenticated` variable in the `Client` class will be `true`, indicating successful authentication. Otherwise, it will be `false`, indicating authentication failure.
8. Based on the value of `isAuthenticated`, we can proceed with server access if authentication is successful or handle authentication failure accordingly.

Iteraton design Pattern

Intent It is a behavioral pattern that lets you traverse element of collection without exposing its underlying nepresentation

Applicability

(1) use when youn collection has a complex datastructure under the hood but you want to hide its complexity from clients.

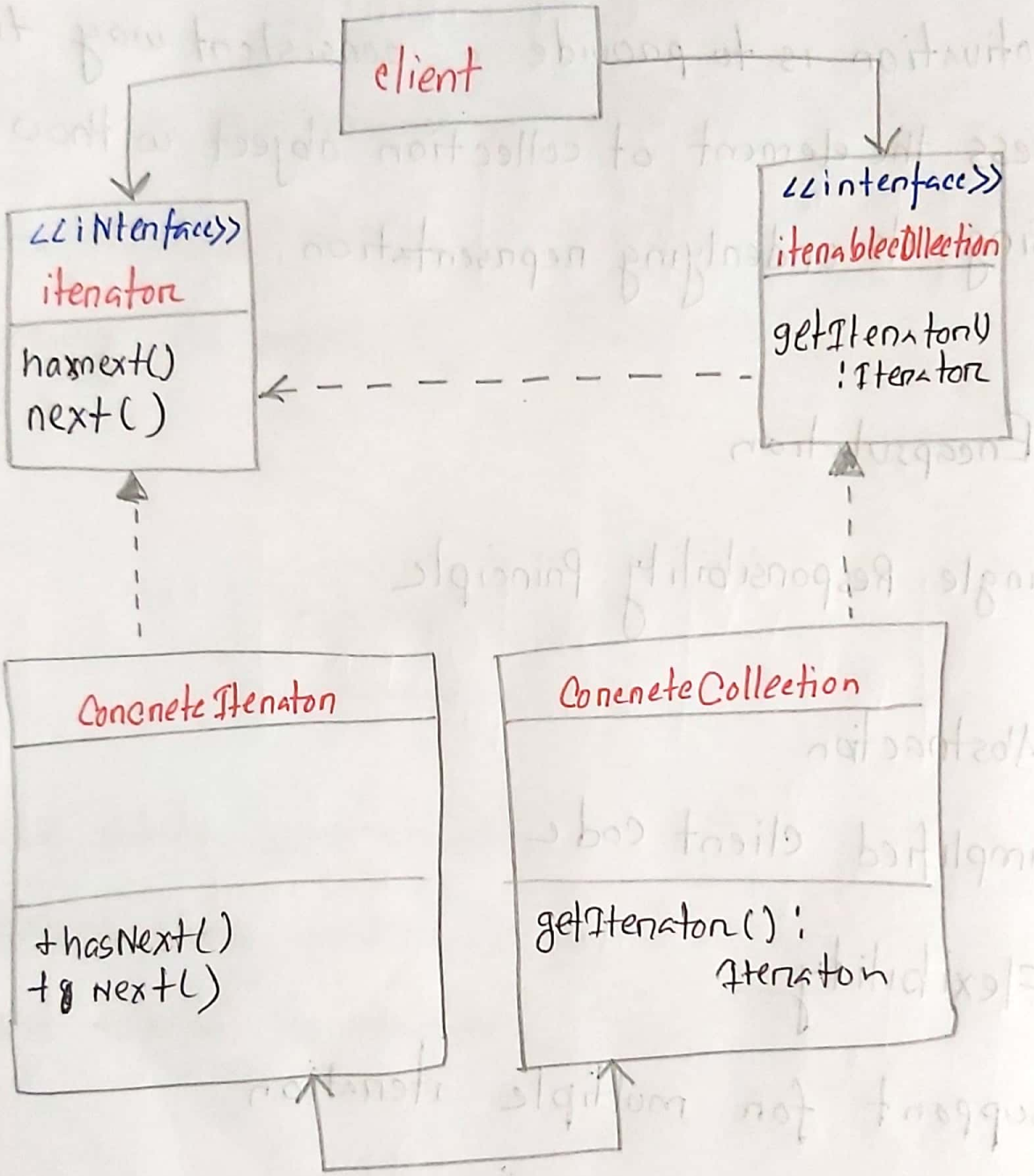
(2) use to reduce duplication of traversal code across your app

Motivation

motivation is to provide a consistent way to acess the element of collection object without exposing its underlying representation.

1. Encapsulation
2. Single Responsibility Principle
3. Abstraction
4. Simplified client code
5. Flexibility
6. Support for multiple iteration

UML Diagram


client
<<interface>>
iterator
hasnext()
next()
<<interface>>
iterableCollection
getIterator()
:Iterator
ConcreteIterator
+hasNext()
+next()
ConcreteCollection
getIterator():
Iterator

1. We start by defining an `Iterator` interface. It declares two methods: `hasNext()` to check if there are more elements, and `next()` to retrieve the next element. This interface serves as a contract for all iterators.
2. Next, we define a `Collection` interface. It declares a single method `getIterator()` that returns an instance of the `Iterator` interface. This interface represents a collection of elements and provides a way to access them using an iterator.
3. We implement the `NameIterator` class, which is a concrete implementation of the `Iterator` interface. It maintains a reference to an array of names (`names`) and a `position` variable to keep track of the current position while iterating. The `hasNext()` method checks if there are more names in the array by comparing the current position with the length of the array. The `next()` method retrieves the next name from the array and increments the position.
4. We implement the `NameCollection` class, which is a concrete implementation of the `Collection` interface. It takes an array of names in its constructor and stores them internally. The `getIterator()` method creates a new instance of the `NameIterator` class and passes the array of names to it. It returns the created iterator, which allows accessing the names in the collection.
5. In the client code (`Main` class), we create an array of names (`names`) containing "John," "Emily," "David," and "Sarah."
6. We create an instance of `NameCollection` called `collection` and pass the `names` array to its constructor. This initializes the collection with the names.
7. We retrieve an iterator from the collection by calling the `getIterator()` method. This gives us an instance of the `NameIterator` class.
8. We use a `while` loop to iterate over the collection. The loop condition checks if the iterator has more elements using the `hasNext()` method.
9. Inside the loop, we retrieve the next name from the iterator using the `next()` method and store it in the `name` variable.
10. Finally, we print each name to the console.

1

```
// Step 1: Iterator interface
interface Iterator {
    boolean hasNext();
    String next();
}
```

2

```
// Step 2: Collection interface
interface Collection {
    Iterator getIterator();
}
```

3

```
// Step 3: NameIterator implementation of Iterator interface
class NameIterator implements Iterator {
    private String[] names;
    private int position;

    public NameIterator(String[] names) {
        this.names = names;
        this.position = 0;
    }

    public boolean hasNext() {
        return position < names.length;
    }

    public String next() {
        String name = names[position];
        position++;
        return name;
    }
}
```

Participant:

1. Iterator
2. IteratableCollection : Collection
3. ConcreteIterator: NameIterator
4. ConcreteCollection : NameCollection

4

```
// Step 4: NameCollection implementation of Collection interface
class NameCollection implements Collection {
    private String[] names;
    public NameCollection(String[] names) {
        this.names = names;
    }
    public Iterator getIterator() {
        return new NameIterator(names);
    }
}
```

5

```
public class Client {
    public static void main(String[] args) {
        String[] names = {"John", "Emily", "David", "Sarah"};

        Collection collection = new NameCollection(names);
        Iterator iterator = collection.getIterator();

        while (iterator.hasNext()) {
            String name = iterator.next();
            System.out.println(name);
        }
    }
}
```

what is separation of concerns in design pattern ?

separation of concern is a principle in software engineering that advocates for dividing a system into distinct parts with each part addressing a separate concerns / responsibility.

It is a practice of isolating different aspects of system's functionality into separate components or modules allowing for better organization, maintability and reusability.

ways

1. Encapsulation
2. Single Responsibility Principle
3. Loose coupling
4. separation of interface and implementation