Agenda

Structural Design Patterns

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

Recap

Design Patterns

What are they?

- Existing, battle tested solutions to common Engineering problems

Why are they needed?

- Save time solving problems
  - Battle tested
  - Steal/Reuse existing code
  - Commonly known solutions
- Reading code
  - you can skip 90% of the code because you already know what it does

Creational Design Patterns

Tell us how to `create an object`

5 common ones

- Singleton
- Builder
- Factory
    - Abstract Factory
- Prototype

What's common b/w them?

- function that will return the desired object

Structural Design Patterns

---

tell us how to `stucture our objects`

Common Feature

- we will have some class X
- we won't use the class directly
- create another class Y
- we will use class X via class Y

Adapter Pattern

Data Adapter:

```
class XMLToJSONAdapter {
    static JSON toJSON(XML item) {
    }
}

class Client {
    void main() {
        XML data = getDataFromSomewhere();
        // mould / adapt the data into the expected format
        JSON jsonData = XMLToJSONAdapter.toJSON(data);
        processor.processJSONData(data);
```

```
    }
}
```

Class Adapter

Phone Adapter

“Interface” for your phone charger has certain features
* earth pin
* live pin
* neutral pin
Wall Socket does not have an earth socket

If the interfaces don’t match, you can’t use the two objects together

Type C USB port
MicroUSB port
Patli pin waala charger

Somehow create a new device that “adapts” b/w these two things

PhonePe - Yes Bank APIs
forced to chage from Yes Bank to ICICI

```
interface BankProvider {
    void makePayment(UPIId from, UPIId to, Float amount);

    float checkBalance(UPIId account);
}


class YesBank { } // Old API
class YesBankToBankProviderAdapter implements BankProvider {
    private YesBank yesBank;
}

// new API code

class PaymentDetails {
    UPIId from, to;
    Float amount;
}

class AccountDetails {
```

```
    float balance;
}

class ICICI {
    void pay(PaymentDetails details) {}

    AccountDetails getAccountDetails(UPIId account) {}
}


class ICICIToBankProviderAdapter implements BankProvider {
    private ICICI icici;

    void makePayment(UPIId from, UPIId to, Float amount) {
        return icici.pay(new PaymentDetails(from, to, amount));
    }

    float checkBalance(UPIId account) {
        return icici.getAccountDetails(account).getBalance();
    }
}




class PhonePe {
    void main() {
        // 100s of thousands of lines of code like this
        BankProvider provider = new YesBank();

        provider.makePayment(102, 100, 100.0);
        provider.checkBalance(102);
    }
}
```

You have dependencies
- there are some libraries that you rely on
- libraries can change
- you have to completely switch libraries

- you use the Interfaces and Adapater pattern in conjunction

All you codebase will be implemented according to some “custom” interface that you want to use

example: All of PhonePay’s codebase will internally use `BankProvider`

originally: For library X create an XToBankProviderAdapter

tomorrow: For library Y create an YToBankProviderAdapter

Adapter vs Wrapper

Wrapper - general case

- protect it
- modify its behavior (adapter)
- logging / extra hidden functionality (decorator)
- delegates commands somewhere else (proxy)

ExpectedCode

```
interface BankProvider {
    makePayment();
    getBalkance();
}

class PhonePe {
    void main() {
        // all the code here codes to the interface BankProvider
    }
}

class YesBank {

}

class YesBankToBankProviderAdapter implements BankProvider {

}


class ICICIBankToBankProviderAdapter implements BankProvider {

}


class YesBank {
    void makePayment();
    float getBalance();
}

class PhonePe{
```

```
    void main() {
        obj.makePayment();
        obj.getBalance();
    }
}

class ICICI {
    void pay();

    AccountDetails getAccountDetails();
}

class ICICIToYesBankAdapter {}


// Yes Bank public library code

class YesBank {
    AccountDetails getAccountDetails(UPIId account);

    void pay(PaymentDetails details);
}




interface BankingProvider {
    float getBalance(UPIId account);

    void makePayment(UPIId from, UPIId to, Float amount);
}


class PhonePe {
    BankProvider provider;

    public PhonePe(BankProvider provider) { this.provider = provider; }

    float getBalance(String username) {
        // get the acccount if for this user
        provider.getBalance(accountId);
    }

    void pay(String from, String to, float amount) {
        provider.makePayment(...);
    }
}


class YesBankToBankingProviderAdapter implements BankingProvider {
```

```
    private YesBank yesBank;
}
```

// Scene 1

ICICI - API gateway
REST endpoints

ICICILibrary

```
class ICICI {
    float getBalance(String username) {
        // get the acccount if for this user
        provider.getBalance(accountId);
    }

    void pay(String from, String to, float amount) {
        provider.makePayment(...);
    }
}
```

Interfacing with another business (3rd party service) - always use an interface + adapters
Database, Messaging Queue - use interfaces and adapters

Math, fileIO, raqndom number ... will not use adapters (very very unlikely to change)

Design Patterns reduce the amount of code - WRONG

```
class MySQLProvider {
    String executeQueryFast(String query);
    void connect(String domain, String port, String username, String password);
}


interface DBProvider {
    DBResult executeQuery(Query query);

    void connect(ConnectionParameters params);

    void commit();

    void rollback();
}


class MySQLDBTODBPRoviderAdapter implements DBProvider {
    private MySQLProvider provider;
```

```
    DBResult executeQuery(Query query) {}

    void connect(ConnectionParameters params) {}

    void commit() {}

    void rollback() {}
}


class Client {
    DBProvider provider;

    public Client(DBProvider provider) { this.provider = provider; }

    void main() {

        DBResult result = provider.executeQuery(new Query("select count(1) from user
        print(result.getInt());
    }
}
```

Service ML (API - domain:port/speech POST) REST

MLRESTToGraphQLAdapterService - GraphQL endpoint
internally call the ML Service

Service 1 (graphQL)

Adapter pattern - Purpose of wrapping - change the interface of the object

Decorator Pattern

We will create a wrapper around an object - purpose of wrapping - modify the behavior / add-on / decorate the object

```
class Pizza {
    public Pizza(PizzaBase base) {
    }
```

```
}


class MeatDecorator {
    public Pizza addSausages(Pizza pizza, Int amount) {
    }
}


class Dominos {
    void placeOrder(String orderDetails) {
        Pizza pizza = new Pizza(PizzaBaseFactor.getBase(...));
        MeatDecorator.addSausages(pizza, 10);
    }
}



class HTML {
    public String div(String content) {
        return "<div>" + content + "</div>";
    }
    public String span(String content) {
        return "<span>" + content + "</span>";
    }
    public String p(String content) {
        return "<p>" + content + "</p>";
    }
}


class MYHTMLDecorators {
    // not really a class
    // we're using this class as a "NameSpace"
    // as a way to "club" together similar functions
    public static String center(String html) {
        return "<div style='margin: auto; width: 60%'>" + html + "</div>"
    }

    public static String makeBold(String html) {
        return "<b>" + html + "</b>";
    }

    public static String spin(String html) {
        ...
    }

}

class CountryFlagDecorators {
```

```
    HashMap<String, String> countryFlagASCIIArts;

    public CountryFlagDecorators() {
        // loads the ASCII art from the DB
    }

    public String addIndianFlag(String html) {
        return html + "<div>" + this.countryFlagASCIIArts.get("India") + "</div>";
    }
}

class Client {
    public void main() {
        HTML html = new HTML();
        String myComponent = html.div();
        String centeredComponent = HTMLDecorators.center(HTMLDecorators.makeBold(myC
    }
}
```

Builder Pattern - you want to create a complex object BEFORE using it

Decorator Pattern - you might ALREADY have an object and you want to modify/extend it

---

Functions vs Methods

Functions - any piece of code that takes input and gives output

```
# stand alone function

def make_bold(html):
    return '<b>' + html + '</b>'



// Functions - I don't know what they're
// I always work with Classes

class MyClass {
    // this is NOT a function - this is a "Instance Method"
    String makeBold(String html) {
        // special pointer hidden inside here
        // this
        // -> an object of the class MyClass
        this.doSomething();
    }
}
```

```
MyClass obj = new MyClass();
obj.makeBold(); // invoking this function "on" the obj
```

Interface

- these are the methods that will be available in the class

Object -> Decorated it
Object Decorator

Code that we wish to decorate

Method decorator
Class decorator

```
class Fib {
    static int fib(int n) {
        if(n <= 1) return n;
        return fib(n-1) + fib(n-2);
    }
}

class MemoFib {
    private static values[];

    static int fib(int n) {

        if(n <= 1) return n;
        int ans = fib(n-1) + fib(n-2);

        return ans;
    }
}
```

```
if(values[n] != 0)
    return values[n];


values[n] = ans;
```

```
def memoizer(fn):
    dp = {}

    def memoized(n):
        if n in dp:
            return dp[n]
```

```
        ans = fn(n)
        dp[n] = ans
        return ans

    return memoized


def fib(n): # O(2^n)
    if n <= 1: return n
    return fib(n-1) + fib(n-2)

fib = memoizer(fib) # O(n) time
```

Logging

```
String myLogDecorator(String message) {
    return "date:time:machine:process" + message;
}

logger.addDecorator(myLogDecorator);

logger.logInfo("User clicked button");
```

date: timestamp: machineid: processid: User clicked button

```
from functools import lru_cache


@lru_cache(None)
def recursive_solution():
    solve   # O(2^n)
```

Builder Pattern Doubts

```
class User {
    // make sure that the objects of this class are immutable
    // can't modify them once created

    // simply don't provide any setters

    String name, password;
    Float age;
```

```
    float getAge() { return age; }

    private User(Builder builder) {}

    public static Builder {
        String name, password;
        Float age;

        public Builder() {}

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

        public build() {return new User(this);}
    }
}

class Client {
    void foo() {
        User u = User.Builder()
                     .setName("")
                     .setAge()
                     .build();
    }
}
```

Builder Pattern

Creational
- Singleton
- Builder
- ways of working around language limitations

Structural

Behavioral

```
class Color {
    public User(Float red, Float green, Float blue) {}
    // n parameters -> 2^n different constructors

    static Builder {
        setRed()
        setGreen()
        setBlue()
    }
}
```

```
class Client {
    void foo() {
        Color c = new Color(100, 20, 200);

    }
}
```

Reason 1 - params may have the same type - easy to confuse

Reason 2 - params in different order

Reason 3 - optional parameters

Reason 4 - Default values

```
class Color:
    def __init__(self, red, green, blue=30):
        ...


c = Color(green=10, red=20)
```

UPI interface that is globally followed

ICICI -> internally follow some interface

UPIADapter for myself

pragy@scaler.com
+91-7351769231