# Team work

## 1- Ahmed Ashour Musa

- ✓ solid principles (LSP -ISP - DIP)
- ✓ Design Patterns (Creational Patterns)
- ✓ Architectural Patterns (Layered Architecture RepositoryArchitecture )
- ✓ Test driven development

## 2- Abdul Rahman Hilal

- ✓ solid principles (SRP-OCP)
- ✓ Design Patterns (Structural Patterns)
- ✓ Architectural Patterns(MVP-MVVM)

## 3- Ahmed Al-Sayed Mohamed Ali Sanad

- ✓ Design Patterns (behavioral Patterns)
- ✓ Architectural Patterns(MVC- Clientserver)

# Outline

## 1- solid principles

- Single Responsibility Principle (SRP)
- Open-Closed Principle (OCP)
- Liskov Substitution Principle (LSP)
- Interface Segregation Principle (ISP)
- Dependency Inversion Principle (DIP)

## 2- Design Patterns

### - Creational Patterns:

- Singleton
- Factory Method
- Builder

### - Structural Patterns:

- Adapter Pattern
- Decorator Pattern

### - behavioral Patterns:

- Chain of Responsibility
- Command

## 3- Architectural Patterns

- Model-View-Controller (MVC)
- Model-View-ViewModel (MVVM)
- Model-View-Presenter (MVP)
- Layered Architecture
- Repository Architecture
- Client-server Architecture

## 4- Test driven development

# Solid Principles

**They are rules that must be followed when building code to obtain code**

- **1- Flexible**
- **2- Robust**
- **3- Reusable**

## 5 Principles is:

1. **Single Responsibility Principle (SRP)**
2. **Open-Closed Principle (OCP)**
3. **Liskov Substitution Principle (LSP)**
4. **Interface Segregation Principle (ISP)**
5. **Dependency Inversion Principle (DIP)**

## 1- Single Responsibility Principle

**A class should have a single responsibility.**

**every class should have only one reason to change**

# Python Code Example1 (Violation)

**The Person class violates the single responsibility principle.**

**Why?**

This Person class has **two** responsibilities

– Manage the person's property.

- Store the person in the database

```python
class Person:
    def __init__(self, name):
        self.name = name

    def __repr__(self):
        return f'Person(name={self.name})'

    @classmethod
    def save(cls, person):
        print(f'Save the {person} to the database')


if __name__ == '__main__':
    p = Person('John Doe')
    Person.save(p)
```

# Python Code Example1 (obey)

```python
class Person:
    def __init__(self, name):
        self.name = name

    def __repr__(self):
        return f'Person(name={self.name})'


class PersonDB:
    def save(self, person):
        print(f'Save the {person} to the database')


if __name__ == '__main__':
    p = Person('John Doe')

    db = PersonDB()
    db.save(p)
```

# Python Code Example2 (Violation)

```python
# file_manager_srp.py

from pathlib import Path
from zipfile import ZipFile

class FileManager:
    def __init__(self, filename):
        self.path = Path(filename)

    def read(self, encoding="utf-8"):
        return self.path.read_text(encoding)

    def write(self, data, encoding="utf-8"):
        self.path.write_text(data, encoding)

    def compress(self):
        with ZipFile(self.path.with_suffix(".zip"), mode="w") as archive:
            archive.write(self.path)

    def decompress(self):
        with ZipFile(self.path.with_suffix(".zip"), mode="r") as archive:
            archive.extractall()
```

# Python Code Example2 (obey)

```python
# file_manager_srp.py

from pathlib import Path
from zipfile import ZipFile

class FileManager:
    def __init__(self, filename):
        self.path = Path(filename)

    def read(self, encoding="utf-8"):
        return self.path.read_text(encoding)

    def write(self, data, encoding="utf-8"):
        self.path.write_text(data, encoding)

class ZipFileManager:
    def __init__(self, filename):
        self.path = Path(filename)

    def compress(self):
        with ZipFile(self.path.with_suffix(".zip"), mode="w") as archive:
            archive.write(self.path)

    def decompress(self):
        with ZipFile(self.path.with_suffix(".zip"), mode="r") as archive:
            archive.extractall()
```

## 2- Open-Closed Principle (OCP)

**software entities (classes, functions, modules, etc.) should be open for extension, but closed for modification.**

### Python Code Example1(Violation)

```python
from math import pi

class Shape:
    def __init__(self, shape_type, **kwargs):
        self.shape_type = shape_type
        if self.shape_type == "rectangle":
            self.width = kwargs["width"]
            self.height = kwargs["height"]
        elif self.shape_type == "circle":
            self.radius = kwargs["radius"]

    def calculate_area(self):
        if self.shape_type == "rectangle":
            return self.width * self.height
        elif self.shape_type == "circle":
            return pi * self.radius**2
```

### Python Code Example1 (obey)

```python
from abc import ABC, abstractmethod
from math import pi

class Shape(ABC):
    def __init__(self, shape_type):
        self.shape_type = shape_type

    @abstractmethod
    def calculate_area(self):
        pass
```

```python
class Rectangle(Shape):
    def __init__(self, width, height):
        super().__init__("rectangle")
        self.width = width
        self.height = height

    def calculate_area(self):
        return self.width * self.height
```

```python
class Circle(Shape):
    def __init__(self, radius):
        super().__init__("circle")
        self.radius = radius

    def calculate_area(self):
        return pi * self.radius**2
```

```python
class Square(Shape):
    def __init__(self, side):
        super().__init__("square")
        self.side = side

    def calculate_area(self):
        return self.side**2
```

## 3- Liskov Substitution Principle

**Subclasses (Derived) classes must be substitutable for their base classes**

```
  @property
  def height(self):
      return self._height
  @height.setter
  def height(self, value):
      self._height = value
  def get_area(self):
      return self._width * self._height
```

```
class Square(Rectangle):
    def __init__(self, size):
        Rectangle.__init__(self, size, size)
    @Rectangle.width.setter
    def width(self, value):
        self._width = value
        self._height = value
    @Rectangle.height.setter
    def height(self, value):
        self._width = value
        self._height = value
```

```
class Rectangle:
    def __init__(self, height, width):
        self._height = height
        self._width = width
    @property
    def width(self):
        return self._width
    @width.setter
    def width(self, value):
        self._width = value
```

```
def get_squashed_height_area(Rectangle):
    Rectangle.height = 1
    area = Rectangle.get_area()
    return area
rectangle = Rectangle(5, 5)
square = Square(5)
assert get_squashed_height_area(rectangle) == 5  #
expected 5
assert get_squashed_height_area(square) == 1  #
expected 5
```

## Python Code Example1 (Violatio )

```python
from abc import ABC, abstractmethod


class Notification(ABC):
    @abstractmethod
    def notify(self, message, email):
        pass


class Email(Notification):
    def notify(self, message, email):
        print(f'Send {message} to {email}')


class SMS(Notification):
    def notify(self, message, phone):
        print(f'Send {message} to {phone}')


if __name__ == '__main__':
    notification = SMS()
    notification.notify('Hello', 'john@test.com')
```

```python
class Contact:
    def __init__(self, name, email, phone):
        self.name = name
        self.email = email
        self.phone = phone


class NotificationManager:
    def __init__(self, notification, contact):
        self.contact = contact
        self.notification = notification

    def send(self, message):
        if isinstance(self.notification, Email):
            self.notification.notify(message, contact.email)
        elif isinstance(self.notification, SMS):
            self.notification.notify(message, contact.phone)
        else:
            raise Exception('The notification is not supported')
```

```python
if __name__ == '__main__':
    contact = Contact('John Doe', 'john@test.com', '(408)-888-9999')
    notification_manager = NotificationManager(SMS(), contact)
    notification_manager.send('Hello John')
```

## Python Code Example1 (obey)

```python
class Notification(ABC):
    @abstractmethod
    def notify(self, message):
        pass
```

```python
class Email(Notification):
    def __init__(self, email):
        self.email = email

    def notify(self, message):
        print(f'Send "{message}" to {self.email}')
```

```python
class SMS(Notification):
    def __init__(self, phone):
        self.phone = phone

    def notify(self, message):
        print(f'Send "{message}" to {self.phone}')
```

```python
class Contact:
    def __init__(self, name, email, phone):
        self.name = name
        self.email = email
        self.phone = phone
```

```python
class NotificationManager:
    def __init__(self, notification):
        self.notification = notification

    def send(self, message):
        self.notification.notify(message)
```

```python
if __name__ == '__main__':
    contact = Contact('John Doe', 'john@test.com', '(408)-888-9999')

    sms_notification = SMS(contact.phone)
    email_notification = Email(contact.email)

    notification_manager = NotificationManager(sms_notification)
    notification_manager.send('Hello John')

    notification_manager.notification = email_notification
    notification_manager.send('Hi John')
```

# 4- Interface Segregation Principle

**Clients should not be forced to depend on methods that they do not use.**

## Python Code Example1 (Violatio )

```
from abc import ABC, abstractmetho

class Printer(ABC):
    @abstractmethod
    def print(self, document):
        pass

    @abstractmethod
    def fax(self, document):
        pass

    @abstractmethod
    def scan(self, document):
        pass
```

```
class OldPrinter(Printer):
    def print(self, document):
        print(f"Printing {document} in black and white...")

    def fax(self, document):
        raise NotImplementedError("Fax functionality not supported")

    def scan(self, document):
        raise NotImplementedError("Scan functionality not supported")
```

```
class ModernPrinter(Printer):
    def print(self, document):
        print(f"Printing {document} in color...")

    def fax(self, document):
        print(f"Faxing {document}...")

    def scan(self, document):
        print(f"Scanning {document}...")d
```

## Python Code Example1 (obey)

```
from abc import ABC, abstractmetho

class Printer(ABC):
    @abstractmethod
    def print(self, document):
        pass

class Fax(ABC):
    @abstractmethod
    def fax(self, document):
        pass

class Scanner(ABC):
    @abstractmethod
    def scan(self, document):
        pass
```

```
class OldPrinter(Printer):
    def print(self, document):
        print(f"Printing {document} in black and white...")
```

```
class NewPrinter(Printer, Fax, Scanner):
    def print(self, document):
        print(f"Printing {document} in color...")

    def fax(self, document):
        print(f"Faxing {document}...")

    def scan(self, document):
        print(f"Scanning {document}...")d
```

# 5-Dependency Inversion Principle

**Depend on abstractions, not on concretions**

## Python Code Example1 (Violation)

```python
class FXConverter:
    def convert(self, from_currency, to_currency, amount):
        print(f'{amount} {from_currency} = {amount * 1.2} {to_currency}')
        return amount * 1.2


class App:
    def start(self):
        converter = FXConverter()
        converter.convert('EUR', 'USD', 100)


if __name__ == '__main__':
    app = App()
    app.start()
```

# Python Code Example1 (obey)

```python
from abc import ABC


class CurrencyConverter(ABC):
    def convert(self, from_currency, to_currency, amount) -> float:
        pass
```

```python
class FXConverter(CurrencyConverter):
    def convert(self, from_currency, to_currency, amount) -> float:
        print('Converting currency using FX API')
        print(f'{amount} {from_currency} = {amount * 1.2} {to_currency}')
        return amount * 2
```

```python
class App:
    def __init__(self, converter: CurrencyConverter):
        self.converter = converter


    def start(self):
        self.converter.convert('EUR', 'USD', 100)
```

```python
if __name__ == '__main__':
    converter = FXConverter()
    app = App(converter)
    app.start()
```

```
Converting currency using FX API
100 EUR = 120.0 USD
```

# Design Patterns

**It is a solution that is crafted and designed by brilliant minds in order to solve repeating problem that often occurs in Software development process**

## Gang of Four (GoF) patterns:

- **Creational Patterns** ***(abstracting the object-instantiation process)***
  - **Factory Method** **Abstract Factory** **Singleton**
  - **Builder** **Prototype**
- **Structural Patterns** ***(how objects/classes can be combined)***
  - **Adapter** **Bridge** **Composite**
  - Decorator Façade Flyweight
  - Proxy
- **Behavioral Patterns** ***(communication between objects)***
  - **Command** **Interpreter** **Iterator**
  - Mediator **Observer** State
  - **Strategy** **Chain of Responsibility** **Visitor**
  - **Template Method**

# Creational Patterns

## 1. Singleton pattern:

*A class that has only a single instance*

### Understanding Singleton Pattern by Real Example

```
class Car:
    def __init__(self):
        pass  # default empty constructor

    def start(self):
        print("Car engine start")
```

```
class CarShop:
    def __init__(self):
        self.car_sold = 0  # because we want to keep track of the number of cars tha

    # function to sell our car, as it returns a car
    def sell_car(self):
        self.car_sold += 1
        return Car()
```

```
# Create an instance of the CarShop class
car_shop = CarShop()

# Sell a car from the car shop
car = car_shop.sell_car()
```

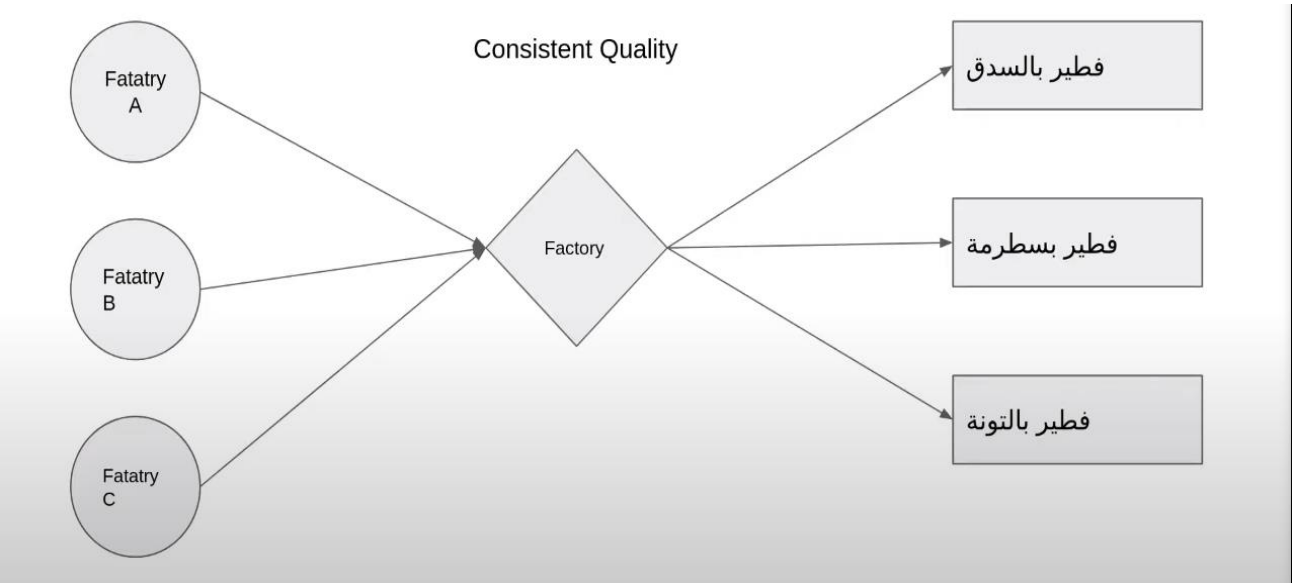
# solve

```
class CarShop:
    # Static variable to hold the singleton instance
    car_shop = None

    def __init__(self):
        # Private variable to keep track of the number of cars sold
        self.car_sold = 0

    # Static method to get the singleton instance
    @staticmethod
    def get_instance():
        if CarShop.car_shop is None:
            CarShop.car_shop = CarShop()
        return CarShop.car_shop

    # Method to sell a car
    def sell_car(self):
        self.car_sold += 1
        return Car()
```

```
car = CarShop.get_instance().sell_car()
```

## 2. Factory pattern:

***Defines an interface for creating an object, but let subclasses decide which class to instantiate***




Bad code

Context X

```
Take input from user
if(input == 'ProductA')
    product = new ProductA()
Else if(input == 'ProductB')
    product = new ProductB()
Else if(input == 'ProductC')
    product = new ProductC()
```

Context Y

```
Take input from user
if(input == 'ProductA' &&
something else)
    product = new ProductA()
Else if(input == 'ProductB')
    product = new ProductB()
Else if(input == 'ProductC')
    product = new ProductC()
```

# Python example

```python
from abc import ABC, abstractmethod

class CardType(ABC):
    def __init__(self):
        self.credit_limit = 0.0

    @abstractmethod
    def set_credit_limit(self):
        pass

    def __str__(self):
        return f"Your card is {type(self).__name__} & your credit limit is {self.cred
```

```python
class SilverCard(CardType):
    def __init__(self):
        self.set_credit_limit()

    def set_credit_limit(self):
        self.CreditLimit = 100000
```

```python
class GoldCard(CardType):
    def __init__(self):
        self.set_credit_limit()

    def set_credit_limit(self):
        self.CreditLimit = 250000
```

```python
class PlatinumCard(CardType):
    def __init__(self):
        self.set_credit_limit()

    def set_credit_limit(self):
        self.CreditLimit = 500000
```

```python
if __name__ == "__main__":
    salary = float(input("Enter your salary: "))

    if salary < 50000:
        card_type = "Silver"
    elif salary < 100000:
        card_type = "Gold"
    else:
        card_type = "Platinum"

    my_card = Factory.get_card(card_type)
    print(my_card)
```

```python
class Factory:
    @staticmethod
    def get_card(card_type):
        if card_type == "Silver":
            return SilverCard()
        elif card_type == "Gold":
            return GoldCard()
        elif card_type == "Platinum":
            return PlatinumCard()
        else:
            return None
```

```
Enter your salary :
85000
Your card is GoldCard & your credit limit is 250000.0
```

```
Enter your salary :
40000
Your card is SilverCard & your credit limit is 100000.0
```

```
Enter your salary :
165000
Your card is PlatinumCard & your credit limit is 500000.0
```

## 3. Builder pattern:

**is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.**

## Python example

```
# Product class
class Person:
    def __init__(self, name=None, age=None, address=None):
        self.name = name
        self.age = age
        self.address = address

    def __str__(self):
        return f"Name: {self.name}, Age: {self.age}, Address: {self.address}"

# Builder class
class PersonBuilder:
    def __init__(self):
        self.person = Person()

    def set_name(self, name):
        self.person.name = name
        return self

    def set_age(self, age):
        self.person.age = age
        return self

    def set_address(self, address):
        self.person.address = address
        return self

    def build(self):
        return self.person

# Usage
person = (PersonBuilder()
          .set_name("Alice")
          .set_age(28)
          .set_address("456 Elm St")
          .build())

print(person)
```

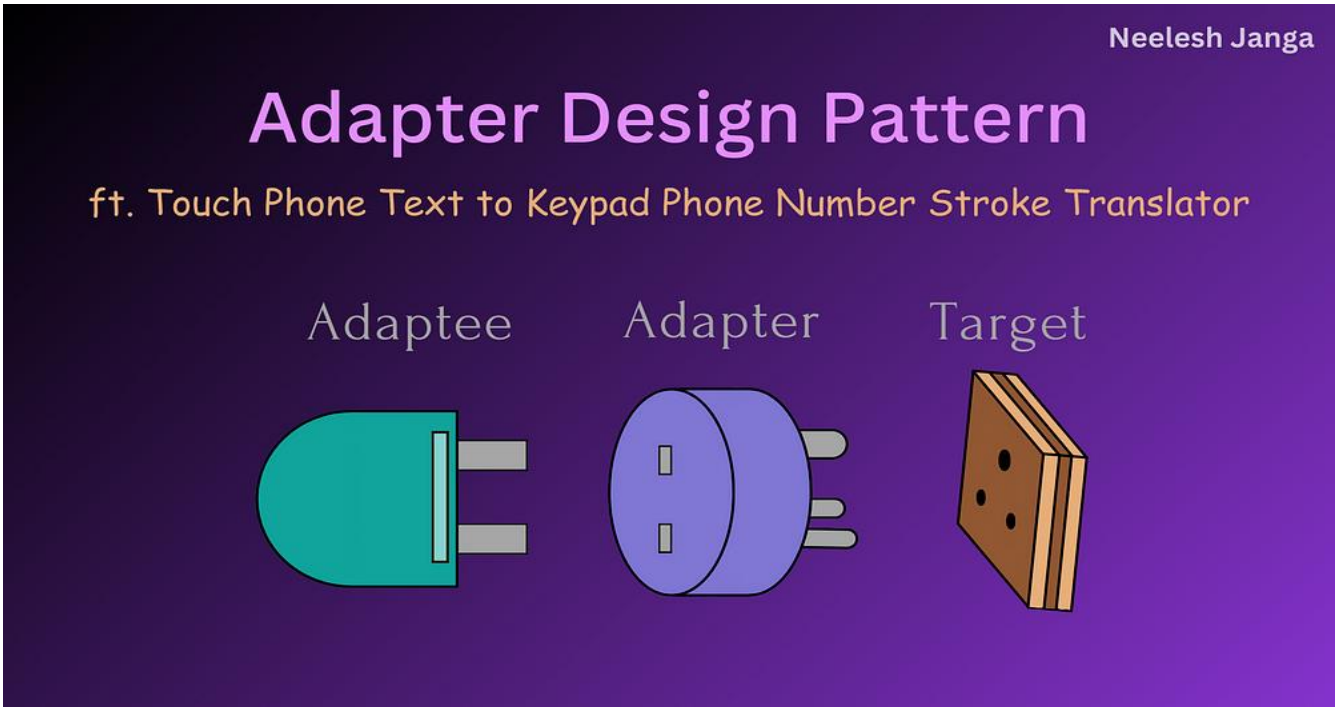
# Python example

```python
# Product class
class Computer:
    def __init__(self, cpu=None, ram_gb=None, storage_gb=None, gpu=None, monitor=None):
        self.cpu = cpu
        self.ram_gb = ram_gb
        self.storage_gb = storage_gb
        self.gpu = gpu
        self.monitor = monitor

    def __str__(self):
        return f"CPU: {self.cpu}, RAM: {self.ram_gb}GB, Storage: {self.storage_gb}GB, GPU: {self.gpu}, Monitor: {self.monitor}"

# Builder class
class ComputerBuilder:
    def __init__(self):
        self.computer = Computer()

    def set_cpu(self, cpu):
        self.computer.cpu = cpu
        return self

    def set_ram_gb(self, ram_gb):
        self.computer.ram_gb = ram_gb
        return self

    def set_storage_gb(self, storage_gb):
        self.computer.storage_gb = storage_gb
        return self

    def set_gpu(self, gpu):
        self.computer.gpu = gpu
        return self

    def set_monitor(self, monitor):
        self.computer.monitor = monitor
        return self

    def build(self):
        return self.computer

# Usage
computer = (ComputerBuilder()
            .set_cpu("AMD Ryzen 9")
            .set_ram_gb(32)
            .set_storage_gb(1024)
            .set_gpu("AMD Radeon RX 6900 XT")
            .set_monitor("32-inch 4K")
            .build())

print(computer)
```

# Structural Patterns

## 1. Adapter Pattern:

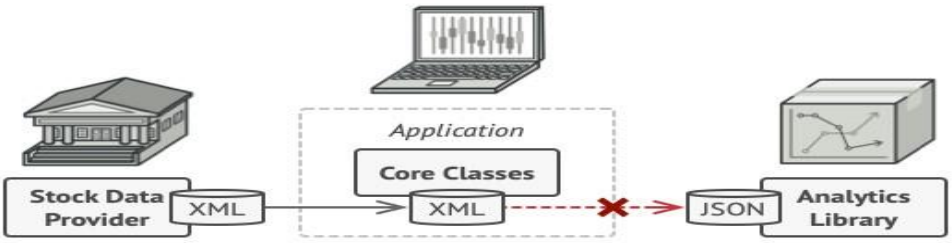
As mentioned earlier, the Adapter Pattern allows objects with incompatible interfaces to work together. It acts as a bridge between two incompatible interfaces, converting the interface of a class into another interface that a client expects.



# What is the problem here?

- **Problem**
  - **Imagine that you're creating a stock market monitoring app.**

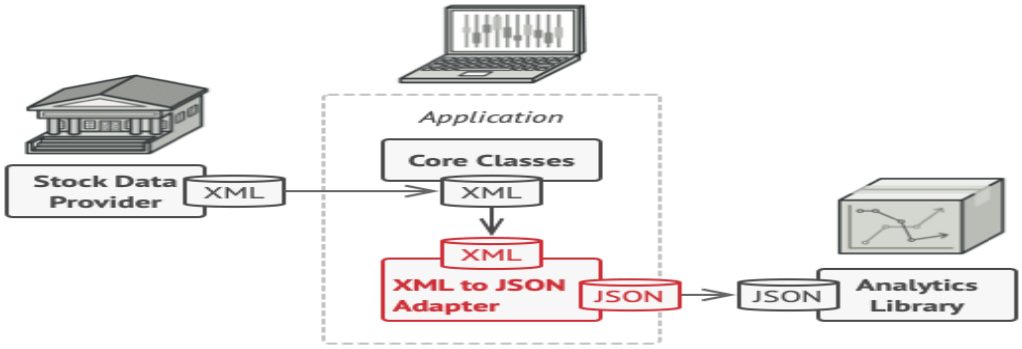
    **The app downloads the stock data from multiple sources in XML format and then displays nice-looking charts and diagrams for the user.**
  - **At some point, you decide to improve the app by integrating a smart 3rd-party analytics library.**
    - **But there's a catch: the analytics library only works with data in JSON format.**




*You can't use the analytics library "as is" because it expects the data in a format that's incompatible with your app.*

# Solution

- **You can create an *adapter*.**
  - **This is a special object that converts the interface of one object so that another object can understand it.**



## Python example:

```python
# Target interface
class Target:
    def request(self):
        pass

# Adaptee (the class we want to adapt)
class Adaptee:
    def specific_request(self):
        return "Adaptee's specific request"

# Adapter
class Adapter(Target):
    def __init__(self, adaptee):
        self.adaptee = adaptee

    def request(self):
        return f"Adapter: {self.adaptee.specific_request()}"

# Client
def client_code(target):
    print(target.request())

if __name__ == "__main__":
    adaptee = Adaptee()
    adapter = Adapter(adaptee)
    client_code(adapter)
```

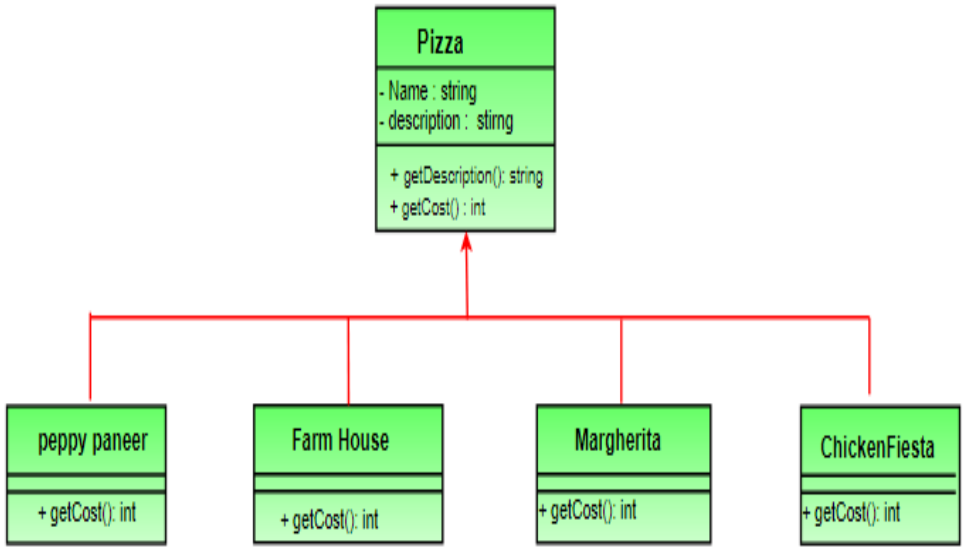
## 2. Decorator Pattern:

**Decorator is a structural design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.**

**"The ability to attach additional responsibilities to an object dynamically. Decorators, provide a flexible alternative to sub classing for extending functionality."**

**Understanding by Example** :

- To understand decorator pattern let us consider a scenario inspired from the book “Head First Design Pattern”. Suppose we are building an application for a pizza store and we need to model their pizza classes. Assume they offer four types of pizzas namely Peppy Paneer, Farmhouse, Margherita and Chicken Fiesta. Initially we just use inheritance and abstract out the common functionality in a **Pizza** class.



# Python example

```python
# Decorator
class CoffeeDecorator(Coffee):
    def __init__(self, coffee):
        self._coffee = coffee

    def cost(self):
        return self._coffee.cost()

# Concrete decorator
class Milk(CoffeeDecorator):
    def cost(self):
        return self._coffee.cost() + 0.5

class Sugar(CoffeeDecorator):
    def cost(self):
        return self._coffee.cost() + 0.2
```

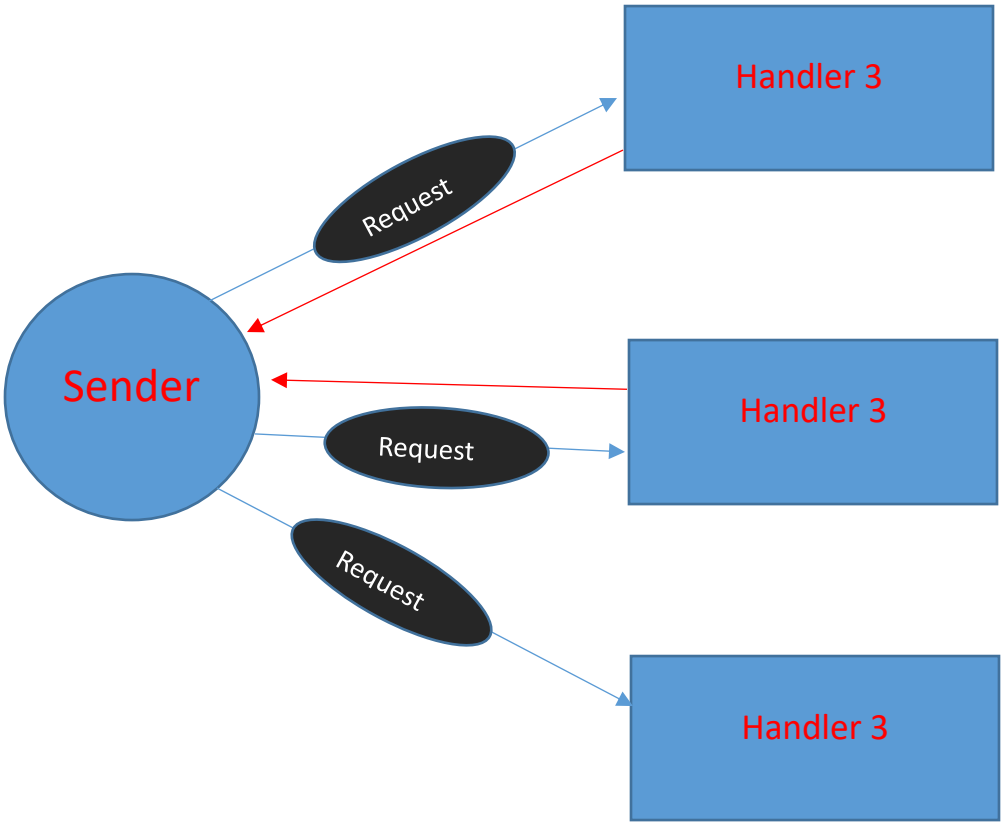
```python
# Usage
coffee = SimpleCoffee()
print(coffee.cost())  # Output: 1

coffee_with_milk = Milk(coffee)
print(coffee_with_milk.cost())  # Output: 1.5

coffee_with_milk_and_sugar = Sugar(coffee_with_milk)
print(coffee_with_milk_and_sugar.cost())  # Output: 1.7
```

# Behavioral pattern

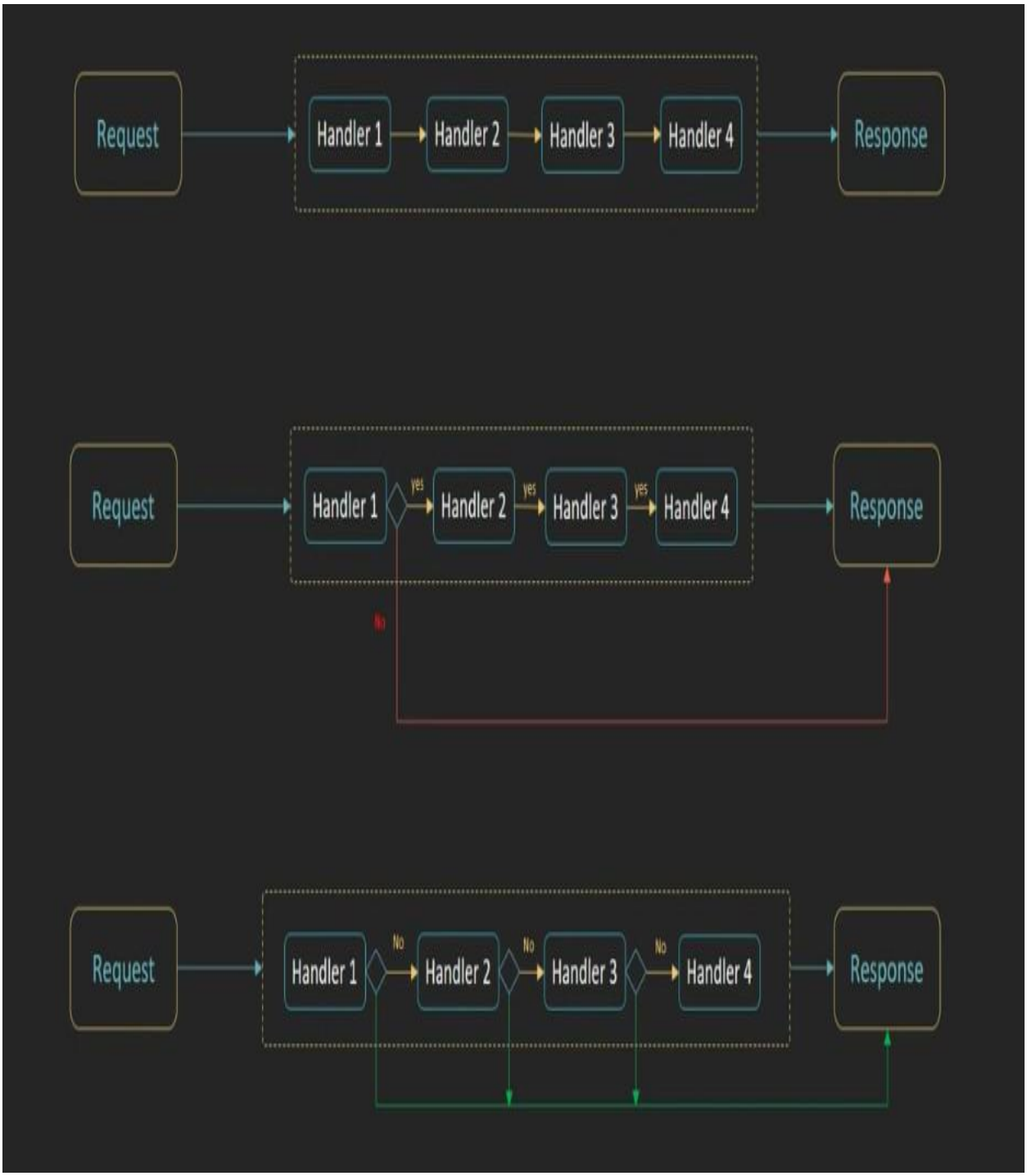
## Chain of Responsibility pattern:

The Chain of Responsibility design pattern is a behavioral pattern that allows an object to pass a request along a chain of handlers until the request is handled. Each handler in the chain has the ability to either handle the request or pass it on to the next handler in the chain.

## Redundancy problem



# Solution


Request
Handler 1
Handler 2
Handler 3
Handler 4
Response
Request
Handler 1
yes
Handler 2
yes
Handler 3
yes
Handler 4
Response
No
Request
Handler 1
No
Handler 2
No
Handler 3
No
Handler 4
Response

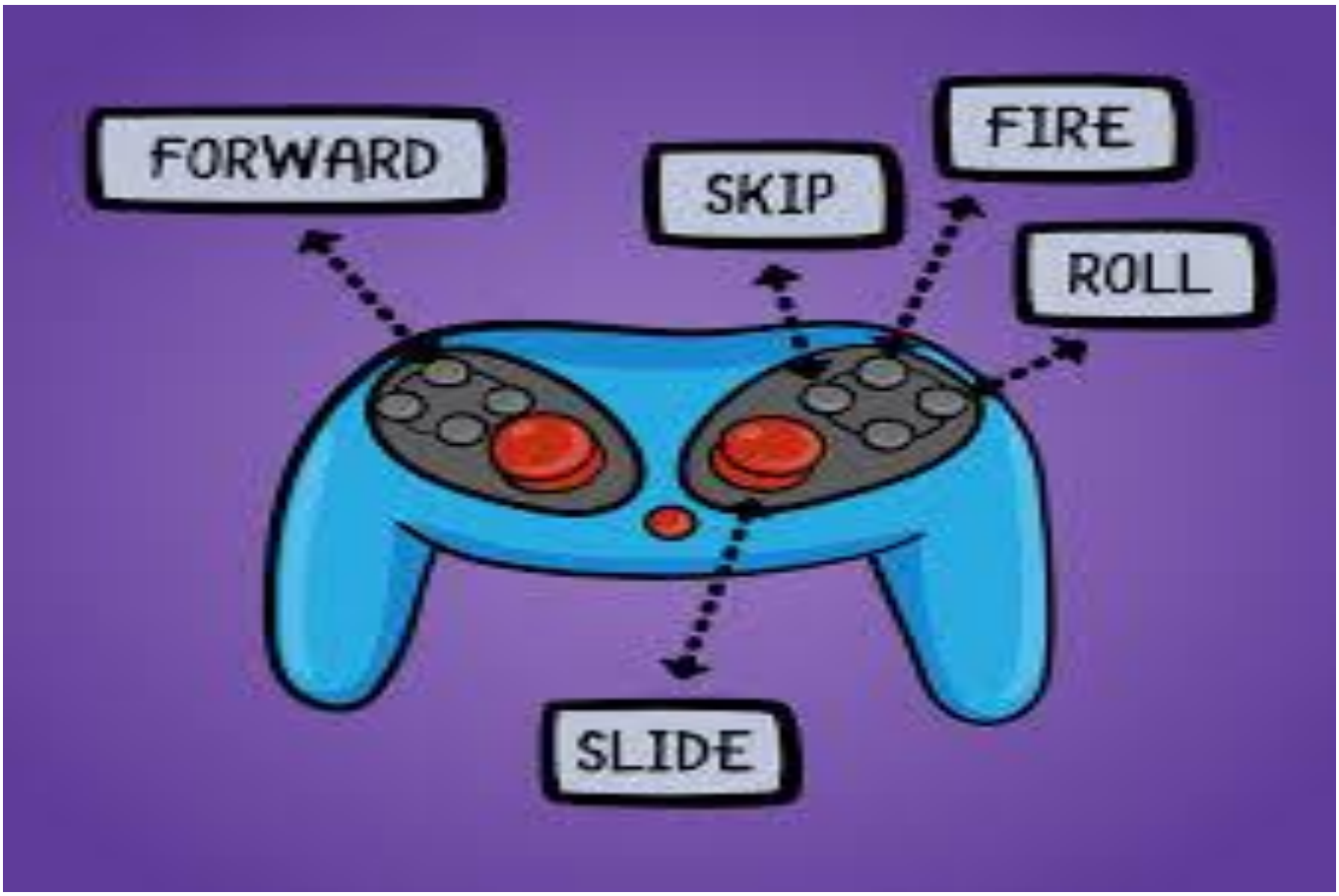
```python
class Handler:
    def __init__(self, successor=None):
        self.successor = successor

    def handle_request(self, request):
        if self.successor:
            return self.successor.handle_request(request)
        else:
            print("Request cannot be handled.")

class ConcreteHandler1(Handler):
    def handle_request(self, request):
        if request == "Type1":
            print("ConcreteHandler1 is handling the request.")
        else:
            print("ConcreteHandler1 cannot handle the request.")
            super().handle_request(request)
```

```python
class ConcreteHandler2(Handler):
    def handle_request(self, request):
        if request == "Type2":
            print("ConcreteHandler2 is handling the request.")
        else:
            print("ConcreteHandler2 cannot handle the request.")
            super().handle_request(request)

class ConcreteHandler3(Handler):
    def handle_request(self, request):
        if request == "Type3":
            print("ConcreteHandler3 is handling the request.")
        else:
            print("ConcreteHandler3 cannot handle the request.")
            super().handle_request(request)
```

```python
# Client code
if __name__ == "__main__":
    handler1 = ConcreteHandler1()
    handler2 = ConcreteHandler2(handler1)
    handler3 = ConcreteHandler3(handler2)

    handler3.handle_request("Type1")  # Output: ConcreteHandler1 is handling
    handler3.handle_request("Type2")  # Output: ConcreteHandler2 is handling
    handler3.handle_request("Type3")  # Output: ConcreteHandler3 is handling
    handler3.handle_request("Type4")  # Output: Request cannot be handled.
```

# Command Design Pattern

The Command Pattern is a behavioral design pattern that encapsulates a request as an object, thereby allowing you to parameterize clients with queues, requests, and operations. This pattern allows the separation of concerns between the sender (client) of a request and the receiver (handler) of the request.



```python
# Command interface
class Command:
    def execute(self):
        pass

# Concrete command classes
class LightOnCommand(Command):
    def __init__(self, light):
        self.light = light

    def execute(self):
        self.light.turn_on()

class LightOffCommand(Command):
    def __init__(self, light):
        self.light = light

    def execute(self):
        self.light.turn_off()
```

```python
# Receiver
class Light:
    def turn_on(self):
        print("Light is on")

    def turn_off(self):
        print("Light is off")

# Invoker
class RemoteControl:
    def __init__(self):
        self.command = None

    def set_command(self, command):
        self.command = command

    def press_button(self):
        self.command.execute()
```

```python
# Client
if __name__ == "__main__":
    light = Light()
    turn_on = LightOnCommand(light)
    turn_off = LightOffCommand(light)

    remote = RemoteControl()
    remote.set_command(turn_on)  # Client selects turn on command
    remote.press_button()  # Invoker executes the command

    remote.set_command(turn_off)  # Client selects turn off command
    remote.press_button()  # Invoker executes the command
```

# Architectural Patterns

- **Architectural design:** **It is the critical link between design and requirements engineering. It identifies the main structural components in a system and the relationships between them.**

## Examples:

**Model-View-Controller (MVC)**

**Model-View-ViewModel (MVVM)**

**Model-View-Presenter (MVP)**

1. **Model-View-Controller (MVC)**:
   - **Model**: Represents the data and business logic of the application. It interacts with the database and processes data.
   - **View**: Presents the user interface. It displays data to the user and sends user inputs to the controller.
   - **Controller**: Handles user inputs, processes them, and updates the model accordingly. It also updates the view with the changes in the model.
   - In Python, frameworks like Django and Flask can be used to implement MVC architecture for web applications.

Example :

```
from flask import Flask, render_template, request

app = Flask(__name__)

class Model:
    def __init__(self):
        self.data = []

    def add_data(self, item):
        self.data.append(item)

class View:
    @staticmethod
    def show(data):
        return render_template('index.html', data=data)
```

```
class Controller:
    def __init__(self, model, view):
        self.model = model
        self.view = view

    def add_item(self, item):
        self.model.add_data(item)
        return self.view.show(self.model.data)

model = Model()
view = View()
controller = Controller(model, view)

@app.route('/', methods=['GET', 'POST'])
def index():
    if request.method == 'POST':
        item = request.form['item']
        return controller.add_item(item)
    else:
        return controller.view.show(model.data)

if __name__ == '__main__':
    app.run(debug=True)
```

2. **Model-View-View Model (MVVM)**:
    - **Model**: Represents the data and business logic of the application, similar to MVC.
    - **View**: Presents the user interface. However, in MVVM, the view is more passive and binds directly to the ViewModel.
    - **ViewModel:** Acts as an intermediary between the view and the model. It exposes data and commands from the model to the view and handles user interactions.
    - Libraries like Kivy and frameworks like PyQT support MVVM architecture for GUI applications in Python

```python
from kivy.app import App
from kivy.uix.boxlayout import BoxLayout
from kivy.properties import StringProperty, ObjectProperty

class Model:
    def __init__(self):
        self.data = []

    def add_data(self, item):
        self.data.append(item)

class ViewModel:
    def __init__(self, model):
        self.model = model
        self.data = StringProperty('')

    def add_item(self, item):
        self.model.add_data(item)
        self.data = ', '.join(self.model.data)
```

```
class View(BoxLayout):
    data_label = ObjectProperty(None)
    item_input = ObjectProperty(None)

    def __init__(self, **kwargs):
        super(View, self).__init__(**kwargs)
        self.model = Model()
        self.viewmodel = ViewModel(self.model)
        self.data_label.text = self.viewmodel.data

    def add_item(self):
        item = self.item_input.text
        if item:
            self.viewmodel.add_item(item)
            self.data_label.text = self.viewmodel.data
            self.item_input.text = ''

class MVVMApp(App):
    def build(self):
        return View()

if __name__ == '__main__':
    MVVMApp().run()
```

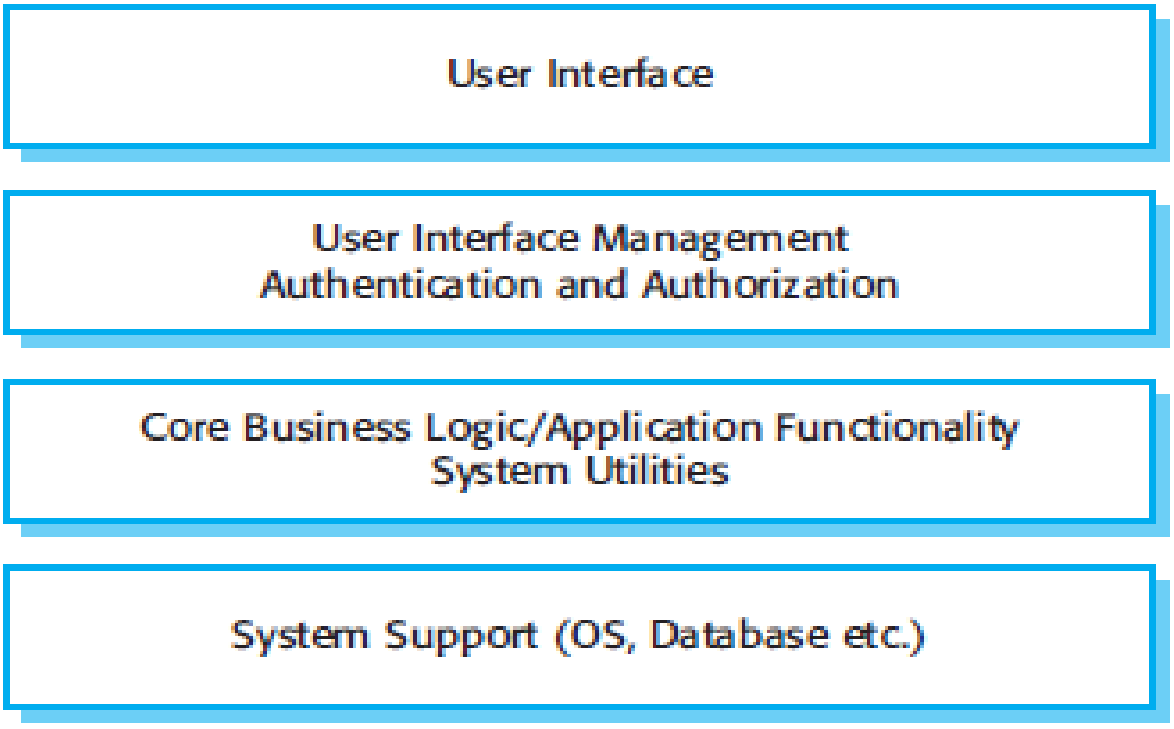
3. **Model-View-Presenter (MVP)**:
    - **Model**: Represents the data and business logic, just like in MVC and MVVM.
    - **View**: Presents the user interface. In MVP, the view is passive and delegates user interactions to the presenter.
    - **Presenter**: Acts as an intermediary between the view and the model. It handles user inputs from the view, updates the model accordingly, and updates the view with changes in the model.
    - MVP is commonly used in GUI applications where the view is kept as simple as possible, and the presenter contains most of the application logic.
    - While not as prevalent in Python as MVC, libraries like PyQT and frameworks like MVPython can be used to implement MVP architecture.

```
from mvpython import View, Model, Presenter

class MyModel(Model):
    def __init__(self):
        self.data = []

    def add_data(self, item):
        self.data.append(item)

class MyView(View):
    def __init__(self, presenter):
        super().__init__(presenter)
        self.presenter = presenter

    def show_data(self, data):
        print("Data:", data)
```

```
class MyPresenter(Presenter):
    def __init__(self):
        self.model = MyModel()
        self.view = MyView(self)

    def add_item(self, item):
        self.model.add_data(item)
        self.view.show_data(self.model.data)

if __name__ == "__main__":
    presenter = MyPresenter()
    presenter.add_item("Item 1")
```

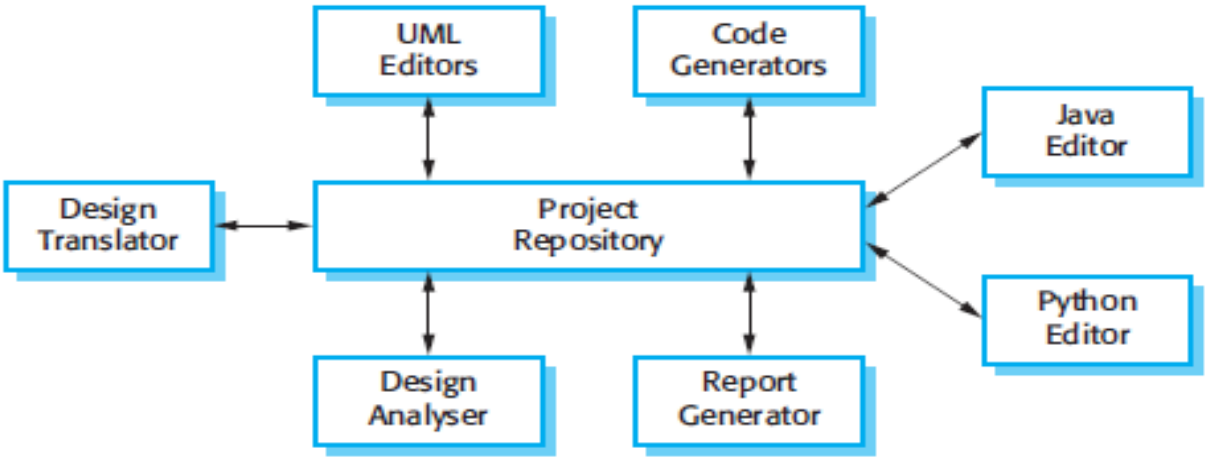
# 4 - Layered Architecture

| | |
|---|---|
| **Name** | Layered Architecture |
| **Description** | Organizes the system into layers with related functionality associated with each layer.<br>A layer provides services to the layer above it so the lowest-level layers represent core services that are likely to be used throughout the system. |
| **When used** | Used when:<br>1. Building new facilities on top of existing systems.<br>2. When the development is spread across several teams with each team responsibility for a layer of functionality. |
| **Advantages** | Allows replacement of entire layers so long as the interface is maintained. |
| **Disadvantages** | In practice, providing a clean separation between layers is often difficult and a high-level layer may have to interact directly with lower-level layers rather than through the layer immediately below it. |



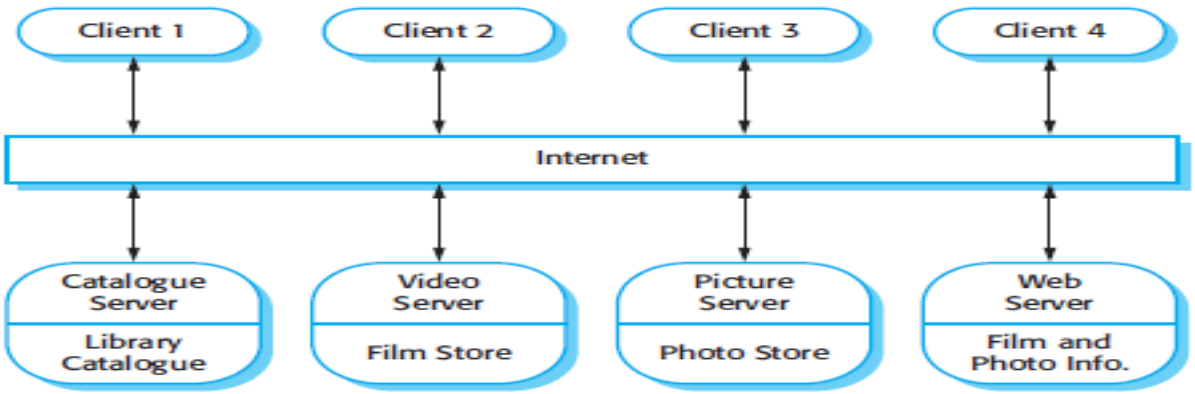
# 5- Repository Architecture

| Name | Repository |
|---|---|
| **Description** | All data in a system is managed in a central repository that is accessible to all system components. Components do not interact directly, only through the repository. |
| **When used** | You should use this pattern when you have a system in which large volumes of information are generated that has to be stored for a long time. |
| **Advantages** | Components can be independent—they do not need to know of the existence of other components. Changes made by one component can be propagated to all components. All data can be managed consistently (e.g., backups done at the same time) as it is all in one place. |
| **Disadvantages** | The repository is a **single point of failure** so problems in the repository affect the whole system. May be **inefficiencies** in organizing all **communication** through the repository. |

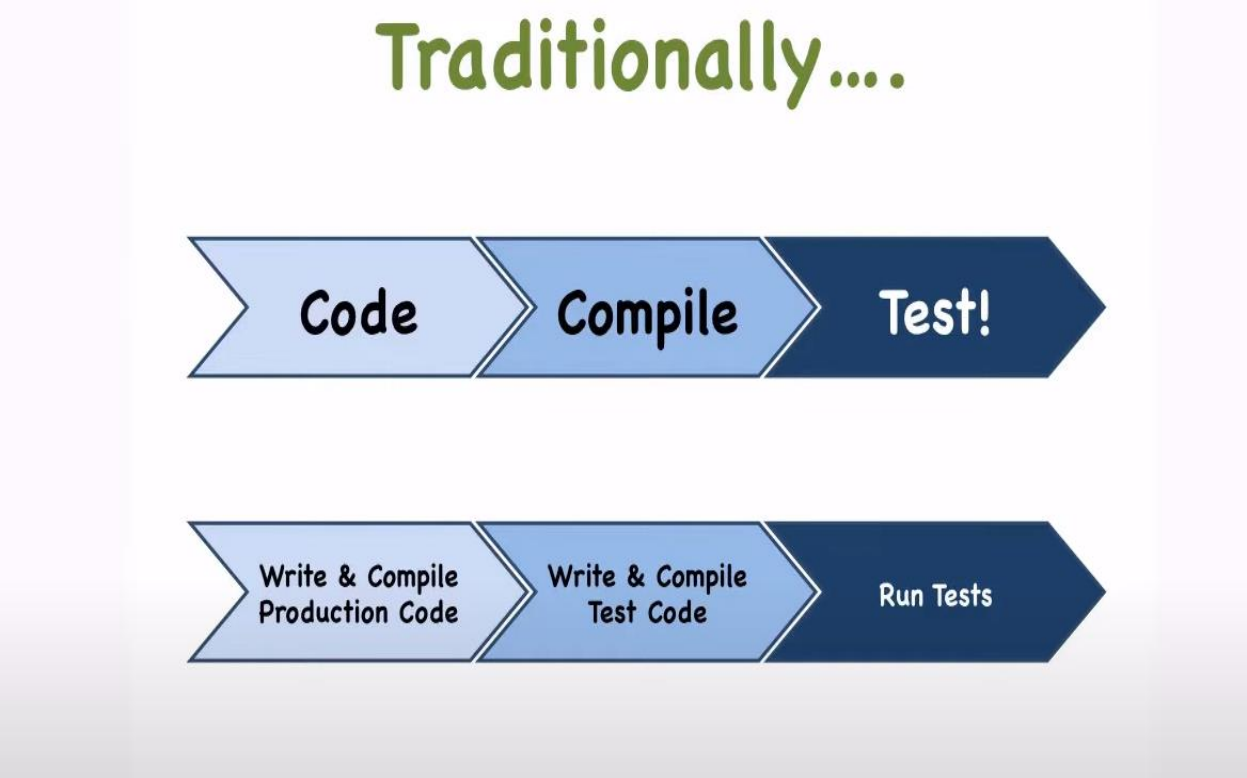


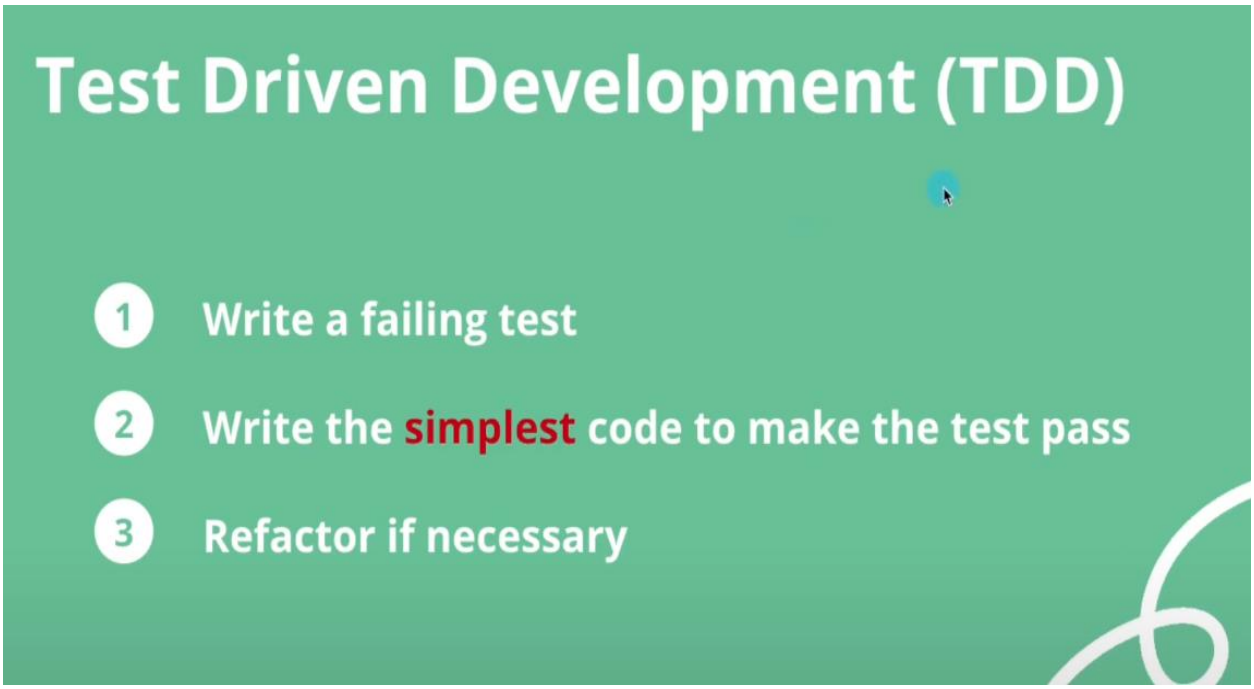
# 6- Client–Server Architecture

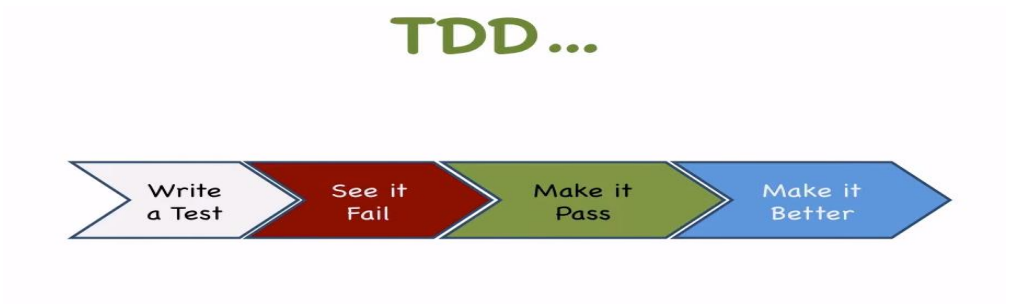
| Name | Client-server |
|---|---|
| Description | In client–server architecture, the functionality of the system is organized into services, with each service delivered from a separate server. Clients are users of these services and access servers to make use of them. |
| When used | 1. Used when data in a shared database has to be accessed from a range of locations.<br>2. Because servers can be replicated, may also be used when the load on a system is variable. |
| Advantages | The principal advantage of this model is that servers can be distributed across a network. |
| Disadvantages | 1- Each service is a single point of failure.<br>2- Performance may be unpredictable because it depends on the network.<br>3- May be management problems if servers are owned by different organizations. |



# Test driven development






## How to get started?

To begin writing tests in Python we will use the `unittest` module that comes with Python. To do this we create a new file `mytests.py`, which will contain all our tests.

Let's begin with the usual "hello world":

```
import unittest
from mycode import *

class MyFirstTests(unittest.TestCase):

def test_hello(self):
        self.assertEqual(hello_world(), 'hello world')
```

```
def hello_world():
    pass
```

**Running `python mytests.py` will generate the following output in the command line:**

```
F
======================================================================
FAIL: test_hello (__main__.MyFirstTests)
----------------------------------------------------------------------
Traceback (most recent call last):
File "mytests.py", line 7, in test_hello
self.assertEqual(hello_world(), 'hello world')
AssertionError: None != 'hello world'
----------------------------------------------------------------------
Ran 1 test in 0.000s
FAILED (failures=1)
```

```
def hello_world():
    return 'hello world'
```

Running `python mytests.py` again we get the following output in the command line:

```
.

----------------------------------------------------------------------

Ran 1 test in 0.000s

OK
```

in the file `mytests.py` this would be a method `test_custom_num_list`:

```
import unittest
from mycode import *

class MyFirstTests(unittest.TestCase):

def test_hello(self):
        self.assertEqual(hello_world(), 'hello world')

    def test_custom_num_list(self):
        self.assertEqual(len(create_num_list(10)), 10)
```

This would test that the function `create_num_list` returns a list of length 10. Let's create function `create_num_list` in `mycode.py`:

```
def hello_world():
    return 'hello world'

def create_num_list(length):
    pass
```

Running `python mytests.py` will generate the following output in the command line:

```
E.
======================================================================
ERROR: test_custom_num_list (__main__.MyFirstTests)
----------------------------------------------------------------------
Traceback (most recent call last):
File "mytests.py", line 14, in test_custom_num_list
self.assertEqual(len(create_num_list(10)), 10)
TypeError: object of type 'NoneType' has no len()
----------------------------------------------------------------------
Ran 2 tests in 0.000s
FAILED (errors=1)
```

This is as expected, so let's go ahead and change function `create_num_list` in `mytest.py` in order to pass the test:

```
def hello_world():
    return 'hello world'

def create_num_list(length):
    return [x for x in range(length)]
```

Executing `python mytests.py` on the command line demonstrates that the second test has also now passed:

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s
OK
```