JOMO KENYATTA UNIVERSITY OF AGRICULTURE AND TECHNOLOGY.

M.Sc. SOFTWARE ENGINEERING

ICS 3105: - OBJECT ORIENTED SOFTWARE ENGINEERING.

ASSIGNMENT 2

Dr Eunice Njeri.

ALEX KEMBOI SCT 313-0530/2023 OBJECT ORIENTED SOFTWARE ENGINEERING.

A Research proposal submitted to Jomo Kenyatta University of Agriculture and Technology, School of Computing. Partial fulfillment of the requirement for the award of degree of Masters of Science in Software Engineering.

2025

Design Patterns

Design patterns are reusable solutions to common software design problems. They help make code more **maintainable, scalable, and flexible**.

Types of Design Patterns

Design patterns are categorized into **three main types**:

1. **Creational Patterns** – Handle object creation.
2. **Structural Patterns** – Organize and structure code.
3. **Behavioral Patterns** – Manage communication between objects.

---

1. Creational Patterns

Creational patterns focus on the best ways to create objects.

1.1 Singleton Pattern

Ensures that only one instance of a class is created.

Example (Singleton in JavaScript)

```
class Singleton {
    constructor() {
        if (!Singleton.instance) {
            Singleton.instance = this;
        }
        return Singleton.instance;
    }
}

const obj1 = new Singleton();
const obj2 = new Singleton();
console.log(obj1 === obj2); // true (Both refer to the same instance)
```

1.2 Factory Pattern

Encapsulates object creation logic in a separate function.

Example (Factory Pattern in JavaScript)

```
class Car {
    constructor(brand) {
        this.brand = brand;
    }
    drive() {
        console.log(`Driving a ${this.brand}`);
    }
}

class CarFactory {
    createCar(brand) {
        return new Car(brand);
    }
}

// Usage
const factory = new CarFactory();
const car1 = factory.createCar("Toyota");
car1.drive(); // Driving a Toyota
```

1.3 Builder Pattern

Separates object construction from representation, allowing step-by-step object creation.

Example (Builder Pattern in JavaScript)

```
class Car {
    constructor() {
        this.brand = "";
        this.color = "";
    }
    setBrand(brand) {
        this.brand = brand;
        return this;
    }
    setColor(color) {
        this.color = color;
        return this;
    }
    build() {
        return this;
    }
}

// Usage
const car = new Car().setBrand("Tesla").setColor("Red").build();
console.log(car); // { brand: 'Tesla', color: 'Red' }
```

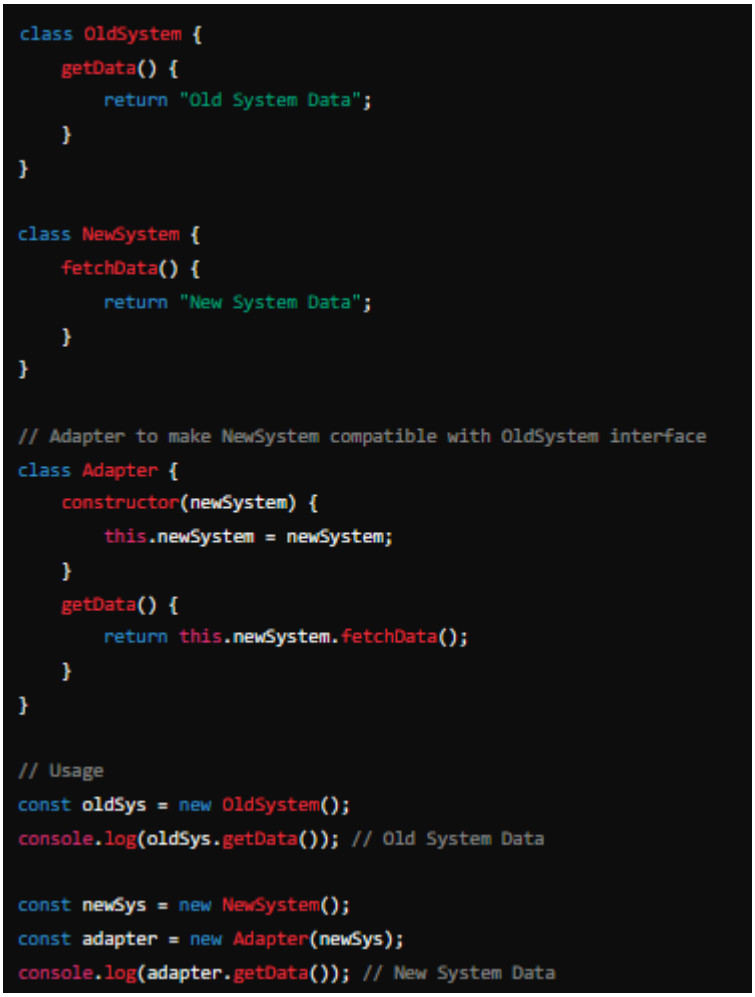
2. Structural Patterns

Structural patterns deal with organizing classes and objects to form larger structures.

2.1 Adapter Pattern

Converts one interface into another expected by the client.

Example (Adapter Pattern in JavaScript)

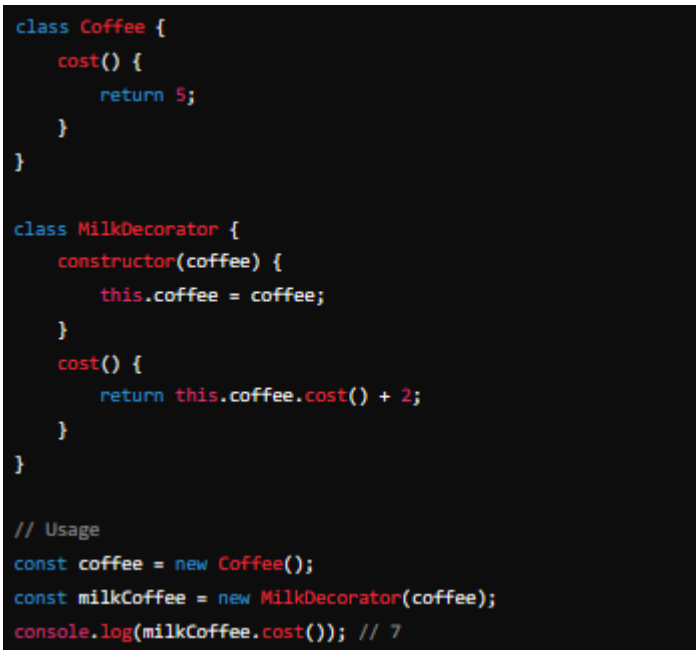


```
class OldSystem {
    getData() {
        return "Old System Data";
    }
}

class NewSystem {
    fetchData() {
        return "New System Data";
    }
}

// Adapter to make NewSystem compatible with OldSystem interface
class Adapter {
    constructor(newSystem) {
        this.newSystem = newSystem;
    }
    getData() {
        return this.newSystem.fetchData();
    }
}

// Usage
const oldSys = new OldSystem();
console.log(oldSys.getData()); // Old System Data

const newSys = new NewSystem();
const adapter = new Adapter(newSys);
console.log(adapter.getData()); // New System Data
```

2.2 Decorator Pattern

Dynamically adds behavior to objects without modifying their structure.

Example (Decorator Pattern in JavaScript)



```
class Coffee {
    cost() {
        return 5;
    }
}

class MilkDecorator {
    constructor(coffee) {
        this.coffee = coffee;
    }
    cost() {
        return this.coffee.cost() + 2;
    }
}

// Usage
const coffee = new Coffee();
const milkCoffee = new MilkDecorator(coffee);
console.log(milkCoffee.cost()); // 7
```

2.3 Proxy Pattern

Controls access to an object, adding security or performance optimizations.

Example (Proxy Pattern in JavaScript)

```
class Server {
    request(url) {
        console.log(`Fetching data from ${url}`);
    }
}

class ProxyServer {
    constructor() {
        this.cache = {};
        this.server = new Server();
    }

    request(url) {
        if (!this.cache[url]) {
            this.cache[url] = `Cached data from ${url}`;
            this.server.request(url);
        }
        return this.cache[url];
    }
}

// Usage
const proxy = new ProxyServer();
console.log(proxy.request("api/data")); // Fetching data from api/data
console.log(proxy.request("api/data")); // Cached data from api/data
```

3. Behavioral Patterns

Behavioral patterns manage communication between objects.

3.1 Observer Pattern

Allows multiple objects to react to changes in another object.

Example (Observer Pattern in JavaScript)

```
class Subject {
    constructor() {
        this.observers = [];
    }
    subscribe(observer) {
        this.observers.push(observer);
    }
    unsubscribe(observer) {
        this.observers = this.observers.filter(obs => obs !== observer);
    }
    notify(data) {
        this.observers.forEach(observer => observer.update(data));
    }
}

class Observer {
    update(data) {
        console.log(`Received data: ${data}`);
    }
}

// Usage
const subject = new Subject();
const observer1 = new Observer();
const observer2 = new Observer();

subject.subscribe(observer1);
subject.subscribe(observer2);
subject.notify("Hello, Observers!"); // Both observers receive the update
```

3.2 Strategy Pattern

Defines a family of algorithms and makes them interchangeable.

Example (Strategy Pattern in JavaScript)

```
class PaymentStrategy {
    pay(amount) {
        throw new Error("Method must be implemented");
    }
}

class CreditCardPayment extends PaymentStrategy {
    pay(amount) {
        console.log(`Paid ${amount} using Credit Card`);
    }
}

class PayPalPayment extends PaymentStrategy {
    pay(amount) {
        console.log(`Paid ${amount} using PayPal`);
    }
}

class PaymentProcessor {
    constructor(strategy) {
        this.strategy = strategy;
    }
    processPayment(amount) {
        this.strategy.pay(amount);
    }
}

// Usage
const payment = new PaymentProcessor(new CreditCardPayment());
payment.processPayment(100); // Paid 100 using Credit Card
```

GIT HUB LINK

https://github.com/alexkemboi/Design-Patterns.git