Design Pattern

EnviroNet is a modern system for monitoring environmental conditions, designed to gather and analyze data from various sources in real time. Some of these data sources use outdated technology that cannot directly communicate with the system's newer components. However, their data is essential for accurate monitoring, so a solution is needed to bridge the gap between the old and the new systems without altering the original setup of the older components.

Another important feature of EnviroNet is its ability to notify users about critical environmental events, such as dangerous air pollution levels. Users should be able to sign up to receive alerts and stop receiving them whenever they wish. The system must handle these notifications efficiently, even as the number of users changes over time.

As a software engineer, you are tasked with solving these challenges. First, you must design a solution that allows the outdated components to work seamlessly with the modern system. Second, you need to build a notification system that can manage user subscriptions and send alerts in real time.

1. Identify and justify the design pattern for integrating the outdated components with the modern system. Explain how your chosen pattern solves the problem and its benefits.
2. Propose and justify a design pattern for the notification system that handles user subscriptions and alerts. Explain how your chosen pattern solves the problem and its benefits.
3. Write pseudocode to demonstrate the integration of the outdated components and the implementation of the notification system with subscription and alert features.

Answer:
1.

The adapter pattern is a good choice for connecting outdated components to modern systems. It works by converting the interface of the outdated component (Adaptee) to match what the modern system (Target) expects. This allows the two systems to work together without changing the original design of the outdated components.

The adapter pattern solves the problem by utilizing the following characteristics:

- The adapter acts as a bridge between the incompatible systems, enabling smooth interaction.
- There's no need to modify the older components or the modern system in order solve the issue.
- It's easy to add new adapters for other outdated components.

The benefits of the adapter pattern are:

- Existing outdated components can be used with the modern system.
- It can connect new components with different interfaces as well.

2.

The observer pattern is useful for handling real-time notifications. It allows multiple users (Observers) to subscribe or unsubscribe to alerts (Subject). When the Subject detects a change, it automatically notifies all subscribed users.

The observer pattern solves the problem by utilizing the following characteristics:

- Users can join or leave the system without affecting its core.
- Ensures users are notified promptly when something changes.
- The Subject and Observers can be updated independently.

The benefits of the observer pattern are:

- Efficiency: Notifications are sent to all subscribers in one go.
- Scalability: It can handle a growing number of users without issue.

3.

Pseudocode for the adapter pattern:

```
class ModernSystemInterface:
    def process_data(self):
        pass

class OutdatedComponent:
    def fetch_data(self):
        return "Outdated data format"

class OutdatedAdapter(ModernSystemInterface):
    def __init__(self, outdated_component):
        self.outdated_component = outdated_component

    def process_data(self):
        data = self.outdated_component.fetch_data()
        return f"Processed {data}"

outdated = OutdatedComponent()
adapter = OutdatedAdapter(outdated)
print(adapter.process_data())
```

Pseudocode for the observer pattern:

```
class User:
    def update(self, message):
        pass

class Subscriber(User):
    def __init__(self, name):
        self.name = name

    def update(self, message):
        print(f"{self.name} received: {message}")

class EnviroNet:
    def __init__(self):
        self.subscribers = []

    def subscribe(self, user):
        self.subscribers.append(user)

    def unsubscribe(self, user):
        self.subscribers.remove(user)

    def notify(self, message):
        for subscriber in self.subscribers:
            subscriber.update(message)

enviro_net = EnviroNet()
user1 = Subscriber("Alice")
user2 = Subscriber("Bob")

enviro_net.subscribe(user1)
enviro_net.subscribe(user2)

enviro_net.notify("Warning: High pollution levels!")
enviro_net.unsubscribe(user1)
enviro_net.notify("Update: Pollution levels normalized.")
```

Diversity is a next-generation game engine designed for creating large-scale multiplayer online games. The engine includes a core system responsible for managing global game settings, such as graphics quality, audio preferences, and server configurations. These settings must remain consistent throughout the game and be easily accessible to all modules. To prevent conflicts and ensure stability, only one instance of the settings manager should exist at any time.

The settings manager must also provide a simple way for developers to access and modify settings without reinitializing or duplicating it across the system.

You have been assigned to design the global settings manager for Diversity.

1. Identify a suitable design pattern for implementing this functionality. Justify your choice with clear reasoning.
2. Describe the key components involved in implementing this design pattern for the settings manager.
3. Write pseudocode to implement the settings manager, ensuring that only one instance is created and shared.

Answer:

The Singleton pattern is the best choice for implementing the global settings manager in Diversity. This pattern ensures only one instance of the settings manager exists and provides a global access point.

The Singleton pattern is suitable because:

1. It guarantees a single instance of the settings manager, ensuring consistency across the system.
2. It provides global access, making it easy for different modules to use the settings manager without needing to pass it explicitly.
3. It centralizes control of global settings, preventing conflicts and ensuring system stability.
4. It avoids redundant instances, improving efficiency and saving resources.

The main components for implementing the pattern are:

1. A Singleton class to manage the single instance of the settings manager.
2. A private constructor to prevent direct creation of multiple instances.
3. A static variable to store the single instance.
4. Some method to provide controlled access to the instance.

Pseudocode for the settings manager:

```
class SettingsManager:
    _instance = None  # Static variable to hold the single instance

    def __new__(cls, *args, **kwargs):
        if cls._instance is None:
            print("Creating the SettingsManager instance")
            cls._instance = super(SettingsManager, cls).__new__(cls, *args, **kwargs)
            # Initialize attributes directly in __new__
            cls._instance.graphics_quality = "High"
            cls._instance.audio_preferences = "Surround"
            cls._instance.server_configurations = {"region": "US-East"}
        return cls._instance
```

Refactoring

Python

```
class Order:
    def __init__(self, order_amount):
        self.order_amount = order_amount

    def calculate_discount(self, customer_type, is_first_time_buyer, location):
        order_amount = self.order_amount
        if customer_type == "VIP" and is_first_time_buyer and location == "Local":
            discount = order_amount * 0.20
        elif customer_type == "Regular" and location == "Local":
            discount = order_amount * 0.10
        else:
            discount = 0

        # Apply tax
        tax = order_amount * 0.05
        final_amount = order_amount - discount + tax
        return final_amount

class PaymentProcessor:
    def __init__(self, order):
        self.order = order

    def process_payment(self, payment_method):
        amount_due = self.order.calculate_discount()
        print(f"Processing payment of {amount_due} using {payment_method}.")
        print("Payment processed successfully.")
```

Java

```
class Order {
    public double order_amount;

    public Order(double order_amount) {
        this.order_amount = order_amount;
    }

    public double calculate_discount(String customer_type, boolean is_first_time_buyer, String location)
{
        double order_amount = this.order_amount;
        double discount;
        if (customer_type.equals("VIP") && is_first_time_buyer && location.equals("Local")) {
            discount = order_amount * 0.20;
        } else if (customer_type.equals("Regular") && location.equals("Local")) {
            discount = order_amount * 0.10;
        } else {
            discount = 0;
        }

        // Apply tax
        double tax = order_amount * 0.05;
```

```
        double final_amount = order_amount - discount + tax;
        return final_amount;
    }
}

class PaymentProcessor {
    private Order order;

    public PaymentProcessor(Order order) {
        this.order = order;
    }

    public void process_payment(String payment_method) {
        double amount_due = order.calculate_discount();
        System.out.println("Processing payment of " + amount_due + " using " + payment_method + ".");
        System.out.println("Payment processed successfully.");
    }
}
```

1. Identify the code smells in the above implementation. Explain what problem each of them indicate.
2. Suggest remedies to address the identified smells and explain how these changes improve the code.
3. Refactor the code to resolve all identified issues. Write only the corrected code.

Answer:

Long Method

The calculate_discount method is lengthy and contains a complex conditional structure.

Problem: It is harder to understand and maintain, as the logic is buried within a single block of code. This increases the chance of errors when modifications are required.
Remedy: Extract Method and Decompose Conditional to simplify the logic and make each part self-explanatory.

Long Parameter List

The calculate_discount method takes multiple parameters (customer_type, is_first_time_buyer, location).

Problem: It makes the method hard to use and prone to errors due to the need to pass multiple values explicitly, which reduces code readability and usability.
Remedy: Introduce Parameter Object to group related parameters into a single object, improving method clarity.

Feature Envy

The calculate_discount method accesses multiple attributes of Order and external inputs without encapsulating the logic.

Problem: It creates tight coupling by scattering logic across classes, making it harder to maintain and test the code.
Remedy: Move Method to the class where the data resides, aligning behavior with the relevant data.

Inappropriate Naming
Class Order and method calculate_discount lack specificity about their purpose.

Problem: It reduces readability and makes it harder for developers to understand their roles at a glance.
Remedy: Rename for clarity to ensure names convey the intention and functionality explicitly.

Refactored Code:

```
class CustomerInfo:
    def __init__(self, customer_type, is_first_time_buyer, location):
        self.customer_type = customer_type
        self.is_first_time_buyer = is_first_time_buyer
        self.location = location

class Order:
    def __init__(self, amount, customer_info):
        self.amount = amount
        self.customer_info = customer_info

    def calculate_amount_after_discount(self):
        discount = self._determine_discount()
        tax = self._determine_tax()
        return self.amount - discount + tax

    def _determine_discount(self):
        if self._is_vip_local_first_time():
            return self.amount * 0.20
        elif self._is_regular_local():
            return self.amount * 0.10
        return 0

    def _determine_tax(self):
        return self.amount * 0.05

    def _is_vip_local_first_time(self):
        return (self.customer_info.customer_type == "VIP" and
                self.customer_info.is_first_time_buyer and
                self.customer_info.location == "Local")

    def _is_regular_local(self):
        return (self.customer_info.customer_type == "Regular" and
                self.customer_info.location == "Local")
```

```
def process_payment(self, payment_method):
    amount_due = self.calculate_amount_after_discount()
    print(f"Processing payment of {amount_due} using {payment_method}.")
    print("Payment processed successfully.")
```

Python

```python
class ReportGenerator:
    def __init__(self, data):
        self.d = data

    def generate_summary(self):
        # This method generates a summary of the data
        total = 0
        for item in self.d:
            total += item
        average = total / len(self.d)
        print(f"Total: {total}")
        print(f"Average: {average}")

    def generate_detailed_report(self):
        # This method generates a detailed report of the data
        total = 0
        for item in self.d:
            total += item
            print(f"Data Point: {item}")
        average = total / len(self.d)
        print(f"Detailed Total: {total}")
        print(f"Detailed Average: {average}")
```

Java

```java
class ReportGenerator {
    private int[] d;

    public ReportGenerator(int[] data) {
        this.d = data;
    }

    public void generateSummary() {
        // This method generates a summary of the data
        int total = 0;
        for (int item : d) {
            total += item;
        }
        double average = (double) total / d.length;
        System.out.println("Total: " + total);
        System.out.println("Average: " + average);
    }

    public void generateDetailedReport() {
        // This method generates a detailed report of the data
        int total = 0;
        for (int item : d) {
            total += item;
            System.out.println("Data Point: " + item);
        }
```

```
        double average = (double) total / d.length;
        System.out.println("Detailed Total: " + total);
        System.out.println("Detailed Average: " + average);
    }
}
```

1. Identify the code smells in the above implementation. Explain what problem each of them indicate.
2. Suggest remedies to address the identified smells and explain how these changes improve the code.
3. Refactor the code to resolve all identified issues. Write only the corrected code.

Answer:

Duplicated Code

Both generate_summary and generate_detailed_report calculate the total and average in the same way.

Problem: Redundant logic leads to increased maintenance effort and a higher chance of bugs.
Remedy: Extract Method to centralize the calculation of total and average.

Inappropriate Naming

The attribute d is not descriptive.

Problem: Reduces code readability and understanding.
Remedy: Rename to data to clearly indicate its purpose.

Long Method

The methods include multiple responsibilities (calculating and printing).

Problem: Reduces clarity and makes methods harder to maintain.
Remedy: Extract Method to separate concerns of calculation and printing.

Refactored Code:

```
class ReportGenerator:
    def __init__(self, data):
        self.data = data

    def generate_summary(self):
        total, average = self._calculate_summary()
        self._print_summary(total, average)

    def generate_detailed_report(self):
        total, average = self._calculate_summary()
        self._print_detailed_report(total, average)
```

```
def _calculate_summary(self):
    total = sum(self.data)
    average = total / len(self.data)
    return total, average

def _print_summary(self, total, average):
    print(f"Total: {total}")
    print(f"Average: {average}")

def _print_detailed_report(self, total, average):
    for item in self.data:
        print(f"Data Point: {item}")
    print(f"Detailed Total: {total}")
    print(f"Detailed Average: {average}")
```

Testing (CFG, Cyclomatic Complexity, SIX)

Python

```
class User:
    def login(self):
        print("User logged in.")

    def logout(self):
        print("User logged out.")

    def view_profile(self):
        print("Viewing user profile.")

class Admin(User):
    def login(self):
        print("Admin logged in.")

    def manage_users(self):
        print("Admin is managing users.")

class SuperAdmin(Admin):
    def login(self):
        print("SuperAdmin logged in with elevated privileges.")

    def view_system_logs(self):
        print("Viewing system logs.")

    def manage_access(self, level):
        print("Managing access permissions.")
```

Java

```
class User {
    public void login() {
        System.out.println("User logged in.");
    }

    public void logout() {
        System.out.println("User logged out.");
    }

    public void viewProfile() {
        System.out.println("Viewing user profile.");
    }
}

class Admin extends User {
```

```
    @Override
    public void login() {
        System.out.println("Admin logged in.");
    }

    public void manageUsers() {
        System.out.println("Admin is managing users.");
    }
}

class SuperAdmin extends Admin {
    @Override
    public void login() {
        System.out.println("SuperAdmin logged in with elevated privileges.");
    }

    public void viewSystemLogs() {
        System.out.println("Viewing system logs.");
    }

    public void manageAccess(String level) {
        System.out.println("Managing access permissions.");
    }
}
```

1. Calculate the SIX of the SuperAdmin class in the above code.

Answer:
[You do not have to explicitly calculate all the values, I am doing this just to show you]

Depth of Inheritance (DIT):
The depth of inheritance for SuperAdmin is 2 (SuperAdmin -> Admin -> User).

Number of Methods Added (NMA):
The SuperAdmin class introduces 2 new methods: view_system_logs and manage_access.

Number of Inherited Methods (NMI):
It directly inherits 3 methods from Admin and User: logout, view_profile, and manage_users.

Number of Overridden Methods (NMO):
The SuperAdmin class overrides 1 method: login.

$$SIX = \frac{DIT \times NMO}{NMO + NMI + NMA} \times 100\%$$

$$SIX = \frac{2 \times 1}{1 + 3 + 2} \times 100\%$$

$$SIX = \frac{2}{6} \times 100\% = 33.33\%$$

Python

```python
class Vehicle:
    def start(self):
        print("Vehicle is starting.")

    def stop(self):
        print("Vehicle is stopping.")

class Car(Vehicle):
    def honk(self):
        print("Car is honking.")

    def open_trunk(self):
        print("Car trunk is opening.")

class SportsCar(Car):
    def start(self):
        print("Sports car is roaring to life.")

    def stop(self):
        print("Sports car is stopping quickly.")

    def honk(self):
        print("Sports car honks with a sharp sound.")

    def activate_sport_mode(self):
        print("Sport mode activated for enhanced performance.")
```

Java

```java
class Vehicle {
    void start() {
        System.out.println("Vehicle is starting.");
    }

    void stop() {
        System.out.println("Vehicle is stopping.");
    }
}

class Car extends Vehicle {
    void honk() {
        System.out.println("Car is honking.");
    }

    void openTrunk() {
        System.out.println("Car trunk is opening.");
    }
```

```
}

class SportsCar extends Car {
    void start() {
        System.out.println("Sports car is roaring to life.");
    }

    void stop() {
        System.out.println("Sports car is stopping quickly.");
    }

    void honk() {
        System.out.println("Sports car honks with a sharp sound.");
    }

    void activateSportMode() {
        System.out.println("Sport mode activated for enhanced performance.");
    }
}
```

2. Calculate the SIX of the SportsCar class in the above code.

Answer:
Depth of Inheritance (DIT): 2 (SportsCar -> Car -> Vehicle).
Number of Methods Added (NMA): 1 (activate_sport_mode).
Number of Inherited Methods (NMI): 1 (open_trunk).
Number of Overridden Methods (NMO): 3 (start, stop, honk).

$$SIX = \frac{DIT \times NMO}{NMO + NMI + NMA} \times 100\%$$

$$SIX = \frac{2 \times 3}{3 + 1 + 1} \times 100\%$$

$$SIX = \frac{6}{5} \times 100\% = 120\%$$