**Selected Labs in software engineering Project**

**Team Members:**

1. Mohamed Ashraf Osman :2101176
2. Seif El-Din Salah :2101581
3. Asmaa Hamdy Abd-Elaleem:2100524
4. Maria Emil:2100971
5. Mennaallah Ahmed:2101082

**Project Name: Weather Monitoring System**

**Overview**

The Weather Monitoring System is a console-based Java application designed to track and display weather data such as temperature, humidity, and wind speed. Additionally, it generates weather alerts for specific conditions like storms, snow, and heatwaves. The system uses the Singleton Design Pattern , Factory Design Pattern ,Observer Design Pattern , Decorator Design Pattern , and Strategy Design Pattern

**1-Singleton Design Pattern:** ensures that a class has only one instance and provides a global point of access to that instance. This is achieved by:

1-Making the constructor private to prevent external instantiation.

2-Creating a static method to return the single instance.

3-Optionally ensuring thread-safety in multi-threaded environments

**Class Descriptions**

**1.UserPreferencesManager:**

**Purpose**: Manages user settings like temperature and wind speed units.

**Why**: Ensures consistent and globally accessible user preferences across the system**.**

**2-WeatherDataFetcher:**

**Purpose**: Retrieves weather data for a city and displays it according to user preferences**.**

**Why**: Leverages the Singleton instance of UserPreferencesManager for consistent settings**.**

**Problems Solved by Singleton**

**1-Duplicate Instances:** Eliminates redundant instances, ensuring only one instance exists**.**

**2-Global State Management**: Provides a centralized way to manage global application state or settings**.**

**3-Thread-Safety:** When implemented properly, it ensures thread-safe access to the single instance.

**4-Performance:** Reduces the overhead of repeatedly creating and destroying instances

**2-** **Factory Design Pattern:** : is a creational design pattern that provides a way to create objects without specifying the exact class of the object that will be created. This pattern defines an interface for creating objects but allows subclasses to alter the type of objects that will be created

In the Weather Monitoring System, the Factory Pattern is applied to:

1. **Create weather data objects** (e.g., Temperature, Humidity, Wind Speed) based on the type of weather data.
2. **Generate weather alerts** (e.g., Storm, Snow, Heatwave) for specific conditions

**Class Descriptions**

**1. Weather Data Factory**

**Class: WeatherDataFactory**

* **Description**: Creates weather data objects based on the specified type

**Product Abstract Class: WeatherData**

* **Description**: Abstract base class for all weather data types.

**Concrete Products:**

1. **Temperature**:
   * Represents temperature data.

2.**Humidity**:

* Represents humidity data.

3.**WindSpeed**:

* Represents wind speed data.

**2.Weather Alert Factory**

**Class: WeatherAlertFactory**

* **Description**: Generates weather alert objects based on the specified condition.

**Product Abstract Class: WeatherAlert**

* **Description**: Abstract base class for all weather alerts.

**Concrete Products:**

1. **StormAlert**:
   * Represents a storm alert.

2.  **SnowAlert**:

* Represents a snow alert.

3.  **HeatwaveAlert**:

* Represents a heatwave alert

**Advantages of Using the Factory Pattern**

1. **Encapsulation**: The logic for object creation is centralized in the factory class, simplifying code maintenance.
2. **Flexibility**: New types of products can be added without modifying existing code.

**Reusability**: The factory can be reused to create different objects based on input parameters.

**3**- **Observer Design Pattern:** defines a one-to-many relationship between objects. A subject (or publisher) maintains a list of dependents (subscribers or observers) that need to be notified of any changes to the subject's state. This pattern allows a flexible and decoupled way of passing updates from the subject to its observers

**In this implementation, the Observer pattern is applied to a weather monitoring system where:**

* WeatherData is the subject (publisher).
* Observers like CurrentConditionDisplay and WeatherAlerts are notified of changes to weather conditions.

**Problem Scenario**

imagine customers (observers) interested in updates from a store (publisher) about a new product release. Instead of customers frequently checking the store, the store notifies all registered customers when the product becomes available.

This analogy is reflected in the weather system:

* Observers like displays and alert systems are notified automatically when the weather changes.
* This avoids unnecessary polling and keeps the system efficient.

**Class Descriptions**

**1. Subject Interface**

**Responsibilities:**

* Acts as the base interface for the subject (publisher).
* Maintains methods for registering, unregistering, and notifying observers.

**2. WeatherData Class (Subject)**

**Responsibilities:**

* Implements the Subject interface.
* Stores a list of observers.
* Updates observers whenever the weather data changes.
* Provides a method setWeatherData to set and update weather conditions.

**3. Observer Interface**

**Responsibilities:**

* + Acts as the base interface for observers.
  + Defines the update method that is called when the subject notifies observers.

**4. CurrentConditionDisplay Class (Observer)**

**Responsibilities:**

* Implements the Observer interface.
* Receives weather updates and displays the current conditions.

**5. WeatherAlerts Class (Observer)**

**Responsibilities:**

* Implements the Observer interface.
* Triggers specific alerts based on the weather conditions.

**4- Decorator Design Pattern:** allows behavior to be added to individual objects dynamically, without affecting the behavior of other objects from the same class.

**Class Descriptions**

**1- WeatherData Interface:**

* Defines the blueprint for weather data objects.
* Includes methods getDescription() and getValue().

**2- BasicWeatherData Class**:

* Implements the WeatherData interface.
* Represents a simple weather data object with a numeric value (e.g., temperature).

**3- WeatherDecorator Abstract Class**:

* Implements the WeatherData interface.
* Serves as a base for all decorators.
* Contains a reference to a WeatherData object (weatherData), allowing decoration of existing objects.

**4-TemperatureDecorator Class**:

* Extends WeatherDecorator to add specific functionality.
* Appends "Temperature" to the description and provides a formatted string representation of the temperature value (e.g., "23.5°C").

**6-Strategy Design Pattern:** is a behavioral pattern that allows an object to change its behavior dynamically based on the selected strategy. Instead of embedding the logic within the object, the algorithm (or behavior) is encapsulated in a strategy class, which can be swapped easily at runtime

- This pattern is useful when multiple algorithms exist for a task and the client class needs to choose which one to use, or when you want to allow clients to choose different algorithms dynamically

**Class Descriptions**

**1*.* WeatherData Class**

* Purpose: Represents the core data (weather information) that will be displayed. It holds values like temperature, humidity, and forecast.

Explanation:

* The WeatherData class is responsible for storing the weather information: temperature, humidity, and forecast. It provides getter methods to retrieve these values, making the class simple and focused on data storage.
* This class does not perform any display logic but is used by other classes (like display strategies) to access the weather information**.**

**2. WeatherDisplayStrategy Interface**

* Purpose: Defines a contract for all display strategies to implement. Each concrete strategy will have its own way of displaying weather data.

Explanation:

* The WeatherDisplayStrategy interface is crucial because it provides a common method signature (displayWeatherData) for all display strategies, ensuring that each display method behaves in a consistent way.
* Any class that implements this interface is guaranteed to be compatible with the WeatherDisplayContext class, which will use the strategy to display weather data.
* By following this interface, new strategies can be added easily without modifying the existing system**.**

**3- *Concrete Strategy Classes***

These are the classes that implement the WeatherDisplayStrategy interface and define specific ways of displaying the weather data.

1-TabularDisplay Class

* Purpose: Displays weather data in a tabular format.

**Explanation:**

* The TabularDisplay class takes the weather data and formats it into a structured table. This makes it easy to read and interpret, especially in console-based applications.
* It is one of the possible ways of presenting the weather data, and the strategy ensures the flexibility of switching display methods.

2-GraphicalDisplay Class

* Purpose: Displays weather data using a simple graphical representation, such as bars or visual indicators.

**Explanation:**

* The GraphicalDisplay class shows the data in a more visual way, using symbols like blocks () to represent the values. This can help users quickly understand the magnitude of values like temperature or humidity at a glance.
* This strategy makes the system more interactive and can be especially useful in applications where visual representation is key (e.g., dashboards, mobile apps).

3-TextualDisplay Class

* Purpose: Displays weather data in a simple, easy-to-read text format.

**Explanation:**

* The TextualDisplay class outputs the weather data in a clean, non-visual format. It's straightforward and provides all the necessary information in plain text.
* This is useful for scenarios where the goal is clarity without needing complex formatting or visual elements

**4. WeatherDisplayContext Class**

* **Purpose**: Acts as the context that manages the current display strategy. It allows for the dynamic switching between different display strategies.

**Explanation**:

* The WeatherDisplayContext class holds a reference to the current strategy, which dictates how the weather data will be displayed. It provides methods to **set** the strategy and **display** the data using the current strategy.
* This class is essential because it allows you to **change** the display method at runtime. For example, you might switch from a tabular view to a graphical one depending on user preference or system conditions.

The separation of concerns here is vital, as the WeatherDisplayContext does not deal with how the data is displayed. It only knows that it needs to use the correct strategy to display the data