**Software Design Patterns**

**Final**

Name of our final project: WeatherApp

Group: SE-2216

Team members: Azamat Yerezhepbayev, Abylay Amandykov, Merei Isenbai

**Project Overview:**

1. Project Information: The WeatherApp is a Java-based project, that uses 6 design patterns to create a flexible and modular weather management system.
2. The main idea of the project: The WeatherApp aims to provide a comprehensive weather management system, allowing users to observe and update weather conditions, receive notifications, and convert temperatures. The application is designed with a user-friendly command line interface.
3. The purpose of the work: The purpose of the WeatherApp is to demonstrate the practical application of design patterns in software development. By putting these patterns into practice, we show how it makes the code flexible and easy to build on, also making the code adaptable, so it can be reused and maintained more efficiently.
4. The objectives of the work: The main objectives of the WeatherApp are to make a user-friendly and easily adaptable weather system. Demonstrate the practical implementation of design patterns (Observer, Decorator, Strategy, Adapter, Singleton, Factory). By using them we're making sure the code is easily extendable and maintainable. Through them we're achieving the practical benefits of design patterns in achieving flexibility, code reusability, and overall software reliability.

**Main body:**

Usage:

The WeatherApp class serves as the main entry point for the application. Users can interact with the system through a menu-driven interface. The application provides the following functionalities:

1. Update Weather for Cities: Allows users to input city names, temperatures, and weather conditions to update the weather data.
2. Send Notification about Weather Updates: Sends weather notifications using the WeatherNotifier interface. The FahrenheitDecorator ensures temperature conversion to Fahrenheit.
3. Convert Celsius to Fahrenheit: Converts temperatures from Celsius to Fahrenheit using the FahrenheitConversionStrategy.
4. Exit: Exits the WeatherApp.

Patterns:

1. Observer Pattern: The Observer pattern is employed to allow various components to receive updates about weather changes. The WeatherStation acts as the subject, while classes such as CelsiusToFahrenheitAdapter and FahrenheitDecorator act as observers.

2. Decorator Pattern: The Decorator pattern is employed in the FahrenheitDecorator class, allowing dynamic extension of the BaseWeatherNotifier class with additional behavior. This pattern enhances the notification system with temperature conversion.

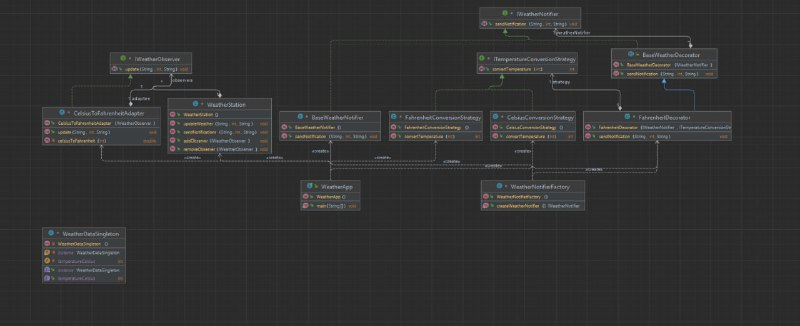
3. Strategy Pattern: The Strategy pattern is implemented through the TemperatureConversionStrategy interface, with CelsiusConversionStrategy and FahrenheitConversionStrategy as concrete strategies. The WeatherApp class uses these strategies for temperature conversion.

4. Adapter Pattern: The Adapter pattern is employed with the CelsiusToFahrenheitAdapter class, allowing seamless integration of different weather observers. This adapter adapts observers expecting temperatures in Celsius to work with the Fahrenheit-based system.

5. Singleton Pattern: The Singleton pattern ensures a single instance of the WeatherDataSingleton class, providing a centralized point for storing and accessing weather data.

6. Factory Pattern: The Factory pattern is used to create instances of the WeatherNotifier interface with the help of the WeatherNotifierFactory class. This pattern abstracts the creation of WeatherNotifier instances, providing flexibility for future extensions.

UML Diagram:



**Conclusion:**

Summary: The WeatherApp project, consisting of 13 classes, successfully implemented 6 design patterns, including Observer, Decorator, Strategy, Adapter, Singleton, and Factory, to create a flexible and user-friendly weather management system. We were able to achieve all our initial plans, the application allows users to observe and update weather conditions, receive notifications, and if need correctly convert temperatures.

Challenges faced: Working in a team of three had its own challenges. Making sure everyone used design patterns consistently and merging our work smoothly needed clear communication. Setting coding rules, having regular meetings, and checking each other's code helped us work together. Even with challenges, teamwork was crucial, bringing in different ideas. It was a great experience that sets the stage for future improvements and more teamwork.

Future improvements: There are several ways to make the WeatherApp even better. One idea is to add more ways to get weather notifications, like through email or text messages. We could also explore including more weather features to make the app more useful. Improving the look and feel of the app's interface, adding GUI would make it even more user-friendly. And if we can get real-time weather data from external sources, that would be a big plus. These improvements would enhance the app's capabilities and user experience, making it even more practical and enjoyable to use.

**References**

Refactoring.Guru. (2023a, January 1). *Adapter*. <https://refactoring.guru/design-patterns/adapter>

Refactoring.Guru. (2023b, January 1). *Decorator*. <https://refactoring.guru/design-patterns/decorator>

Refactoring.Guru. (2023c, January 1). *Factory method*. <https://refactoring.guru/design-patterns/factory-method>

Refactoring.Guru. (2023d, January 1). *Observer*. <https://refactoring.guru/design-patterns/observer>

Refactoring.Guru. (2023e, January 1). *Singleton*. <https://refactoring.guru/design-patterns/singleton>

Refactoring.Guru. (2023f, January 1). *Strategy*. <https://refactoring.guru/design-patterns/strategy>