ICS 372-01
Benjamin Cassidy
Jordan Mielke, Mohamed Yusuf, Steve Hartmann

# Climate Watch

## Introduction

Climate Watch is designed to deliver precise and current weather information, catering to both local conditions and US city forecasts. Our app ensures that you're always equipped with the latest weather updates, whether you're planning your daily activities or arranging a trip. It offers a comprehensive weather monitoring experience, enabling seamless access to real-time temperature data for your current location or any chosen destination.

Ideal for organizing family vacations, Climate Watch provides detailed forecasts, including expected temperature highs and lows. But it's more than just a temperature tracker; the app also offers extensive insights into weather conditions, including wind speed, humidity, visibility, and dew point, to give you a complete understanding of the weather wherever you are or plan to be.

## Feature Overview

### Functional Requirements

- **Display Current Weather For City:** Upon launching the app, a pop-up is shown giving the user the ability to read current weather data using the user's location.
- **Search By ZIP Code:** Enter a valid ZIP code and verify if the weather details are displayed. Test invalid ZIP codes (like "00000" or "Scooby Dooby-Doo") and check for an appropriate error message.
- **Unit Conversions:** Access the User Preferences screen, toggle between different units, and observe if the temperature, distance, and wind speed displays update accordingly.

- **Responsive UI Components:** Interact with various UI components to ensure they respond appropriately to user input and display relevant weather data efficiently.
- **Location Services:** When loading the app, it will prompt the user with a welcome screen asking if they would like to allow for location tracking. If allowed, it will grab the IP address of the user and use that to get their ZIP code. If denied, it will default to the ZIP code for Metro State University.
- **Weather Cache:** When searching for ZIP codes to display weather, if you were to check for a previously entered ZIP code it will load the cached information instead of doing a REST API call.

## Non-Functional Requirements

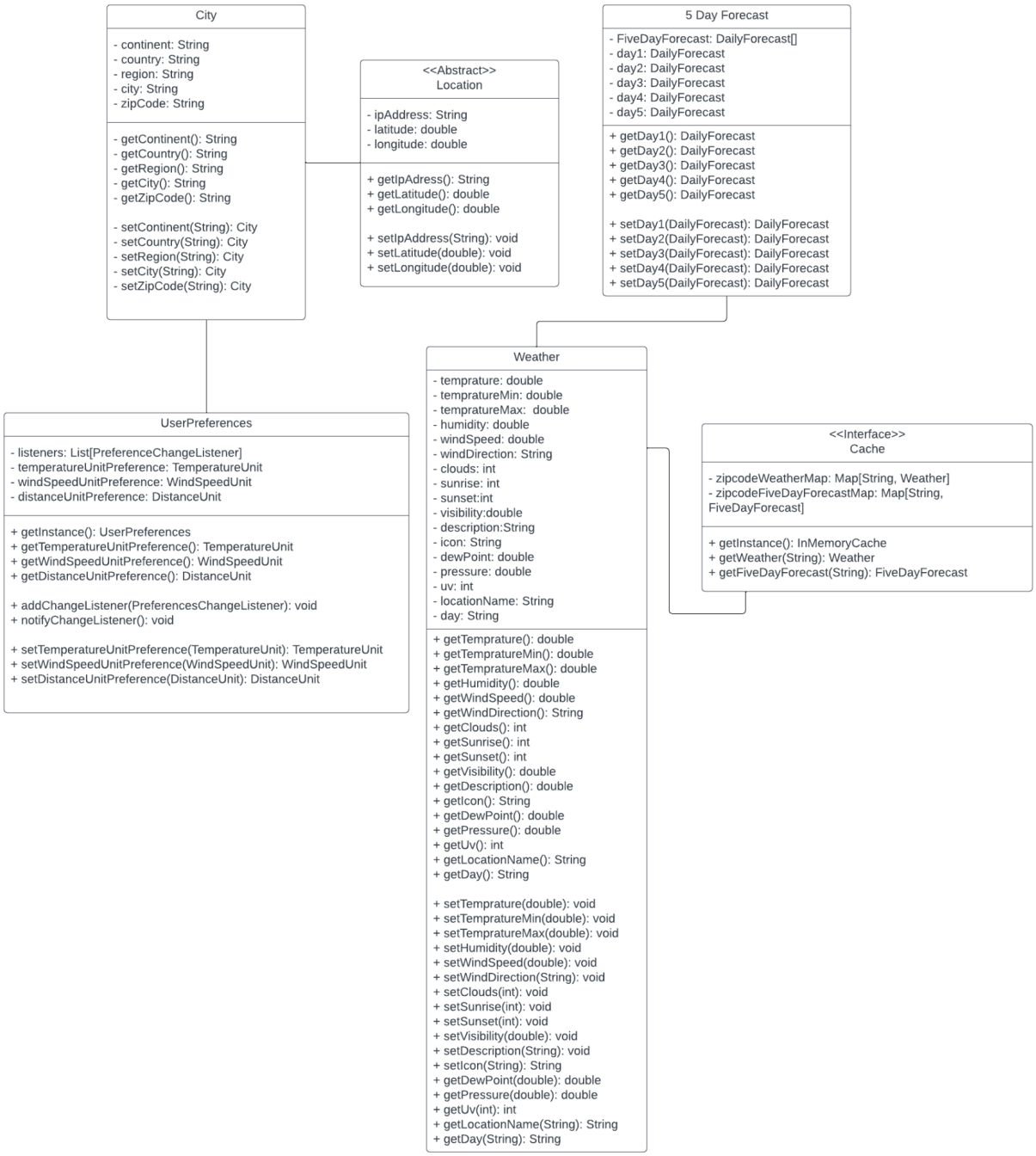
- **JavaFX For GUI:** Our application uses JavaFX to create a user-friendly graphical interface. This ensures an intuitive and responsive experience for users interacting with the application.
- **Use Weather API For Weather Data:** We utilize the OpenWeather API (https://openweathermap.org) to fetch real-time weather data. We've implemented efficient API calls and handled responses to display the data seamlessly within the app.
- **Use IP Geolocation API For Converting User IP Address To Location:** Our application incorporates the Ipstack API (https://ipstack.com) to convert the user's IP address into a geographical location. This feature enables the app to automatically detect and display weather information for the user's current location, enhancing the app's usability and personalization.
- **Error Handling:** We have implemented comprehensive input validation and error handling mechanisms. This ensures that the application responds gracefully to invalid inputs, such as incorrect ZIP codes, and provides helpful feedback to the user.

# Software Architecture & Design

## Internal Implementation

- **Model:** Contains classes such as City, Weather, and User, representing the data model, as well as strongly typed enums to hold units and conversions for temperature, distance, and wind speed.
- **Controller:** HomeController, UserPrefController, and WelcomeController to manage user interactions.
- **Service:** CityApiService and WeatherApiService to handle REST API calls for fetching city and weather data.
- **Utils:** Utility classes like IpUtils, TimeUtils, and ZipCodeUtils to provide auxiliary functionalities.
- **Cache:** An in-memory cache as intermediate storage for fetched weather data to reduce REST API calls.

# UML Diagrams



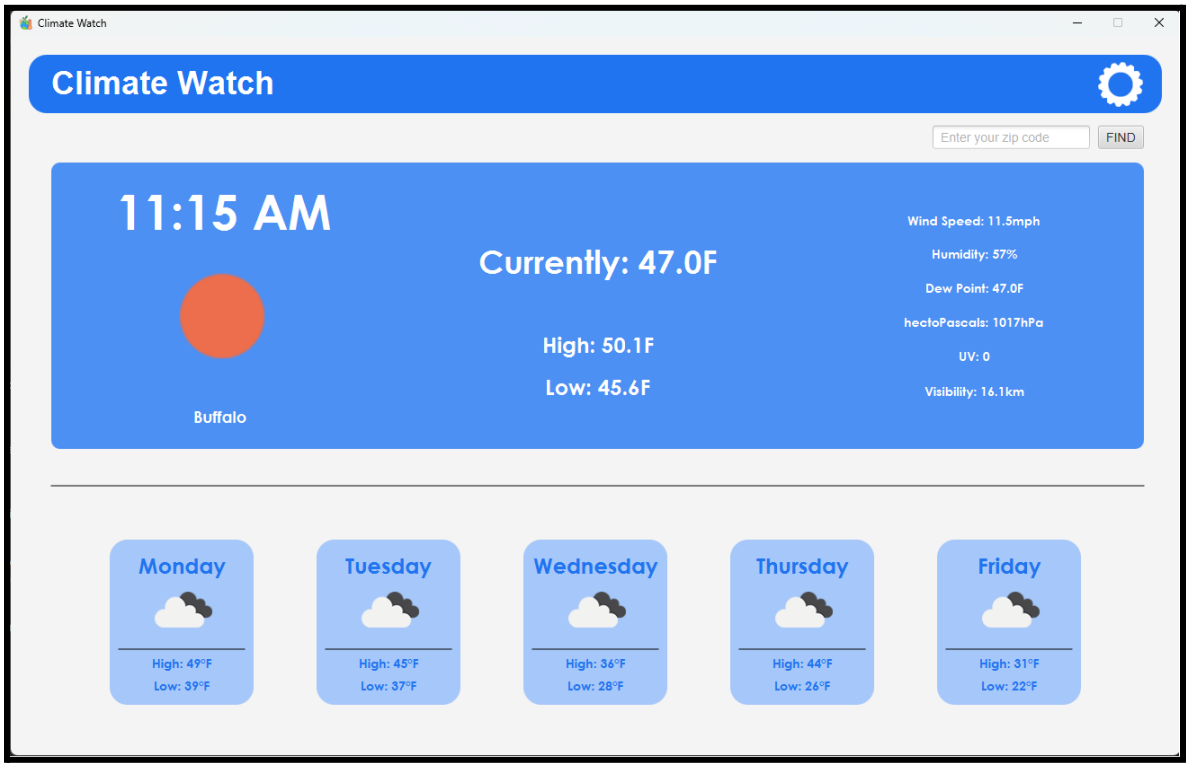
# User Interface Design

The UI is designed using FXML markup files, styled with CSS, and supported by a collection of images.

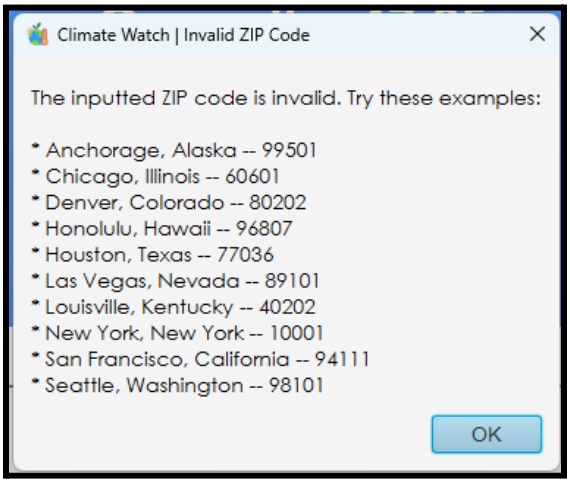
- **home.fxml, welcome.fxml,** and **user-pref.fxml** are the main UI markup files
- **home.css**, **welcome.css**, and **user-pref.css** ensure a consistent and appealing look

**Welcome Screen**

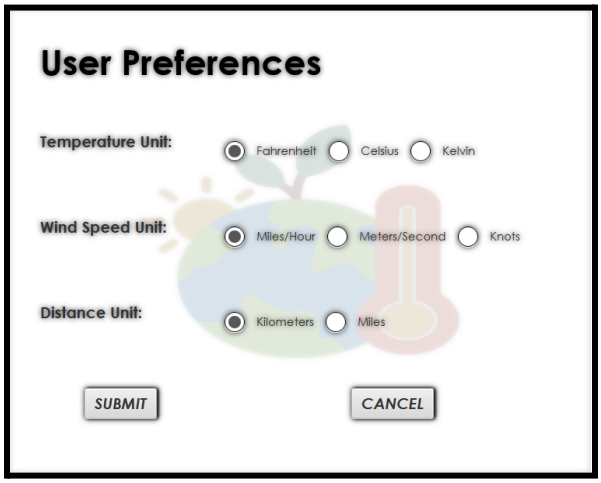

**Home Screen**



**Error State**




**User Preferences**




# Development Process

**What went well:**

Our approach to breaking down the app into different modules was a big win. It made it easier for us to work on separate parts without stepping on each other's toes. Using JavaFX to design the user interface was also a smart move. It helped us create an interface that not only looked good but was also user-friendly.

Communication among group members also went well. We shared phone numbers at the beginning of the project and had a group chat to text updates or progress as the project progressed. We also had a weekly Friday 6 pm meeting scheduled to check in and discuss any upcoming work or changes as needed. This helped to keep everybody engaged and on the same page.

**What didn’t go well:**

There were a number of things that didn’t go well. We ran into some issues trying to incorporate features from the REST APIs we were using. In particular, the forecast information that the weather API provides was not designed for our five-day forecast use case so we had to “massage” the data and do a lot of testing to make it work for our application. We also wanted to

include timezone data when displaying the city but the city API only includes that when you use the paid version of the API key.

We also weren't that well organized in terms of tasks. We could have used a JIRA board to keep track of who is working on what and to decide what functionality to work on next. While we avoided stepping on each other's toes for the most part, we did have some instances of that happening. For example, somebody made a change to the weather API service that reverted using the in-memory cache changes that had been done earlier that week.

**What would you do differently if you had the opportunity to implement the application again:**

Looking back, we could have spent more time in the initial stages planning how to integrate these external APIs. A bit more research and preparation could have saved us some trouble. Another thing we'd change is to focus on making the app responsive from the beginning. It's easier to build this in from the start than to try and add it later.

The final version of the application is also missing some functionality that we had planned on including. In particular, we wanted to list the weather for the 5 most popular US cities and the 5 most popular non-US cities as the main screen when the user first accesses the application. We inverted the order of operations for work, however, as we created the single city details screen first and left the multiple city screen as a todo for later.

**What did you wish you knew at the beginning of the project:**

There were a couple of things we wish we had known at the beginning of the project. We wish we had a better understanding of how tricky it would be to integrate the external APIs. The fact the weather API did not naturally support our five-day forecast design would have impacted what information we display and how we display it.

We also wish we knew how much work it would be to create UI screens using the built-in JavaFX UI elements, layouts, and behavior. There is a lot of trial-and-error testing where you make a change, re-start the application, and rinse and repeat. Deciding which UI element to use requires a considerable amount of digging through the Javadocs and consulting online tutorials and articles. While there is a SceneBuilder tool available, we didn't find it particularly helpful as it didn't integrate very well with IntelliJ IDEA.

**What have you learned about software development and object-oriented analysis and design:**

This project gave us valuable experience in breaking down complex undertakings into more manageable pieces. We also learned how to integrate various components, particularly external features. We came to understand the significance of early planning for various devices. It is far more difficult to make design modifications later in the development process.

In terms of coding, we learned that it is better to keep things simple and to avoid complexity i.e., follow the KISS principle. The more challenging you make things to understand, the more difficult it is to make changes in the future; you spend most of your time on maintainability rather than writing new code. Along the same lines, separation of concerns is essential to writing well-organized code that doesn't turn into a big ball of mud.

# Summary

**How did your team organize work?**

Our team implemented a structured yet flexible approach to work organization. We held weekly meetings every Friday, serving as a platform to assess our progress and set clear goals for the coming week. These meetings were crucial for maintaining alignment on project objectives, distributing tasks based on individual strengths and availability, and ensuring that everyone was on the same page regarding our project milestones.

**How did your team share code?**

We utilized GitHub for code sharing and collaboration needs. This platform allowed us to efficiently manage our codebase. We used features like branches, pull requests, and merge conflict resolution constantly. Through this we were able to work on different aspects of the project simultaneously without overwriting each other's work, thus enhancing productivity and minimizing bottlenecks.

**How did your team come up with features?**

Feature development was started off in brainstorming sessions. As we progressed, we shifted towards a more individual-driven approach, allowing team members to explore and propose new features independently. We ensured that these ideas were discussed and refined

during our Friday meet-ups, which kept them aligned with the overall project vision and user requirements.

**How did your team test features?**

Each team member was responsible for testing the features they developed on their computer before committing their changes to the main branch. After changes were merged, the other group members would each stress test those changes. This approach allowed us to maintain the integrity of the application.

**How did your team decide when the feature was complete?**

A feature was deemed complete only after satisfying the aforementioned criteria as well as receiving collective approval. We also looked at the milestones reports to make sure nothing was missed and we covered all of our bases.