ThreePlus

TrailWeather

Software Design Document

Jyri Hakala, Paavo Jyrkiäinen, Aaro Melchy, Henri Sutinen

01.10.2023

1. Introduction

This document is a comprehensive guide outlining the design and architecture of the TrailWeather application. This software represents a solution that combines trail mapping functionality with weather data, catering to outdoor enthusiasts, hikers, and adventure seekers. By seamlessly integrating trail map information and weather updates, TrailWeather aims to enhance the outdoor experience, improve safety, and provide valuable insights to users during their explorations.

The application is built with Java and JavaFX, which ensures a robust and interactive user experience.

2. Functionality and scope

TrailWeather aims to combine a collection of trail maps with real-time and forecast weather data. TrailWeather encompasses the following key features:

1. **Mapping:** The software will provide a collection of outdoor maps.
2. **Weather Data:** Real-time weather information for the chosen place will be available, including current conditions and forecasts.
3. **Saving preferences:** Users will be able to save locations (home, workplace, etc).
4. **Interactive Maps:** An interactive map feature will display relevant weather information on top of and next to the map in a user-friendly and intuitive manner.

3. API's used

Our project utilizes Trailmap as the foundational map source. Trailmap has maps such as MTB (mountain bike), orienteering and topographical maps which can be displayed in other web services and mobile applications. Trailmap's map tiling service follows the so-called OpenStreetMap (OSM) standard which is one of the most common if not the most common standard used by consumer services. We are using Gluon Maps library to integrate the map to our application.

For geospatial data retrieval, we employ the Nominatim API, a tool designed to search OSM data by name and address and retrieve the corresponding coordinates of specific locations.

For the weather data, we rely on the relevant information sourced from Openweathermap's weather API. Openweathermap's datasets encompass both real-time weather data and forecasts, ensuring we have access to up-to-date and accurate weather information for our project.

4. Design

The application follows the MVC model. We chose MVC as our design architecture, because MVC enforces a clear separation of concerns within the application. Each component has a distinct role, making the code more organized and easier to maintain. MVC also allows for the creation of a more responsive and interactive user interface. The controller handles user interactions, and the view can be updated dynamically to reflect changes in the model. This separation of concerns is crucial for creating modern, dynamic user interfaces. Lastly, MVC is a well-established pattern with a vast developer community and numerous resources available. It is easy to find plenty of tutorials, documentation, and tools that can help you implement MVC effectively.

The class diagram of the application (figure 1) has been expanded from the original prototype component diagram (figure 2). The applications model consists of weather and user data as well as some map related attributes, such as its' current center and type. View comprises the UI components which display the map and its overlays and controls for the map types and the different kinds of weather data. The controller part of the MVC –model is comprised of several controllers with specific responsibilities. They orchestrate data retrieval, updates, and user interactions.

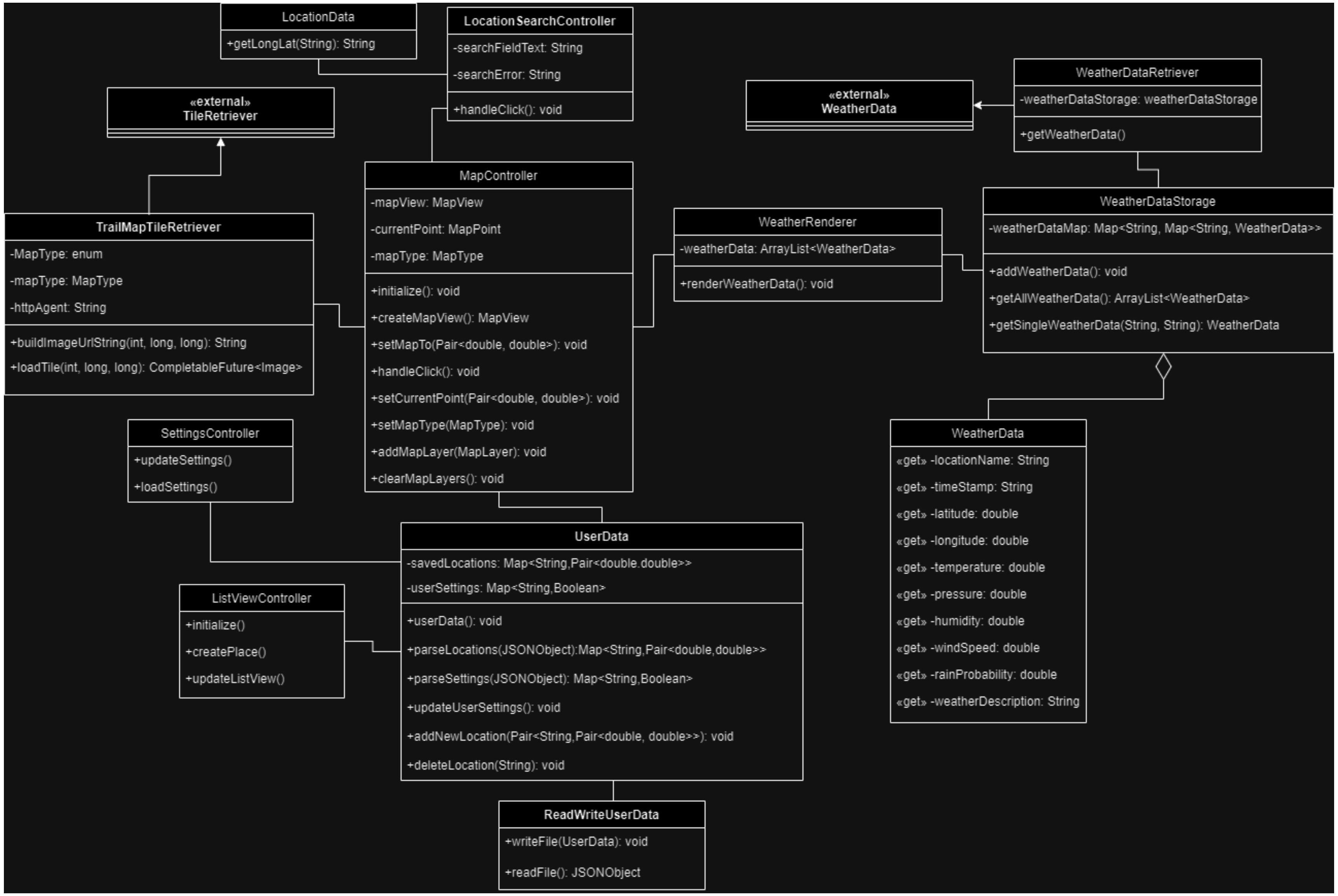
LocationData
+getLongLat(String): String
LocationSearchController
-searchFieldText: String
-searchError: String
+handleClick(): void
«external»
TileRetriever
«external»
WeatherData
WeatherDataRetriever
-weatherDataStorage: weatherDataStorage
+getWeatherData()
MapController
-mapView: MapView
-currentPoint: MapPoint
-mapType: MapType
+initialize(): void
+createMapView(): MapView
+setMapTo(Pair<double, double>): void
+handleClick(): void
+setCurrentPoint(Pair<double, double>): void
+setMapType(MapType): void
+addMapLayer(MapLayer): void
+clearMapLayers(): void
TrailMapTileRetriever
-MapType: enum
-mapType: MapType
-httpAgent: String
+buildImageUrlString(int, long, long): String
+loadTile(int, long, long): CompletableFuture<Image>
WeatherRenderer
-weatherData: ArrayList<WeatherData>
+renderWeatherData(): void
WeatherDataStorage
-weatherDataMap: Map<String, Map<String, WeatherData>>
+addWeatherData(): void
+getAllWeatherData(): ArrayList<WeatherData>
+getSingleWeatherData(String, String): WeatherData
SettingsController
+updateSettings()
+loadSettings()
WeatherData
«get» -locationName: String
«get» -timeStamp: String
«get» -latitude: double
«get» -longitude: double
«get» -temperature: double
«get» -pressure: double
«get» -humidity: double
«get» -windSpeed: double
«get» -rainProbability: double
«get» -weatherDescription: String
UserData
-savedLocations: Map<String,Pair<double.double>>
-userSettings: Map<String,Boolean>
+userData(): void
+parseLocations(JSONObject):Map<String,Pair<double,double>>
+parseSettings(JSONObject): Map<String,Boolean>
+updateUserSettings(): void
+addNewLocation(Pair<String,Pair<double, double>>): void
+deleteLocation(String): void
ListViewController
+initialize()
+createPlace()
+updateListView()
ReadWriteUserData
+writeFile(UserData): void
+readFile(): JSONObject

Figure 2. Class diagram

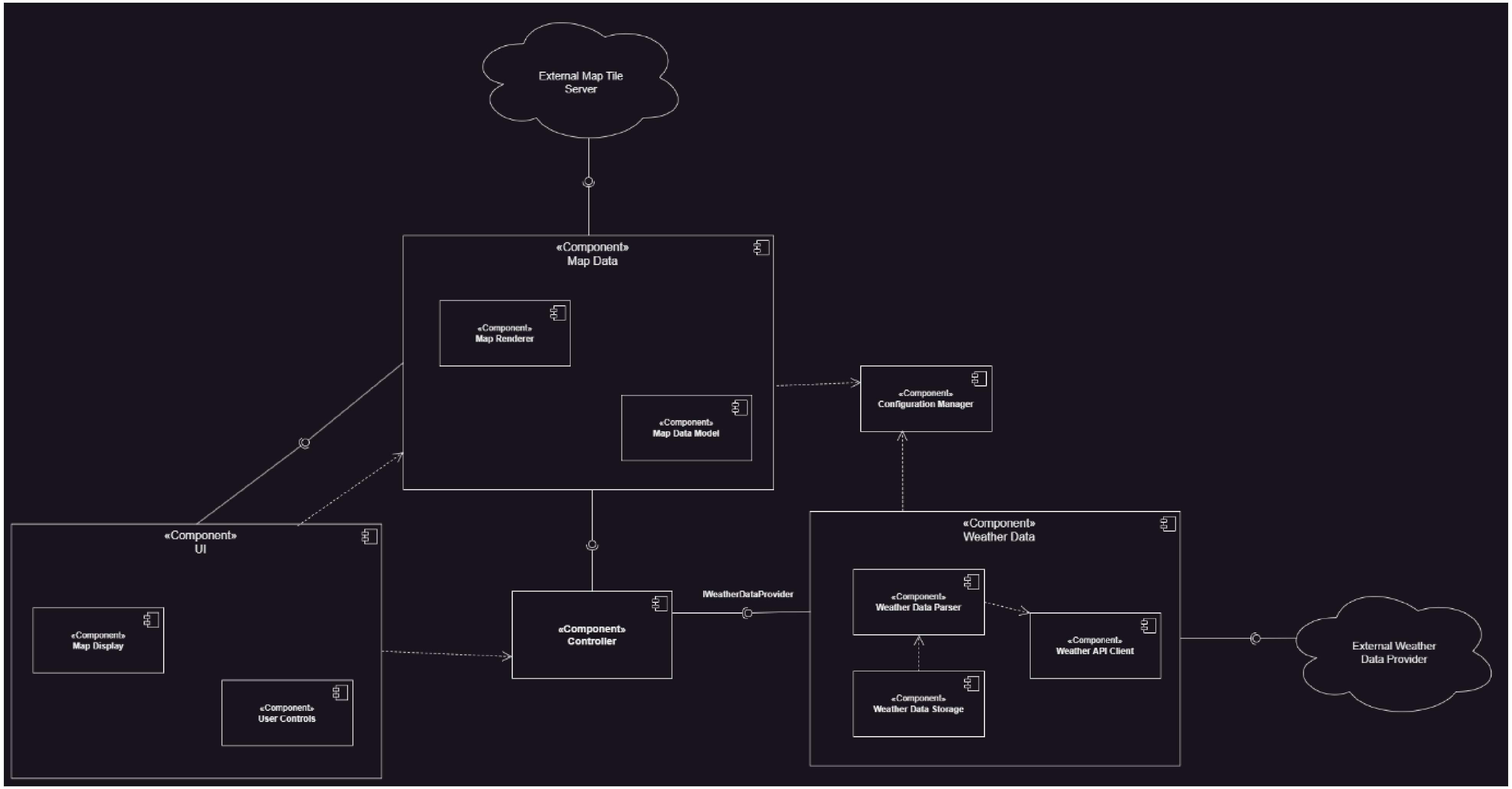
External Map Tile Server
«Component» Map Data
«Component» Map Renderer
«Component» Map Data Model
«Component» Configuration Manager
«Component» UI
«Component» Map Display
«Component» User Controls
«Component» Controller
IWeatherDataProvider
«Component» Weather Data
«Component» Weather Data Parser
«Component» Weather API Client
«Component» Weather Data Storage
External Weather Data Provider

Figure 1. Component diagram (PROTOTYPE)

4.1 Component descriptions:

1. **TrailMapTileRetriever:**
    - **Responsibility:** Fetches map tiles from the tile provider (trailmap.fi)
    - **Role:** Extends the functionality of gluon maps to use trailmap as a provider instead of openstreetmaps.
    - **Relationships:**
        - Used by MapController to set type of map to display.
        - Other interactions are handled by gluon maps.
2. **MapController:**
    - **Responsibility:** The Map controller handles map-related functionalities, such as displaying the correct map and its' zoom levels.
    - **Role:** Manages all user interactions related to the map. Also controls which type of map to render.
    - **Relationships:**
        - The controller communicates with a MapView object provided by our chosen library (gluon maps).
        - Utilizes a TrailMapTileRetriever to fetch the actual tiles to be rendered.
3. **LocationData:**
    - **Responsibility:** Provide coordinates to locations.
    - **Roles:**
        - Acquiring matching coordinates to a building name / street name / city name.
    - **Relationships:**
        - Gets location from LocationSearchController.
        - Communicates with OpenStreetMap API to retrieve coordinates.
4. **UserData:**
    - **Responsibility:** Saves and modifies user data during runtime of the program.
    - **Roles:**
        - Parses JSONObject that contains user's data.
        - Modifies user data real time and updates it to different controllers.
    - **Relationships:**
        - Gets and sends user's data to ReadWriteUserData.
        - When the user changes options, UserData informs these new options to ListViewController and MapController.
5. **ReadWriteUserData:**
    - **Responsibility:** Reads and writes user data to a file.
    - **Roles:**
        - Reads a file that contains user's data.
        - Saves user data to a file.
    - **Relationships:**
        - Gets and sends user data from UserData.
6. **WeatherDataStorage:**
    - **Responsibility:** Stores all of the retrieved weather data.
    - **Roles:**
        - Saves weather data into a map that contains dates weather data objects mapped for locations and dates.
        - Provides retrieval of the weather data that is saved into the storage.
    - **Relationships:**
        - Gets the weather data from WeatherDataRetriever.

        - Provides weather data for WeatherRenderer when it requires the data to render it to the map.
        - The storage itself consists of multiple instances of the WeatherData class.

7. **WeatherData:**
    - **Responsibility:** Represents and encapsulates individual weather information. It stores all relevant details like place, temperature and more into its attributes.
    - **Roles:**
        - Structure for each weather data instance.
    - **Relationships:**
        - Offers model for the instances of weatherDataMap in the class WeatherDataStorage.
8. **WeatherRenderer:**
    - **Responsibility:** Renders the weather data on the map.
    - **Roles:**
        - Renders typical weather icons on the defined locations on the map based on the weather data.
        - Updates the weather icons when the user changes time settings.
    - **Relationships:**
        - Gets the weather data from WeatherDataStorage.
        - Updates the map of MapController.
9. **WeatherDataRetriever:**
    - **Responsibility:** Fetches the weather data from the weather API.
    - **Roles:**
        - Uses Openweathermap's API to retrieve the weather data for the program to use.
    - **Relationships:**
        - Provides the retrieved data for WeatherDataStorage.
10. **SettingsContoller:**
    - **Responsibility:** Handles user actions regarding settings.
    - **Roles:** "Bridge" between UI and the user data model.
    - **Relationships:** Loads and updates data of UserData.
11. **ListViewContoller:**
    - **Responsibility:** Manages the "saved places" -listview element.
    - **Roles:** Provides funcitonality for the listview and manages the saved places.
    - **Relationships:** Loads and updates data of UserData.

5. AI tool usage

The original ideas for this project were partly generated by ChatGPT and some parts of this document were written with the help of AI tools (ChatGPT and DeepL translator). The rough outline for the architecture of the application was created with the help of ChatGPT. Some of the IDEs used by the developers also included some AI tool extensions such as GitHub Copilot to enhance coding efficiency.

6. Self-evaluation

The original prototype component diagram (figure 2) gave some direction for us, but the end result differentiated from it to some extent. Also, the UI turned out to be different compared to the prototype as the controls are now on the left panel instead of on top of the map.

Specifics on changes made:

- There is no central controller, but several of them instead. All the controllers have their own responsibilities.
- No configuration manager. The application only has one API key and making a class for managing that seems like overkill.
- Originally, we planned to use The Finnish Meteorological Institute's open data for our weather data. However, Openweathermap’s API was better suited for our needs and documentation was clearer.

We don’t currently have a plan to make any major changes. The original design provided a reasonable basis for the application that we’ve been able to modify and extend. The MVC – model has proved to be a good design choice.

References

- Trailmap: https://web.trailmap.fi/
- Gluon Maps: https://github.com/gluonhq/maps
- Openweathermap's API: https://openweathermap.org/api
- Nominatim API: https://nominatim.org/release-docs/latest/api/Overview/