SW Design project design document

Project Idea

Our project idea is a desktop application you can use to search for astronomical events happening near you. The application contains a map and calendar to select desired coordinates and time range and based on those inputs we send API requests to various services to fetch information on various happenings in the sky. These events can be things like the International Space Station flyover, full moon, meteor flying over you, solar eclipse or a picture of the night sky above you.

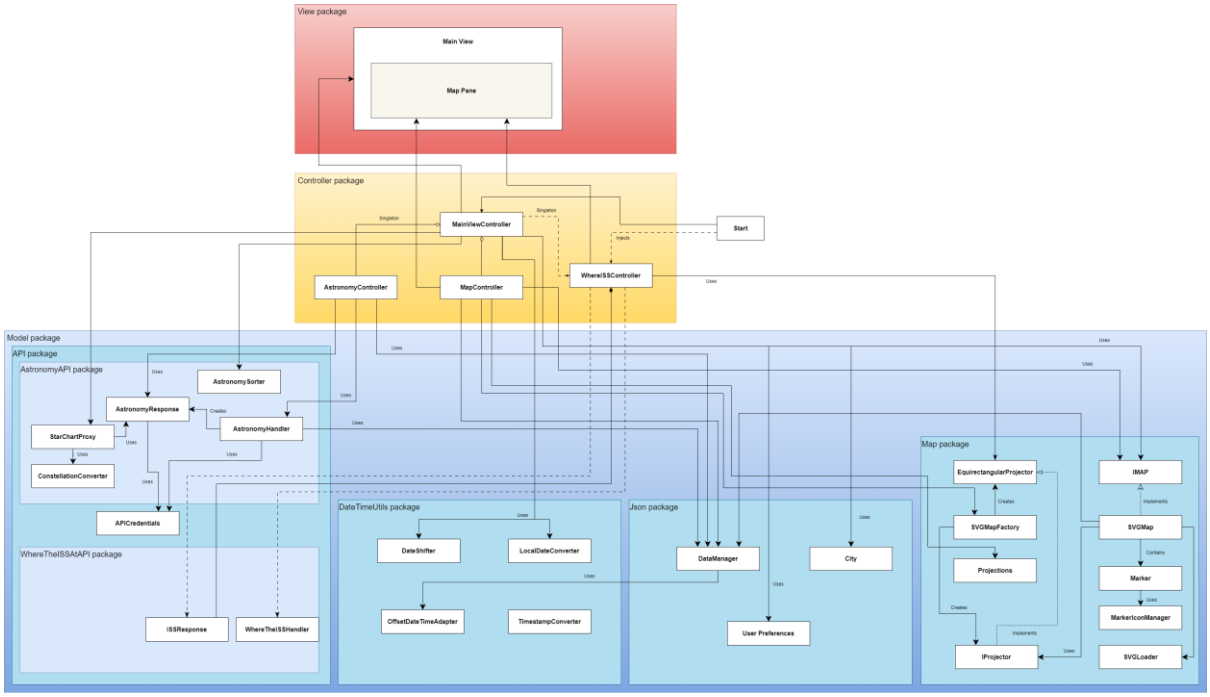
The idea was produced using ChatGPT with the following prompt: “Give me an idea for a JavaFX app that has a map with location and time selectors, and uses those to display data from various sky related events happening near given parameters. The information is fetched from external APIs.”

UI Prototype & Software architecture

Our project follows the MVC model and is composed of four main parts: The GUI or the view which is built with SceneBuilder and composed of .fxml files. Our Controller classes are declared in .fxml files and handle updating the UI as well as passing requests to the Model, which contains the main application logic.

The model contains three major components. The map package contains all classes pertaining to the map shown on screen, as well as logic for calculating the real-world coordinates from the pixel grid and back. Json package contains classes for saving and loading data, such as the state of the program when it was closed, as well as the list of suggestions when you type a city name into the search bar. Finally, the API and dateTimeUtils packages contain the actual API requests, and formatting as well as other helper methods which we need to display the data in a presentable fashion.

Since our View is relatively simple and is composed of a single .fxml file, and at the same time the controller methods are much more complex, we split the controller methods handling the API and Map methods to their own controller classes to ensure proper separation of responsibilities.



Picture 1. UML-diagram of the program structure

UI

For the UI of this application, we ended up using JavaFX with FXML for its ease of integration into our Maven project and the easy-to-use Scene Builder application. This does limit our options a bit when it comes to more advanced functionality compared to for example JavaScript or HTML based environments, but smooth integration quickly outweighs these concerns. Even though splitting the controllers for a single window UI can be quite clumsy with just one fxml-file, we still decided to go for it for the ease of organizing and development. The UI is essentially laid out in two parts, the map pane and the sidebar. Both are implemented in the Scene Builder app, but the map pane is left empty, and it's handled completely by the specific map controller

The clearest design pattern implemented in the UI and its controller is the Factory pattern in the form of the addEventCard-method, which generates data cards for each event given during a search operation based on just the Event-object. In a way there is also a Strategy pattern being utilized, by creating multiple different types of cards for events and planets using the same method.

We've also utilized the Asynchronous pattern in multiple instances in the code to smooth out the user experience in situations with longer wait times caused by the APIs. One of these is the loading of event cards after a search, as the API can sometimes be a bit slow. We've added a Text box, which both indicates when the new events are being

fetched from the API, but also tells the user if the given input was incorrect in formatting or if it wasn't able to be read for some other reason.

The only part of the UI which utilizes a separate window is the star map, which is available for every planet and gives an idea where the planet can be found in the sky relative to stars and constellations. Here we were once again plagued by the slow load times of the Astronomy API, this time it causes an error almost half the time. We still wanted to implement this feature, so we needed to apply the Asynchronous pattern once again to keep the application functional while the picture was being fetched from the API and also an error message when the fetching was unsuccessful.

Map

Since there were no good map libraries available or working at this point in time, we had to implement our own. To accomplish this we created an interface of the general methods our map is supposed to provide called IMap, and then prototyped different implementations for it before deciding on using .svg map and doing the marker and coordinate mapping on our own. IMap is also technically an example of the strategy pattern though it is only implemented by one class.

We did this with a map image that is in SVG file format and has a known projection type. With this information we can calculate the pixel coordinates to real world coordinates and back and draw markers and other information on top of the map seamlessly.

The lifecycle of SVGMap begins with SVGMapFactory, which uses the factory pattern to create our SVGMap with the associated Projector class, which is responsible for converting the pixel coordinates to and from real world coordinates. Maps have different projections, which is why those calculations are separated into their own group of classes that implement the IProjector interface. SVGMap can be considered a Mediator, since it handles communication with any marker objects, which also require methods provided by the Projector to work properly. The controllers and view also only interact with SVGMap / SVGMapFactory, which means that the SVGMap is also a sort of Facade pattern.

Right now, we only have Equirectangular projection implemented, but to use for example Mercator projection all we would need is the corresponding map file and a projector class implementation. This is an example of the Strategy pattern.

Projector is also a Bridge pattern in a way, since most of the application uses the pixel coordinates to draw and update marker positions for example, while our APIs and user inputs use real world coordinates. Projector is the translation layer between these classes.

When SVGMap is first created, the .svg file is loaded from our resources. JavaFX does not natively support .svg filetype so we need to use Apache Batik library to convert it into ImageView object which can be shown on screen. This is done by SVGLoader.

When the map is clicked, a marker is added to the screen. All of the map markers share the same icon image, which is loaded from disk once by MarkerIconManager, which is a Singleton responsible for this. It's also a Flyweight pattern since the icon file is the most expensive part of a Marker instance and is thus shared across all of them.

API communication

Our project uses two main APIs to fetch data: the Astronomy API for celestial events and the WhereTheISSAt API for tracking the International Space Station. These APIs are integrated into the program through dedicated controller and model components to keep the code organized and easy to maintain. API communication and logic is kept in the model, while the controllers control and communicate with the view.

The controllers mimic interface-like behavior for the handlers, as they are intended as the access point for them, though they also include other features.

API controllers

AstronomyController handles requests to the Astronomy API. It provides methods for getting data about celestial body positions, their visibility, and events such as lunar and solar eclipses. To improve performance, it caches API responses, reducing unnecessary calls.

WhereISSController is responsible for managing the ISS icon on the map. It fetches the ISS's position, updates its coordinates on the map, and ensures the icon's correct scaling and placement. It also provides methods to retrieve the ISS data when needed.

API model packages

The model includes two packages for working with these APIs:

The astronomyAPI package contains the AstronomyHandler, which makes the actual API calls and converts the data into AstronomyResponse objects. These objects store the information retrieved from the API. The package also includes a StarChartProxy class, which shows a placeholder image while the actual starchart is being fetched.

The whereTheISSAtAPI package includes the WhereTheISSHandler, which handles API calls to fetch the ISS's position and velocity. It also loads the ISS icon from the hard drive. The data is stored in ISSResponse objects.

In both cases, the logic for handling API requests and responses is kept separate from the rest of the application, which makes it easier to manage and test.

The project uses some design patterns to simplify the implementation. The AstronomyResponse uses the Builder Pattern to make it easier to create objects with multiple fields as the API responses can vary significantly. The StarChartProxy uses the Proxy Pattern to manage star chart images, showing a placeholder until the actual image is available.

The Singleton Pattern is utilized for both controllers in the application as they are implemented to so that only one instance exists throughout the program. The Observer Pattern is used to react to changes in windows sizes. This ensures a responsive and user-friendly interface without requiring a manual refresh.

API issues

The Astronomy API does unfortunately not work perfectly. It's a known issue, that the API returns endpoint timeouts (HTTP 504) sometimes when generating a star chart. This is even specified in the API's documentation (https://docs.astronomyapi.com/known-issues). The star charts also take a while to generate. This is where caching and the Proxy pattern come in handy.

Calling the API again with the same parameters will work. In this application that means pressing the star chart image button again. If the endpoint is very busy this may even take a couple tries.

Data persistence using JSON

To meet the requirement for storing and loading data persistently between application runs, we chose JSON as the data format. JSON was selected due to its lightweight structure, human readability, and ease of debugging.

Since Java does not include native support for JSON handling, we evaluated various libraries and chose Gson for its lightweight design, simplicity, and flexibility. Gson provides extensive support for working with JSON, including the ability to implement custom serialization and deserialization, making it a suitable choice for our use case, which initially focused on small JSON files.

The DataManager class in our project is responsible for converting data to and from JSON format. It provides three key methods that enable seamless saving and loading of data from JSON files. These methods are designed to offer flexibility, allowing users to easily save or load any Java objects in a straightforward manner.

In our application, data loading is performed during the initialization of classes that require persistent data. When the application starts, these classes utilize the methods

provided by DataManager to load their data from JSON files, ensuring that the app is preloaded with the necessary information.

Saving is performed just before the application shuts down. Since we have classes that hold data that needs to be persistent and follow the Singleton pattern, the save commands are coordinated through the MainViewController. This ensures that all necessary data is properly saved before the application exits.

Unit testing

Unit tests were created for most of the Model classes. Controllers are very dependent on the JavaFX view and user inputs, so the optimal way to test those would be with a UI testing framework like Robot Framework. Controllers also utilize heavily methods from model classes, so it benefits from those tests indirectly. View is also not Java code at all so unit testing there using JUnit was not possible.

Self-evaluation

From the beginning our application was built for JavaFX and SceneBuilder. This means that .fxml files, their controllers and the application logic classes very naturally can be put into their own packages. Which are currently named as view, controller and model respectively. Model is further split into subpackages for different major components.

Whether this is a true MVC-pattern is a little confusing, since in our application the application logic files, which correspond to model and .fxml files in view don't communicate with each other. Instead, model passes the information back to controller which is responsible for updating the view.

As of now all the problems have been due to JavaFX, and its poor/nonexistent support of for example Open Street Maps or Google Maps which forced us to make our world map implementation.

Despite that though, we have been able to follow our original design remarkably closely. Even if something was more difficult to implement than we originally thought, we haven't needed to give up anything yet.

As for dividing up our work responsibilities, we met up at least once every week and paid considerable attention that everyone had always something to do, which worked well for us.

AI usage

We used GitHub Copilot and ChatGPT throughout the project in writing the code and ease the transition to Java from other languages that we had more experience in.

For the map-package, Copilot was used to as a very fancy autocomplete, and most majorly the SVGLoader class was implemented almost entirely by AI, after the initial libraries and structure of the class was made.

The saveData method in the DataManager class was initially generated by ChatGPT and modified. The loadDataAsObject method was created using saveData as a template.

The OffsetDateTimeAdapter is the only class that was entirely generated by ChatGPT without modifications.

On the User Interface side, the panes that hold the event information was first designed as its own fxml-file in Scene Builder to make the design process easier, but as we needed the panes to be generated individually inside the controller, we used ChatGPT to transfer the contents of the fxml file into a pane generator, which we could then modify into what it currently is.

Within the API package in the model, including AstronomyController and WhereISSController, Copilot was utilized as an autocomplete tool to accelerate development. ChatGPT played a role in refining ideas, assisting with debugging, and suggesting new approaches when challenges arose.

Notably, the API response classes were initially generated using ChatGPT. However parsing the data into these classes proved to be too high a hurdle for ChatGPT, requiring the work to be done manually. Additionally, the ChatGPT-generated response classes were eventually scrapped and recreated manually, as they were exceedingly difficult to work with, they included a lot of useless data, and the code was hard to read.

The map file we used (under creative commons licence):
https://commons.wikimedia.org/wiki/File:Equirectangular_projection_world_map_without_borders.svg