Assignment 2

Team number: 44

Team members

Name	Student Nr.	Email
Daniel Volpin	2659162	d.volpin@student.vu.nl
Daniel Verner	2671724	d.berzak@student.vu.nl
Marco Deken	2646923	m.j.deken@student.vu.nl
Osman Abdelmukaram	2708829	o.abdelmukaram@student.vu.nl

Implemented features

ID	Short name	Description
F1	Visualise route	Based on a GPX input file provided by the user, visualise the route on a map.
F2	Display user activity metrics	After the user selects the GPX file, the app should compute relevant metrics calculations, and display them to the user.
F3	Display weather information	After the user selects the GPX file, the app should show weather metrics based on the GPX data.

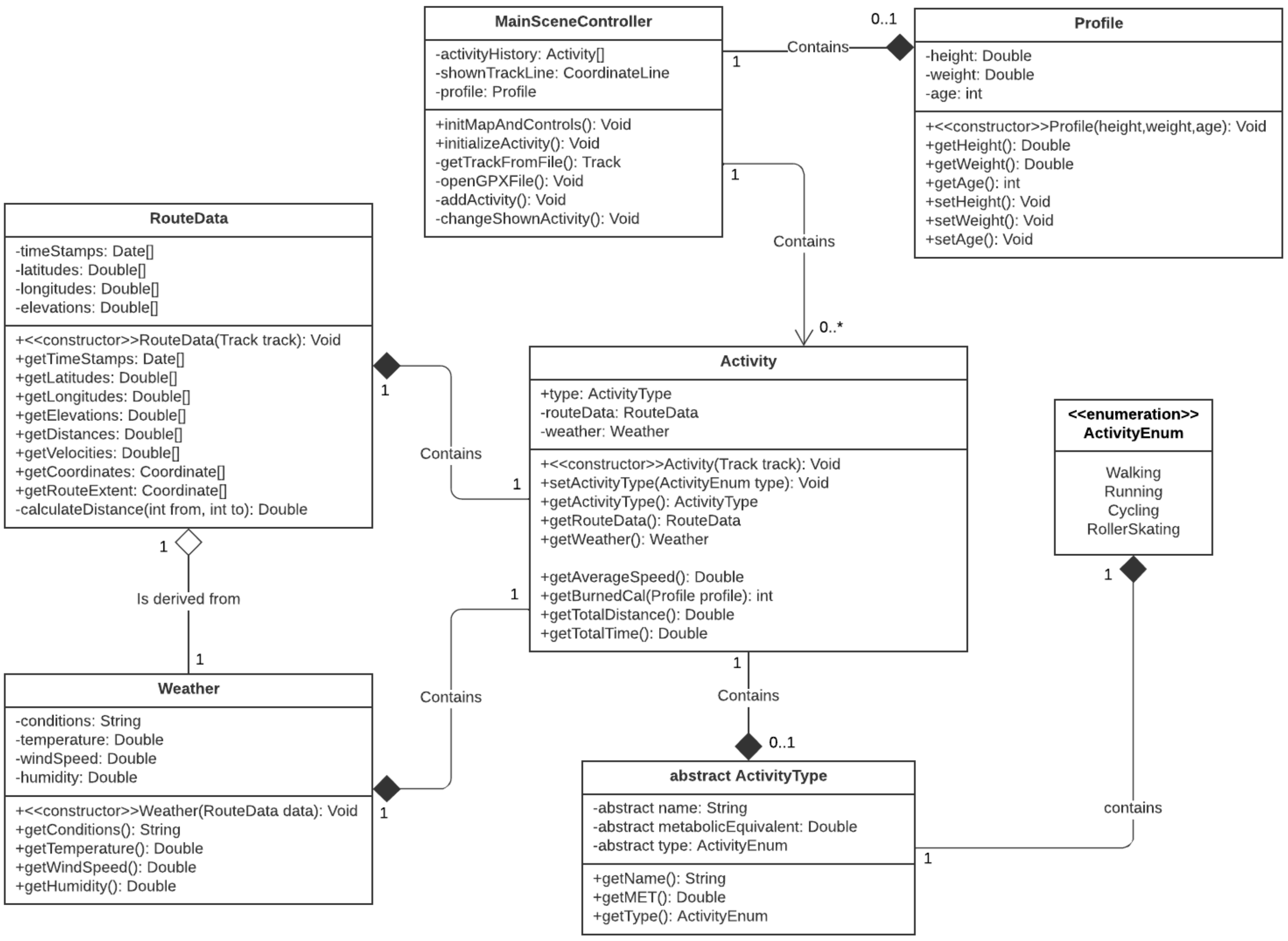
Used modelling tool: https://lucid.app

About our assignment 1 feedback

In our previous assignment, our group established several quality requirements that would be used in our project. However, one quality requirement in particular was vague and lacked description. The removed quality requirement was discussing how to improve the application's performance through the use of caching. This quality requirement can be interpreted both positively and negatively. As a result, we eliminated that one quality requirement as it was deemed too vague for our project. Furthermore, we already save the provided GPX file data in an Activity array (which we will elaborate on further in this document), so the requirement was unnecessary as well.

Class diagram

Author(s): Marco Deken and Daniel Volpin



MainSceneController

The **MainSceneController** acts as the data flow controller and the main class to display our application. When the user starts up the application, an instance of the *MainSceneController* is initialised. This initialisation (initMapAndControls()) means that the map will be rendered on screen, and buttons will be shown to allow for further functionality (e.g. provide a GPX file).

If the user decides to provide a GPX file by pressing the corresponding button, openGPXFile() is called. This function allows the user to select a GPX file, upon which the getTrackFromFile() and initializeActivity() function are called. The contents of the GPX file will namely be stored in an instance of the *Activity* class. To allow the user to provide multiple file uploads, we store all the **activities** in an *Activity array* attribute called activityHistory. Furthermore, if the user has provided multiple GPX files, then the user can switch between showing different activities on the map, which will be done using the changeShownActivity() function. Out of all initialised activities, the **activity** that is currently visualised on the map is stored in the shownTrackLine attribute.

Furthermore, if the user wants, he/she can store information about their profile in a *Profile* class. See *Profile* paragraph for more information.

Profile

A **profile** will store the user's height, weight and age. This information is relevant in order to be able to calculate the amount of calories the user has burned doing a certain type of activity. The class has getters as well as setters for all three properties, because the user should be able to change its **profile** in the application.

However, the user does not have to provide a **profile**. The application is not dependent on it and can therefore run fine without it. However, only if the user wants his burned calories to be calculated, this information is required.

We have made the design choice that the **profile** is associated to the *MainSceneController* class, and not to the *Activity* class. We chose this, as now the user will have to set his **profile** information no more than only once. The alternative is that we allow the flexibility of a **profile** per **activity**. This would allow a user to provide GPX files of activities that were performed by different users, and then set the activity-specific **profile** for the different people that did the activities. However, this would imply the user has to set a **profile** for every provided GPX file. Therefore, we chose for one instance of a *Profile* to be associated to the *MainSceneController*.

Activity

An **activity** of type *Activity* clusters together all functionality that is relevant for the activity corresponding to a provided GPX file. To be able to construct it , a **track** is needed (from the GPX file). Then, the positional data from the **track** is stored in routeData (see *RouteData* paragraph), for which a get-function is available. Also, a feature of the application is to show some weather information about the place and time the activity took place. This information is stored in a *Weather* object, with the name weather and corresponding getWeather() function.

Next to routeData and weather, **activity** has a third property, type. This property stores the activity type in an instance of *ActivityType*. Upon initialisation of an *Activity* object, the type of activity will not be initialised. After the *activity* is initialised and visualised on the map, the user can provide the type if he/she wants. Only if the user is interested in calculating the amount of calories burned during the activity, the type of activity will be required, since it is a vital piece of information for that calculation. See *ActivityType* paragraph for more information.

Furthermore, the *Activity* class is where all the implemented metrics are calculated. Therefore, from this place, it is easy to add another metric if deemed desirable. The current metrics that are implemented are the total travelled distance (getTotalDistance()), the total travelled time (getTotalTime()), and the average speed in metres per second (getAverageSpeed()). The getBurnedCal() still has to be implemented. Furthermore, if we want to show more metrics, the list of metrics is easily extensible by writing a function for it in the *Activity* class.

RouteData

The **RouteData** class is used to store the positional data from the **track**, where its properties are timeStamps, latitudes, longitudes, and elevations. Each property has a get-function that was implemented to display meaningful information on the map and the right-side panel. In addition, to construct it, an *Activity* instance is needed to compose the relevant track information.

When the user clicks on the “Upload GPX file” button and uploads the file, an instance of the **Activity** class is created, which in turn creates an instance of the **RouteData** class once the **track** is added. Following this, the track points are loaded (getTrackPoints()), to initialise the above mentioned properties. Moreover, from these properties the user can view the velocities, distances and coordinates points through the getDistances(), getVelocities(), and getCoordinates() functions. These derived properties are meaningful to present to the user, as it creates a short overview of the activity that has taken place. Furthermore, when an instance of routeData is created, the getRouteExtent() function is called from the class to centre the map around the coordinates given by the GPX file.

Also, the **RouteData** has a shared aggregation with the **Weather** class which utilises the timeStamps, longitude and latitude to display weather properties to the user.

Weather

The **Weather** class is used to store information regarding the weather at the time and place of the activity. The information is fetched using a call to the “Visual Crossing Weather API”, received in JSON formatting. It is later parsed and stored in an instance of the **Weather** class. The class has four properties: conditions, temperature, windSpeed and humidity. Each property has a get-function which, similarly to the **RouteData** class, is used to present the user with meaningful weather metrics. This happens only once the *activity* instance is created.

Depending on the conditions, the user is presented with a visual weather overview, which is presented on the right pane of the application. Together with the chosen measurements, it was decided that the main four properties and the visual weather overview would present the user with the most meaningful information in a concise manner.

The class has a shared aggregation relation with **RouteData** from which it extracts the timeStamps, longitudes and latitudes to display correct weather information.

Last, the API call returns a wide variety of variables regarding the weather conditions at the time given from the GPX file, making this feature widely extensible.

ActivityType

An **activity** can have an optional <u>type</u> as an instance of the *ActivityType* class. The *ActivityType* class itself is an abstract class. The *ActivityType* class can be extended by classes that represent an activity, such as a *Walking* class or a *Running* class. These non-abstract classes have to implement the abstract properties of *ActivityType*, which are the <u>name</u>, <u>metabolicEquivalent</u>, and <u>type</u>. The <u>name</u> would be the type of activity in string-format (i.e. “Walking”), and <u>metabolicEquivalent</u> (MET) is the activity-specific variable necessary to calculate the amount of calories burned during the activity. Lastly, for completeness, the <u>type</u> as *ActivityEnum* is included to be able to compare the different *ActivityType* classes.

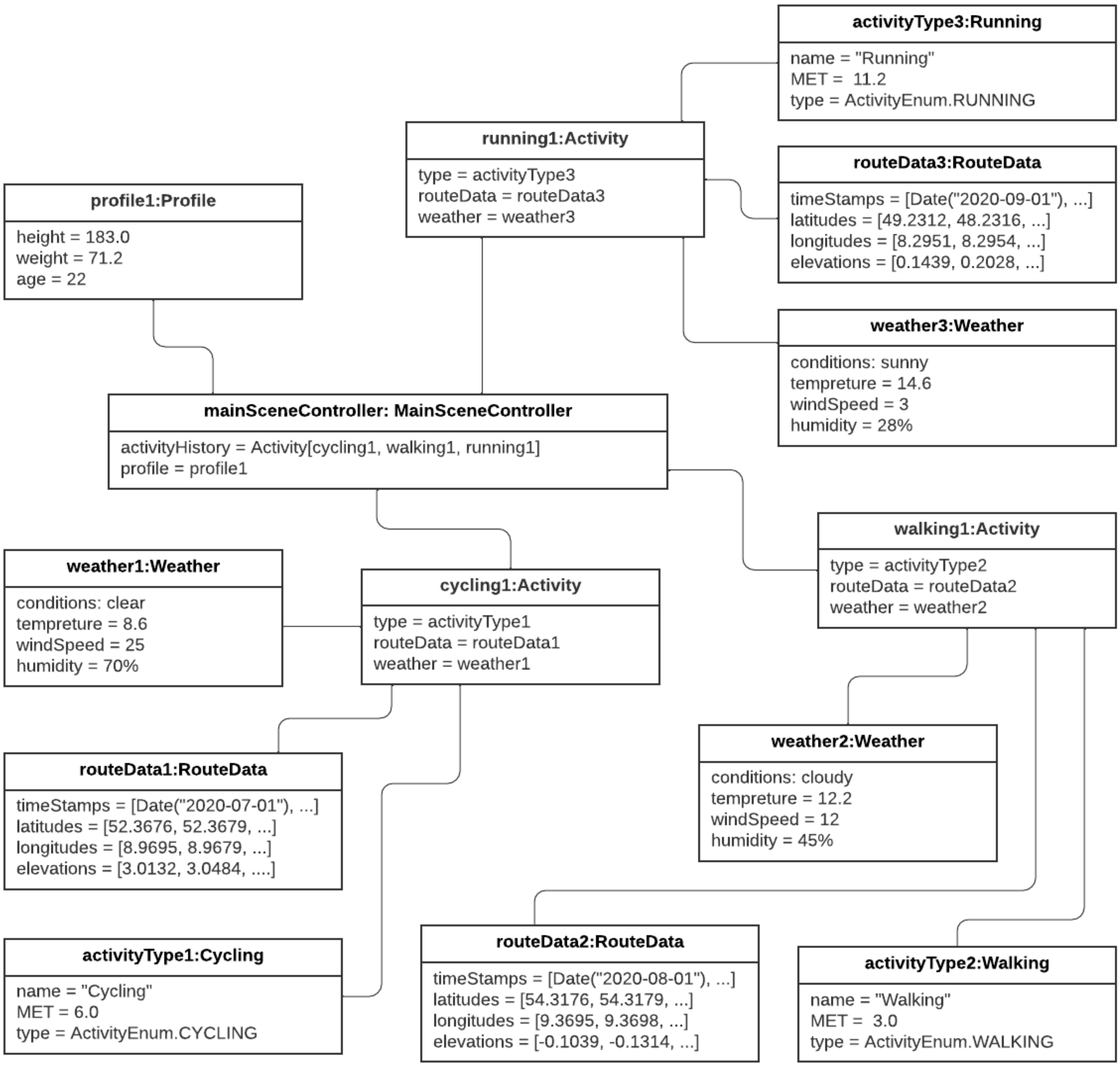
So, we chose for the <u>type</u> attribute of an *Activity* instance to be of the (abstract) type *ActivityType*, a class that “wraps” the *ActivityEnum*, instead of having it be of type *ActivityEnum* directly. Both solutions allow for easy extension of the support activity types. However, if we only used the enumerator, then this would not allow for the easy incorporation of the activity type-specific MET values. Furthermore, a class per activity type also allows for easy customisation per type, if this were ever to be required in the future.

ActivityEnum

This enumerator is used to present the user with a list of options from which he/she can choose from in order to assign a type to an activity. Any activity type that is not supported but is desirable, can be added here.

Object diagram

Author(s): Daniel Volpin

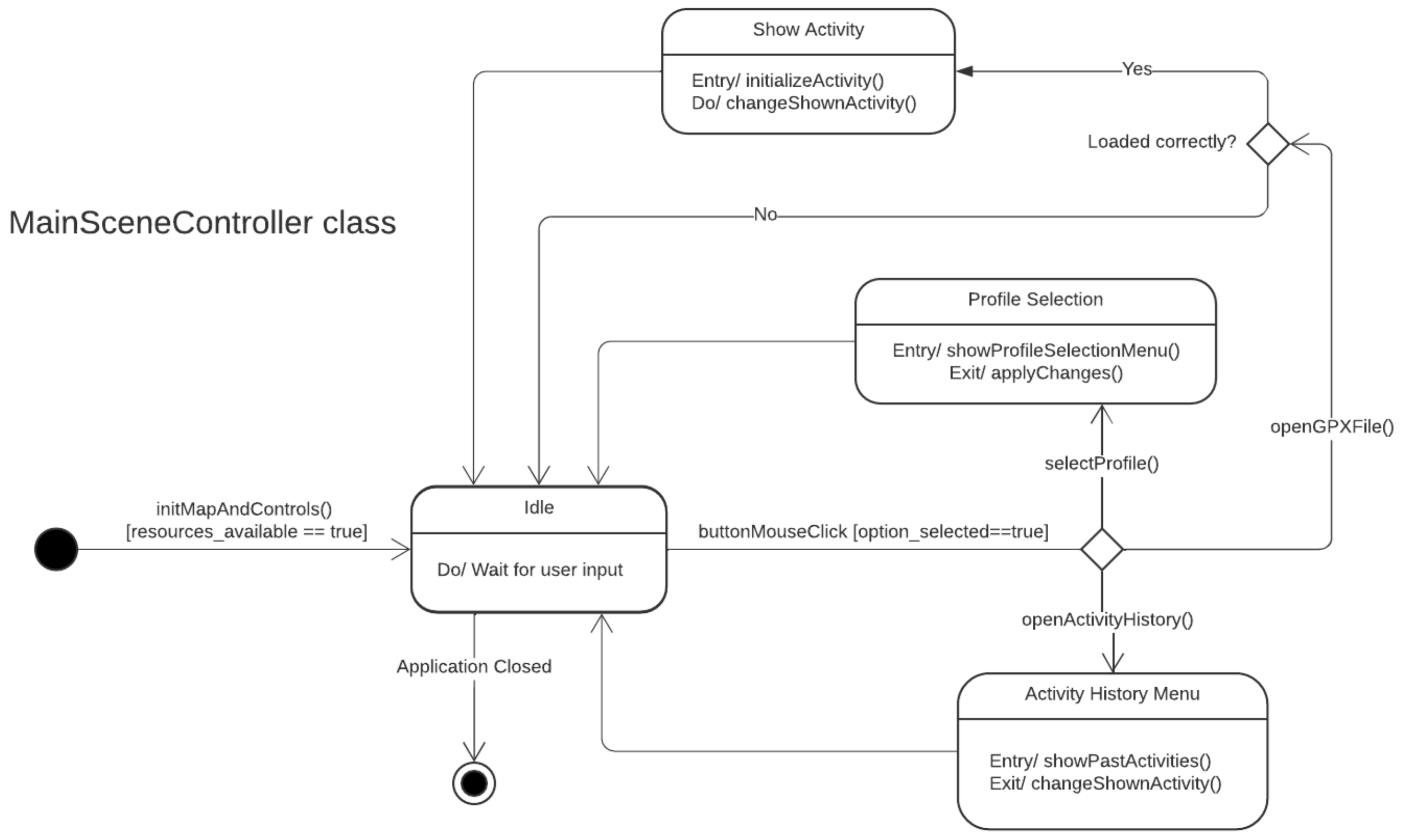


The following diagram represents our AerobixTracker system which shows instances of the system after the user has loaded three GPX files. The diagram has the following objects: **MainSceneController**, **Activity**, **RouteData**, **Weather**, **Profile**, and the **ActivityType**.

Once the *mainSceneController* instance is created, it composes a *Profile1* instance of the user chooses to input personal information. Also, it can be seen above that three *activity* instances (*walking1*, *biking1*, *running1*) were created from uploading three different GPX files. In turn, these *activity* instances compose and contain *weather* (*weather1*, *weather2*, *weather3*), *routeData* (*routeData1*, *routeData2*, *routeData3*), and *ActivityType* (*activityType1*, *activityType2*, *activityType3*) instances based on the corresponding *activity* instance.

State machine diagrams

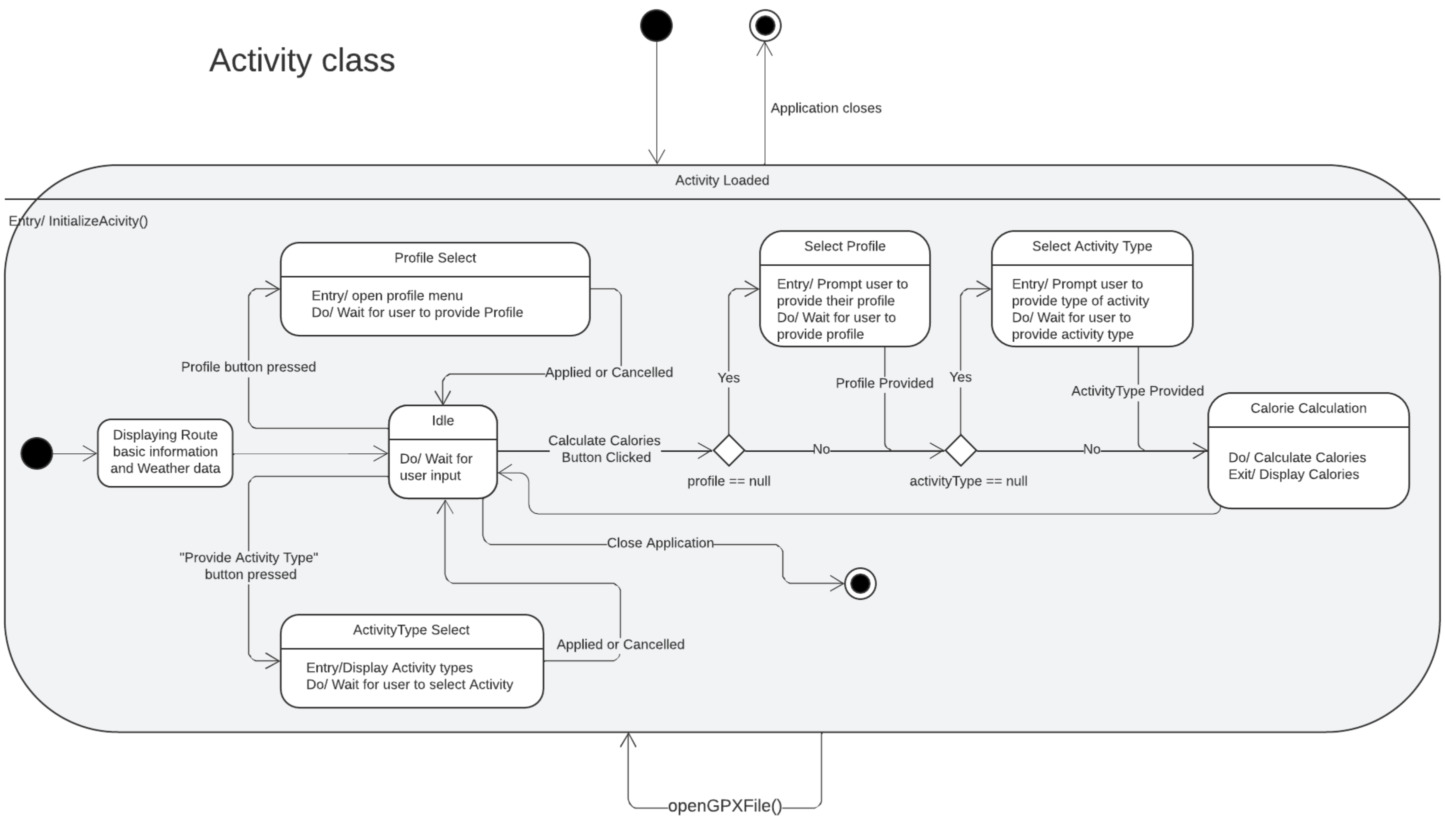
Author(s): Daniel Berzak and Osman Abdelmukaram


MainSceneController class
Show Activity
Entry/ initializeActivity()
Do/ changeShownActivity()
Yes
Loaded correctly?
No
Profile Selection
Entry/ showProfileSelectionMenu()
Exit/ applyChanges()
openGPXFile()
selectProfile()
initMapAndControls()
[resources_available == true]
Idle
Do/ Wait for user input
buttonMouseClick [option_selected==true]
openActivityHistory()
Application Closed
Activity History Menu
Entry/ showPastActivities()
Exit/ changeShownActivity()

The state machine diagram represents the behaviour of the **MainSceneController** class. We chose to implement a single class with the purpose of displaying the different parts of the application on the user interface. Because this class would then need access to all of the information that should be displayed, we made the design decision for this class to hold the state of the uploaded GPX files. Upon starting the application the function initMapAndController() is called and loads the starting scene of the application (map and menu bar). If everything is loaded correctly (meaning the resources were available, i.e. an internet connection), the application goes into idle state. Once there, the user has the option to choose one of 3 mouse click events.

If the user clicks on the profile button, he/she will be provided with a window in which there is the option to provide personal information such as: height, weight and age. In the case where the user chooses to load a GPX file (using the designated button) the class loads a new activity, recenters the map, and draws the tracked lines through the waypoints provided (changeShownActivity()). However, if the provided file is not in the correct format or another issue occurs such that the *track* can not be visualised on the map, the application returns back to its idle state, and does not change anything about the map. The third and final option the user has is pressing the activity history button, from which they are presented with all of the previously loaded activities (<u>activityHistory</u>). They can select any of these activities to show them again on the map.

Once any of those states is over the class will go back to its idle state and the user will be presented with the same 3 options again. The program only terminates when the user closes the program manually (an action that can only be done from the idle state).


Activity class
Application closes
Activity Loaded
Entry/ InitializeAcivity()
Profile Select
Entry/ open profile menu
Do/ Wait for user to provide Profile
Profile button pressed
Applied or Cancelled
Select Profile
Entry/ Prompt user to provide their profile
Do/ Wait for user to provide profile
Select Activity Type
Entry/ Prompt user to provide type of activity
Do/ Wait for user to provide activity type
Yes
Profile Provided
Yes
ActivityType Provided
Displaying Route basic information and Weather data
Idle
Do/ Wait for user input
Calculate Calories Button Clicked
No
No
profile == null
activityType == null
Calorie Calculation
Do/ Calculate Calories
Exit/ Display Calories
Close Application
"Provide Activity Type" button pressed
ActivityType Select
Entry/Display Activity types
Do/ Wait for user to select Activity
Applied or Cancelled
openGPXFile()

The diagram above represents the transition between states of the *Activity* class. This state machine diagram becomes relevant when the user provides a first GPX file. Then, we enter the composite state, and the entry event, <u>InitializeActivity()</u>, is performed and the instance of the class is initialised with the data provided by the GPX file. Any additional trigger of <u>openGPXFile()</u> event would lead to entering the Activity Loaded composite state again. After the initial state, the class transitions to the state where the route is displayed on the map, along with the weather data about the route (if a timestamp is attached to the GPX file).

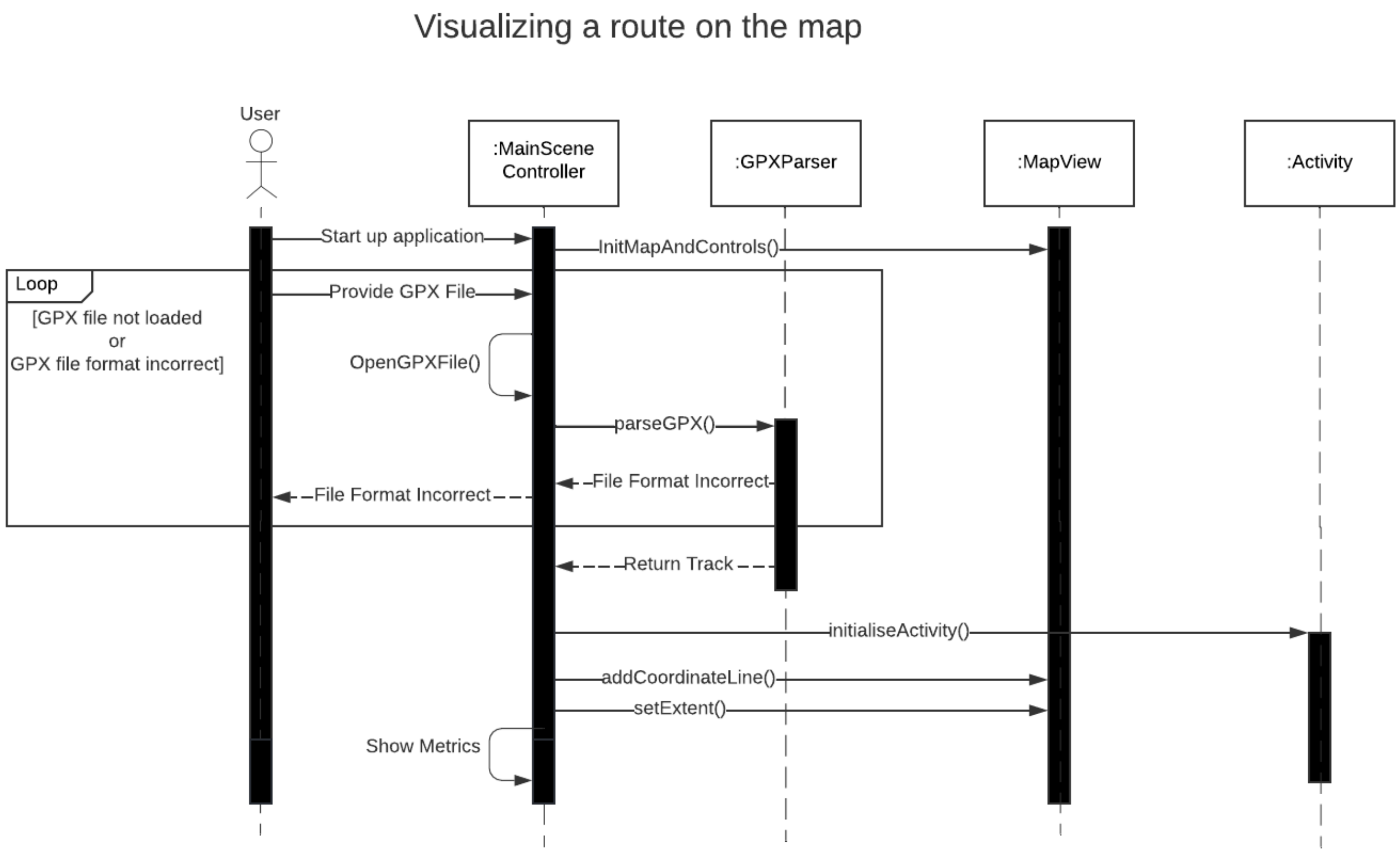
Following the next transition, the class enters the idle state, where it waits for the user to select one of the following events: <u>profile selection</u>, <u>activity type selection</u>, <u>calculation of calories</u>, or <u>closing the instance of application</u>. In case the user pressed the “provide activity type” button, the state of the class transitions to a state where the user can select an activity type, after it again waits for the user input. If the user applies his changes, or if he cancels, the *activity* instance returns back to the central idle state waiting for user input. A similar path occurs if the user presses the “select profile” button; the class transitions to a state where it opens a window to where the user can provide its profile. Once the user applies or cancels the changes, it returns to the central idle state.

The **Activity** class was made in a stateful manner where for most users it will simply remain idle while holding the relevant data, and in some cases receive input of an activityType and profile to provide the “calories burned” metric. We decided to make the providing of a profile and activity type optional. This design decision was made to give the flexibility for users to visualise the route without the necessity of their (private) data.

In the case of the “calculate calories button clicked” event, the *activity* instance wants to calculate the amount of calories burned. However, for this to be possible, it needs both the <u>profile</u> and the activity <u>type</u>. Therefore, the class checks if these exist. If the profile or the activity type do not exist, the user is prompted to provide it. Only once the user has provided both, the calories burned during the activity can be calculated by the class reaching the calorie calculation state. Once this is finished, it returns to the idle state.

Sequence diagrams

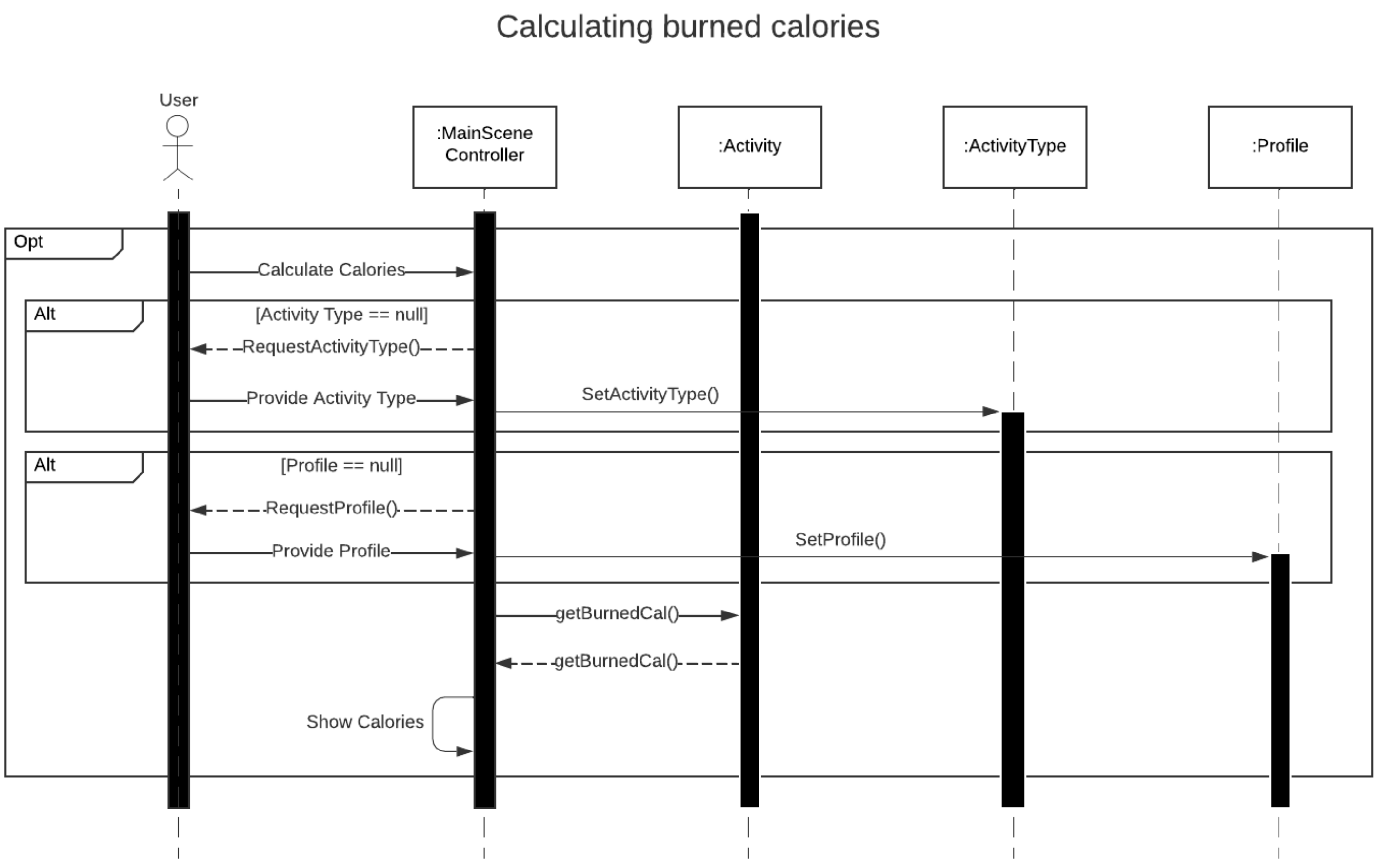
Author(s): Marco Deken and Osman Abdelmukaram



This sequence diagram starts off with the interaction between the user and the MainSceneController. The user starts up the application, upon which an instance of the MainSceneController is made, which then does a function call to initMapAndControls(). Then, once the instance of the MapView class is initialised, the user has the possibility to provide a GPX file. Alternatively, we could have made the design decision to force the user to first upload a valid GPX file, before we initialise the *MapView*. However, we decided that it would give a nicer feel to the application if the user would first be able to see the map, and get “access” to the basic functionality like *Profile*. Therefore, the MapView is first initialised, and if the user wants to unlock more functionality of our application, a GPX file has to be uploaded.

Once the user provides a GPX file, the MainSceneController will open the file, and then tell the newly created instance of GPXParser to parse the file. However, if the GPXParser fails, it will return an error stating that the file format is incorrect. The MainSceneController will pass on this message to the user. This design decision was made such that the user understands the provided file will not be visualised on the map. This will trigger the user to provide a different file, which is now hopefully in the correct format.

As soon as the user provides a file that can be interpreted correctly, the loop is ended and the GPXParser can return the **track** from the file. Then, an instance of the *Activity* class can be initialised, after which the addCoordinateLine() and setExtent() functions are called to visualise the **activity** on the map. Lastly, the MainSceneController shows the metrics in the right pane of the window, which concludes the series of events triggered by the user uploading a GPX file.



The sequence diagram above represents the optional situation where the user requests calories burnt to be computed. The interaction is initialised by the user clicking on the calculate calories button. The next sequence of the events checks if the activity type and profile objects are initialised with non-null values. Information provided in both objects is required for the accurate computation. If one, or both, of the objects are not initialised, the user is prompted to provide the input data that will create new instances of objects respectively. Once activity *type* and *profile* objects are created, the MainSceneController passes derived information to the getBurnedCal() function call. The Activity instance calculates the burned calories using the profile and activity type information, and returns it to the MainSceneController, who subsequently displays the information on the screen.

Implementation

Author(s): Daniel Volpin and Marco Deken

After we had decided upon the external libraries we would use, we started the implementation part of the project by trying to incorporate some of the libraries into the provided framework. We started by incorporating the library for the UI, for which we are using JavaFX. After we got this working, we incorporated the second vital library for our project, which is the GPXParser library. After incorporating this library into our project, we began to consider how we could build the rest of our GPX Manager around the framework, and we structured our UML models accordingly.

We used an MVC structure to implement the project based on our UML models (Model View Controller). The Model handles data logic, the View displays model information to the user, and the Controller directs the flow of data into a model object and updates the view when the data changes. The class diagram shows that we have represented our controller with the MainSceneController class, which will create instances of the composed classes from it once a GPX file has been uploaded.

Following the establishment of the main controller, we examined the functional and quality requirements model of the composed and associated classes in order to code the main entities. That would be the Activity, Weather, Profile, and RouteData classes in this case. We discussed which properties and functions would be required for our application after we had decided on the entity classes. As stated in the Class Diagram description of the entity classes, the properties and functions were chosen to be as meaningful to the user as possible. For example, for the Weather class, it was decided that the most useful properties to display to the user would be humidity, conditions, temperature, and wind speed. After deciding on the properties, get-functions for each property were created so that the MainSceneController could call them once an activity instance was created by uploading a GPX file. This approach was applied to all entity classes.

The menu on the left side of the border pane is one of the most important features of our application. The Menu was built with a VBox element that holds Buttons. Each Button has an event handler that records the mouse click of the user. The first menu Button is "Upload GPX file," which uses the FileChooser library to open the file explorer for the user. Then, to make things easier for the user, a file type filter was applied, so that the user could only upload GPX files. The filtering option eliminates the possibility of the user receiving an error as a result of uploading the incorrect file type. Furthermore, when the user hovers the mouse over the individual menu buttons, the colours change to make it easier for the user to see where the mouse is hovering.

As the “Upload GPX File” button gets pressed, as explained in the State Machine Diagram and Class diagram, an Activity instance is initialised. To then visualise this on the map, the Coordinates of the route of the Activity are used to create a CoordinateLine object. This is an object provided by the library we use for visualising the map (mapjfx). This object can be added to the MapView with a simple function call, which completes the “journey” from opening the GPX file to visualising the route on the map. Moreover, a zoom slider was added to the bottom of the border pane, allowing the user to easily zoom in and out. This is true if the user's trackpad is unable to use the 'Pinch Zoom' feature available on some laptop touchpads.

Furthermore, not only the route but also the metrics and weather information are displayed on the screen. This is accomplished using JavaFX, which injects a dedicated FXML file into the right side of the borderpane once the activity has been initialised. The FXML file's properties are changed based on the weather and route data metrics that will be displayed to the user. Labels were created for each property and then added to a VBox element added to the right side of the main border pane. When the activity instance is created, the right side of the FXML file is loaded, allowing for a more dynamic experience.

All in all, the main Java class that starts up the first window when the application is run, is the AppLauncher class. It can be found at *src/main/java/softwaredesign/AppLauncher.java*. A short video demonstrating the execution of our system can be found on youtube through the following link: https://youtu.be/pJJA8V0EOK4

The location of the Jar file that directly executes our program is located at *out/artifacts/software_design_vu_2020_jar/software-design-vu-2020.jar*. To generate the aforementioned Jar file, a manifest was added to the *build.gradle* file, which specifies the project's main class (AppLauncher). By doing so, the user was able to run the program by executing the ./gradlew file with the command "run".

Time logs

Assignment 2

Member	Activity	Week number	Hours
Osman	Work on state machine diagram	4	2
Marco	Work on state machine diagram	4	2
Daniel Berzak	Make class diagram	4	2
Daniel Volpin	Make class diagram	4	2
Marco	Researching and incorporating the map view library	4	4
Daniel Volpin	Researching and incorporating the map view library	4	5
Daniel Berzak	Researching and incorporating weather API	5	4
Osman	Researching and incorporating JSON parsing for weather class	5	5
Osman	Work on sequence diagram	5	3
Marco	Work on sequence diagram	5	6
Daniel Berzak	Work on class diagram	5	4
Daniel Volpin	Work on class diagram	5	4
Daniel Berzak	Work on state machine diagram	5	4
Osman	Work on state machine diagram	5	2
Marco	Visualising route on the map	4	2
Daniel Volpin	Improving map functionality	5	2
Daniel Volpin	Making buttons and showing metrics on the screen	5	3
Daniel Berzak	Make the final document	5	4
Osman	Make the final document	5	4
Marco	Make the final document	5	5
Daniel Volpin	Make the final document	5	4
Osman	Finding images and implementing basic UI for weather app	5	1

Osman	17
Marco	19
Daniel Berzak	18
Daniel Volpin	20
TOTAL	**74**