Assignment 3

Team number: 44

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

Summary of changes of Assignment 2

Author(s): Daniel Volpin

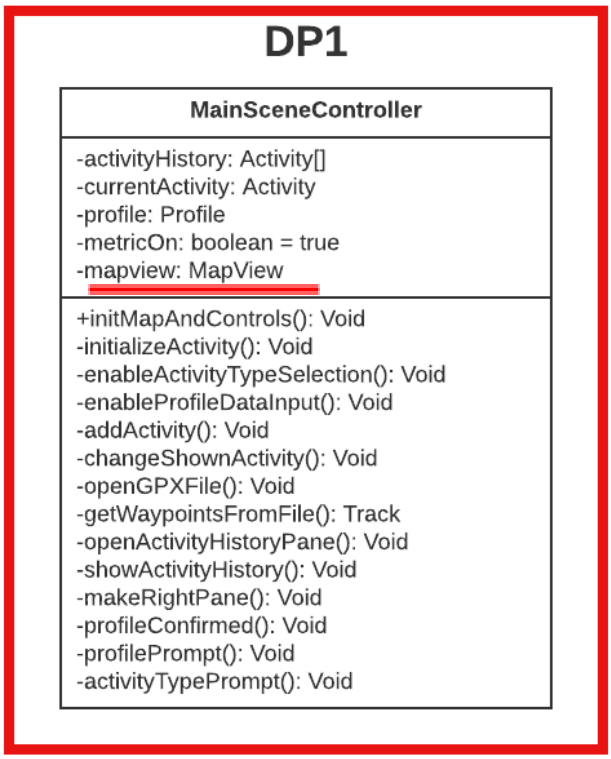
- Extended the **state machine** to include the following things:
    - Modifies relation arrows that were done incorrectly
        - Missing flow and wrong arrow format
    - Added additional description
    - Changed the modeling of calculateCalories()
- Extended the **class diagram** to include the following things:
    - An abstract class for four activity types: Running, Cycling, Waling, and Roller Skating
    - A factory method for the activity type class
    - Renamed relationships to have a more meaningful description
    - Given extra reasonings for attributes being public
- Extended the **sequence diagram** to include the following things:
    - Message calls and replies name conforms to the operations specified in the class diagram
    - Added additional description
    - Changed the modeling of calculateCalories()

Application of design patterns

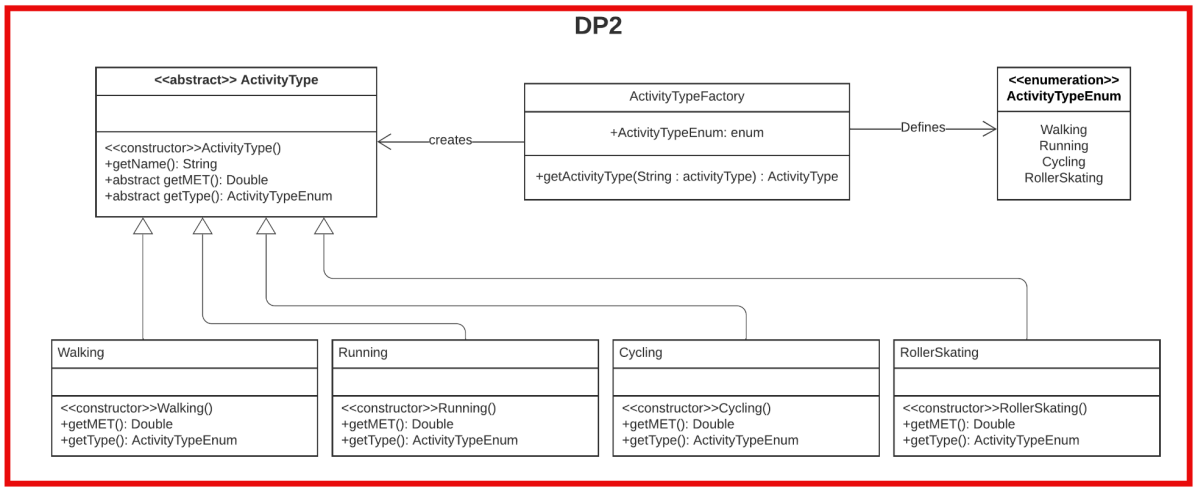
Author(s): Marco Deken, Daniel Volpin and Osman Abdelmukaram

Design Patterns:

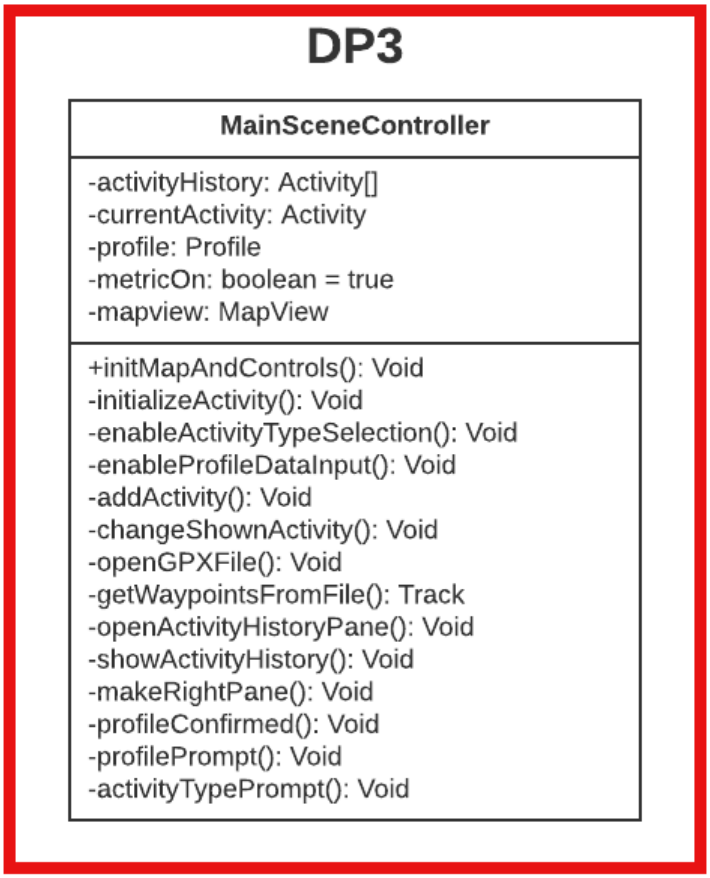
ID	DP1
Design pattern	Observer (MVC architectural pattern)
Problem	We want to visualize a map on the screen that is responsive to the user input. So, something has to listen to user input, which then manipulates the mapview.
Solution	The MVC architectural pattern solves our problem. We create a controller class called MainSceneController that listens to requests from the user. This class then manipulates all the entities (i.e. Activity, Profile, etc.) in the correct way. Then, the view object called MapView can be updated to show the information from the Model
Intended use	The most important part of our application, the mapView and everything around it, is structured as the MVC.
Constraints	-
Additional remarks	-



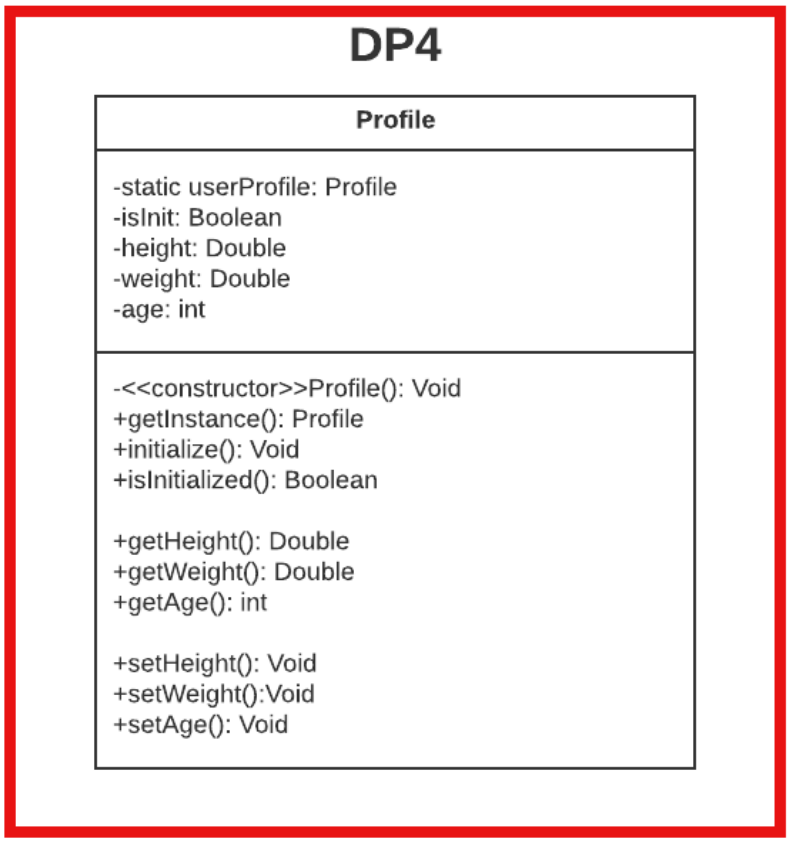
ID	DP2
Design pattern	Factory Method
Problem	When assigning an activity type to an Activity, we want to create an instance of a specific class based on the type that the user inputted. The activityType-specific class then has properties that are specific to the activity type. Somewhere, there has to be a switch statement to decide which activity type class to create.
Solution	Using the Factory Method, we create a Factory class which takes care of all the activity type object creation for us. We just pass the user-inputted activity type name to the Factory's getInstance() method, and the Factory handles the rest.
Intended use	We are going to use the instantiated classes, when the user wants to calculate his calories. Then, the classes hold the properties of the activityType to be able to calculate the amount of burned calories.
Constraints	-
Additional remarks	-



ID	DP3
Design pattern	Inversion of Control (IoC)
Problem	When the user wants to calculate the calories burnt from doing a specific activity, they should be able to fill in their age, height and weight in any order. In addition, the user should be able to select any menu item and proceed with a new event without worrying about the order of execution.
Solution	Utilizing a framework that utilizes event handlers from JavaFX to handle the flow of the program.
Intended use	Achieve loose coupling that lets the user decide the flow of the program based on different mouse events that are registered throughout the program to increase modularity of the program.
Constraints	- makes code difficult to trace (read)
Additional remarks	-

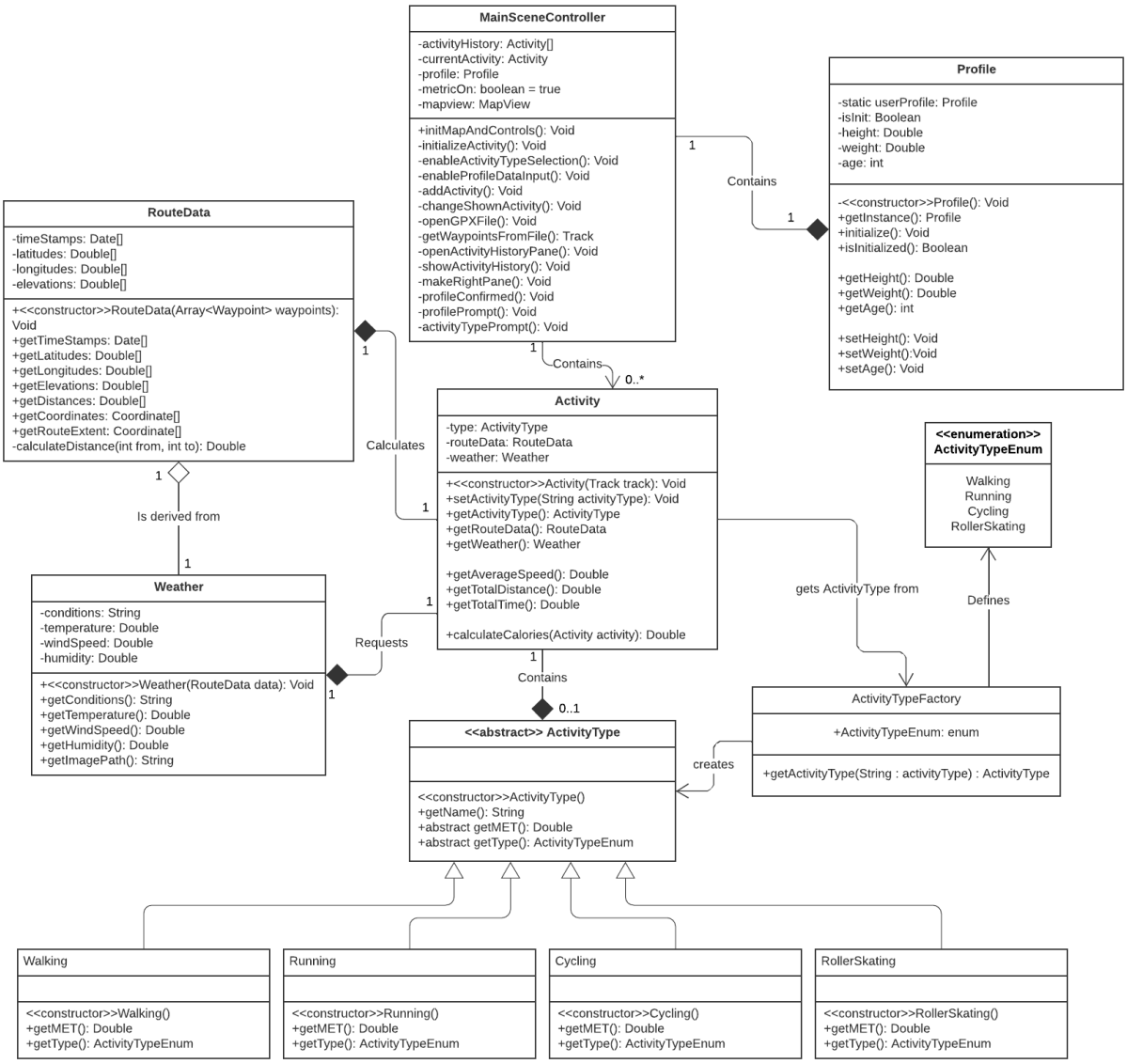


ID	DP4
Design pattern	Singleton
Problem	Since the usage of the application is personalized, there is only a single user of the application per session. However, the profile class was instantiable, enabling the existence of the multiple profile objects. The instance profile had to be created as a global variable, adding additional complexity to the system.
Solution	By applying the singleton design pattern to profile class, we ensured the existence of a single instance of the profile in the system.
Intended use	Enabling users to set their profile characteristics once per session, and keep using them throughout different instances of activities. This achieves controlled access to the user profile, and reduces complexity in the code.
Constraints	- Long chain function calls when accessing profile members
Additional remarks	- Since the instantiation of the object is performed statically, the setter methods are used to apply values to the object.



Class diagram

Author(s): Daniel Volpin, Marco Deken, Daniel Verner and Osman Abdelmukaram



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 initialized. 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 getWaypointsFromFile() 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 initialized activities, the **activity** that is currently visualized on the map is stored in the currentActivity 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 the **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 only once. Therefore, we have decided to make the **profile** a singleton class. This would allow a user to provide GPX files of activities that were performed by the user, and then be able to choose the type of activity to calculate calories burnt for each GPX file.

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, an array of **waypoints** is needed (from the GPX file). Then, the positional data from the **waypoints** are 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 initialized. After the *activity* is initialized and visualized 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 traveled distance (getTotalDistance()), the total traveled time (getTotalTime()), the average speed in meters per second (getAverageSpeed()), and the calories burnt for each activity (calculateCalories()). Furthermore, if in the future one would

want to show more metrics, the list of metrics can be easily extended 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 to initialize the above mentioned properties. Moreover, from these properties the user can view the distances and coordinates points through the getDistances(), 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 changeShownActivity() function is called from the class to center the map around the coordinates given by the GPX file.

Also, the **RouteData** has a shared aggregation with the **Weather** class which utilizes 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. The icons representing the weather conditions are stored in the resources directory, and by using the “getImagePathFunction()” function we retrieve the icon representing the weather conditions. 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 type 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 *Walking*, *Running, RollerSkating, Cycling* classes. These non-abstract classes have to implement the abstract properties of *ActivityType*, which are represented with the following getter functions: getMET(), and getType(). The MET property stands for Metabolic Equivalent, which is the activity-specific variable necessary to calculate the amount of calories burned during the activity. The getType() method returns the type of the class as an ActivityTypeEnum. The only non-abstract property is the getName(), which returns the names of the classes that are shown to the user as options when selecting an activity type. These getName() functions do not have to be implemented by every non-abstract extension of the ActivityType class, as they work automatically based on the getType() function (which is an abstract function in the ActivityType class and therefore has to be implemented in the specific *Walking* class for example.

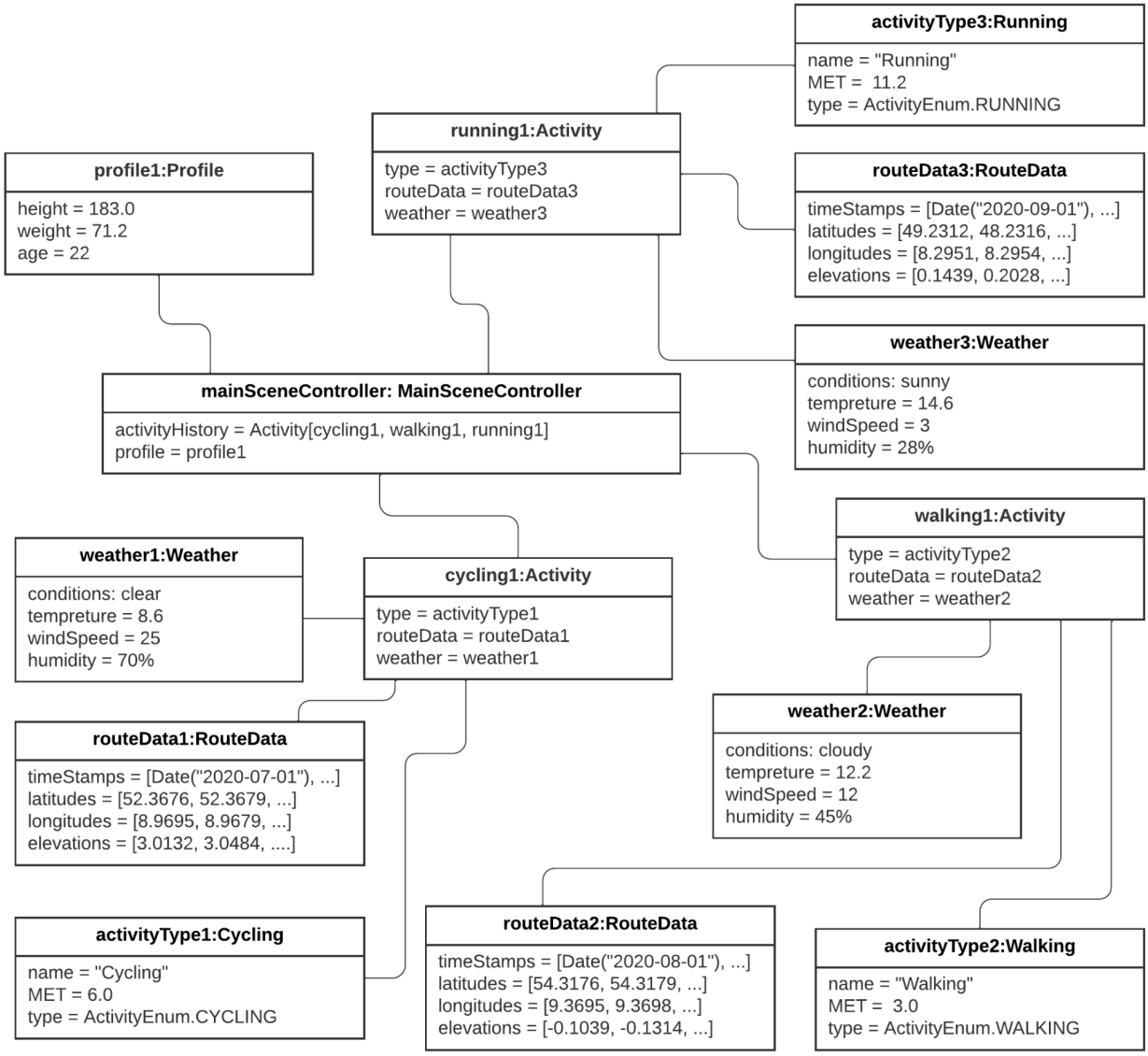
We made the design decision to choose for the type attribute of an *Activity* to be of the (abstract) type *ActivityType*, a class that "wraps" the *ActivityEnum*, instead of having it be of type *ActivityEnum* directly. Both solutions allow for easy extension of the supported activity types. However, if we only used the enumerator, then this would not allow for the easy incorporation of the activity type-specific MET values in a class (like we do now). Furthermore, a class per activity type also allows for easy customisation per type, if this were ever to be required in the future.

ActivityTypeEnum

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, and then a class to extend the abstract ActivityType has to be made. The fact that a developer would have to add a new activity type to the ActivityTypeEnum, is also directly enforced by the fact that if the developer instantiates a new non-abstract ActivityType class, i.e. HorseRiding, then the abstract function getType() has to be implemented, which would have to return an activity in the ActivityTypeEnum.

Object diagram

Author(s): Daniel Volpin

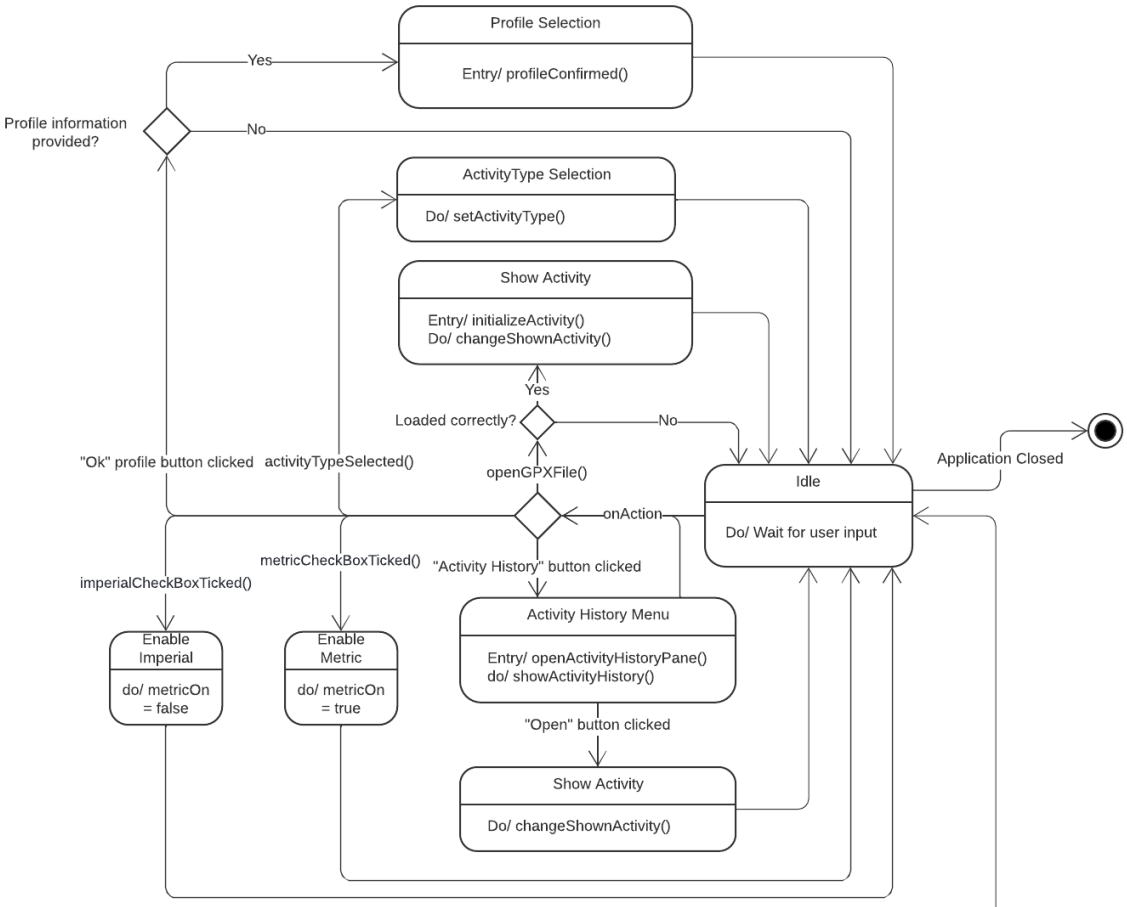


The following diagram represents our AerobixTracker system which shows instances of the system after the user has loaded three GPX files. The diagram has the following objects: **MainSceneController**, **Activity**, **RouteData**, **Weather**, **Profile**, and the **ActivityType**.

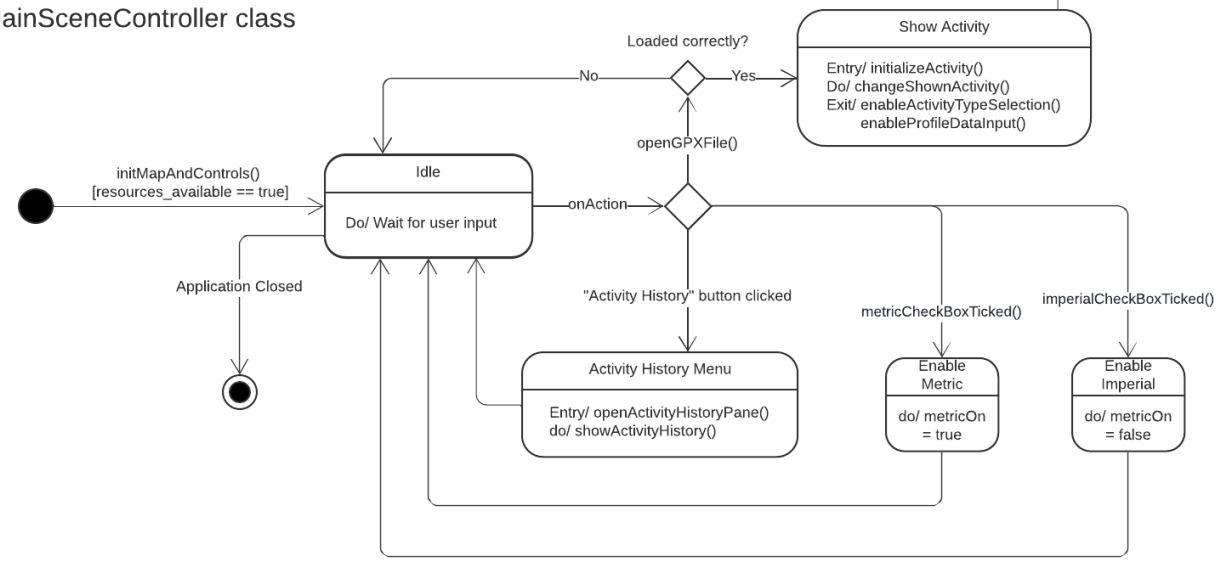
Once the *mainSceneController* instance is created, it composes a *Profile1* instance of the user chooses to input personal information. Also, it can be seen above that three *activity* instances (*walking1*, *biking1*, *running1*) were created from uploading three different GPX files. In turn, these *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 Verner and Daniel Volpin



MainSceneController class



The state machine diagram on the previous page represents the behavior of the **MainSceneController** class. We chose to implement a single controller 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, user profile, and the currently selected activity.

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 several mouse click events.

As the behavior of our application has been extended from the previous assignment (with the addition of switching between imperial/metric systems of measurement, access to activity history etc..) we extended the state machine to cover all the different mouse click events offered on the interface of the app that are handled by this class.

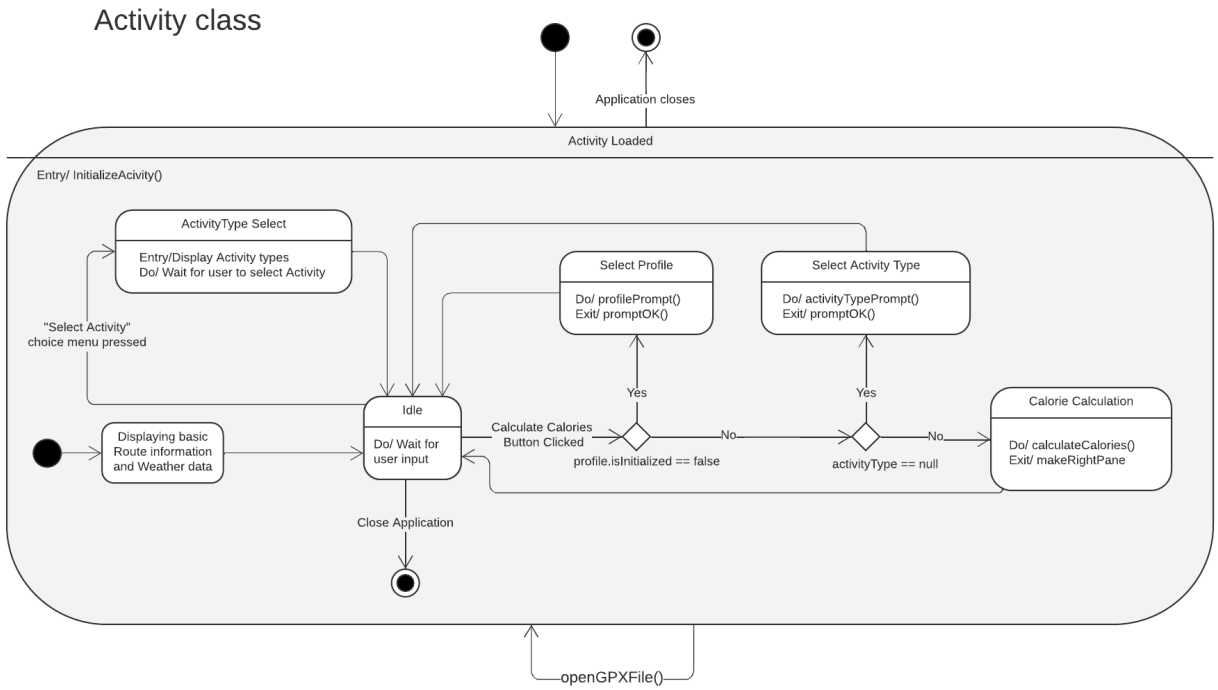
Once the app is loaded the user is provided with a map view and a menu bar on the left side of the app in which the user can select to load a GPX file. In the case where the user chooses to load a GPX file (using the designated button) the class loads a new activity, recenters the map, and draws the 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 *waypoints* can not be visualized on the map, the application returns back to its idle state, and does not change anything about the map.

In the case where the first GPX file was loaded correctly two functionalities become available that are not available in the initial idle state: The **profile** menu, and the **ActivityType** dropdown menu (which are made available in the exit activity of the first Show Activity state, which are enableActivityTypeSelection() and enableProfileDataInput()). The fact that two more functionalities are enabled, can be seen in the model by seeing how the state shifts to the top part of the diagram. Then, a user can add a profile or select an activity type. For adding their profile the user is provided with text fields in which they are able to provide their personal information (height, weight and age) and once they press on the button “ok” their profile is saved for the specific activity they have loaded.

Another feature we implemented is the activity history feature, which occurs once the user has pressed on the “Activity History” button. Each time the user uploads a GPX file, it is added dynamically (activityHistory) and presented nicely with old values from route information. Also, the user can select any of the old activities and reload them in the main scene. However, the user does not have to “Open” any of the old activities, but the other options from the idle state are still available (as modeled in the diagram).

The last feature the MainSceneController class handles is the imperial/metric checkbox. When the checkbox is changed (imperial to metric and vice versa) the metrics on the right pane change dynamically.

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



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, initializeActivity(), is performed and the instance of the class is initialized with the data provided by the GPX file. Any additional trigger of openGPXFile() 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: activity type selection, calculation of calories, or closing the instance of application. In case the user pressed the “select activity” choice menu, the state of the class transitions to a state where the user can select one of four activity types. Given that the selection is provided in a form of choice box, after clicking on the desired activity type, the *activity* instance returns back to the central idle state waiting for user input.

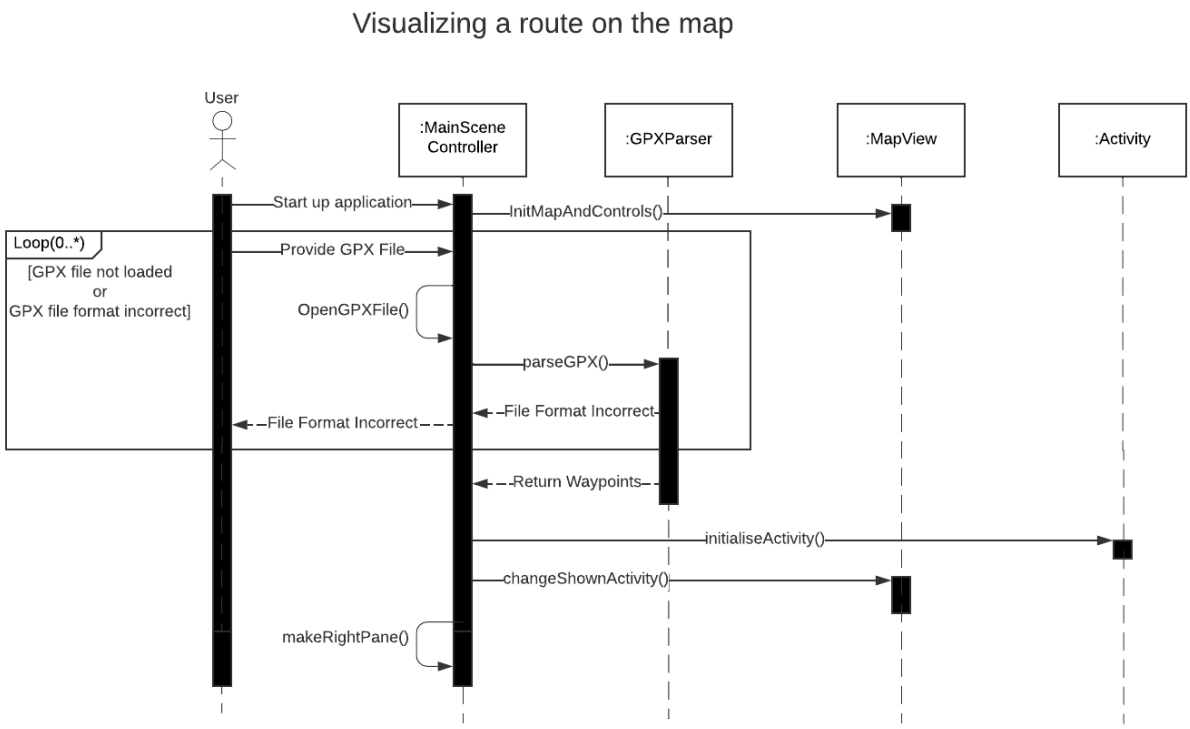
The **Activity** class was made in a stateful manner where for most users it will simply remain idle while holding the relevant data, and in some cases receive input of an activityType and profile to

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 visualize 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 profile information to be provided, and the activity type initialized. Therefore, the class checks if these conditions are met. If not, the user is prompted to provide inputs respectively. 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

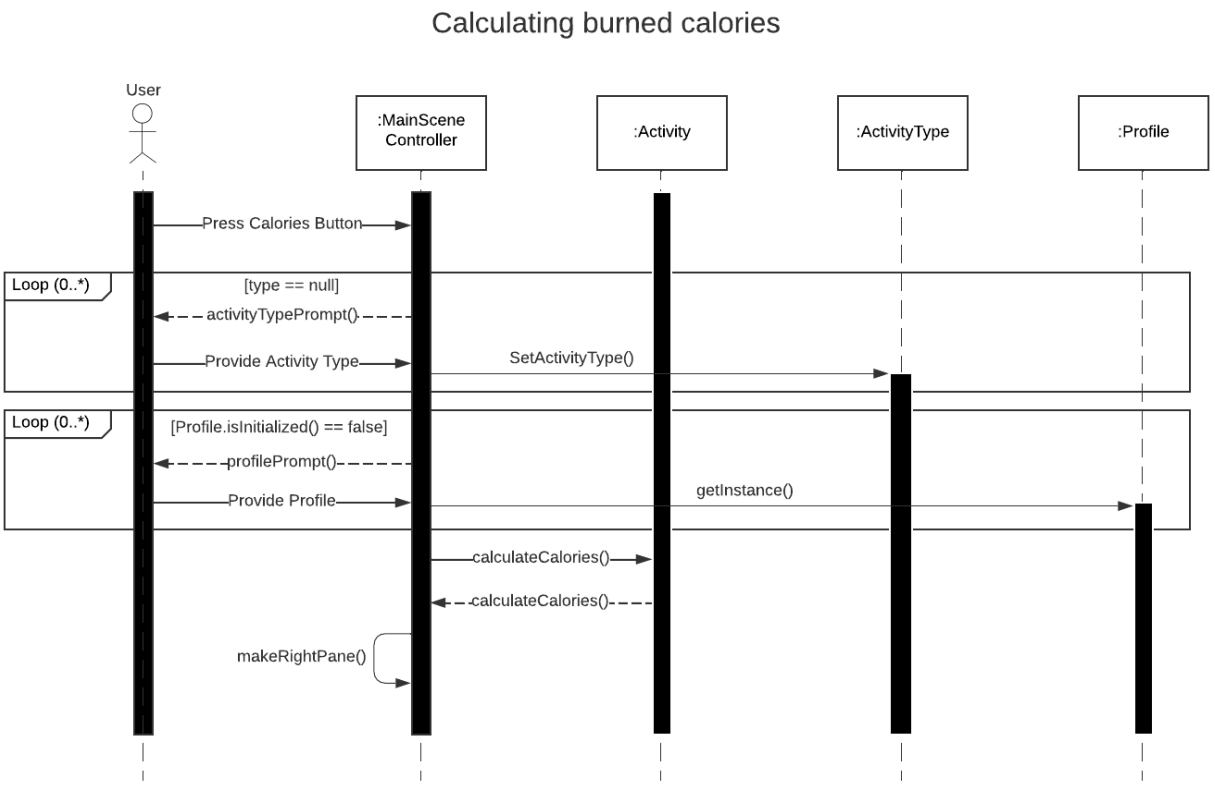


This sequence diagram starts off with the interaction between the user and the MainSceneController. The user starts up the application, upon which an instance of the MainSceneController is made, which then does a function call to initMapAndControls(). Then, once the instance of the MapView class is initialised, 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. Hence the loop of

interactions of the user providing a GPX file and the program checking for a correct file, can be executed 0 times if the user never provides a GPX file, or more as long as the file is incorrect.

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 **waypoints** from the file. Then, an instance of the *Activity* class can be initialised, after which the changeShownActivity() function is called to visualise the **activity** on the map. Lastly, the MainSceneController shows the metrics in the right pane of the window by calling the makeRightPane() function, which concludes the series of events triggered by the user uploading a GPX file.



The sequence diagram above represents the situation where the user requests the amount of calories burnt to be computed. The interaction is initialised by the user clicking on the calculate calories button. In order for the program to be able to calculate the total amount of calories burned during the activity, the activity type has to be set. So, if the value of type is null, we enter

the loop and let the user know that he first has to provide an Activity Type by calling activityTypePrompt(). Then, as long as the selection of an activity is not successful, it will not be possible to continue calculating the calories. If the type is non-null, the loop is broken and the second requirement of calculateCalories is checked.

The second requirement is that also a profile has to exist in order to be able to calculate the total amount of burned calories. As the Profile class is a singleton class, we check for its existence using the Profile.isInitialized() function call. As long as this is false, the user will be sent a prompt (profilePrompt()) to let the user know that it is momentarily not possible to calculate the calories, as the user first has to give his/her profile information. If the user provides their profile information for the first time, the getInstance is called and the profile is initialized.

Once activity *type* and *profile* objects are created, the MainSceneController passes derived information to the calculateCalories() function call in the Activity class. 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 by reloading the right pane (makeRightPane())

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 colors 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 initialized. To then visualize 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 visualizing 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 visualizing 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 initialized. 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.

In addition, we created a "Select Activity" button, which is disabled by default but becomes active once the user uploads a GPX file. Running, cycling, walking, and roller skating are the four types of activities available to the user (these activities extend the ActivityType abstract class). Following that, when the user enters their weight, height, and age and clicks the "ok" button. When an activity is loaded with all the different information they are also provided with a “Calculate Calories” button which if pressed, a function called calculateCalories() is called to display the calories burned based on the activity that was chosen and the profile provided. This will only occur in the case that a user provided the information required. If not, they will be prompted with a request for the missing information. Based on the current GPX file (and with the corresponding data provided by the user), this feature informs the user of how many calories they have burned from participating in a specific activity.

Another feature that was added was the conversion of the currently displayed route and weather data from metric to imperial. We're using a private boolean to keep track of whether the current data is metric or imperial (it's "true" by default). On the bottom left pane, the user can see two check boxes that allow them to select whether they want weather and route information displayed in imperial or metric units. The metric checkBox is selected by default. Furthermore, to reduce user-side errors, each checkBox is disabled when the current box is checked. This feature broadens the application's accessibility; users from various continents can use our application without feeling left out.

Finally, whenever a user uploads a GPX file, an activity instance containing the route and weather data is added to the array of type Activity, which is then added to an anchor pane

that appears when the user selects the fourth menu item "Activity History." The right anchor pane is extended from the application's right side, and the user can see their activities numbered there (starting from 1). If a user wants to view an older uploaded activity, they can click on a "Open" button within the individual titled panes to open the right pane that displays all of the route and weather information that is initially displayed when a user uploads a GPX file.

The AppLauncher class is the main Java class that launches the first window when the application is launched. It's in src/main/java/softwaredesign/AppLauncher.java. A short video demonstrating our system's execution can be found on YouTube at the following link: https://youtu.be/n5B-b21ZubA

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 3

Member	Activity	Week number	Hours
Daniel Berzak	Creating activity history feature	6	2
Daniel Volpin	Creating activity history feature	6	2
Osman	Creating profile information input container	6	2
Daniel Volpin	Integrating represenation of activity in metric and imperial units	6	2
Marco	Creating prompt windows	6	1
Marco	Fixing right side pane container	6	1
Dainel Volpin	Fixing profile information input container	6	1
Osman	Adding new icons for weather conditions	6	2
Marco	Completing calorie calculation feature	7	1.5
Daniel Berzak	Fixing button handlers	7	1
Marco	Code refactoring and cleaning	7	2
Daniel Volpin	Implementing observer design pattern	7	2
Daniel Berzak	Implementing factory method design pattern	7	2
Osman	Implementing singleton design pattern	7	2
Daniel Berzak	Work on class diagrams	7	1.5
Daniel Volpin	Work on class diagrams	7	1.5
Daniel Berzak	Work on state machine diagrams	7	1.5
Osman	Work on state machine diagrams	7	1.5
Marco	Work on sequence diagrams	7	3
Osman	Work on sequence diagrams	7	2
Daniel Volpin	Improvement of UI design	7	1
Daniel Berzak	Make the final document	7	4
Daniel Volpin	Make the final document	7	4
Marco	Make the final document	7	4
Osman	Make the final document	7	4

Osman	13.5	
Marco	12.5	
Daniel Berzak	12	
Daniel Volpin	12.5	
TOTAL	**51.5**	