Assignment 2

Team number: 22
Team members

Name	**Student Nr.**	**Email**
Martynas Rimkevičius	2708302	m.rimkevicius@student.vu.nl
Erik Vunsh	2696857	e.vuns@student.vu.nl
Baher Wahbi	2707520	b.wahbi@student.vu.nl
Abhisaar Bhatnagar	2694178	a2.bhatnagar@student.vu.nl

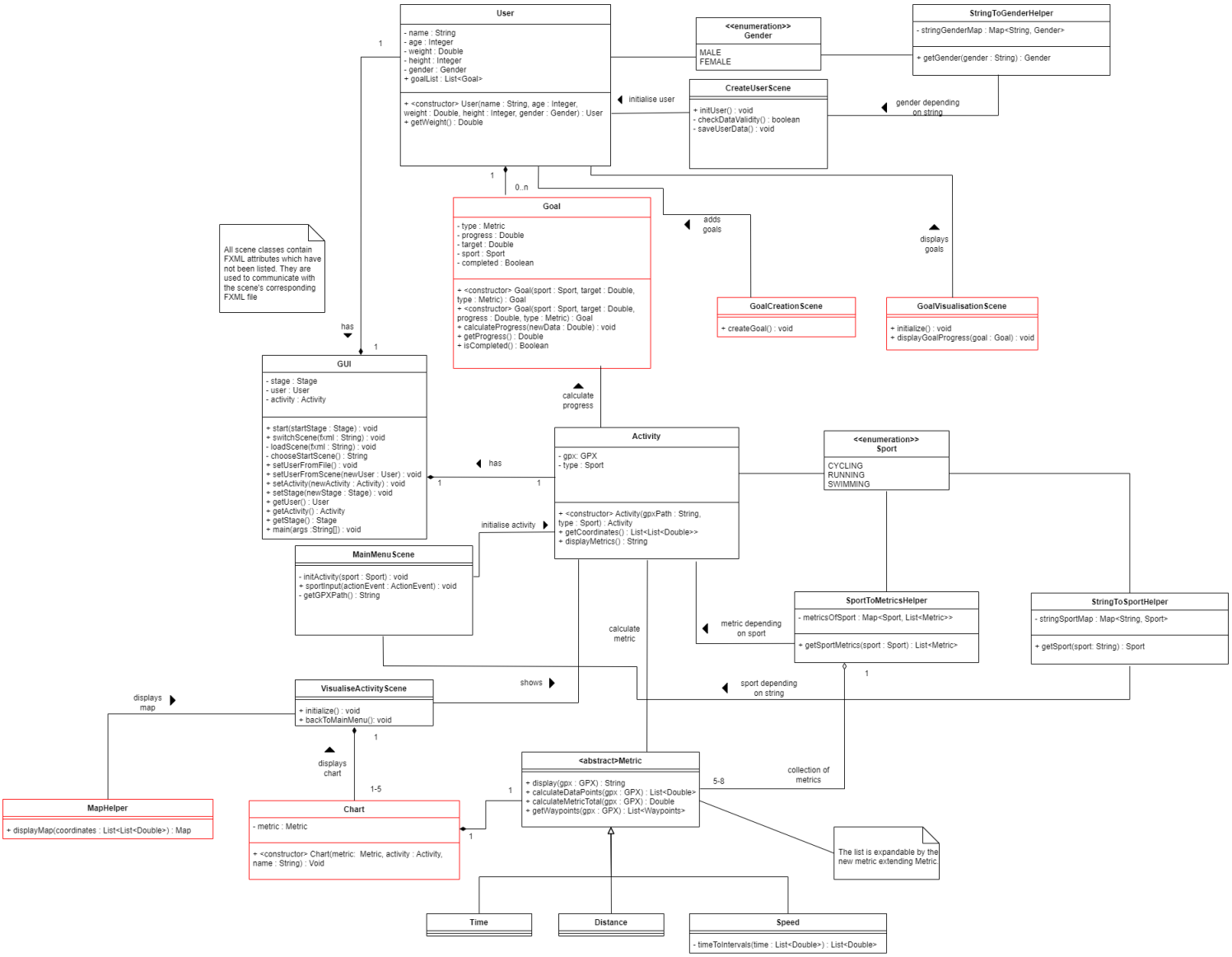
Implemented features

ID	**Short name**	**Description**
F1	GUI	Graphical User Interface to interact with all the features.
F2	Metrics	A list of all available metrics. For example: ● Elevation ● Power ● Distance
F3	Developer Extensibility	A developer can expand the metric list and add newly required metrics. Upon adding a new sport the developer chooses relevant metrics from the metrics list. Example: The addition of swimming as a sport would require new metrics “pace per 100m” and “stroke length” while also making use of existing metrics such as distance and calories burnt.
F4	Account System	Users are prompted to input their: ● Weight ● Height ● Gender ● Name ● Age This information will be used to calculate metrics which need user specific data such as calories burnt. This increases the usability of the app as each user does not have to input their information repeatedly with every gpx analyzation. The User object which is used to represent the account system will also hold the user’s goals.

Used Modelling tool: diagrams.net

Class diagram

Author(s): Martynas Rimkevičius & Baher Wahbi



(Use this link to view the diagram)

User

User contains personal user data and goals. This data is used to calculate user-dependent metrics such as calories burnt which requires the application user's weight. **User** is stored locally in a JSON file to improve the flow of the app as data is only input once.

Attributes

- name : String
- age : Integer
- weight : Double
- height : Integer
- gender : Gender
- goalList : List<Goal>- List of all the **Goal** objects that the user has set.

Operations:

- <constructor> User(name : String, age : Integer, weight : Double, height : Integer, gender : Gender) : User
- getWeight() : Double

Associations

- **GUI** composition - The **GUI** initialises the *user* : **User**. Since there is only one instance of **GUI** there can only ever be one instance of a **User**.

- **User** has a one-way association with **StringToGenderHelper** from which it gets a **Gender** based on a string**.**
- **User** has an enumeration association with **Gender**.

<abstract> Metric

Metric encapsulates the calculation methods of each metric from the GPX file.

Operations

- display(gpx : GPX) : String - returns a string containing generalised data of the metric, thus providing a way to display any metric.
- calculateDataPoints(gpx : GPX) : List<Double> - extracts or calculates points of the metric and returns a list of those points.
- calculateMetricTotal(gpx : GPX) : Double - calculates the average or the sum total of the metric and returns a single value.
- getWaypoints(gpx : GPX) : List<WayPoint> - used by all metrics and activity to extract the waypoints from the GPX object, which contain relevant data points.

Association

- **Metric** has a shared association with **SportToMetricsHelper**, which contains from 5 to 8 **Metric** objects for each **Sport** object in a map.
- **Metric** is a generalisation of all metrics such as **Time**, **Distance**, **Speed**

Time

Time is a class that contains all the methods necessary to extract the time and time points of the activity from the GPX object.

Operations

- display(gpx : GPX) : String - returns a string of total time spent doing the activity.
- calculateDataPoints(gpx : GPX) : List<Double> - creates a list of time points from the GPX object.
- calculateMetricTotal(gpx : GPX) : Double - returns the time difference between first and last points of the list calculated by calculateDataPoints()

Distance

Distance is a class that contains all the methods necessary to extract the distance covered of the activity from the GPX object.

Operations

- display(gpx : GPX) : String - returns a string of full distance covered in the activity.
- calculateDataPoints(gpx : GPX) : List<Double> - creates a list of distances between each WayPoint point extracted from a GPX object.
- calculateMetricTotal(gpx : GPX) : Double - returns a sum of all the distance points from the calculateDataPoints() method

Speed

Speed is a class that contains all the methods necessary to extract the speed of the activity from the GPX object.

Operations

- display(gpx : GPX) : String - returns a string of the average speed of the activity.
- calculateDataPoints(gpx : GPX) : List<Double> - creates a list of speed values calculated by finding speed of each point (distance point / time interval of that distance)
- calculateMetricTotal(gpx : GPX) : Double - returns an average of all speed points in the list returned by calculateDataPoints() method
- timeToIntervals(time : List<Double>) : List<Double> - converts **Time** metric returned list into a list of intervals between time points.

Activity

Activity is the class that is representative of the whole GPX file, meaning it gets the extracted metrics depending on the sport.

Attributes

- gpx : GPX - object containing information about the activity metrics.
- type : Sport - enumerator variable that describes which sport is selected for this activity.

Operations

- <constructor> Activity(gpxPath : String, type : Sport) : Activity- creates the activity while parsing the GPX file to the object.
- getCoordinates() : List<List<Double>> - extracts all coordinates from the *gpx* and puts them in a list thus preparing them to display on the map.
- displayMetrics() : String - puts all metric totals of the activity for the particular sport into a single string.

Association

- **Activity** has a one-way association with **SportToMetricsHelper** from which it gets a list of **Metric** objects that correspond to the provided **Sport.**
- **Activity** has an enumeration association with **Sport.**
- **Activity** has a one-way association with **Metric** class, from which it gets the strings to display data.

SportsToMetricsHelper

SportsToMetricsHelper is a helper class that provides corresponding **Metric** objects depending on the **Sport** object.

Attributes

- metricsOfSport: Map<Sport, List<Metric>> - map that contains translation from **Sport** object to a list of **Metric** objects.

Operations

- getSportMetrics(sport : Sport) : List<Metric>- depending on the provided **Sport** object, returns a list of **Metric** objects.

Association

- **SportsToMetricsHelper** has an enumeration association with **Sport.**

<enumeration> Sport

Sport is an enumerator class that contains all the available sports in the application.

Attributes

- CYCLING
- RUNNING
- SWIMMING

GUI

The entry point for JavaFX applications is the **GUI** class which extends the abstract **Application** class provided by JavaFX. It handles the loading/switching of scenes and allows different parts of the application to interact with the current user and activity.

Attributes

- stage : Stage - The primary stage for this application, onto which the application scenes can be set.
- user : User - The *user* using the application.
- activity : Activity - The most recent *activity* the application user has imported in this session. Is null until an *activity* is imported.

Operations

- start(startStage : Stage) : void - The main entry point for all JavaFX applications. Prepares the stage and sets the first scene.
- switchScene(fxml : String) : void - Switches from one scene to another on the stage.
- loadScene() : void - Loads a scene from an fxml file into a JavaFX *Parent*.

- chooseStartScene() : String - Chooses the first scene to set based on whether user data exists in JSON or not and sets the *user* if JSON exists.
- setUserFromFile() : void - Sets the *user* with data from the JSON file “user-data”.
- setUserFromScene() : void - Sets the *user* with data from the scene ‘CreateUser.fxml”.
- setActivity(newActivity : Activity) : void
- setStage(newStage : Stage) : void
- getUser() : User
- getActivity() : Activity
- getStage() : Stage
- main(args : String[]) - Launches the BEAM app.

Association

- **GUI** has one-to-one composite association with **Activity** class, which object it contains.
- **GUI** has one-to-one composite association with **User** class, which object it contains.

CreateUserScene

CreateUserScene is the controller of the scene “CreateUser.fxml”, a first-time launch scene where the application user fills in their data and the data is saved in *user*.

Attributes

- FXML Attributes

Operations

- initUser() : void - set *user* in **GUI** with the data provided by the application user. Switch to MainMenuScene.
- checkDataValidity() : boolean - checks if the data provided by the application user is reasonable.
- saveUserData() : void - saves *user* to a JSON file.

Association

- **CreateUserScene** has a one-way association with **StringToGenderHelper** which returns **Gender** depending on the string.
- **CreateUserScene** also has a one-way relation with **User** since it initialises **User** attributes depending on the application user input.

<enumeration> Gender

Gender is an enumerator class that contains gender titles so that they are easily represented in the system.

Attributes

- MALE
- FEMALE

StringToGenderHelper

StringToGenderHelper is a helper class that provides a **Gender** variable based on the given String variable.

Attributes

- stringGenderMap : Map<String, Gender> - map that contains translation from String variable to **Gender** variable.

Operations

- getGender(gender : String) : Gender - depending on the provided String, returns a **Gender** variable.

Association

- **StringToGenderHelper** has an enumeration association with **Gender**

MainMenuScene

MainMenuScene is the controller of the scene “MainMenu.fxml” , a scene used to start interactions with different app functionalities.

Attributes

- FXML Attributes

Operations

- initActivity(sport :Sport) : void - Fills *Activity* attributes with the GPX path and sport provided by the application user. Switch to “VisualiseActivity.fxml”.
- getGPXPath() : String - Creates the *fileChooser* which prompts the application user to input their GPX file. Then extracts the absolute file path.
- sportInput(actionEvent : ActionEvent) : void - Calls initActivity() with the appropriate *sport* parameter.

Association

- **MainMenuScene** has a one-way association with **StringToSportHelper** which returns **Sport** depending on the string.
- **MainMenuScene** also has a one-way relation with **Activity** since it initialises **Activity** attributes depending on the application user input.

StringToSportHelper

StringToSportHelper is a helper class that provides a **Sport** variable based on the given String variable.

Attributes

- stringSportMap : Map<String, Sport> - map that contains translation from String variable to **Sport** variable.

Operations

- getSport(sport : String) : Sport- depending on the provided String, returns a **Sport** variable

Association

- **StringToSportHelper** has an enumeration association with **Sport**.

VisualiseActivityScene

VisualiseActivtyScene is the controller of the scene “VisualiseActivity.fxml”, a scene responsible for displaying the activity details such as metric totals/averages, a map and charts.

Attributes

- FXML Attributes

Operations

- initialize() : void - Sets all text labels to display calculated activity data.
- backToMainMenu() : void - Switches scene to “MainMenu.fxml” when ‘Back’ button is pressed.

Association

- **VisualiseActivityScene** has a composite association with the **Chart** since it can display 1-5 **Chart** objects and they cannot exist without **VisualiseActivityScene**.

MapHelper

MapHelper is the class used to create a map representing the activity path.

Operations

- displayMap(coordinates : List<List<Double>) : Map - creates a Map object and puts passed coordinates on that map.

Association

- **MapHelper** is used to display the map by **VisualiseActivityScene** class to get the map of the activity.

Chart

Chart is the class used to create a chart representing a **Metric** from an activity.

Attributes

- metric : Metric - **Metric** object that would return the data points to be displayed.
- name : String - name of the chart

Operations

- <constructor> Chart(metric : Metric, activity : Activity, name : String) : Chart

Association

- **Chart** has a composite association with **Metric**, so that it cannot exist if the **Metric** does not exist.

Goal

Goal is the class used to create and update the application user's goal. A **Goal** could be defined as Run 1000 Km or burn 5000 calories. Every time the application user imports a GPX file, if the **Goal** is relevant to the **Activity** then the **Goal** is updated. The application user could later track their progress through the VisualiseGoalScene and see, for example, that they have run 400 Km out of the set 1000 Km.

Attributes

- type : Metric - Each **Goal** tracks one **Metric** such as **Distance**.
- progress : Double - Tracks the progress through the **Goal**. If we use the example in the description above then progress would hold the value 400.
- target : Double - The target that the application user is trying to reach. If we use the example in the description above then the target would be 1000.
- sport : Sport - Each **Goal** must be associated to a **Sport**, this is to stop swimming distance from being added to a cycling distance **Goal**.
- completed : Boolean - **Goal** completed.

Operations

- <constructor> Goal(sport : Sport, target : Double, type : Metric) : Goal - This constructor is used when the application user creates a new goal from a scene.
- <constructor> Goal(sport : Sport, target : Double, progress : Double, type : Metric) : Goal - This constructor is used on application launch to load in-progress goals from JSON into objects stored in the *user*.
- calculateProgress(newData : Double) : void - Called when an **Activity** is relevant to a **Goal** to update the progress on that specific **Goal.**
- getProgress() : Boolean
- isCompleted() : Double - Checks if progress is equal to target.

Association

- **Goal** has a composite association with one **User**, which contains a list of all goals (0 to n).
- **Goal** has a one-way association with an **Activity** from which the progress is updated.

GoalCreationScene

GoalCreationScene is the controller of the scene "CreateGoal.fxml", a scene used to create **Goal** objects.

Attributes

- FXML Attributes

Operations

- createGoal() : void - Create a new *goal*, save it to JSON, and update the list of **Goal** objects in *user***.**

Association

- **GoalCreationScene** has a one-way association with the **User** class of creating a **Goal** object.

GoalVisualizationScene

GoalVisualizationScene is the controller of the scene "VisualiseGoal.fxml", a scene used to visualise **Goal** progress.

Attributes

- FXML Attributes

Operations

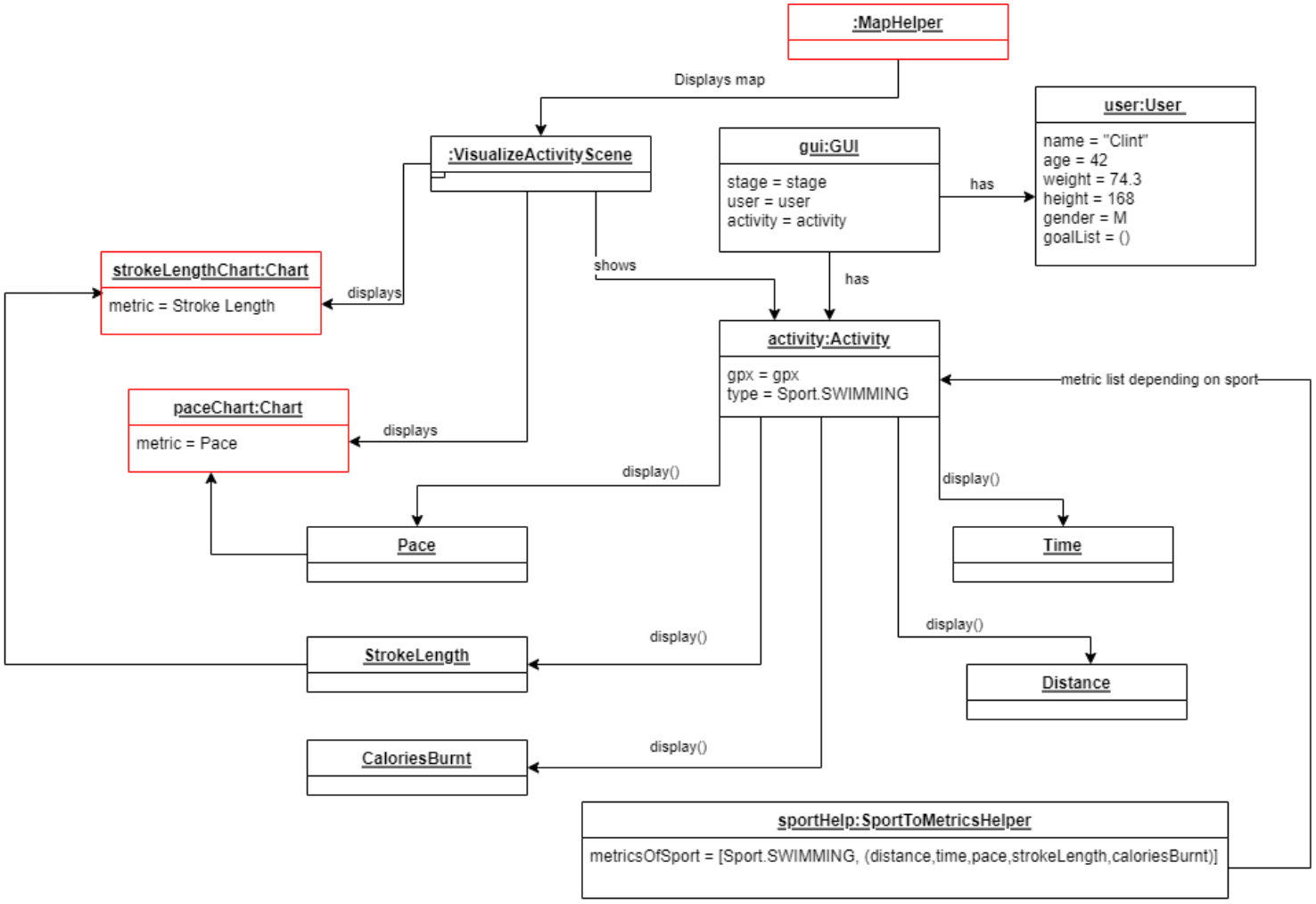
- initialize() : void - Displays all *goals* in *user* goalList.
- displayGoalProgress(goal : Goal) : void - Sets FXML attributes to display a specific *goal's* progress.

Association

- **GoalVisualizationScene** has a one-way association with the **User** class of displaying the goals' progress.

Object diagram

Author(s): Erik Vunsh



(Use this link to view the diagram)

The state of the system that is depicted in the diagram above is when a new user analyses a swimming GPX file, the user being new is a key factor as it explains the reason for no goals being tracked.

- **GUI:** The GUI serves as the starting point for our system, it allows the user to communicate with the system and switch scenes. The attribute *stage* is set to the object “stage”,the *user* attribute is set to “user” object and the final attribute *activity* is set to “activity”.
- **User:** The user object contains personal data and goals of the user. The characteristics are derived from the CreateUserScene, where on first launch the user is prompted to give their details. In the current instance the user is a 42 year old male named Clint, who weighs 74.3 kilos and is 168 centimetres tall. This instance also does not have any goals listed for Clint. Some attributes like weight for example will be further used to calculate metrics.
- **VisualiseActivityScene:** This object acts as a controller for the “VisualiseActivity.fxml” scene that will visualise the analysis of the GPX file. The VisualiseActivityScene will bring together the mapped out GPX file and the appropriate charts and metrics for a certain activity.
- **MapHelper*:** One of the key features of our application is GPX file mapping, which will be done through this object. The coordinates will be extracted from the GPX file by the *activity* and layed out on a map which will be displayed by VisualiseActivityScene.
- **Activity:** The *activity* embodies the GPX file in our system. It gets initialised in the MainMenuScene as a GPX file is selected and obtains all of the sport specific metrics. In the current instance the *gpx* attribute is “gpx” and the activity *type* is Sport.SWIMMING.
- **SportToMetricsHelper:** This helper object provides sport specific metrics to the activity object depending on the *type* attribute. In the occurrence of swimming the list; *distance*, *time*, *pace*, *strokeLength* and *caloriesBurnt*.

- **Time:** Time is an object that holds all the necessary functions to extract the time of an activity from a GPX file.
- **Distance:** The Distance object consists of methods that are required to find the distance travelled during an activity from a GPX file.
- **CaloriesBurnt:** Calories burnt is an object that holds functions for the measurement of calories burnt during an activity.
- **StrokeLength:** StrokeLength is an object that is unique to the swimming instance of our system. StrokeLength as an object contains methods which aid to determine the length of the user's stroke during swimming.
- **Pace:** The pace object consists of functions that are required to determine the pace of the user during an activity.
- **Chart*:** Chart is an object that helps visualise metrics from the imported activity to the user. In the instance of the object diagram the only metrics that displayed are Pace and StrokeLength.

***** - As these objects are yet to be implemented they are prescriptive

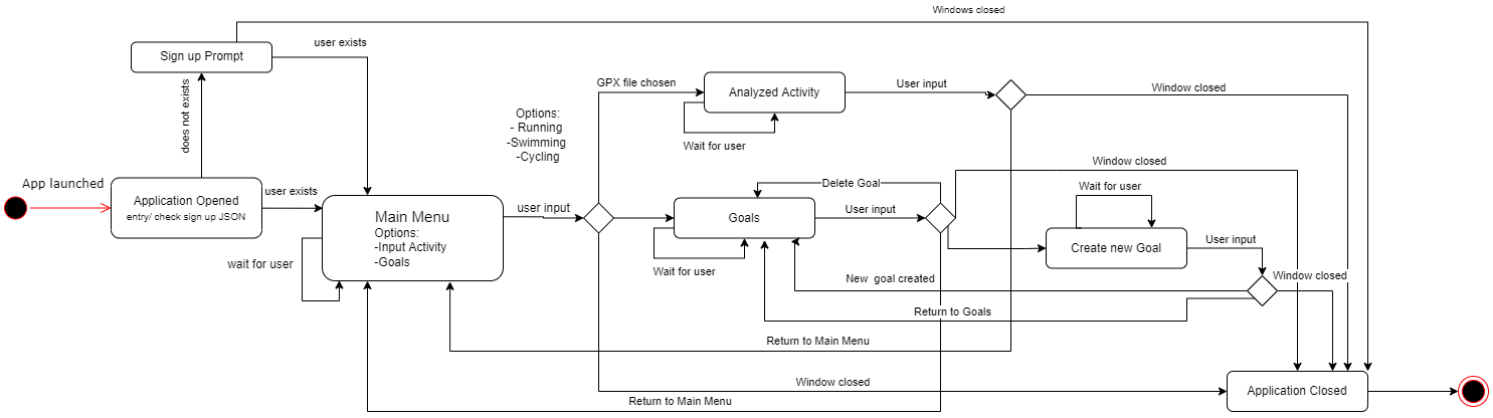
Maximum number of pages for this section: 1

State machine diagrams

Author(s): Erik Vunsh & Abhisaar Bhatnagar

The following section covers two significant sections and their respectively different states. The two classes that the diagrams in this section shall cover are the **GUI** class and the **GOAL** class.

State Machine Diagram 1: ***GUI***



(Use this link to view the diagram)

The **GUI** class manages scene functions and user interaction in our system. Once the application is launched the **GUI** first checks for the presence of the 'user-data' JSON. If the JSON does not exist a Sign up Prompt state is invoked, after its completion, the *user* is initialised and the state is switched to the Main Menu. If the JSON exists the scene goes straight to the Main Menu state.

The Main Menu state presents the user with an option to either add an activity or view goals. The GUI waits for the user until user input is detected, the GUI then changes states depending on which of the options is selected.

The first option prompts for a choice between the 3 available sports, then it opens the file explorer and a GPX file can be imported, in which case the **GUI** redirects to the Analysed Activity state. The second option the user can make is to select the goals option at which the GUI redirects to the Goals *state*. An option which is always

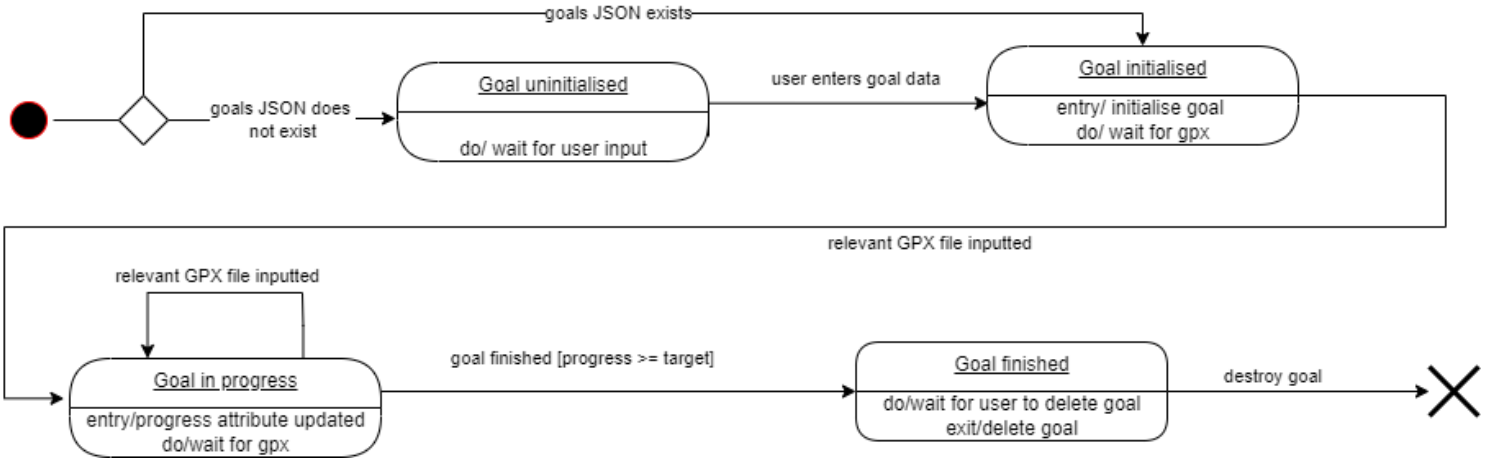
available is to quit the application, in which case the GUI enters the Application Closed state and the final state is reached.

In the Analysed Activity state the user is posed with an option to either return to the main menu or to close the window. Returning to the main menu will invoke the Main Menu state. Closing the window will lead to the final state of the GUI.

In the Goals state the user is given four options. The first two consist of a return to the main menu or closing of the window, both of which have been described above. The second two allow the creation or deletion of goals. Creating a goal changes takes the **GUI** into the Create New Goal state, while deleting a goal will reinvoke the current Goals state.

During Create New Goal state the user is given three options. The first two consist of a return to Goals state and a closing of the window. The last allows the creation of a new goal. When a new goal is created the state is reverted back to the Goals state where the user can view all their goals.

State Machine Diagram 2: ***Goal***



(Use this link to view the diagram)

Goal object is created and enters first state when the GUI is loaded. Since it is empty at first, it tries to read the JSON file where the goals are stored, and if such file does not exist its state is Goal uninitialised. While in this state it waits for a user to choose a **Goal**'s requirements.

The second state is Goal initialised. Its entry is the initialisation of a **Goal**, either by the application user in the **CreateGoalScene** or via JSON on launch, and while the **Goal** is in such a state it waits for the application user to upload a relevant gpx file.

Goal's third state, Goal in progress, is entered after a relevant gpx file is inputted. On entry, the progress of the **Goal** is updated and while the progress has not reached its target, **Goal** waits for more relevant gpx files and redoes the transition to the same state.

Once the goal is completed ([progress >= target]) the user arrives at the last state, Goal finished. In this state it waits for the application user to delete the goal (this is because the user might like to see the completed goals), and once the goal is deleted it exits the state and the **Goal** object is destroyed.

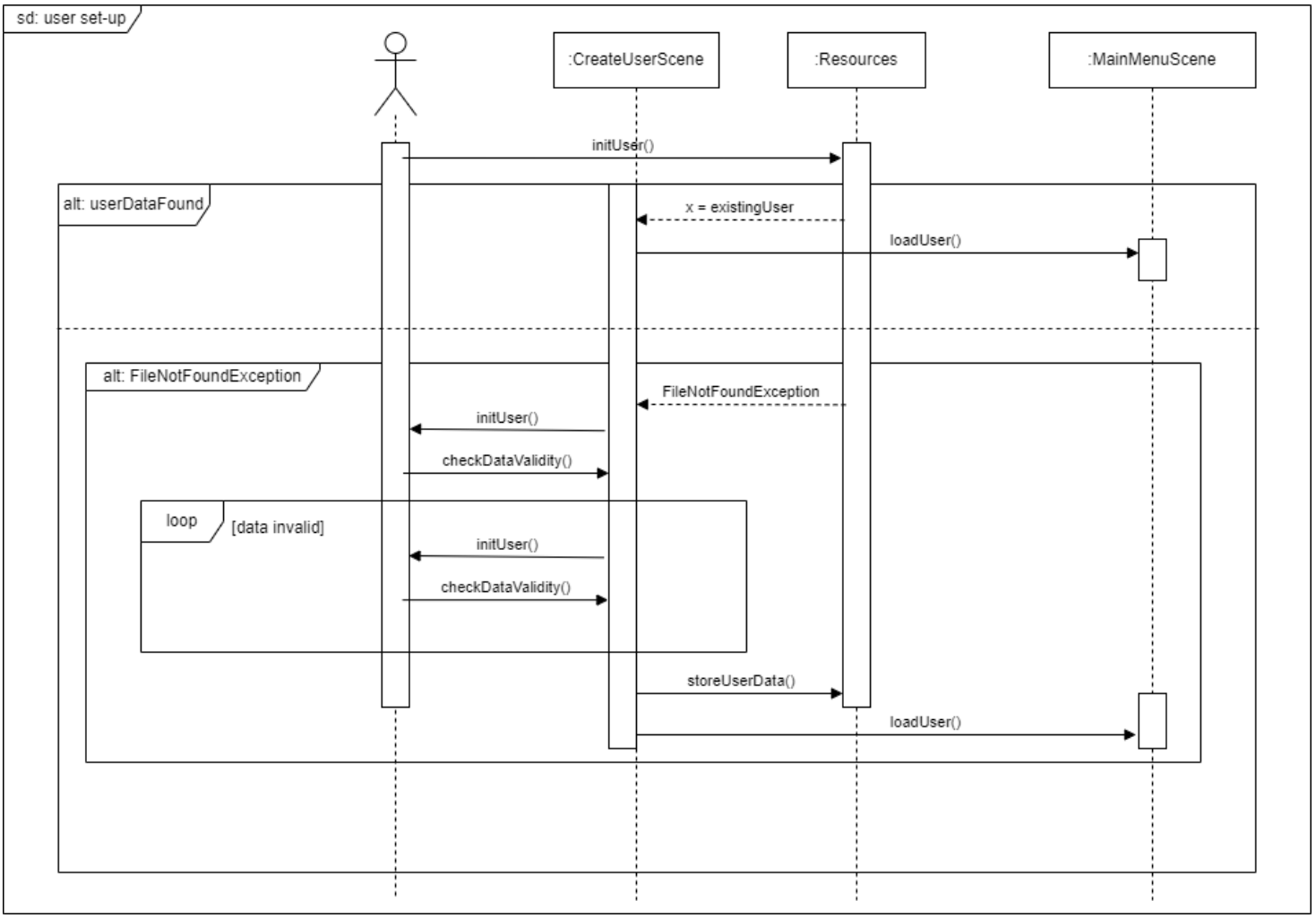
Sequence diagrams

Author(s): Abhisaar Bhatnagar & Baher Wahbi & Martynas Rimkevičius

In the following section, we will delve into the sequential functionality of two components of our application. The first of the two consists of a sequence diagram demonstrating the initial set up conducted by the user whereas the second of the two diagrams illustrates the visualisation of activity on text, maps, and charts. The diagrams aim to give a view of essential functionality of our application while also assisting in the provision of documentation in reference to the timeline of the functions and their runtime processing.

Sequence Diagram 1: ***User Set-Up Sequence***

UML Sequence Diagram:



(Use this link to view the diagram)

The process of user set-up is a fairly simple sequence divided into two key components. The first half of the diagram outside of the alts depicts the initial check the system conducts when the application starts with the appropriately corresponding objects, lifelines, types and functions whereas the bottom half inside the alts illustrates the second component which can be further subdivided via the means of two alternate situations that can occur during the user set up. A more detailed account of the components of user set up can be found as follows:

Pre-Existing User Data Check: Upon the user's starting of the system, the system invokes a check on whether the user is a new user or an existing user. This is demonstrated in the following manner within the diagram:

The user running the application causes the system to access the :Resources object (resources directory in the project) and check for the existence of a JSON file in the :Resources. If the file is not found then the object replies representing whether or not a user exists.

Post-Check Interaction: depending on the check result (JSON file was found or not found), two alternative scenarios are executed.

userDataFound: If user data is found within the :UserDataStorage object then the system simply redirects the user to the :MainMenuScene gaining access to whatever functionality they require.

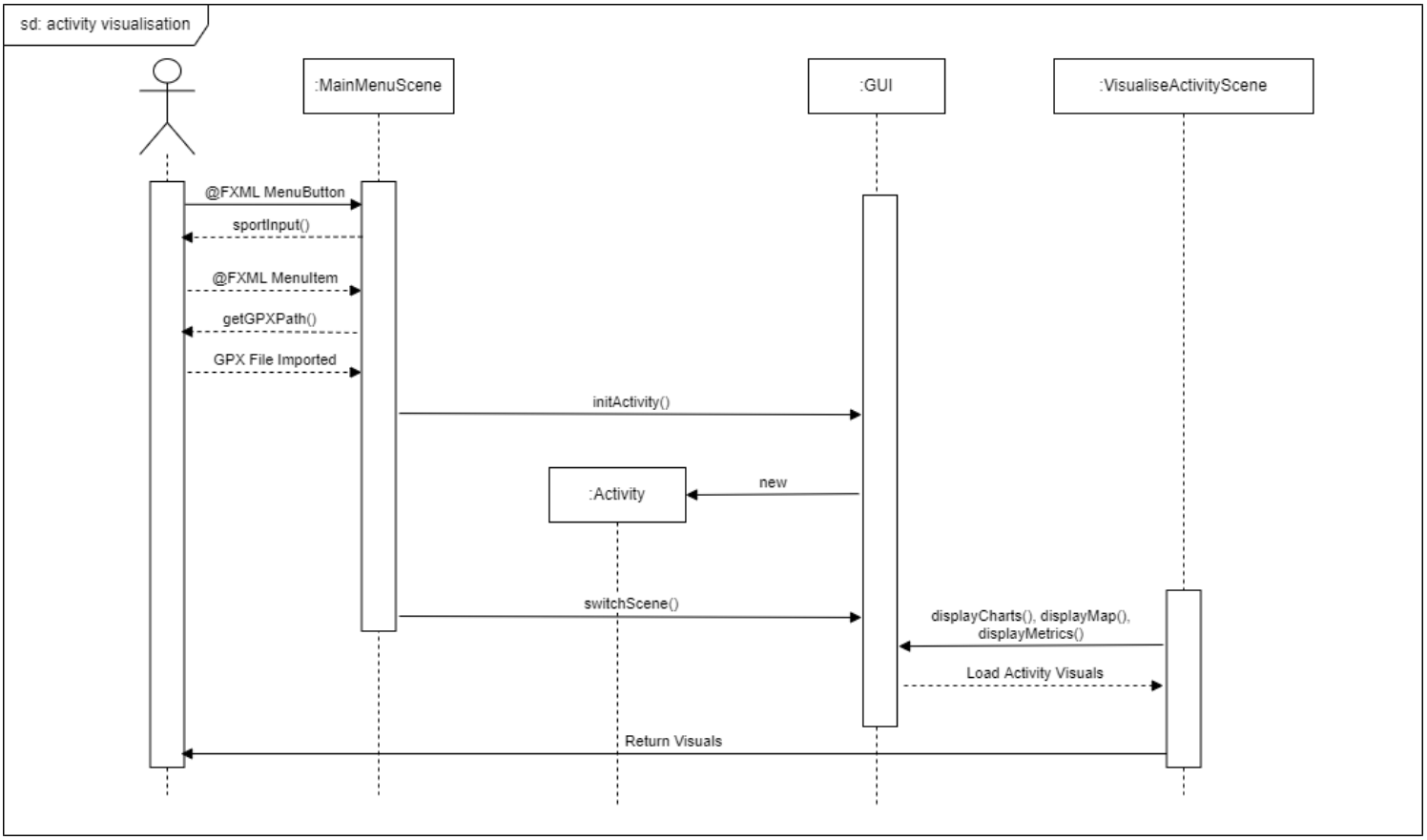
userDataNotFound: This alternative happens when the :UserDataStorage finds no existing user data. In such a case, a FileNotFoundException is thrown. Consequently the :UserDataScreen asks the user to enter their data: name, height, weight, age, and gender for their new user account.

The interaction of filling the data is in a loop as :CreateUserScene checks the validity of data. If it is valid, then :CreateUserScene stores the user's data to the :Resources object.

The :UserDataStorage object then redirects the user to the :MainMenuScene object where the user can access the functionality of the application.

Sequence Diagram 2: *Sports Activity Visualisation*

The UML Sequence Diagram:



(Use this link to view the diagram)

This sequence diagram denotes the post user set-up visualisation, occurring when the user initiates an activity, with the appropriately corresponding objects, lifelines, types and functions. This can be seen as five interaction stages ranging throughout the diagram. A more detailed account of this sequence of activity visualisation can be found as follows:

User - Main Menu Interaction: Initially the sequence starts with an interaction between the application user and the main menu scene. For any functionality to be used the user would firstly have to initiate a sequence with the main menu. This stage is initiated when the user chooses to import an activity.

Firstly the user clicks to import an activity. Upon which, the :MainMenuScene object calls a sportInput() function request asking the user to select a corresponding sport for the activity, via the means of a dropdown menu.

Upon selecting the desired sport the :MainMenuScene opens the file explorer via the fileChooser() function allowing the user to select their GPX file.

MainMenuScene - Activity - GUI Interaction: After all user interaction with the :MainMenuScene and the importing of the GPX File, this phase of interaction consists of the creation of the **Activity** and setting it in the **GUI**.

Once the :MainMenuScene has received an imported GPX file and knows the sport, it sets the **Activity** in **GUI** and calls a switchScene() to the VisualiseActivityScene..

GUI - Visualise Activity Scene Interaction: This respective phase is where the visuals are actually created (visuals refer to maps, charts and metrics).

During this stage, after the switching of scenes, the :VisualiseActivityScene object calls upon the :Activity object using the displayCharts(), displayMap(), and displayMetrics() functions to get the visuals.

Visualise Activity Scene - User Interaction: Perhaps, the simplest of stages, this is the final interaction of this diagram where the :VisualiseActivityScene object simply relays the visuals back to the user.

Implementation

Author(s): Martynas Rimkevičius & Baher Wahbi

UML to Java

We have followed an iterative process of moving classes from a prescriptive class diagram to Java classes because we found that we gained a better understanding of the structure of the model while creating code. This allowed us to improve our class diagram while making sure we adhered to the design principles. Object diagram, being a snapshot representation of the class diagram, followed the same update process.
State-machine and sequence diagrams helped us to create a communication structure between classes and model the behaviour inside objects themselves. This meant that even though the implementation was based on the diagrams, the diagrams had to go through few iterations of changes when the class diagram was updated.
By the end of the assignment 2 timeframe we were left with unimplemented Charts, Maps and Activity History features and modelling for them was left prescriptive.

Key Solutions

Metric: To calculate sport metrics we chose an abstract class approach representing a generic Metric, such that it would be extended by implementing calculation of specific metrics as speed or elevation. This allowed us to create a reusable and extendable structure for future metrics. Each metric is created in a way so that it can be used by other sports (this is for the developer to decide which metrics best suit a sport). This is done by only creating operations which return and do not save data, thus allowing us to reuse the metric for different sports. We have also created a linking system between sports and metrics using a helper mapping which helps with the system's expandability.

User: Early on in development we realised that in order for developer extensibility to function for creating new metrics and subsequently sports, the developer must have access to some general user data such as their weight, height, age and gender. This would in turn ensure that the developer would have no limitation on which metrics or sports they could extend that app with. To start with, we envisioned the user inputting their personal data with each GPX analysis, but quickly it seemed impractical and we opted for a one time user input that would be stored in JSON. Using the GSON library we are able to read the JSON into the **User** object on each launch of the application. Other classes can use getUser() to retrieve the user information and use it in their calculations.

GUI: The general goal throughout modelling and building the **GUI**, was to create a system that would allow the developer to write short snippets of code which allow the loading and switching of scenes seamlessly. This was achieved with the function switchScene() which internally called the function loadScene() both taking an FXML file as an input. This way switchScene() could simply be called with the name of the desired scene in any scene controller and end up in the right location.

Helpers: we came up with 3 small classes, which we are using to translate one type of information to another. Two of them translate String variable to enumerator class variable (**StringToGenderHelper** and **StringToSportHelper**), whereas the third one, **SportToMetricsHelper**, maps **Sport** type variable to a list of corresponding **Metric** objects, such that it could return only the relevant metrics. This map is created by the developer, thus it allows them to expand the list of sports and easily assign new or already created metrics.

Small Note on Implementation:

Exception handling: - not implemented - Currently the application expects complete gpx files, meaning each defined sport metrics have to be in the file. Exceptions, which could arise if some metric field does not exist, handling has not been implemented yet. This can be expected in the assignment 3.

Location of main Java class:

'\src\main\java\org\softwaredesign\GUI.class'

Location of JAR file:

We were able to create the executable but once executed it throws the error "**Error: JavaFX runtime components are missing, and are required to run this application**"
It's discussed in "https://canvas.vu.nl/courses/60274/discussion_topics/506191" that if we do get such errors it's okay as long as a TA can run the code on their machines. You can do so through the GUI class.

Demo

Time logs

Team number	22		
Member	**Activity**	**Week number**	**Hours**
Martynas	Class diagram	3	10
Martynas	Implementation	3	14
Baher	Library Integration (ERRORS)	3	7
Baher	Implementation	3	2
Abhisaar	1st Sequence Diagram Creation	3	3
Abhisaar	1st Sequence Diagram Textual Description	3	2
Baher	Diagram Creation	4	2
Baher	Textual Discription	4	16
Baher	Implementation	4	18
Erik	State Machine Diagram	4	10
Erik	Object Diagram	4	9
Erik	Class diagram review	4	2
Martynas	Class diagram	4	7
Martynas	Class diagram textual description	4	8
Martynas	Implementation	4	3
Martynas	Implementation textual description	4	3
Martynas	Final edits of textual descriptions	4	6
Abhisaar	2nd Sequence Diagram Creation	4	2
Abhisaar	2nd Sequence Diagram Textual Description	4	2
Abhisaar	State Machine Textual Descriptions	4	2
		TOTAL	104