Assignment 3

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

Summary of Changes of Assignment 2

Author(s): Abhisaar Bhatnagar

Class Diagram:

- ***Problem 1:*** *"While the description did pose some limited discussion, you can elaborate a bit more…This document should guide such discussion."*
- **Solution:** The descriptions corresponding to the classes were elaborated on where deemed possible.

- ***Problem 2:*** *"I think you mean 0..* instead of 0..n. Where is n defined?"*
- **Solution:** The '0..n' adjoining Goals Class to the User Class was adjusted to '0..*'.

- ***Problem 3:*** *"Nice design for the enumeration. Note you are missing some multiplicities for these classes."*
- **Solution:** The appropriate multiplicities corresponding to the respective classes of enumeration were adjusted into the Class Diagram.

- ***Problem 4:*** *"Great design for Metrics. But why is there a limit on the maximum number of metrics?"*
- **Solution:** Initially it was for the design of our charts' window - to limit the cluster of displayed metrics, but now the design has evolved and the limit on the number of metrics has been removed.

Object Diagram:

- ***Problem:*** *"This discussion could do a slightly better job of explaining why classes are designed as they are."*
- **Solution*:*** Enhanced explanations for each class and why they are designed the way they are.

State Machine Diagrams:

- ***Problem:*** *"The first diagram has some syntactic issues. In the 'create a new goal' state, the 'wait for user' should be part of do/relation. The state is not left when waiting for user input (that is the state)"*
- **Solution:** The 'wait for user' activity is no longer represented by a recurring loop to the state rather as a 'do' activity within the 'Create New Goal' state given that 'wait for user' does not leave the state, but is an inner state action.

Sequence Diagrams:

- ***Problem:*** *"There is one minor mistake: Missing new operators when instantiating objects of a class in the Diagram1."*

- **Solution:** The new operators upon instantiating objects that were missing have been added in Sequence Diagram 1.

Code:

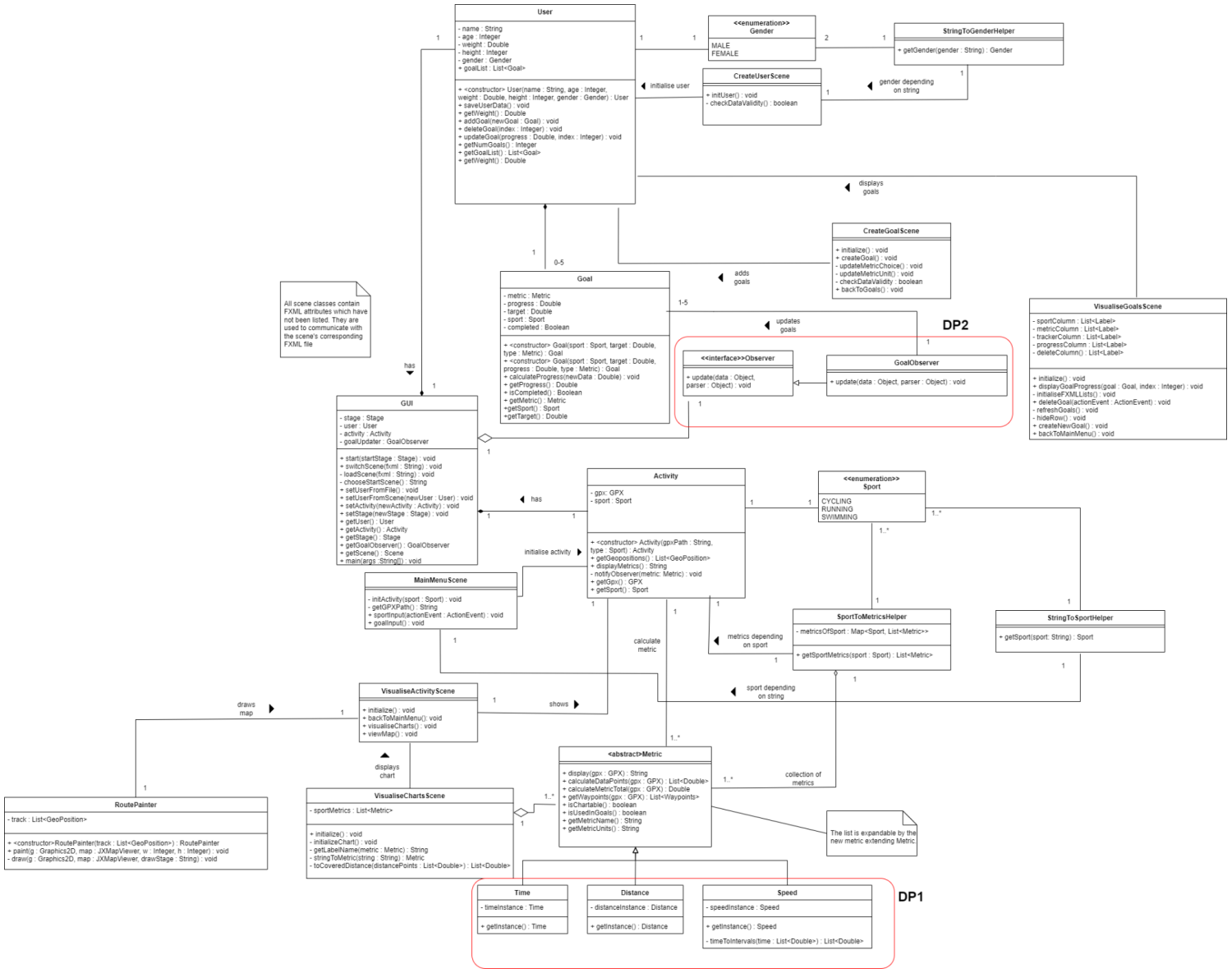
- ***Problem 1:*** *"Perhaps you want to make this a private member? Is there a reason for it to be accessible by other classes?"*
- ***Solution:*** Cadence metric constructor has been made private, it was a mistake that had not been noticed.

- ***Problem 2:*** *You may want to think of the case that there is no data. So, instead of displaying "heartRate 0 bpm", you can write N/A if a nonetype is returned.*
- ***Solution:*** Missing data handling has been implemented by catching DOMException for the GPX file fields that could be missing. In such case data is displayed as "N/A"

- ***Problem 3:*** *"Maybe print this out to stderr."*
- ***Solution:*** We decided that since the user is shown an "N/A" on the specific metric in the GUI, we do not need to also print these errors out to any output stream. Instead we have opted to use the catch to return an empty list when a list of metric points cannot be calculated. The same applies for the saveUserData() where we used to output a 'Can't save user data' to stderr, but have now opted to just print the stack trace, since SonarLint found the print using standard output interface to be an issue .

Application of design patterns

Author(s): Martynas Rimkevičius



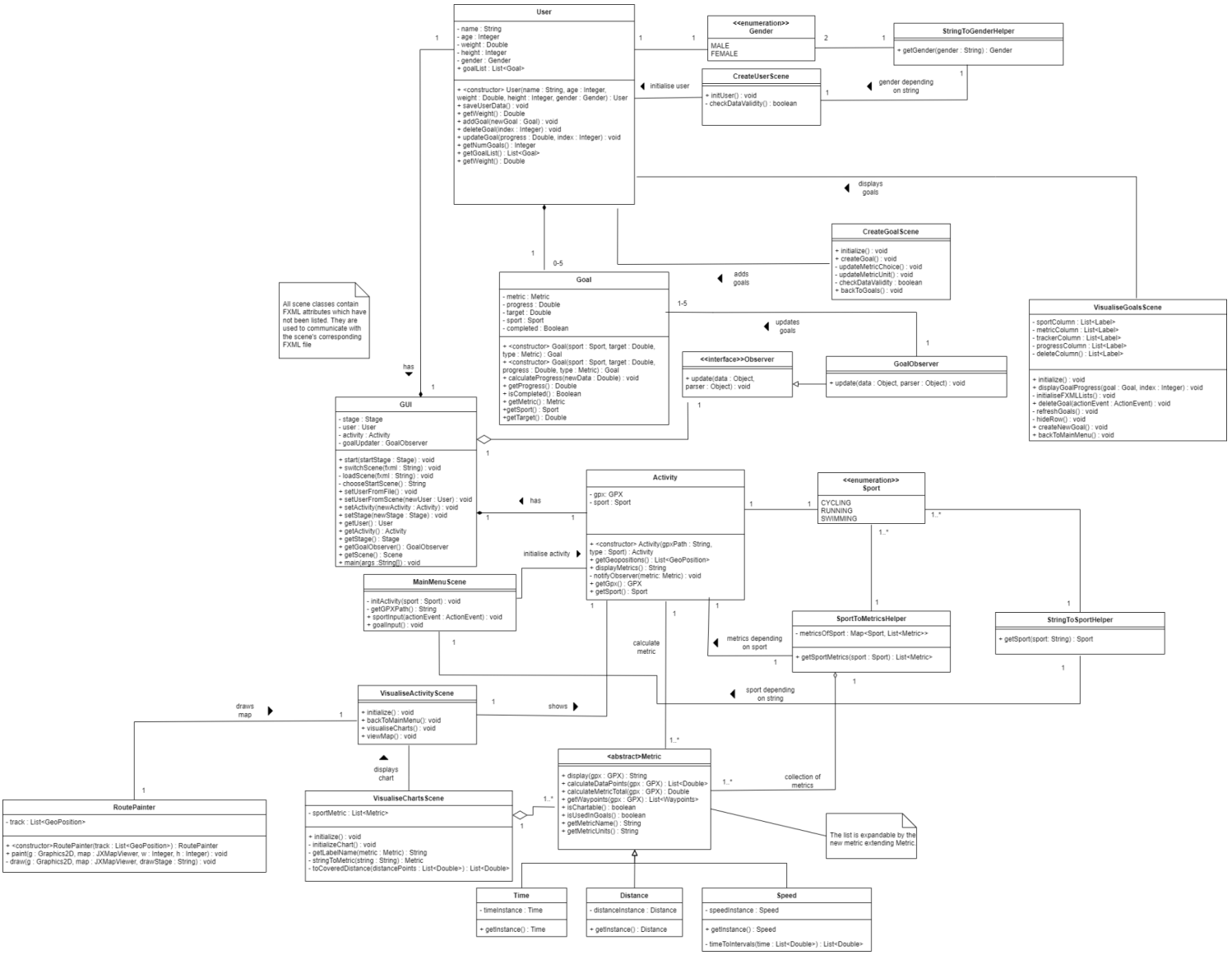
	DP1
Design pattern	Singleton
Problem	Inefficient code. Single use of the application (launch, upload and analyse a single activity) would generate up to 40 objects. With each new activity uploaded, the object count would increase by around 20 objects.
Solution	Each metric (the child class, e.g. Time) is a singleton object, meaning it will be instantiated only once. It has one private instance which is returned when getInstance() is called instead of constructor.
Intended use	Implementation of the singleton object allows us to call metrics freely to calculate data points for textual description of themselves, charts, and goals. This means that instead of 40 or more objects

	we will always have just the same number as there are metric classes (in the current implementation 10).
Constraints	This design pattern implementation did not raise any additional constraints in our system.
Additional remarks	The child class, not the parent, has to be the singleton object because it is necessary to have an instance of each metric at the same time. Having a one and only instance of Metric object would cause the system to be stuck on the first instantiated metric, thus not allowing a display of other available metrics.

	DP2
Design pattern	Observer (Goals)
Problem	Each activity upload means that potential goals related to the activity need to be updated, therefore this has to be checked. However, keeping the goal updating algorithm as part of the Activity object is not semantically correct, since the Activity object encapsulates data of the activity, not the goals. Also, other potential areas, which the system could be extended to, might require updating its data, which would need to be encapsulated in such areas.
Solution	Observer interface is created to cover the data flow observation (therefore it needs one object for the data, and another for optional parsing of it), and for the specific case GoalObserver is created which updates goal progress with the Activity object as the data and the Metric object as the parser.
Intended use	The update function has been created with data and parser parameters given a situation arises where the Observer interface requires extensibility. These parameters could aid in narrowing the data down to the necessary amount required.
Constraints	This design pattern implementation did not raise any additional constraints in our system.
Additional remarks	Although not ideal, the GoalObserver object lives in the GUI, as it must be alive for the entire duration of the application, and since GUI is forced to be the entry point by JavaFX, then the observer must also live there.

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
- saveUserData() : void - saves *user* to a JSON file.
- addGoal(newGoal : Goal) : void - add a new **Goal** object to the *goalList*.
- deleteGoal(index : Integer) : void - removes **Goal** object from the *goalList* at index *index.*
- updateGoal(progress : Double, index : Integer) : void - calculates progress with new data for the *index* element in *goalList*

- getNumGoals() : Integer
- getGoalList() : List<Goal>
- 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.
- isChartable() : boolean - Marking if the metric can be displayed in a chart.
- isUsedInGoals() : boolean - Marking if the metric can be used for a goal
- getMetricName() : String
- getMetricUnits() : String

Association

- **Metric** has a shared association with **SportToMetricsHelper**, which contains many **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.

Attributes

- timeInstance : Time - a single instance of the class as part of the singleton design pattern implementation.

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.

Attributes

- distanceInstance : Time - a single instance of the class as part of the singleton design pattern implementation.

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.

Attributes

- speedInstance : Time - a single instance of the class as part of the singleton design pattern implementation.

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.
- sport : 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.
- displayMetrics() : String - puts all metric totals of the activity for the particular sport into a single string.
- notifyObserver(metric : Metric) : void - call for an update from the observer
- getGeopositions() : List<GeoPosition> - extracts all coordinates from the *gpx* and puts them in a list thus preparing them to display on the map.
- getGpx() : GPX
- getSport() : Sport

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 extracted specific type of 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. Each item in the enumeration has an implemented toString() method for easier showcase.

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.
- goalUpdater : GoalObserver - the object of the observer pattern implementation.

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.
- switchSceneInNewWindow(fxml : String) : void - Displays scene in another window.
- 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
- getGoalObserver() : GoalObserver
- getScene() : Scene
- main(args : String[]) - Launches the BEAM app.

Association

- **GUI** has one-to-one composite association with **Activity** class, which object it contains. This is because only one **Activity** object can exist at a time, and the **Activity** object will be deleted when the **GUI** object is deleted.
- **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.

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. Each item in the enumeration has an implemented toString() method for easier showcase.

Attributes

- MALE
- FEMALE

StringToGenderHelper

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

Operations

- getGender(gender : String) : Gender - depending on the provided String, returns a **Gender** variable found in a created map of all **Gender** enumerated values.

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.
- goalInput() : void - opens the **VisualiseGoalsScene**.

Association

- **MainMenuScene** has a one-way association with **StringToSportHelper** which returns **Sport** depending on the string. It is used to get **Sport** after the string is returned from the menu button of selecting the uploaded sport.
- **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.

Operations

- getSport(sport : String) : Sport- depending on the provided String, returns a **Sport** variable found in a created map of all **Sport** enumerated values.

Association

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

VisualiseActivityScene

VisualiseActivityScene 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.
- visualiseCharts() : void - function used to open **VisualiseChartScene** window to show the user charts of the activity.
- viewMap() : void - function used to show the map window of the activity to the application user.

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

- metric : 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.
- getMetric() : Metric
- getSport() : Sport
- getTarget() : Double

Association

- **Goal** has a composite association with one **User**, which contains a list of all goals (0 to 5).

CreateGoalsScene

CreateGoalsScene is the controller of the scene “CreateGoal.fxml”, a scene used to create **Goal** objects.

Attributes

- FXML Attributes

Operations

- initialize() : void - Displays fields to create a new goal.
- createGoal() : void- checks if the input is valid, creates the **Goal** object, saves it and leaves back to the **VisualiseGoalsScene**.
- updateMetricChoice() : void - puts the relevant metric types into the *metricChoice* depending on the **Sport** object.
- updateMetricUnit() : void - show the selected metric unit in the input field.
- checkDataValidity() : boolean - checks if the data provided by the application user is reasonable.
- backToGoals() : void - changes the scene to **VisualiseGoalsScene.**

Association

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

VisualiseGoalsScene

VisualiseGoalsScene is the controller of the scene “VisualiseGoal.fxml”, a scene used to visualise **Goal** progress.

Attributes

- FXML Attributes
- sportColumn : List<Label> - a list of FXML attributes
- metricColumn : List<Label>
- trackerColumn : List<Label>
- progressColumn : List<Label>
- deleteColumn() : List<Label>

Operations

- initialize() : void - Displays all *goals* in *user* goalList.
- displayGoalProgress(goal : Goal) : void - Sets FXML attributes to display a specific *goal’s* progress.
- initialiseFXMLLists() : void
- deleteGoal(actionEvent : ActionEvent) : void - delete a selected goal from the view and remove it from the *user*’s *goalList*.
- refreshGoals() : void - show goals’ progress.
- hideRow() : void
- createNewGoal() : void - shows **CreateGoalScene.**
- backToMainMenu() : void - shows **MainMenuScene.**

Association

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

VisualiseChartsScene

VisualiseChartsScene is the controller of the scene “VisualiseCharts.fxml”, a scene used to display metric change over the distance throughout the activity in the chart form.

Attributes

- FXML Attributes
- sportMetrics : List<Metric> - a list of all metrics related to the activity sport.

Operations

- initialize() : void - sets up the *chartChoice* with available **Metric** objects of which isChartable() is true, and sets an event listener to call initializeChart().
- initializeChart() : void - fills up the *chartDisplay* with the calculated data and sets up the title, X and Y axes, accordingly.
- getLabelName(labelMetric : Metric) : String - creates a string which shows the **Metric** object name as well as its units.
- stringToMetric(string : String) : Metric - function used to get the selected **Metric** object from the *string* returned by the *chartChoice* selection.
- toCoveredDistance(distancePoints : List<Double>) : List<Double> - function used to convert **Distance** metric calculated distance points into covered distance points so that it is neatly displayed on chart X axis.

Association

- **VisualiseChartsScene** has a one-way relation with **VisualiseActivityScene** which starts up **VisualiseChartsScene**
- **VisualiseChartsScene** has a shared association relation with the **Metric** class, which is one to many, since **VisualiseChartsScene** can display many metrics (only those that are available to be charted).

<interface> Observer

Observer is the interface following the observer design pattern. It is used to implement different data flow observers.

Operations

- update(data : Object, parser : Object) : void - Updates the observable object with *data* that is (optionally) passed through the *parser*

Association

- **Observer** has a shared aggregation association with **GUI**, which instantiates the observer, and calls its update whenever it needs to be updated.

GoalObserver

GoalObserver is the implementation of the **Observer** interface. It is used to update **Goal** objects' progress when a new **Activity** object is created.

Operations

- update(data : Object, parser : Object) : void - updates all **Goal** objects with the *data*, which in this case is **Activity** object. Those **Goal** objects have to cover *parser*, or in this case it is **Metric** object,

Association

- **GoalObserver** has a one-way association with the **Goal** class, where one **GoalObserver** object could update as many **Goal** objects as there are (up to 5).

RoutePainter

RoutePainter class is used to create the image of the map and then draw the route on the image.

Attributes

- track : List<GeoPosition> - list of points containing coordinates that collectively make up the path of the activity.

Operations

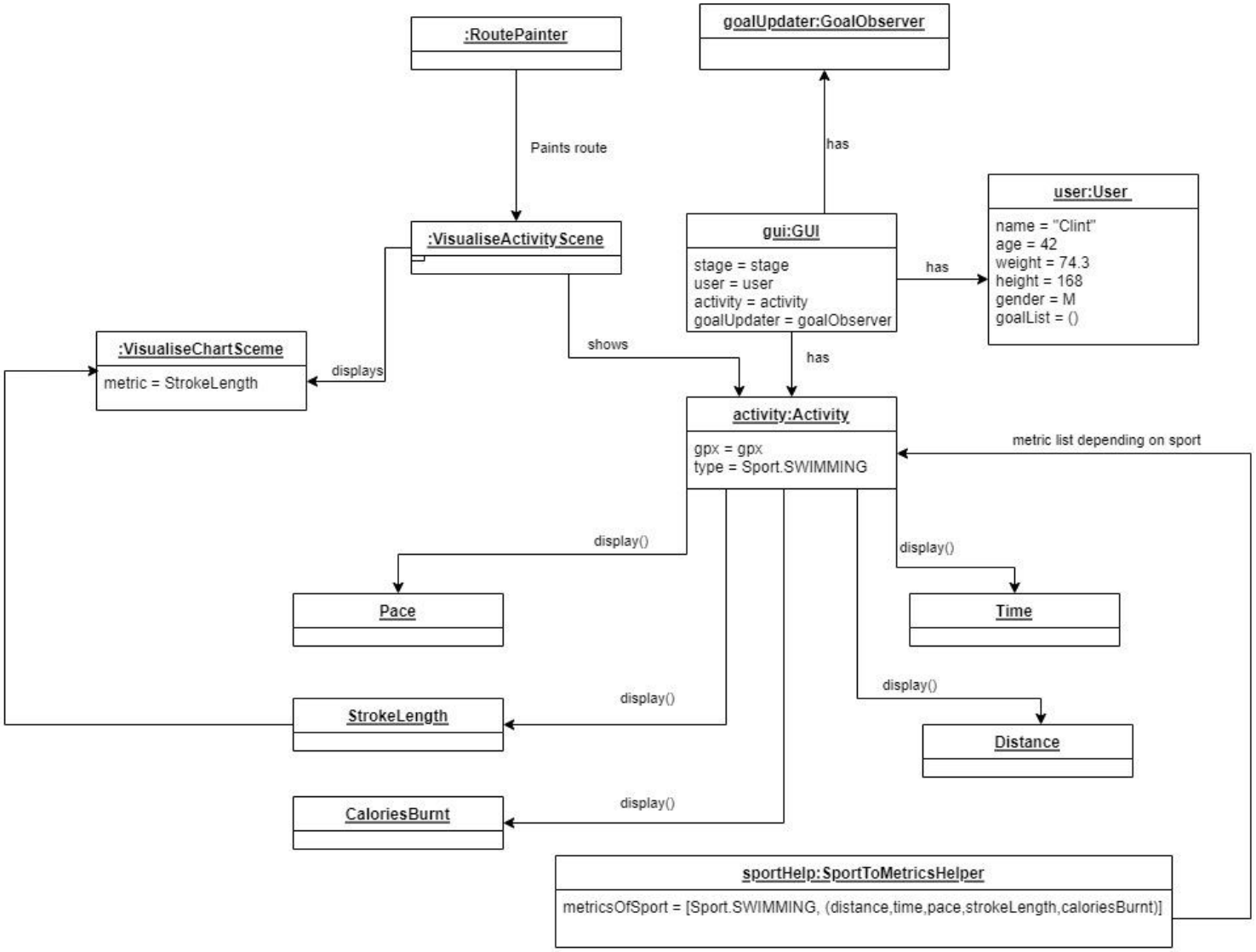
- paint(g : Graphics2D, map : JXMapViewer, w : Integer, h : Integer) : void - function that renders the map image and handles the logic of drawing the map, path, and start and end points.
- draw(g : Graphics2D, map : JXMapViewer, drawStage : String) : void - physically draws points on the map. drawStage tells the function whether to draw the route, the startpoint or the endpoint.

Association

- **RoutePainter** has a one-way association with **VisualiseActivityScene** and is used when the viewMap() function is called to show the map of the activity.

Object diagram

Author(s): Erik Vunsh & Baher Wahbi



(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 and main controller for our system, JavaFX requires that the entry point of the program is an extension of the JavaFX Application class . It contains all the logic for switching between scenes and grabbing inputs from the user. It fills the *user* and *activity* with the correct data and allows the application user to interact with those objects. 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 stores 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. We designed this class in order to help ease the creation of new metrics by giving the developer access to user data that could be used in calculations such as height or gender. In the current instance the user is a 42 year old male named Clint, who weighs 74.3 kilos and is 168 centimetres tall. In this instant, the user, Clint, has not created any goals. 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 and gives the user the option to view them.
- **VisualiseChartScene:** This object acts as a controller for the “VisualiseChart.fxml” scene that will visualise the charted metrics over distance, such as the change in StrokeLength over the distance covered in the

activity. The user will be able to select which metric he wants to visualise the chart for. In the instance of the object diagram the only metric that can be charted are StrokeLength.

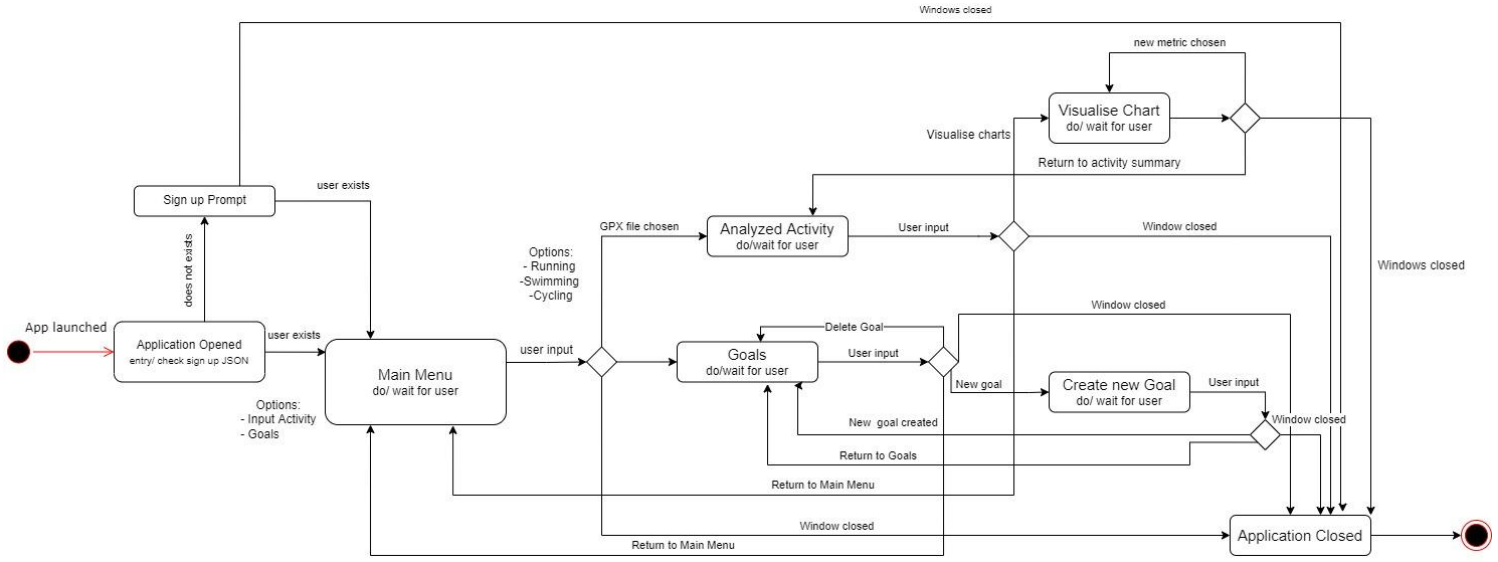
- **RoutePainter:** One of the key features of our application is GPX file mapping, a map object created by the library Jxmapviewer2 will be painted (converted to an image and drawn the route on it) by the RoutePainter. The coordinates will be extracted from the GPX file by the *activity* and painted on the 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 calculates all of the sport specific metrics. Activity also plays a role in updating the user's goals while invoking the observer notification when calculating all the metrics for a new activity input. In the current instance the *gpx* attribute is "gpx" and the activity *type* is Sport.SWIMMING.
- **GoalObserver:** This object exists in the GUI as an observer design pattern implementation for updating existing goals contained in the *user.* As currently there are no goals tracked, the observer will be inactive - will not update anything - when invoked by new activity upload.
- **SportToMetricsHelper:** This helper object provides sport specific metric object instances to the activity object depending on the *type* attribute. In the occurrence of Sport.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.

State machine diagrams

Author(s): Erik Vunsh

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.

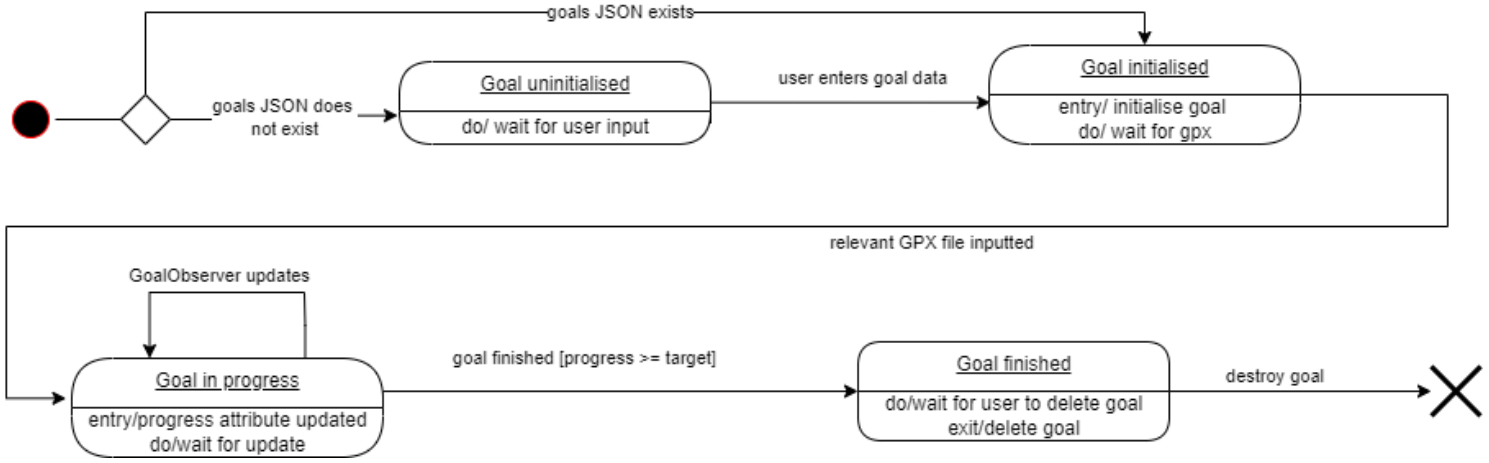
In the Analysed Activity state the user is posed with an option to either view charts for the inputted activity, return to the main menu or to close the window. If the visualise charts option is chosen it will redirect the user Visualise Chart state. Returning to the main menu will invoke the Main Menu state. Closing the window will lead to the final state of the GUI.

In the Visualise Chart state, the user is prompted with 3 options. The first option is to choose a metric to chart, upon which the Visualise Chart state will be re-invoked with the newly chosen metric. The second option is to return to the activity summary which leads to the Analysed Activity state. The last option is to close the window which results in the final state of the program.

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 the GoalObserver object to update it 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.

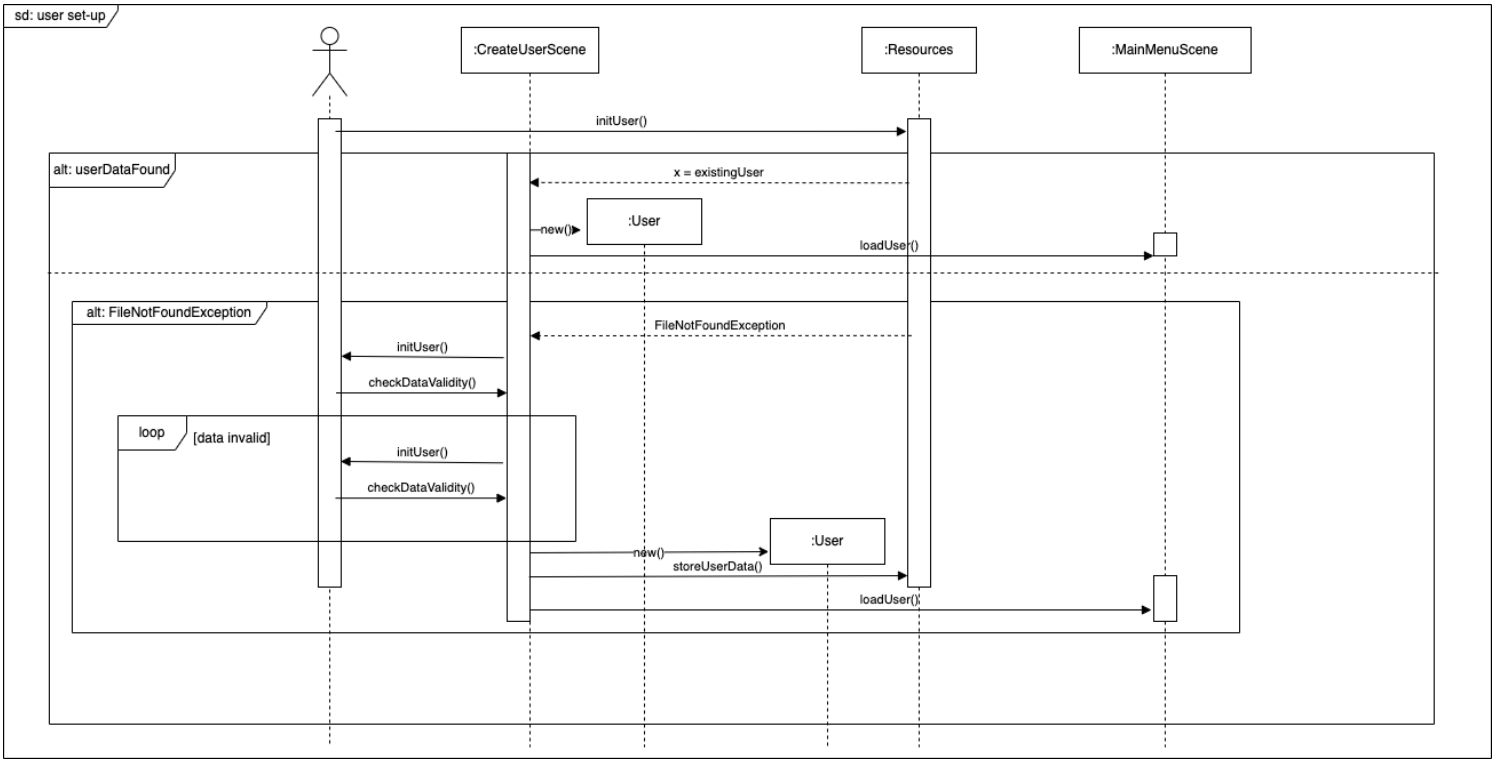
Sequence diagrams

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***

The 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 an encrypted 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 (file was found or not found), two alternative scenarios are executed.

userDataFound: If user data is found within the :Resources object then the system simply redirects the user to the :CreateUserScene where a new user is instantiated after which the User is redirected to the :MainMenuScene gaining access to whatever functionality they require.

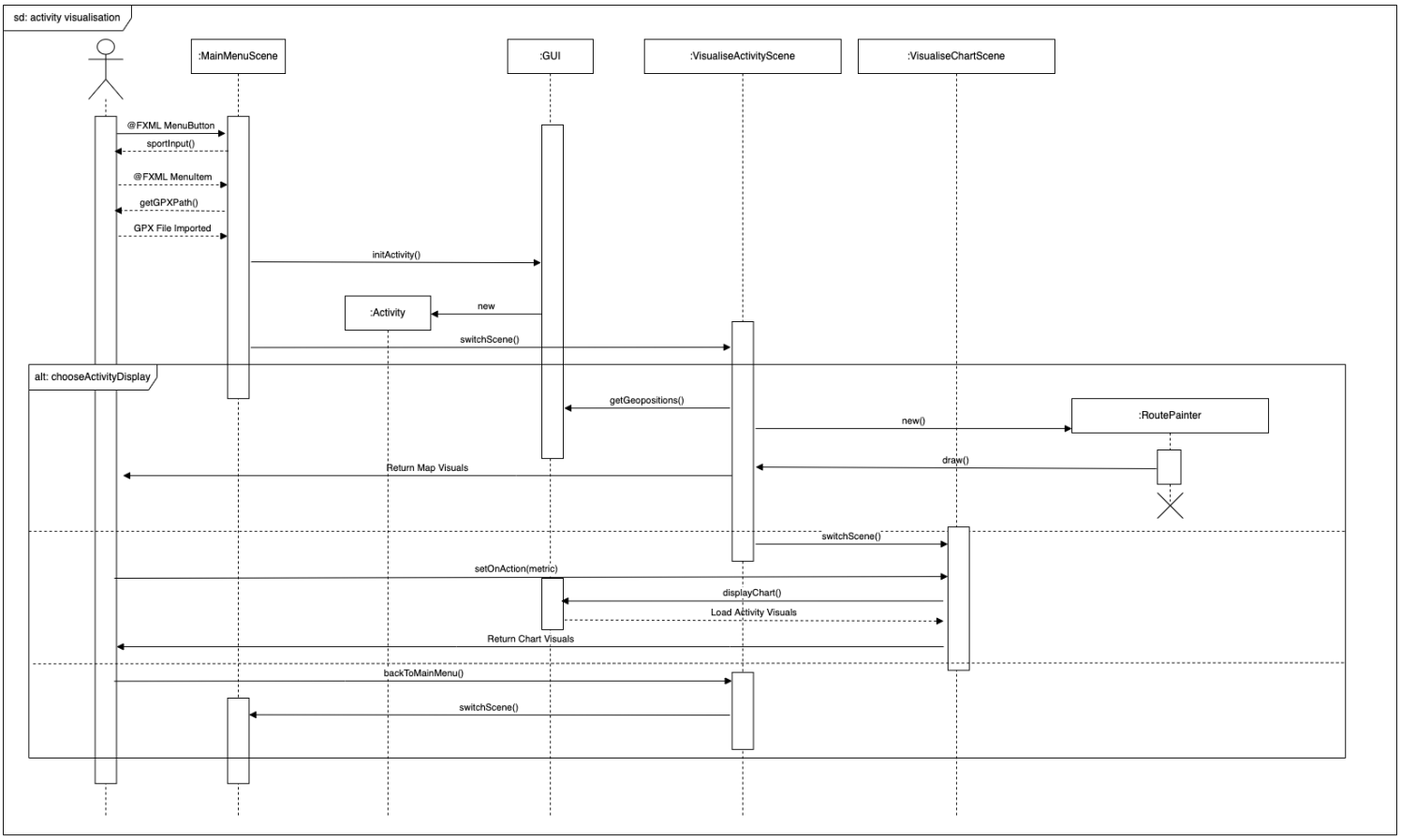
FileNotFoundException: This alternative happens when the :Resources finds no existing user data. In such a case, a FileNotFoundException is thrown. Consequently, the :CreateUserScene asks the user to enter their data: name, height, weight, age, and gender for their new user account.

The interaction of filling the data is in a loop as :CreateUserScene checks the validity of data. If it is valid, then a new user object is instantiated and the user's data is stored into the :Resources object.

The :CreateUserScene 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. The first two interaction stages outside the alt depict the initial interactions the user navigates in order to visualise an activity when the main menu launches with the respective correlating objects, lifelines, types and functions whereas the other three interactions are subdivisions of the alt which demonstrate the three possible scenarios that can occur

upon the choosing of the desired activity displays. 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 getGPXPath() 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 object.

GUI - VisualiseActivityScene - User Interaction: The first respective phase of this alt is started when the application user chooses to view the map. This is when the map visuals are actually created. This stage is initiated when the :VisualiseActivityScene object calls the getGeopositions() function from the :GUI for the display of map visuals.

After the coordinates are received from getGeopositions() call, the :VisualiseActivityScene instantiates a new :RoutePainter object, which paints the map object, and in response :RoutePainter draws the map visuals into :VisualiseActivityScene.

The final stage of this interaction is when :VisualizeActivityScene displays a complete map window corresponding to the user's data back to the :User.

VisualiseActivityScene - VisualiseChartScene - GUI - User Interaction: The second phase of this alt consists of where the chart visuals are created and is quite similar to the previous stage. This stage is yet again initiated post a call to switchScene() by :MainMenuScene to :VisualizeActivityScene.

During this stage, the :VisualiseActivityScene object calls another switchScene() to arrive at the :VisualiseChartScene. When the user selects the metric type to be charted, the :VisualiseChartScene then calls upon the :GUI object using the displayChart() functions to get the visuals. The :GUI object responds to this call by loading the appropriate values to put into the chart.

The final stage of this interaction is when :VisualizeChartScene displays the completed chart window corresponding to the appropriate metric requested by the user back to the :User.

User - VisualiseActivityScene - MainMenuScene Interaction: This is the final possible alt interaction of this diagram where the user can choose to return to to the main menu which calls the

backToMainMenu() function to the :VisualiseActivityScene object which simply relays the user back to the :MainMenuScene.

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.

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. To reuse the given metric and not create a bloated system of calculators, we have implemented a singleton design pattern for each metric child class, so that there would exist only a single calculator for each metric. 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, which is encrypted with base64 encoding so that it cannot be edited maliciously or viewed externally. Using the GSON library we are able to read the JSON into the **User** object on each launch of the application after decrypting the text. 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.

Map: we decided to display the route of the activity in a new window which we draw on using the **routePainter** object by getting the snapshot image of the track area and connecting all the tracked points on that image. For clarity we chose to mark the start and end points of the activity as green and red respectively.

Charts: to display in-depth information in a concise manner for the application user of their activity we chose to show a chart of the metric change over the covered distance. Each metric has an indication isChartable(), which is used to find which metrics of that activity could be displayed. After the metric is selected, the chart's axes are adjusted to show only the relevant number line area, and are renamed to display the metric name and unit. We chose to show metric over the covered distance rather than over time because it was a more interesting and less convoluted display of the information.

Goals: we have a goal tracking system for the created **User** object, which is saved in a file and is available through multiple application uses. **Goal** class encapsulates the progress of a user-selected sport metric (that has to have isUsedInGoals() indication as true). The **User** object can have up to 5 tracked goals, which is done due to limited display space. To update all goals we use the observer design pattern, which updates all goals of a specific metric, once a new relevant **Activity** object is created.

Exception handling: if an erroneous or incomplete gpx file is uploaded to the application, a sport metric that is missing data will not be calculated and its display will be “N/A”, thus indicating to the user that the information is not available or is corrupted. This is done by checking if all relevant optional fields of the GPX object exist and if at least one field is missing, the output is the exception case.

Code cleanliness: we have pushed for a clean and commented code that would adhere to important Java code practices. We have achieved that by using SonarLint plugin in our IDEs, which marked issues and code smells. All of the issues have been tackled except one minor code smell, which is the Metric abstract class not being an interface in Java 8.

Location of main Java class:

‘\src\main\java\org\softwaredesign\GUI.class’

NOTE: Please follow this tutorial in order to run the code. You will have to download the JavaFX SDK locally, add it to the project structure, and link it in the VM options. It is extremely well explained and easy to do. Apologies that this is the only way to run, but JavaFX has been difficult to say the least and we appreciate your cooperation. If you have any questions on how to follow the tutorial please contact us and we will reply immediately.

Location of JAR file:

We were able to create the executable but once executed it throws the error “**Error initialising QuantumRenderer: no suitable pipeline found**”
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	Charts code	5	3
Martynas	Updating metric for goals and charts	5	1
Baher	Goal code	5	8
Baher	Charts code	5	4
Erik	Goal Code	5	6
Abhisaar	Charts code	5	2
Baher	Map code	6	9
Martynas	Error detection in Metrics	6	2
Martynas	Singleton design patter	7	1
Martynas	Class diagram changes	7	6
Martynas	Design patterns	7	2
Martynas	Implementation description	7	1
Martynas	Code cleanup	7	2
Baher	Diagram and Textual descriptions	7	5
Martynas	document fix ups	7	1
Erik	Final review, edits, adjustments on doc	7	3.5
Abhisaar	Summary of Changes	7	2
Abhisaar	Sequence Diagram Changes	7	1
Abhisaar	Sequence Diagram Textual Descriptions	7	3
Abhisaar	Transferring of Content, Final Reviews, Grammar Edits, & Formatting	7	1
All	Uni coffee meeting :) hehehe	7	1
Martynas	Trying to create JAR files (unsuccessful)	7	1
Bahr	Trying to create JAR files (unsuccessful)	7	2
		TOTAL	65.5