Assignment 3

Team number: 26
Team members

Name	Student Nr.	Email
Harsh Vijay	2696378	h.vijay@student.vu.nl
Sarthak Bajaj	2699648	s.bajaj@student.vu.nl
Vilen Geghamyan	2703673	v.geghamyan@student.vu.nl
Philip Vacca	2712798	p.p.p.vacca@student.vu.nl

Summary of changes of Assignment 2

Author(s): Sarthak

Modified parts from Assignment 2:

General: Changed the way we were implementing the project with new classes and different structure than before
Class Diagram:

- Added User interface which interacts with the user
- Added Map visualisation
- Changed the sports class
- Added new abstract Metrics class and it subclasses
- Made new relations among classes
- Added new classes for user interface and other necessary classes

State Machine:

- Changed the state machine diagram according to the new class diagram and the improved implementation

Sequence Diagram:

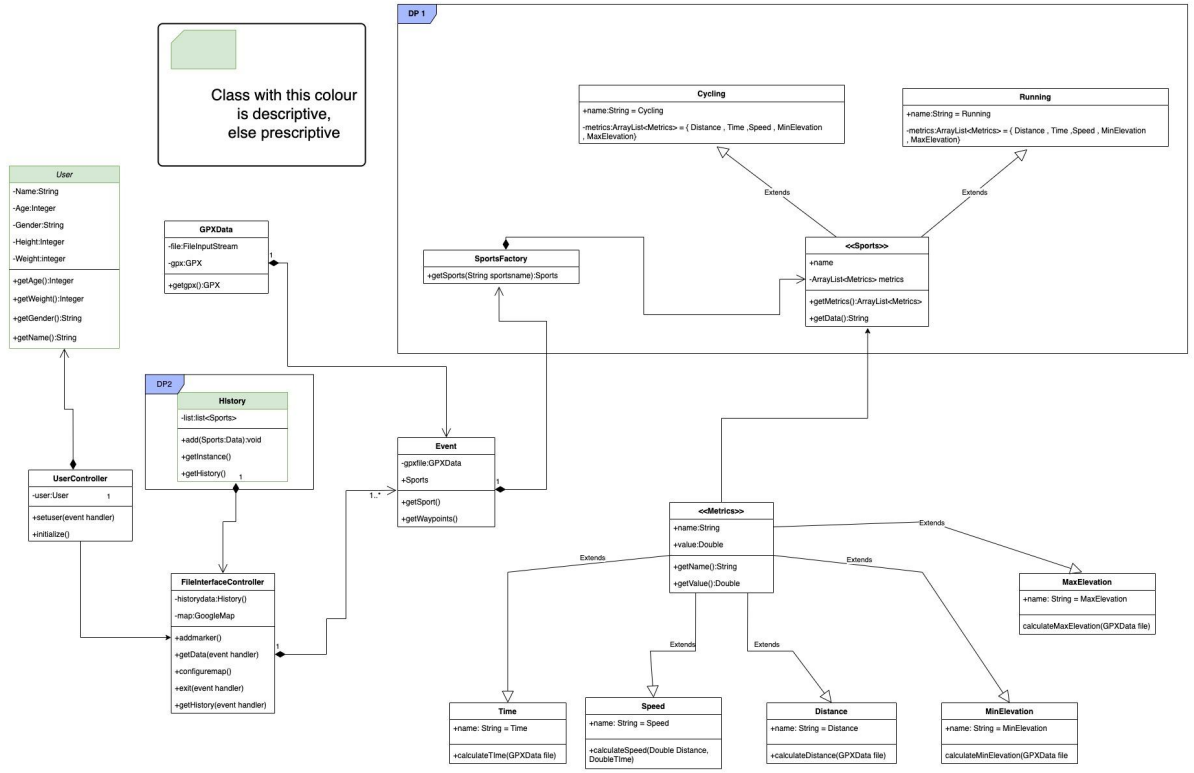
- Changed the sequence machine diagram according to the new class diagram and the improved implementation
- The sequence diagram is more precise.

Implementation:

- Added Map
- Added User Interface with the help of javafx
- Improved the code and added history as a bonus feature

Application of design patterns

Author(s): Sarthak



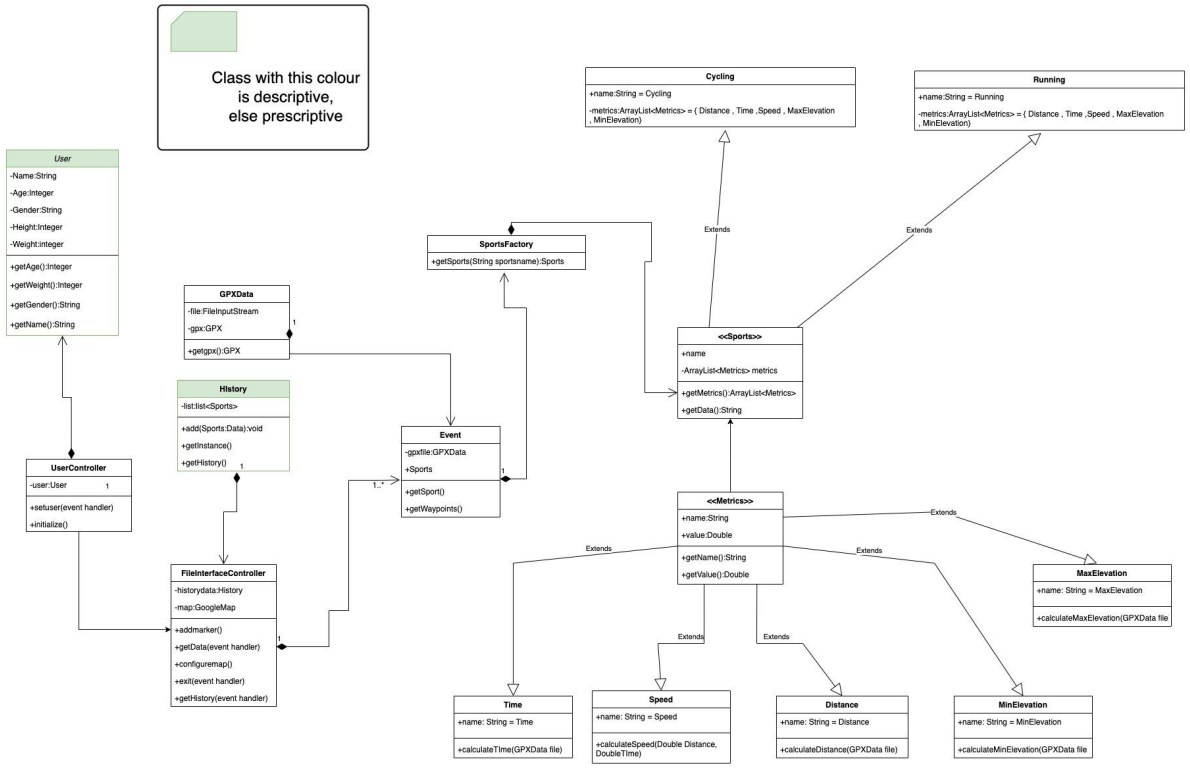
	Design Pattern 1
Design pattern	Factory
Problem	Sports being an abstract class that has subclasses and their initialisation depends on the sports name.
Solution	By applying the factory method, we use a single class to tackle the problem of the initialization of classes.
Intended use	The SportsFactory would initialise the subclasses of sports depending on the sports name provided. This will create an instance of any subclass of sports which will save the metrics as a list.
Additional remarks	The class provides a better and easy way how the initialization of an abstract class works depending on the input provides by the user

	Design Pattern 2
Design pattern	Singleton
Problem	In our implementation, the gpx manager should only have one instance of

	history that stores all the past entries of sports. If there are multiple entries of history we will not be able to access all the data together
Solution	By applying Singleton design patterns, we can prevent any new instance of history to be created during the entire implementation.
Intended use	We will use this to create a private constructor for the class and use getInstance to access the instance of history. You can get access to the data by using gethistory()
Constraints	The class with the singleton method does not have a public constructor so they cannot be initiated outside. This makes it slightly difficult for other people to understand but at the same time, it prevents any mistakes in our implementation.
Additional remarks	This is a good way to prevent the class from getting initiated more than once.

Class diagram

Author(s): Sarthak



User

The **User** class is used to store the user data.

Attributes

- **Name** - It stores the user name that they wish to be called by the GPX manager.
- **Age** - It stores the age of the user as an Integer.
- **Gender** - It stores the gender of a person as a string.
- **Height** - It stores the height of a person as an integer in centimetres.
- **Weight** - It stores the weight of a person as an integer in kilograms.

Operations

- ***getAge()*** - It returns the age of the user they put into the manager
- ***getWeight()*** - It returns the age of the user they put into the manager
- **getGender()** - It returns the gender of the user they put into the manager
- ***getName()*** - It returns the name of the user they put into the manager
- ***getHeight()*** - It returns the height of the user they put into the manager

History

It stores the sports data in a list.

Attributes

- List<Sports>- an array list of type sport which stores the data of sports which can be accessed when you want to see the history of tasks
- History - It is a private static instance of history that is used to return the instance of history when getInstance is called. It is used as a part of singleton design patterns.

Operations

- getInstance() - Used to return the Constructed instance of History
- getHistory() - Used to return all the data of files parsed so far.

History class is in a shared aggregation with file interface controller which uses it to access the data . Without this File interface controller cannot perform the desired functions hence would not work.

GPXData

This class is used to parse and store the gpx file. It checks if the file exists and if it exists it converts the xml file into different components such as waypoints that store the coordinates.

Attributes

- file **-** This attribute creates a file input stream that helps to read the desired file from the system.
- gpx- This attribute creates an instance of GPX class from the library which parses the gpx file into its subcomponents.

Operations

- getgpx() - Returns the parsed data of gpx file, of datatype GPX(Class from the jpx library)

Event

This class creates an event which has an instance of a parsed gpx file in GPXData and Sports. At any moment there can be only a single instance of Event.

Attributes

- Gpx - This stores an instance of parsed gpx file
- sport - This stores an instance of subtypes of abstract sports class which depends on the sports name chosen by the user

Operations

- getSport() - returns the instance of sport
- getWayPoints() - returns the list of waypoints that is the coordinates from the gpx file

Event has a shared aggregation relationship with SportsFactory class due to the reason, SportsFactory is used to store an instance of abstract Sports class following the "Factory Method" design pattern. Therefore Event class cannot exist without SportsFactory class.

Event has a shared Aggregation with GPXData because event contains an instance of gpxdata which is needed to calculate metrics of sports. Which is done inside Event class as sports is an attribute of Event class. Hence Event class cannot exist without GpxData

FileInterfaceController

This class controls the UI for asking file path and sports name and also the map which is displayed after parsing the file.

Attributes

- Map - This is an instance of google map from the library which initialises the map component.

- historydata - This is an instance of history which stores the history of sports data.

Operations

- configureMap() - This initialises the basic map component
- getData() - it initialises event class which further calculates sports metrics and then it is stored in the object of History class and also adds markers to the map with respect to the coordinates of the gpx file.
- addmarkers() -it adds marker to the map
- exit() - it is an event handler that exits the program when the exit button is pressed
- getHistory() - displays the history of the gpx file parsed so far.

FileInterfaceController has a shared aggregation with the event class because it is used to get input from data which is further used to calculate data in the event class. So without any input event class cannot exist.

UserController

This class controls the UI for getting the user data from the user. It uses labels , text fields and a button as components of the UI.

Attributes

- user - It is an instance of the User class that stores all the data that the user provides.

Operations

- setUser() - it stores the data in the user class when the submit button is pressed by gathering information from the text field through which the user gives input to the program.

UserController has got a shared aggregation relationship with "User" as it stores an Instance of "User". Therefore UserController cannot exist without "User".

Metrics

This is an abstract class that defines attributes of the sport class: Distance, MaxElevation, MinElevation, Speed, Tim. It calculates the metrics data taken from the gpx file which is further stored in the history class and returns backs the results to the abstract *Sports* class which returns it to the *Event* class which afterwards gives the results to the FileInterfaceController.

Attributes

- Name - it stores the name of the sport
- Value - it gets the calculated values of the sport

Operations

- getName() - returns a string which contains the name of the sport
- getValue() - returns the calculated values of the sport

Time

It is an abstract class which extends metrics and calculates the time from the data.

Attributes

- Name - it stores the name as Time

Operations

- calculateTime() - it calculates the time and then returns it to the *Metrics* class

Speed

It is an abstract class which extends metrics and calculates the speed from the data.

Attributes

- Name - it stores the name as Speed

Operations

- calculateSpeed() - it calculates the speed and then returns it to the *Metrics* class

Distance

It is an abstract class which extends metrics and calculates the distance from the data.

Attributes

- Name - it stores the name as Distance

Operations

- calculateDistance() - it calculates the Distance and then returns it to the *Metrics* class

MinElevation

It is an abstract class which extends metrics and calculates the minimum elevation from the data.

Attributes

- Name - it stores the name as MinElevation

Operations

- calculateMinElevation() - it calculates the minimum elevation and then returns it to the *Metrics* class

MaxElevation

It is an abstract class which extends metrics and calculates the maximum elevation from the data.

Attributes

- Name - it stores the name as MaxElevation

Operations

- calculateMaxElevation() - it calculates the maximum elevation and then returns it to the *Metrics* class

SportsFactory

This is a class that initiates the sports class. It acts as an initializer for the class. This works as a factory method for Sports abstract class.

Operations

- getSports() - initialises subclasses of sports class based on the provided sportname.

SportsFactory class shares a shared aggregation relationship with abstract closs sports due to the reason that it stores instances subclasses of abstract class sports. Therefore it cannot exist without abstract class Sports

Sports

It is an abstract class which acts as a base for all the sports.

Attributes

- name - it stores the name of the sport
- metrics - it is a list of metrics which stores them and it can be unique for each sport which makes the code extensible

Operations

- getData() - returns a String which contains the name and value of every metrics.

Running

Running extends Sports, which is an abstract class with time, distance, speed, max elevation, min elevation.

Attributes

- name - it is equal to running
- metrics - it is a list of metrics which stores data such as distance, speed , etc.

Operations

- getData() - returns a String which contains the name and value of every metrics.

Cycling

Cycling extends Sports, which is an abstract class with time, distance, speed, max elevation, min elevation.

Attributes

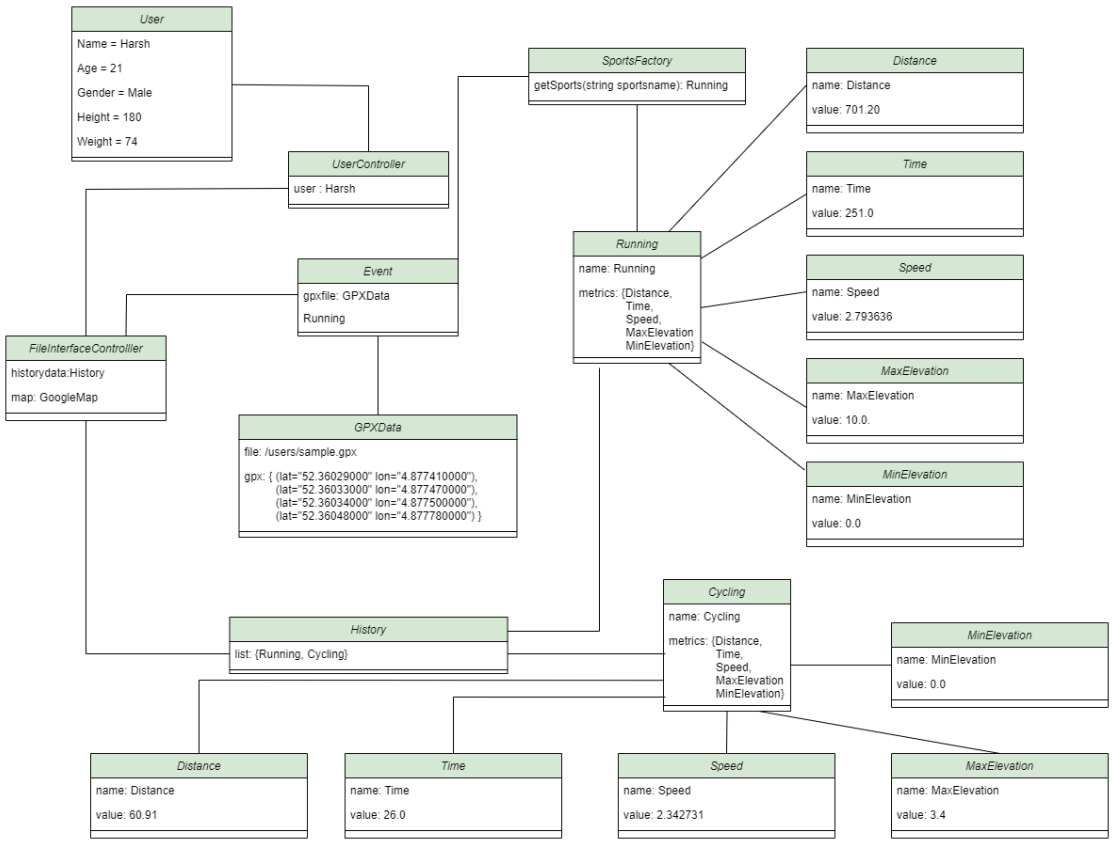
- name - it is equal to cycling
- metrics - it is a list of metrics which stores data such as distance, speed , etc.

Operations

- getData() - returns a String which contains the name and value of every metrics.

Object diagram

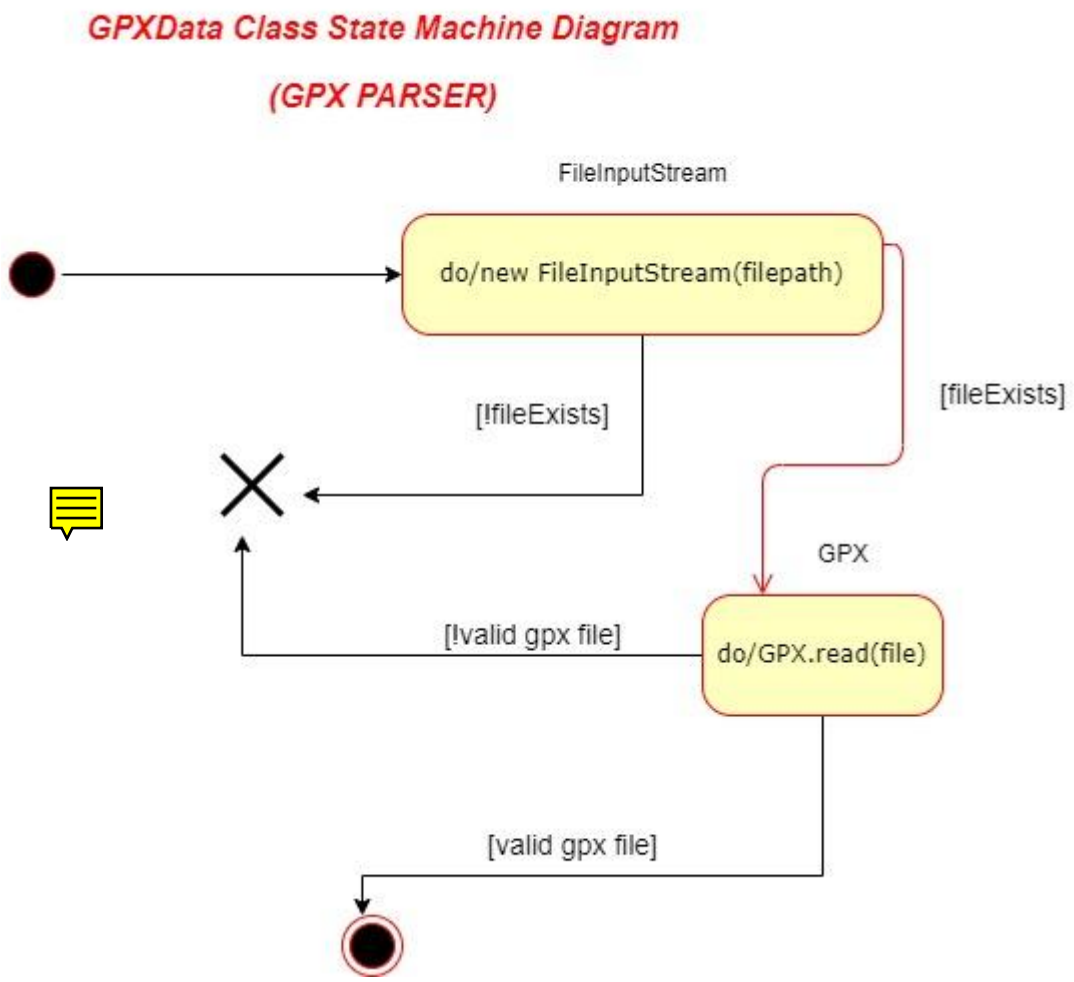
Author(s): Harsh



The diagram shows the object diagram for the GPX manager. The purpose of this diagram is to better illustrate, with an example, how the classes described in the Class Diagram will interact with each other. This is the snapshot of an instance of our manager. The *UserController* has an instance of a user which is “Harsh” in our snapshot. *UserController* is getting the data of the user from the *User* where we have the name, age, gender, height and weight as instances.*FileInterfaceController* object is an interface of our manager which has two instances History and GoogleMap, the first instance shows the old data saved in our manager and the latter instance is showing the visualization of data into the map. Afterwards, it goes to the *Event* Object where we have two instances, first one is GPXData which is getting the data from the *GPXData* object. In *GPXData* we are getting the file path and the waypoints from the gpx file. The second instance for Event is Running which will go into the *SportsFatory* which takes the sport “running” as its instance and extends it to *Running* where metrics of running are calculated and then its is saved into a *History* object. In *History,* we have a list of running and cycling. Cycling is the saved data that was calculated before and was saved into the *History*.

State machine diagrams

Author(s): Vilen Geghamyan

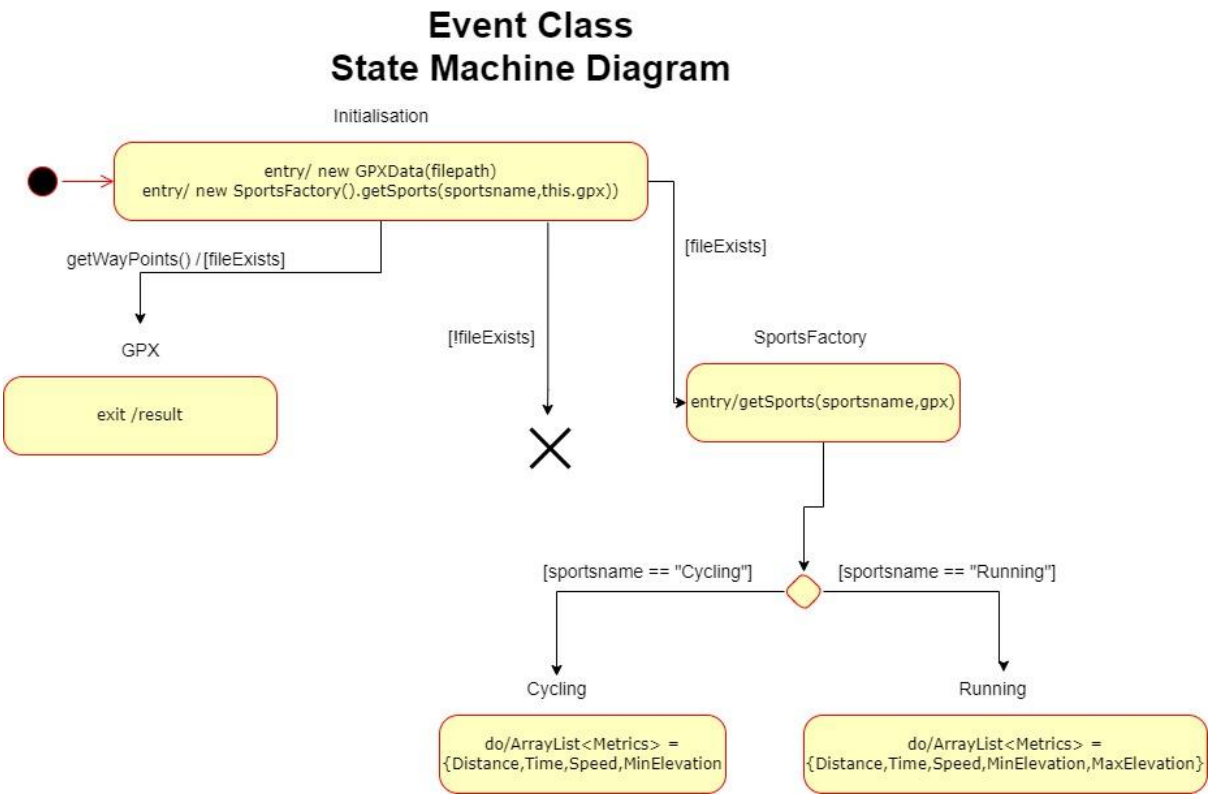


In this state machine diagram, we describe the GPXData class(which serves as parsing the gpx file). We begin with creating a new object of class “FileInputStream” with the argument the filepath and then we check the file. If it exists we transit to the new state “GPX”, if the file does not exist, we terminate. While being in the new state GPX. The GPX built-in methods check the type of the parsed file. If the file is of type gpx, the file is validated and we reach to our end state, else if the file is not of type gpx, we get an invalid file error and if the file is valid it is parsed and broken down into sub-components which are further used to carry out the calculation and visualise into the map.

In our previous implementation we carried out a bit of a different approach. There besides our "GPXPaser" class, we had implemented an additional Input class whose object we used to initialise an input for the "filepath". Only after that we used the initialised filepath input to pass it to the GPXParser class to create an new Instance of "FileInputStream" with argument the filepath. The remaining of our File-Parsing design was the same as our current one.

Our new approach has reduced the classes which take care of parsing the gpx file, to two classes, GPXData made manually by us, and FileInputStream java built-in object class.

With this design, we simplify in such a way that the GPXData class has one single responsibility which is to instantiate a FileInputStream object, who on the other hand takes care of parsing the file.



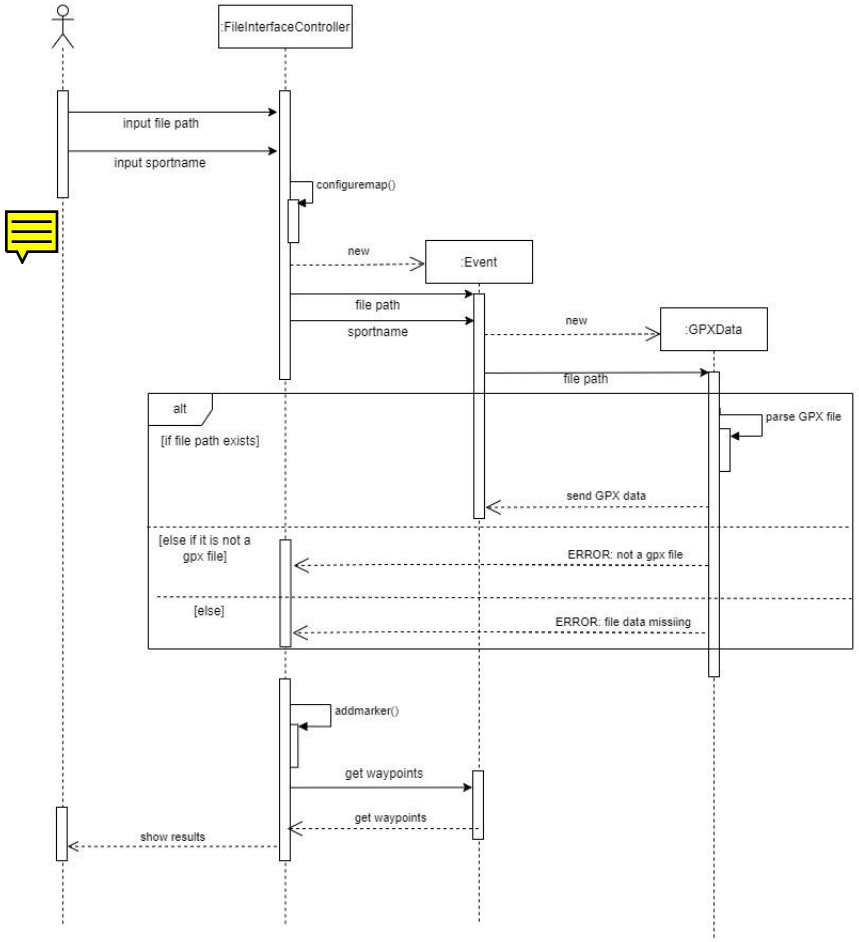
In this state machine diagram, we take a look a the present states of our "Event" object class. The module itself is being initialised by two objects of different classes. The first object is an instance of GPXData with an argument the filepath. The second object is an instance of our newly created "SportsFactory" class, which follows the creational design pattern called "Factory Method". With the help of that method we are able to get the extensions types (subclasses) of abstract class Sports without hard coding and with that design, our subclasses are more maintainable. Before that, it's checked if the file exists. If it doesn't, we terminate, or else if we check the sportsName's string value. We have two options here, two subclasses: Cycling and Running. Our guard checks the value of the string whether is equal to "cycling" or "running", it also ignores cases, and then it proceeds to its corresponding state. Another possible state is in the GPX class, where it uses the built-in libraries to get the WayRoutes() gpx data.

With the newly learned design patterns, we have managed to significantly improve our design. In the current state machine diagram as mentioned above we can see an applied "Factory Method" design pattern, in order to unify the place for object creation which are subclasses of abstract class "Sports", which results in more cuztomisable anti hard-coded design.

Sequence diagrams

Author(s): Philip and Harsh

Visualization of the map



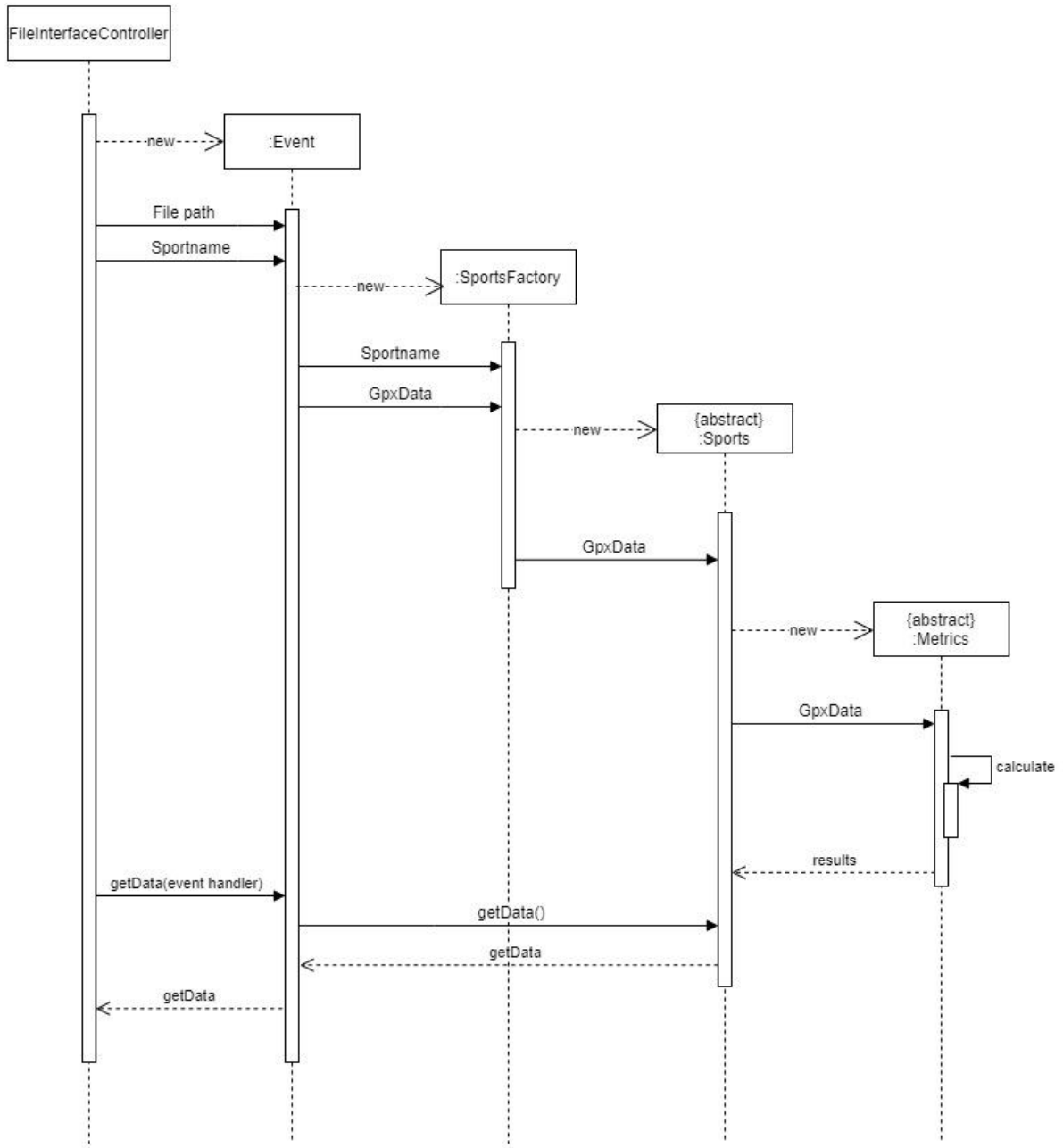
This sequence diagram describes how the user input is processed, parsed and turned into a map that can be seen by the user. The first event described in the sequence diagram above is the user inserting the file path of the GPX file that contains the sporting data, together with the name of the sport.

The file path and the name of the sport will be inserted into the FileInterfaceController class which acts as our UI, whereafter it will call the function configuremap(), which creates an empty map. Then a new instance of the Event class is created and the path and sportname are sent to it.

Within the Event class, an instance of the object GPXData is created and the path is sent to it. Inside the GPXData object, the file path will be checked: If the file path does not exist then it will send a return message to the command interface, stating that the file path they entered does not exist. If the file path exists but it does not point to a GPX file, then a return message will be sent to the command interface, stating that the file that corresponds to the file path they entered does not point to a GPX file. In both cases the user is prompted to submit a valid file path. If the file path exists and it is indeed pointing to a GPX file, the file that belongs to the corresponding file path will be parsed and the GPX data which as an object will be stored in the event class.

After the GPX data class and event class is initialised FileInterfaceController calls addmarker() which uses the waypoints which are retrieved by the help of getwaypoints() function in the event class. The add markers add the marker to respective coordinates which is displayed to the user in the map.

Metrics Calculation and displaying it on the Interface



This sequence diagram describes how a metric is calculated. It will only describe one (abstract) metric, but in reality there will be multiple metrics calculated at the same time.

This diagram starts after the user has submitted the filepath and the sportname into the FileInterfaceController. The first thing that happens is that a new instance of Event is created with the file path an sports name as parameters. Then, an instance of SportsFactory is called. The sportname and GPXData are sent to it; not that how the gpxf file is parsed and the GPX data is retrieved from the file path is already shown in the previous diagram, so we will not show it here again. The SportsFactory clas takes the sportname an initiates the relevant class (E.g, if sportname = running, then the running class is initiated) and the GPXdata is sent to it.

From the relevant class a new instance of the relevant Metric class is created and the GPXdata is put into it. Withing the Metric class, the appropriate calculations are done with the GPXdata and the results of those calculations are sent back to the abstracts sports class. This class then contains the results of all the metrics. Whenever the GO button is pressed in the FileInterfaceController UI, the function getData(event handler) is called, which in turn calls the function getData() to the Sports class which contains the results of the calculated metrics. After that, the results are sent back to the Event class and consequently sent back to the FileInterfaceController.

Implementation

Author(s): Sarthak

Our focus, when we started designing the class diagrams, was to make the system extensible and also easy to understand. While progressing through the diagrams there were many changes that were done due to the variety of challenges that came up during the implementation of the design. We started with just trying to parse a gpx file, which was not easy in the first place. Much of our time was spent on figuring out how we can access the data that has been parsed. Reading through the readme file of the library that we use to parse gpx gave us an idea of how we can access different attributes of the parsed data.

In this assignment we had to change our class diagram which also drastically affected our implementation. Our first few days were taken into account to fix our implementation according to the new class diagram. During this time we came across different problems. For instance, one of them being how to get every coordinate from the gpx file because of the different standard attributes that are in a gpx file.

The further challenge that we encountered while implementing the project was to get the map working. While working with different java libraries for maps we encountered different problems, some libraries were not able to work with our java code and some needed csv files to visualise the data. After many hits and trials we tried the very first library which was suggested by the professor in the project description. Which worked for us but at the same time it had no feature to represent routes so we had to work with markers to visualise the route of the gpx file.This further took much of our time

The strategy that we implemented this time was to first reimplement the implementation and work along the problem that came across. One of them was to make use of the parsed data to calculate different metrics. As we progressed we encountered different problems such as problems with javafx and also with maps. Which were resolved with time by the help of contribution from different team members who tried to see what is causing the problem and how can we tackle it

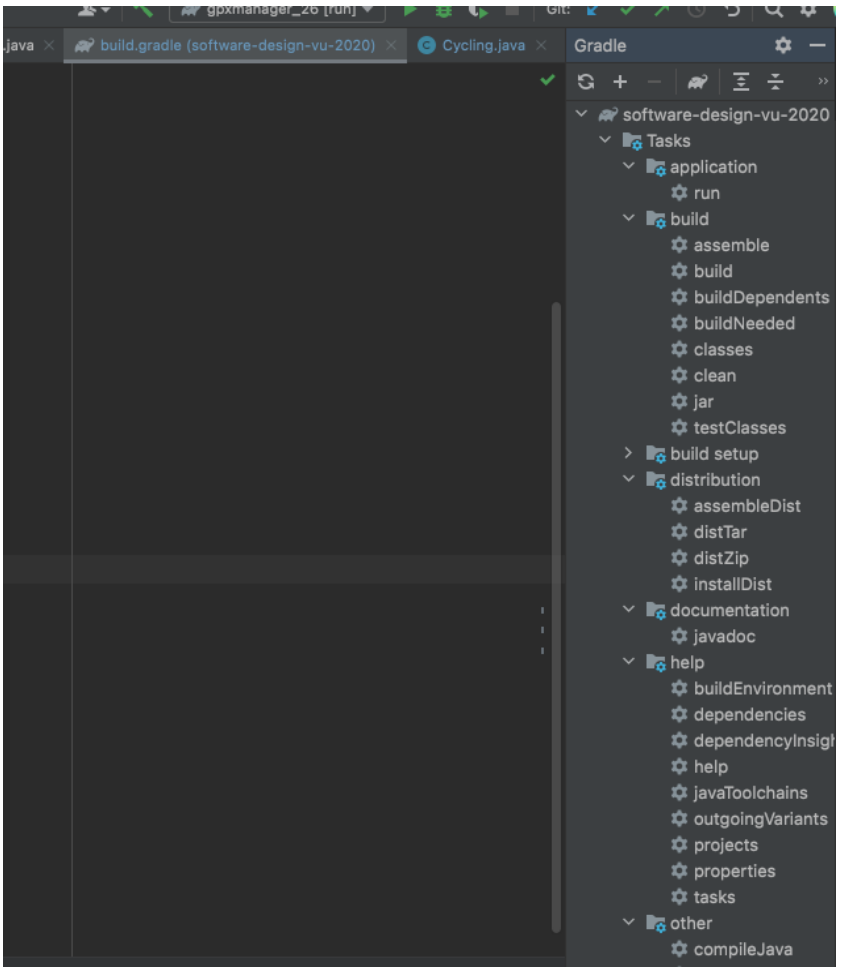
The key Solutions for our system were

- FactoryMethod for Sports abstract class- We used factory method to initialise the sub classes of sports which uses a different class to initialise the sports class depending on the sports name which makes it more convenient for a developer to understand and add things at a single space if they ever wish to extend the subclasses of sports
- History of tasks -Storing the history of tasks in a list. In our implementation, we used a list to store all the instances of the subtypes of Sports. We have a basic function in the class: add() something to the list. In our implementation , we can get the entire history with gethistory() which returns a String with the data of the entire history of sports.
- Initialising the map - we use Gmapsfx to visualise the map that has javafx components such as Google map which is used in an fxml file to visualise the map. And the library also provides us with a helper function to add markers. We convert every coordinate from wayPoint to lat long which can than be parsed in google maps to get the marker pointing at the coordinate.

How to run the project

Rather than running the main class. We need to use the run configuration under the application folder of gradle to run the project.

You will able to find it on the right side of the screen as shown below



The location of the main java class is

src/main/java/softwaredesign/main.java

The location of the jar file

out/artifacts/software_design_vu_2020_jar/software-design-vu-2020.jar

Video

Maximum number of pages for this section: 2

Time logs

Team number	26		
Member	**Activity**	**Week number**	**Hours**
Group	Reviewing of Assignment 2	6	2
Group	Correction of Assignment 2	6	3
Sarthak and Vilen	Class diagram	7	4
Sarthak	Design Pattern	7	3
Vilen	state machine diagram	7 & 8	3
Sarthak	class diagram	7 & 8	2
Harsh and Philip	sequence diagram	7 & 8	4
Harsh	object diagram	8	3
Philip	sequence diagram	8	2
Group	Implementation	7 & 8	5
Sarthak	Final Implementation	8	2
Group	Meeting for the report	8	1
Group	writinng the report	8	3
Group	Formatting the report	8	1
		TOTAL	38