Overview

In this third and final CS1002 practical, I was asked to design, implement, and test the core classes required for an activity tracking software system. This practical was, in my opinion, quite different from the previous two, as there wasn't a specific format to follow for the output. Another very important difference is with the testing: there were no stacscheck tests for this assignment, so I had to hard-code the tests for my program.

All in all, I have achieved the previously established goals:

- Create an initial UML class diagram, and implemented it in Java
- Edit and complete the Java code generated by the initial UML class diagram
- Produce a final UML class diagram
- Test my code (see **Testing** for more details)

Initial Design

The following UML class diagram is the initial UML class diagram for the W11 Practical. It was made on the lab machines, using Modelio. It is labelled *InitialUML.png*, and I will refer to it as the initial UML.

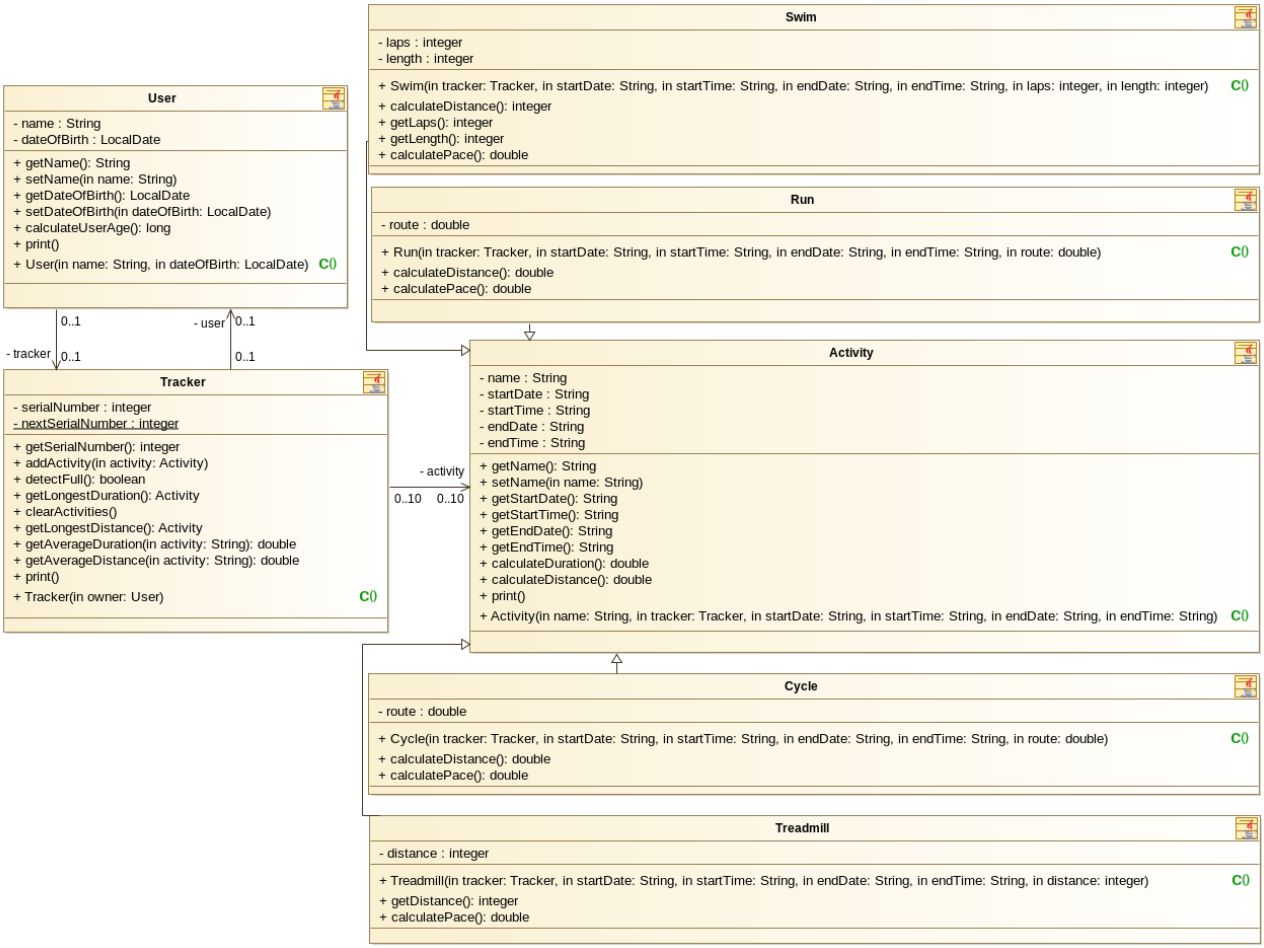

Figure 1: *InitialUML.png*, the initial UML class diagram

I will now explain this diagram in more details, giving a quick overview of the different classes, how they are linked, and then studying them one by one.

There are seven Java classes in the initial UML: *User*, *Tracker*, *Activity*, *Swim*, *Run*, *Cycle*, and *Treadmill*. What each class do is relatively straightforward: *User* represents the human users, *Tracker* the activity trackers, *Activity* the activities in general, and *Swim*, *Run*, *Cycle* and *Treadmill* four special types of activities.

As you can see in Figure 1, some classes are linked to each other. I have used two different types of links: associations and specialisation/generalisation relationships.
According to the System Specification, "[each user] is associated with an activity tracker." Therefore, I created an association link from the *User* class to the *Tracker* class. Similarly, as "each activity tracker has […] an owner" (System Specification), I created an association link from *Tracker* to *User*. Knowing that "each activity tracker maintains a list of up to 10 activities logged for its owner" (System Specification), I created an association link from *Tracker* to *Activity*.
There are four specialisation/generalisation relationships in the initial UML: *Swim*, *Run*, *Cycle*, and *Treadmill* being four special types of activities, they are linked to the "main" *Activity* class through these specialisation/generalisation links, from the subclasses to *Activity*.

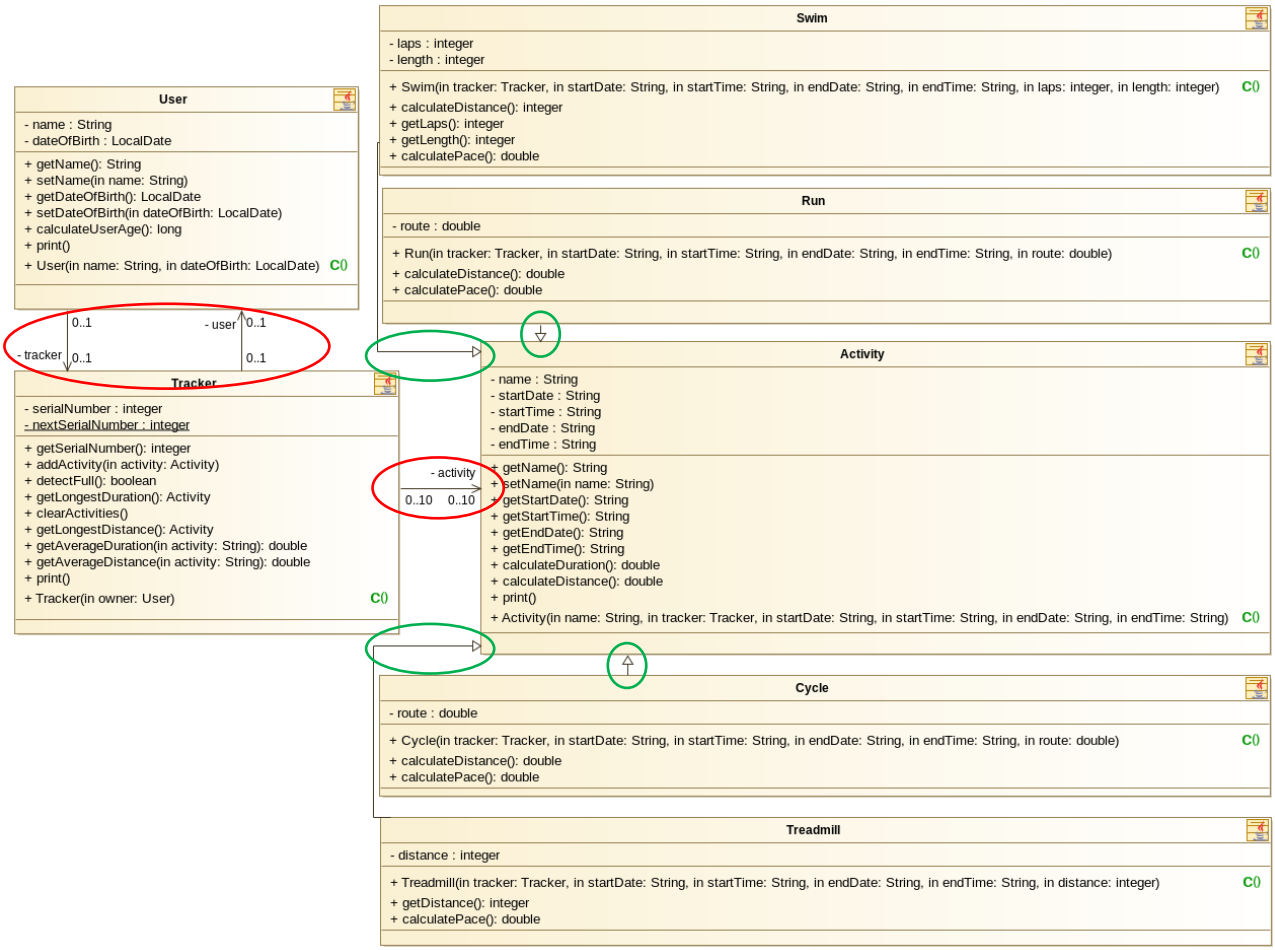

Figure 2: The association links and the specialisation/generalisation relationships

The first class is *User*. It has three attributes, the last one being generated thanks to the association link: a String *name*, a LocalDate object *dateOfBirth*, and an activity tracker *tracker*. The class has a constructor *User(String name, LocalDate dateOfBirth)* (denoted with **C()** next to it in Figure 1) and various operations, according to the System Specification: getters and setters for *name* (*getName()* and *setName(String name)*) and *dateOfBirth* (*getDateOfBirth()* and *setDateOfBirth(LocalDate dateOfBirth)*), *calculateUserAge()*, and a *print()* method.

The second class is *Tracker*. It has four attributes, the last two being generated thanks to the association links: an integer *serialNumber*, a static integer *nextSerialNumber*, a user *user*, and a list of activities *activity*. Just like the *User* class, *Tracker* has a constructor (*Tracker(User owner)*) and various operations: getter for *serialNumber* (*getSerialNumber()*), methods to add an activity and to clear all previously logged activities (*addActivity(Activity activity)* and *clearActivities()*), detect if the log is full (*detectFull()*), get the activity with the longest distance or duration (*getLongestDistance()* and *getLongestDuration()*), get the average distance or duration for an inputted activity name (*getAverageDistance(String activity)* and *getAverageDuration(String activity)*), and a *print()* method.

The third class is *Activity*. It has five attributes, all Strings: *name*, *startDate* and *startTime*, and *endDate* and *endTime*. [One obvious change that I did not notice in my initial UML was to set the type of *startDate* and *endDate* to LocalDate (just like *dateOfBirth* in the *User* class) and not to String. However, this has been updated in my final UML, and I will cover it later in **Final Design**.] It also has a constructor *Activity(String name, Tracker tracker, String startDate, String startTime, String endDate, String endTime)* and multiple operations: getters for all five attributes (*getName()*, *getStartDate()*, *getStartTime()*, *getEndDate()*, and *getEndTime()*), a setter for *name* (*setName(String name)*), calculate the duration or the distance of the activity (*calculateDuration()* and *calculateDistance()*), and a *print()* method.

The fourth class is *Swim*. It has 7 attributes in total: the first five are the five attributes of the inherited class, *Activity*, and the last two are two integers, the number of laps *laps* and the length of the pool *length*. *Swim* inherits of every operation that *Activity* has, as well as its own constructor, getters for the number of laps completed and the length of the pool (*getLaps()* and *getLenght()*), a *calculateDistance()* method, and a *calculatePace()* method as required.

The last three classes, *Run*, *Cycle*, and *Treadmill*, are similar to *Swim*. They have their own constructors, *calculateDistance()* (called *getDistance()* in the *Treadmill* class) and *calculatePace()* methods. *Run* and *Cycle* have an extra attribute called *route*, a list of double, and *Treadmill* has an integer *distance* (the distance covered).

Final Design

The following UML class diagram is the final UML class diagram for the W11 Practical (after editing and completing the generated Java code). It was made on the lab machines, using Modelio. It is labelled *FinalUML.png*, and I will refer to it as the final UML.

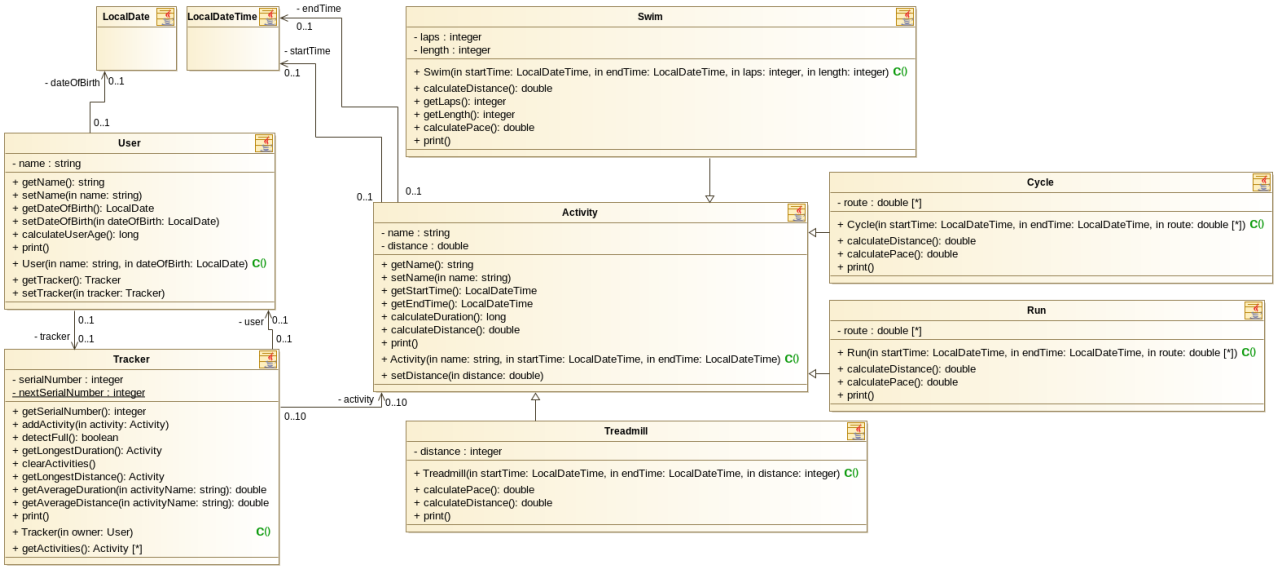

Figure 3: *FinalUML.png*, the final UML class diagram

The first differences between the initial and final UML class diagrams are the new *LocalDate* and *LocalDateTime* classes. These are classes that I've imported in the *User* and *Activity* classes respectively, to use and create LocalDate and LocalDateTime objects.

New classes mean new links. The *dateOfBirth* attribute being a LocalDate, it does not appear anymore in the *User* class, but is generated thanks to the association link from *User* to *LocalDate*. The same thing applies for *startTime* and *endTime*: they are not Strings anymore, but LocalDateTimes.

I added two new operations in the *User* class: a getter and a setter for the activity tracker, *getTracker()* and *setTracker(Tracker tracker)*. I decided to modify this as I could not directly access the methods from the *Tracker* class in the *User* class without a getter. The setter is used in the constructor of the *Tracker* class, to link this tracker to the inputted user.

The *Tracker* class was also modified. I added a new getter for the activities, *getActivities()*, for the same reason as for the *User* class. Another (minor) change was to change the name of the inputs for *getAverageDuration* and *getAverageDistance* methods, from *activity* to *activityName*: I believe that *activity* should refer to an *Activity* object, therefore changing the name to *activityName* for Strings.

Activity is one of the classes that has been modified the most. It only has four attributes, not five, and the attribute types have changed. Firstly, I merged the *startDate* and *startTime* variables into one *startTime* attribute (and I did the same for *endDate* and *endTime*). The new *startTime* is now a LocalDateTime object, which contains a date and a time at the same time – no need to create two separate variables anymore! The new type also helps with calculating the duration between *startTime* and *endTime*, using *duration.BETWEEN(LocalDateTime1, LocalDateTime2)*. I also created a new *distance* attribute.
The methods are also different: I created a new setter (*setDistance(double distance)*) to set the distance, and I deleted the getters for *startDate* and *endDate* as we don't use these attributes anymore. The constructor has also been modified accordingly, and I removed the inputted *tracker* in the constructor, as we don't need to set an activity to a tracker.

The four inherited classes have also been heavily modified to cope with the new changes in the *Activity* class. Adding/deleting attributes in *Activity* implies modifying the constructors of *Swim*, *Run*, *Cycle*, and *Treadmill*. The constructor of the *Swim* class now takes four inputs instead of seven (*Swim(LocalDateTime startTime, LocalDateTime endTime, int laps, int length)*), and all three of *Run*, *Cycle* and *Treadmill*'s constructors takes three arguments instead of six (*Run(LocalDateTime startTime, LocalDateTime endTime, List<Double> route)*, *Cycle(LocalDateTime startTime, LocalDateTime endTime, List<Double> route)*, and *Treadmill(LocalDateTime startTime, LocalDateTime endTime, int distance)*).
The *route* attributes in *Run* and *Cycle* has changed: in the initial UML, its type was just a double ("- route : double" in Figure 1), whereas in the final UML, *route* is an ArrayList of double entries ("- route : double [*]" in Figure 3). This change happened automatically when I generated the UML class diagram from my updated code, as I manually changed the type of the variable from *private double route;* to *private List<Double> route = new ArrayList<Double> ();*.
Finally, I added, in each of *Swim*, *Run*, *Cycle*, and *Treadmill*, a *print()* method, that uses the *print()* method of the *Activity* class (with *super.print()*) and "adds" the appropriate statistics for each activity to the output (see System Specification to check the appropriate statistics for each activity).

Testing

The W11 Practical was different to the previous two, as there were no compulsory stacscheck tests to test our code. Therefore, I had to manually test my classes, by creating an eighth Java file, W11Practical.java, that contains a *main* method. In this method, I created different objects from the previously explained classes, and called some methods on them, to test the output of these invoked methods.

The following table represents the different tests I ran in W11Practical.java. To check the outputs, simply run W11Practical.java in the terminal, and compare the outputs of each test to the theoretical ones in the table.

Method	Test	Result
Swim: *print()*	Create a swim from 10:00 to 11:00 on 25/11/2022, with 20 laps of a 50m pool	Print the activity's name, start/end time, duration, distance (in m), and appropriate statistics (number of lengths completed, length of pool, average number of minutes per length)
Cycle: *print()*	Create a cycle from 11:00 to 11:20 on 25/11/2022, with a route containing the grid references (0;0), (0:10), and (10;10)	Print the activity's name, start/end time, duration, distance (in km), and appropriate statistics (average speed as kilometres per hour)
Run: *print()*	Create a run from 12:00 to 12:30 on 25/11/2022, with a route containing the grid references (0;0), (0:10), and (10;10)	Print the activity's name, start/end time, duration, distance (in km), and appropriate statistics

		(average number of minutes per kilometre)
Treadmill: *print()*	Create a treadmill session from 12:00 to 12:10 on 25/11/2022, with a total distance covered of 5.0km	Print the activity's name, start/end time, duration, distance (in km), and appropriate statistics (average number of minutes per kilometre)
Activity: *print()*	Create a workout from 9:00 to 9:30 on 25/11/2022, with no distance covered	Print the activity's name, start/end time, duration, and distance (0.0km)
User: *print()*	Create a new user called "Anton", born on 13/12/2004	Print the user's name, age, and "No activities logged yet"
Tracker: *print()*	Create a new empty tracker, linked to the user Anton	Print the serial number, user's name and age, and "No activities logged yet"
Tracker: *addActivity()*	Add the previously defined swim to the tracker	Print the serial number, user's name and age, "Summary statistics of all currently logged activities:", and "1. " followed by the activity's details
Tracker: *clearActivities()*	Clear the tracker	Print the serial number, user's name and age, and "No activities logged yet"
Tracker: *addActivity()* Tracker: *detectFull()*	Attempt to add 11 activities to the tracker	"The tracker is full", the tracker's size, and if the tracker is full (true) or not (false)
Tracker: *getLongestDuration()* Activity: *print()*	Clear the tracker, add all five of the previously defined activities, and get the activity with the longest distance	Print the swim's details
Tracker: *getLongestDistance()* Activity: *print()*	Get the activity with the longest distance	Print the cycle's details
Tracker: *getAverageDuration()*	Create a new treadmill session from 12:00 to 12:30 on 25/11/2022, with a total distance covered of 15.0km, add the new treadmill session to the tracker, and get the average duration of the treadmill activities	Print the average duration in minutes of the two treadmill activities
Tracker: *getAverageDistance()*	Get the average distance covered during the treadmill activities	Print the average distance covered in kilometres during the two treadmill activities

User: *print()*	Check the updated tracker of the user Anton	Print out the user's name, age, and last three logged activities (should be Treadmill, Workout, Treadmill)
User: *getTracker()* Tracker: *print()*	Check the updated tracker from the user's interface	Print the serial number, user's name and age, "Summary statistics of all currently logged activities:", and, for each logged activity, the activity's index in the tracker (1 for the first logged activity, 2 for the second, etc.) followed by the activity's details
Tracker: *getSerialNumber()* [N.B. I did not really want to check that method specifically, rather the system of automatically updating the serial number]	Create four new users, each with a tracker	Print the serial numbers of each activity tracker

Figure 4: The different tests ran in W11Practical.java

The methods that weren't explicitly mentioned in the table were tested while using these methods. For example, the *calculateDistance()* methods in the *Activity*, *Swim*, *Run*, *Cycle*, and *Treadmill* classes were not explicitly tested, but we can access the return values of these methods when we print out the activity's details using *print()*.

Evaluation

After thoroughly testing my different classes, I am proud to say that my program works: I created an initial UML class diagram, generated Java code from it, completed and tested the code, and produced a final UML class diagram for my final version of the code.

What I find the most striking in this third and final CS1002 Practical is the difference between my initial and final UML class diagrams. The number of missing methods and badly defined attributes in the initial UML is considerably large. This made me realise two things:

- One, my modelling skills are far from perfect.
- Two, my initial thoughts and ideas are usually wrong/incomplete, and I should always change them or think thoroughly before developing them.

Conclusion

In my opinion, this CS1002 W11Practical was by far the hardest practical of the three. It required much more high-level reflection (association/inheritance between classes, how to access attributes/methods from one class to another) and involved new concepts, such as inheritance, and objects (for example Lists and ArrayLists). Not to mention the whole modelling part, which we never encountered before in a previous Practical.

As explained in the long comments in the Tracker.java and Run.java files, there are numerous ways to code the different methods. If I had more time, I would've loved to test and compare the outputs/time complexity of the different implementations that I mentioned in my code. Furthermore, I would've loved to try to code a Graphical User Interface (GUI) for my activity tracking system.