220002145 (Nnamdi Ekwe-Ekwe) 25/11/2022

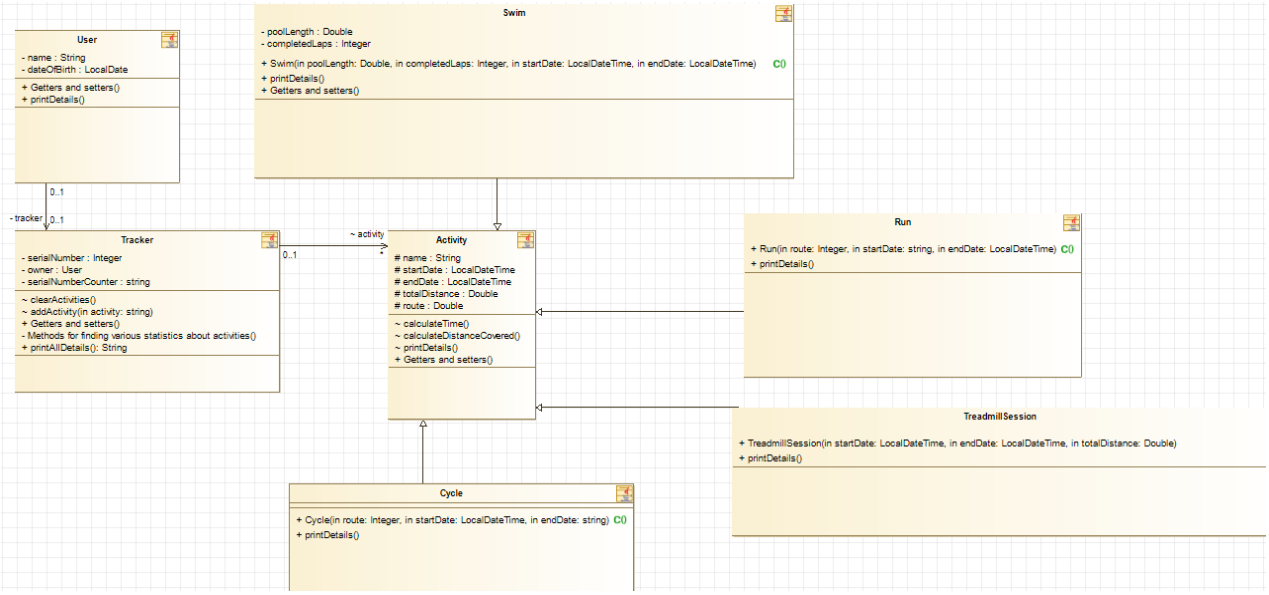
CS1002 WEEK 11 PRACTICAL

Overview:

The practical required us to build a set of classes that would form part of a hypothetical fitness tracker software system, according to a given system specification. They need to support the ability to create trackers for users, store activities in said trackers, and perform various statistical analyses/manipulations on the activities. The design starts with an initial UML diagram, which is then to be transformed into written java code. After the necessary modifications and additions have been implemented, the java code is then used to generate a final UML diagram displaying the classes. I have completed all requirements of the week 11 practical, including implementing all parts of the system specification, at all stages aiming to design the program in object oriented fashion.

Initial Design:

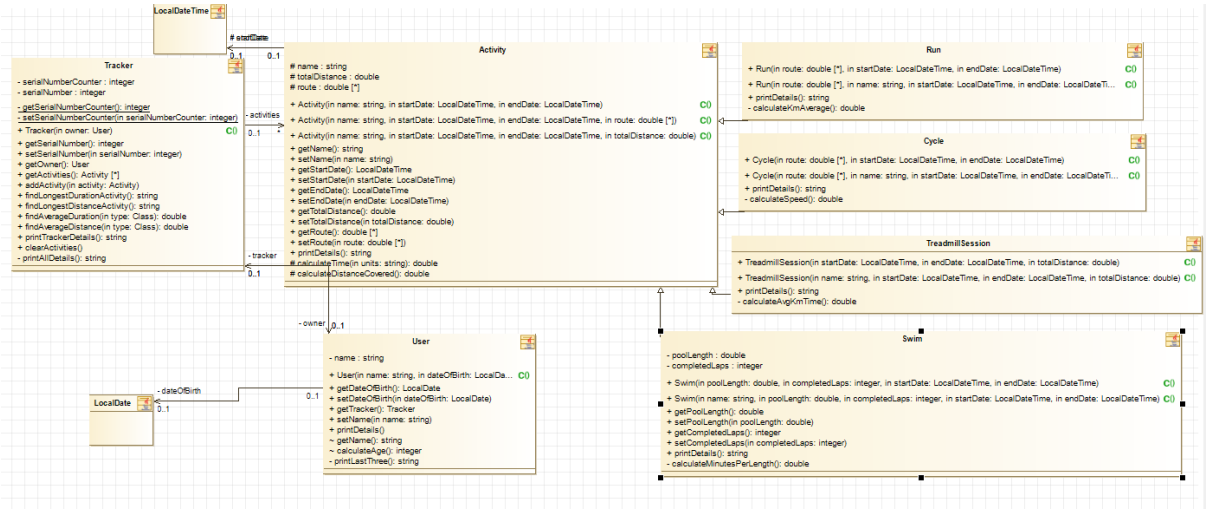
My implementation contains 3 main classes: Activity, Tracker and User. Activity is further subdivided into the subclasses Swim, Cycle, TreadmillSession and Run. (200% zoom to see diagram)



I have used noun identification to decide the main classes constituting the implementation, and the subclasses of Activity follow naturally from the system specification. I have chosen to store all attributes that appear in more than one subclass of Activity in Activity itself in order to reduce code repetition, and have kept the initial diagram quite vague on purpose - the aim of this was to give me a rough idea of what the project will entail and reduce development time by producing some starting code, not to provide a fully accurate representation of the final system.

Final design

My final design ended up being significantly more similar to the initial UML diagram than I had expected, with all major classes and subclasses remaining the same, as well as most methods - the main difference being several private methods that were needed to aid in the calculation (trying to keep to one method -> one purpose) of the various statistics, and changing the visibility of many methods and attributes as I understood what my implementations would look like more during coding. (200% zoom to see diagram)



During development, I have experienced firsthand why OOP is so useful and convenient. Thanks to polymorphism and inheritance in the methods and classes associated with the Activity class, when something went wrong or didn't behave as expected during initial testing, it was enough to modify one field/method in Activity to resolve many errors. Further, designing adaptable methods and inheriting from Activity in the subclasses resulted in low repetition. This drastically sped up development, as many methods ended up just amounting to calling a method from the superclass with little modifications. Because of encapsulation, development over multiple days was a breeze, and the increasing size of the codebase did not result in needless confusion or wasted time attempting to work out what each part of the program does. Abstraction has allowed me to model the real world situation outlined in the system specification using the relevant algorithms and data structures available in java, many of which have proven to be suited perfectly for it, as the specification lends itself very naturally to an object oriented programming style: it contains easily identifiable classes, subclasses and methods. Within the implementation, it is very difficult to "break" the program and cause it to behave unexpectedly, as I have ensured that each user can only hold one tracker and one tracker can only belong to one user, and therefore one can't make use of the getter and setter methods to shuffle around activities and tracker ownership. One of the few ways that one can "break" this program is by creating activities that aren't stored in any tracker, but this is actually a relevant use case: one may create an activity, intending to add it to multiple trackers, and therefore it makes sense that it is created independently (say if someone was to go running with their friends). All attributes in the program are only accessible and modifiable by getters and setters, as this (in my opinion) improves code readability and gives me a consistent way of accessing data through the program.

Testing:

The way in which I have decided what to test and how is by just reading each line of the system specification and writing code to test that the program fulfils it correctly. All testing is included in the W11_Practical class in my src folder.

Test	Method	Result	Pass (Y/N)
Creating a user with an empty tracker	User constructor	Succesfully creates user with empty tracker	Y
Creating multiple users with trackers	User constructor	Succesfully creates multiple users with multiple empty trackers	Y
Printing the details of a user	printDetails()	Succesfully returns the users details as a string formatted to fit the desired output, ready to be passed to a print method	Y
Adding activity to tracker	addActivity(Activity)	Succesfully adds activity to the specified tracker	Y
Adding multiple activities	addActivity(Activity)	Succesfully adds multiple activities	Y
Adding more than 10 activities	addActivity(Activity)	The activity is constructed such that this is not possible; therefore no new activities are created beyond 10, and for each one an error message is printed.	Y
Clearing the activities a tracker holds	clearActivities()	Sets all the activities stored in the tracker to null	Y
Finding the activity with the longest duration	findLongestDurationActivity()	Succesfully finds the activity with the longest duration	Y
Finding the activity with the longest distance	findLongestDistanceActivity()	Succesfully finds the activity with the longest distance	Y
Finding the average duration of a given type of activity (in minutes)	findAverageDuration()	Succesfully finds the average duration	Y
Finding the average distance of a given type of activity (in km)	findAverageDistance()	Succesfully finds the average distance	Y
Printing all details of the tracker	printTrackerDetails()	Succesfully prints the details of the tracker as well as all summary statistics logged for each type of activity	Y
Creating an activity	Activity constructor(s)	All constructors associated with the Activity superclass work as intended	Y
Calculating the statistics for each type of activity	Several methods	All statistics are calculated correctly with the correct units	Y

Further stress testing

Creating an ability with start date later than end date	The constructors	The program displays an error message	Y
Creating activities in several ways where various parameters are negative when they should be positive	The constructors	The program displays an error message	Y

I have decided that, if the user wants to change these parameters to invalid ones using the getter and setter methods, they are free to do so, as they may have a personal reason why they would like it to be that way - say for example if they miss a swimming session, they may make the “completedLaps” negative to signify a sort of “activity debt”. Further, the program could be annoying to a potential user if they try to change the start date of an activity, and it forces them to always change the end date first; With this implementation, a potential user could change them in whatever order they desire. I think this strikes a good balance between giving the users freedom to use the software in whichever way they want, as well as protecting them from accidental data misuse.

Evaluation

The program fulfils all aspects of the specification, and has little room for improper use of the code. The main possible misuse case is passing wrong types or unreasonable numbers/names to the constructor methods, which can always just be edited using the getter and setter method if done by mistake. The program is (in my opinion) readable, maintainable, and clear in what each part of it accomplishes. In my testing - which includes tests for each part of the specification, the program has always behaved as expected, passing all tests.

Conclusion

I am overall satisfied with my final implementation. The methods and attributes are structured in the fashion that VS Code deemed to be optimal (the application has proven to be very useful, generating my getters and setters as well as sorting my code in standard order and providing development advice). Given more time, it would have been reasonable to add more error handling - perhaps displaying error messages to the user for wrong types that are more human-friendly than the in-built java ones, but this is not important as the implementation given here is not user facing, but rather meant to be part of a larger software system, and java stops the user from doing unintended things anyway by not completing any action that results in an error being thrown.