SOFTWARE ENGINEERING COURSEWORK 2

Blanca Fernandez Salamanca s1611086

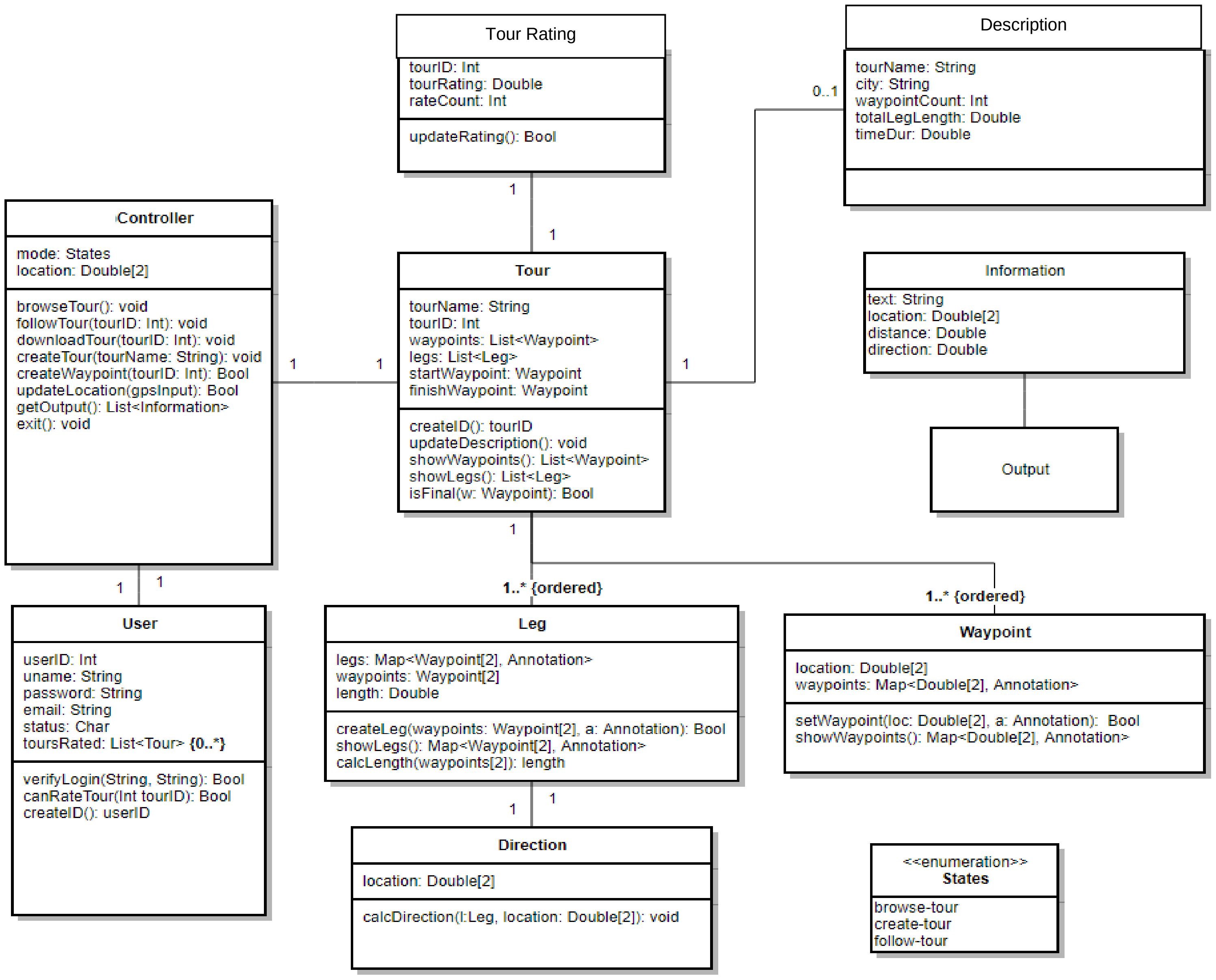
Evangelos Dimitriou s1657192

2.1 INTRODUCTION

TourWay provides a tour-giving service for its users by having previously-established authors (chosen by the app administrator) design tourist tours around a city, that the users can choose from and download. Once a tour is downloaded and started, the app will show the user the route they must take to their next waypoint with map navigation. For details on the constraints placed on each of the apps modes, Public, Author, Admin, and more, refer to the Stakeholder and System State description.

In this document you will see the utility classes and their accessibility, through the use of UML class and sequence diagrams.

2.2 STATIC MODEL



High Class description of diagram:

Controller: The controller class includes the operations necessary for navigating the app such as browseTour() and createTour(), and demonstrates how the actors interact with the system. We decided to use a Controller class instead of separating the user class into its three types; Public, Author and Admin. By creating separate states we achieve the same result of separating permissions and also increase cohesion. This class is also important because it is the one to access output and to update one's location, which are essential aspects of the app and class diagram.

Tour Rating: This class keeps track of user's ratings of a certain tour by adding whatever score is given to a total amount and increasing the count by one, giving the average as the final tour rating.

Waypoint: The Waypoint class, similarly to the Leg class, appoints certain necessary variables to a part of a tour, as well as providing editing methods. It is associated with the Tour class with an unspecified multiplicity, allowing for a tour to have any length, instead of being part of an Element class as previously discussed. An important difference between these two ideas, having an Element class or not, is found in the simplicity of the multiplicity between these classes, as having this class would mean there is a strict one-to-one relation between the Leg/Waypoint and the Element class. It may seem as if this would increase coupling, but its corresponding increase in cohesion was preferred.

Leg: The legs of the tours explored in this app are instances of this class, which is a particularly important regarding the Tour class. It is simple as it has only one operation, the create option, which decreases coupling, and is deliberately separated from the Waypoint class so as to increase cohesion through the sharing of details appointed and needed. The idea to create an Element class that would join both these was considered, however, this would mean a decreased cohesion even though it would also decrease coupling, but for simplicity purposes, reduced coupling was prioritised.

Description: This class contains all relevant information about a tour that can help the users decide when Browsing, including things such as name, city, and total length. This class is particularly helpful for the public user mode, and its instances are created by the Authors at the same time as their corresponding tours. It could be considered as an attribute of the class Tour due to its lack of methods, which we decided to separate from the main Tour class so as to increase cohesion but instead found that the coupling was increased as there are common variables between the classes. However, it is an important part of the app's user-friendliness.

Direction: This class exists to increase the user's experience with the app, as it will keep track of the direction the user must go in. This class could've been held within the Leg class, but to increase cohesion we separated into a different class. This will make the gps implementation with the code slightly easier to write and debug.

Behaviour Descriptions:

Browse Tours:

```
theController --- browseTour() --- aTour

LOOP [while browsing tours]

      aTour --- getDescription(tourID:Int) --- aDescription

      aTour< ------------ aDescription

      theController< -----aTour

      theController --- getOutput()--- aInformation

      theController< -----aInformation

END LOOP

theController --- downloadTour(tourID:Int) --- aTour

theController< ----- aTour
```

Notes: Because of the while loop, you may see the description for any number of tours until you are done browsing, as some users may not be satisfied with the first tour and may want to keep browsing tours, until you decide to download one. If the loop were not there users would only be able to see a single description.

We use tourIDs for reference.

The end of this behaviour description assumes the user has chosen a tour to download, which is then added to his personalised list MyTours.

Importantly, the control flow for this use case ends in theController, meaning the user can keep using the app.

Create a Tour:

```
TheController --- createTour(tourName: String) --- aTour
LOOP [while in CreateTour mode]
    aTour --- updateDescription() --- theController
    theController --- updateLocation(gpsInput) --- theController
    theController --- createWaypoint(tourID)--- aTour
    aWaypoint< ----- aTour
    aWaypoint --- setWaypoint(location: Double[2], description:
    annotation) --- aTour
    LOOP [if updated location is different from previous waypoint
AND there is more than one waypoint in Waypoint list]
        aLeg< ----- aTour
        aLeg --- createLeg(waypoints: Waypoint[2],
    description: annotation) --- aTour
        theController< ----- aTour
    ELSE
        TheController< ---- aTour
END LOOP
```

Notes: This behaviour description is subject to several pre-conditions. It starts by assuming the User is an Author or Admin, meaning they can create tours (public users cannot) which would require a login step to check validity of their ID (Int type).

The loops are in place so as to keep the user in the right mode, the while loop, and to avoid the app generating a leg with the first waypoint, as a leg is only added if the waypoint being marked is not the first (checks this by seeing if there are any elements in the waypoint array created as waypoints are added) and if the guide has moved and wants to create the next waypoint. This is because if he hasn't moved then you don't want a new waypoint, meanwhile you must also check it is not the first.

Something else to note in the beginning of this description is that although the method to create a description is in fact updateDescription, meaning to alter rather than create, it also works as a creator as when a tour is created an automatic empty description is made.

Creating a waypoint requires the tourID to access the chosen tour, where the option to set a waypoint is then available. Once a waypoint is created the system will ask for input, for the author to add a description, which will be kept along with its location.