EdWelcome

Implementation - A tour app for the University of Edinburgh

INF2C Software Engineering (inf2c-se) — Coursework 3

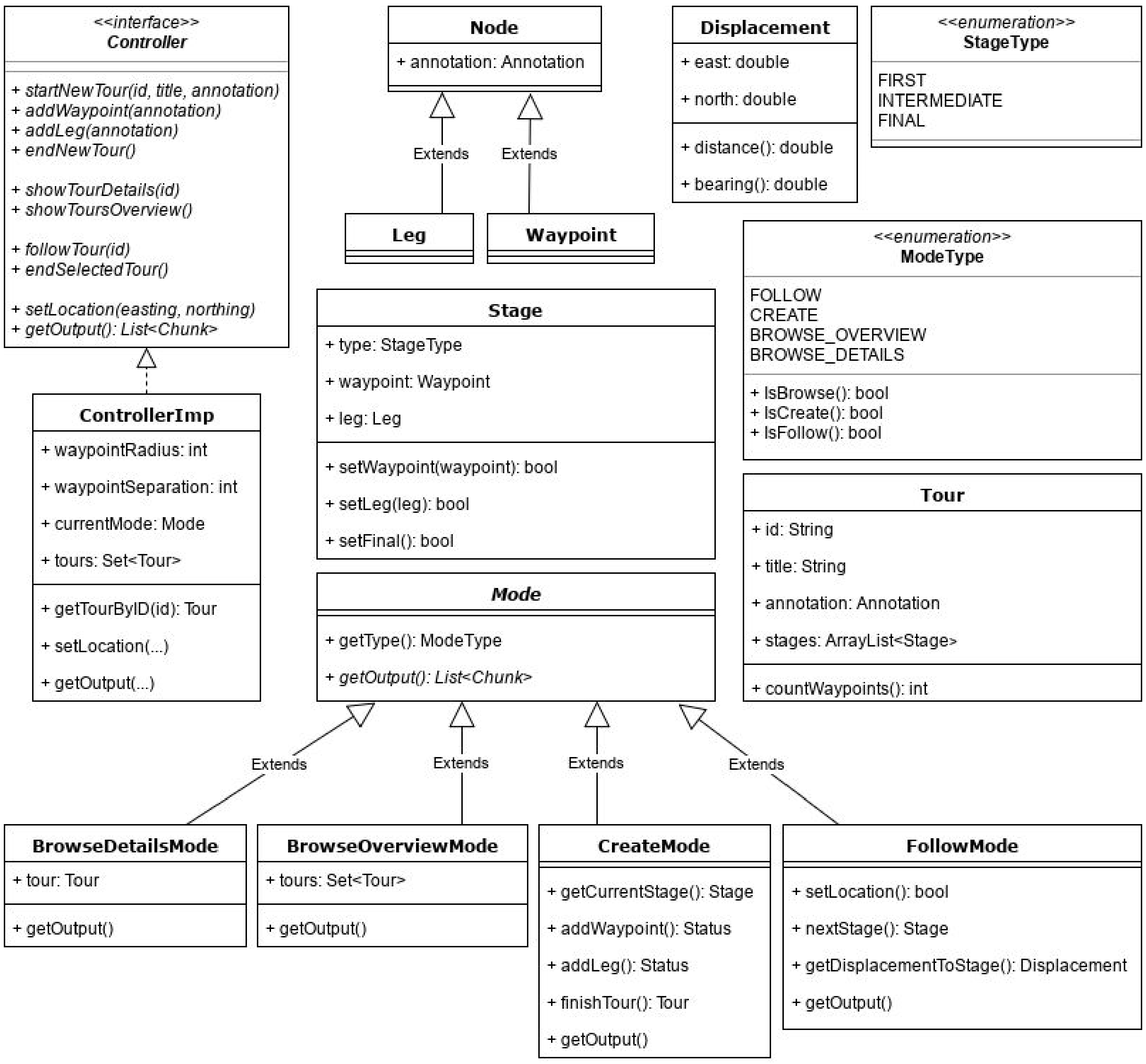
Team:
- Elena Lapinskaite (s1605619@inf.ed.ac.uk)
- Qais Patankar (s1620208@inf.ed.ac.uk)

The application

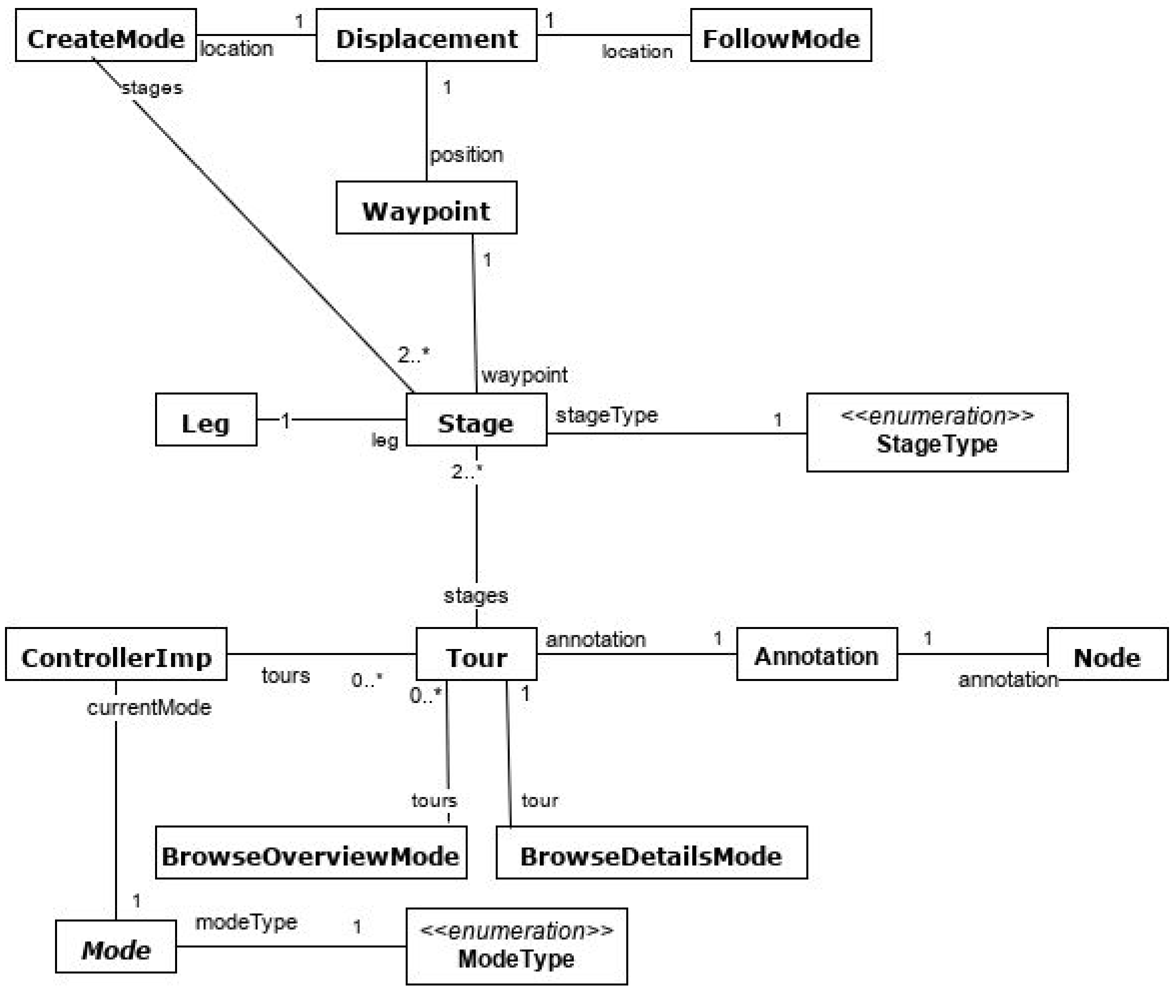
EdWelcome is a GPS-based mobile tour app for prospective students of The University of Edinburgh, who want to get to know the university campuses.

UML Class Diagram

Overview



Associations



High-level design description

Node, Leg, Waypoint

These classes stem from our design created in coursework 2. We noticed a design flaw in our previous piece of coursework, where Legs had locations even though they were supposed to present a user being between two Waypoints.

We instead made Node simply contain an Annotation (and toString definition), and chose to allow Leg/Waypoint to be fairly self contained.

In our final production, Leg/Waypoint no longer had code tracking the previous or next node, and were completely self contained.

Mode, ModeType

An abstract class representing the general requirements of a Mode, and an implementation of *getType*, requiring each Mode to have a corresponding type. It also mandates that all Modes must defined a getOutput method, described in the next section.

ModeType is defined a subclass, and exposes four enums: BrowseMode(Overview, Details), Follow, and Create.

To simplify the use of ModeType, we created methods (IsBrowse, IsCreate, IsFollow) that would check whether the current mode is for a specific use case - this proved especially useful for dealing with the sub-modes presented by BrowseOverviewMode and BrowseDetailsMode.

ControllerImp

ControllerImp keeps the very basic cross-mode state in this class, and offloads all effort away to individual modes when it comes to sending output messages.

Since output messages are very mode-centric, we could simply “pass the message on” to the current mode (stored as *Mode* in *currentMode*) and let them deal with chunking their state out.

Since tours need to be created and browsed, we store all tours this class. Not all classes needed to support *setLocation*, and so we dealt with this particular method on a mode-by-mode basis, rather than mandating that all *Mode*s define a *setLocation* method.

Stage, StageType

We adopted the same rules for StageType here as we did for ModeType. The only difference here is that we use enums in their very raw form - we don’t define any special methods to be used on enums.

To quickly make use of enums in *Stage* we imported StageType as a static class, allowing us to refer to the constants simply as **INTERMEDIATE** (etc), rather than **Stage.StageType.INTERMEDIATE**.

To support the creation of tours, we made it possible to update a Stage after instantiation. In fact, there is no way to “finish a Stage” without instantiating it, and calling the required method functions to add nodes.

The ordering of waypoints and legs are enforced by *Stage*, and the instantiator has rule over what type of Stage it is: the FIRST stage, an INTERMEDIATE stage, or the FINAL stage.

Since it’s not possible to determine whether a Stage is FINAL before ending tour creation, we also make it possible to convert an INTERMEDIATE Stage into a FINAL Stage. This also enforces Stage finality rules.

BrowseDetailsMode

… is an incredibly simple class. It stores the Tour passed to it during instantiation, and plucks the correct values to be returned when *getOutput* is called.

BrowseOverviewMode

… is on a similar level. It stores a list of tours. If no set was provided, tours is an empty set. Otherwise it copies the set, and plucks the correct values when *getOutput* is called.

CreateMode

This class has most of the logic in these three functions: addWaypoint, addLeg, and finishTour. It contains an ArrayList of stages, and defines getCurrentStage, which is simply the last Stage currently defined.

At all times during CreateMode, except during finishTour, the current Stage will be partially completed, and so the above node addition methods are responsible for setting up the stage for the next node to be added.

The primary logic here is to deal with waypoint proximity and to deal with errors returned (in the form of booleans) by Stage methods. This makes this method incredibly logic light.

FollowMode

This class is similar to the above CreateMode class as we only need to advance the stage number and get data from a Stage. This makes the class incredibly logic light, offloading everything except proximity checking to individual Stages.

A getDisplacementToStage method has been defined alongside currentStage() and nextStage(), facilitating the confirmation of whether or not the stage should be advanced when a user is close enough to a waypoint.

Implementation decisions

Mode

We decided to create separate classes (*BrowseOverviewMode, BrowseDetailsMode, FollowMode, and CreateMode*) to encapsulate each of the major use cases as defined the specification, and we knew that we would create a *Mode* class to represent the general case.

However, we were not sure about whether *Mode* should be an interface or an abstract class.

Since we wanted to enforce the existence of some methods (*getOutput*), and once we realised that the implementation for *getType* would be consistent across inherited classes, we decided that defining *Mode* as an abstract class would be better suiting, as we could provide a method body for *getType*.

Another reason we did not choose *interface* is because we would not be making use of the core *Mode* class in more than one location.

stage.getNodes

We initially had a method called *getNodes* that would return a list of nodes involved in that particular stage (i.e, just a list containing the leg and the waypoint, in that order, omitting the correct nodes dependent on the stage).

This might have been useful if we were required to return node data on a tour level, but since this was not expected nor needed, we removed this method.

StageList vs ArrayList<Stage>

We were not sure whether to model a list of *Stage*s as a simple ArrayList, or as a separate class *StageList*.

When glossing out our original plan for this class, we had intended to *extend* from *ArrayList<Stage>* and add the following features:

- Enforce waypoint/leg ordering in the data store (here)
- Automate Stage creation as well as Leg creation
- Make it possible to follow through a StageList during the lifetime of a tour
- Maintain the idea of the “current/next state”, “current/next leg/waypoint”, and “waypoint finality”.

In the end, we decided against keeping this class as state is sufficiently kept in the corresponding mode classes.

Annotatable

We wanted to create an Annotatable interface as per our Design plan in coursework 2, but in the end decided against this as annotations were not being generically read by any sort of user interface.

Storing Tours as a Set

Our list of *Tour*s is stored in various places as a Set (backed by HashSet).

To enforce ID uniqueness here we defined a *hashCode* method in Tour as HashSet looks for uniqueness by comparing values returned by *hashCode.*

We could have use a HashMap but felt this would be a much more elegant solution, although we did have to implement our own getter: *getTourByID*.

ModeType and StageType

ModeType were initially “external” enums, but we decided that since they were very centric to a particular class they should be made a “subclass”. And so we did that.

We contemplated whether we should make the enums private, but realised that it would be too inconvenient and counterproductive to block simple constants from being read and used.