Curtin University – Department of Computing

**Assignment Cover Sheet /**

**Declaration of Originality**

Complete this form if/as directed by your unit coordinator, lecturer or the assignment specification.

|  |  |  |  |
| --- | --- | --- | --- |
| Last name: | Payne | Student ID: | 18925997 |
| Other name(s): | William |  |  |
| Unit name: | Object Orientated Software Engineering | Unit ID: | COMP2003 |
| Lecturer / unit coordinator: | David Cooper | Tutor: | Chitra |
| Date of submission: | 22/05/19 | Which assignment? | (Leave blank if the unit has only one assignment.) |

I declare that:

* The above information is complete and accurate.
* The work I am submitting is *entirely my own*, except where clearly indicated otherwise and correctly referenced.
* I have taken (and will continue to take) all reasonable steps to ensure my work is *not accessible* to any other students who may gain unfair advantage from it.
* I have *not previously submitted* this work for any other unit, whether at Curtin University or elsewhere, or for prior attempts at this unit, except where clearly indicated otherwise.

I understand that:

* Plagiarism and collusion are dishonest, and unfair to all other students.
* Detection of plagiarism and collusion may be done manually or by using tools (such as Turnitin).
* If I plagiarise or collude, I risk failing the unit with a grade of ANN (“Result Annulled due to Academic Misconduct”), which will remain permanently on my academic record. I also risk termination from my course and other penalties.
* Even with correct referencing, my submission will only be marked according to what I have done myself, specifically for this assessment. I cannot re-use the work of others, or my own previously submitted work, in order to fulfil the assessment requirements.
* It is my responsibility to ensure that my submission is complete, correct and not corrupted.

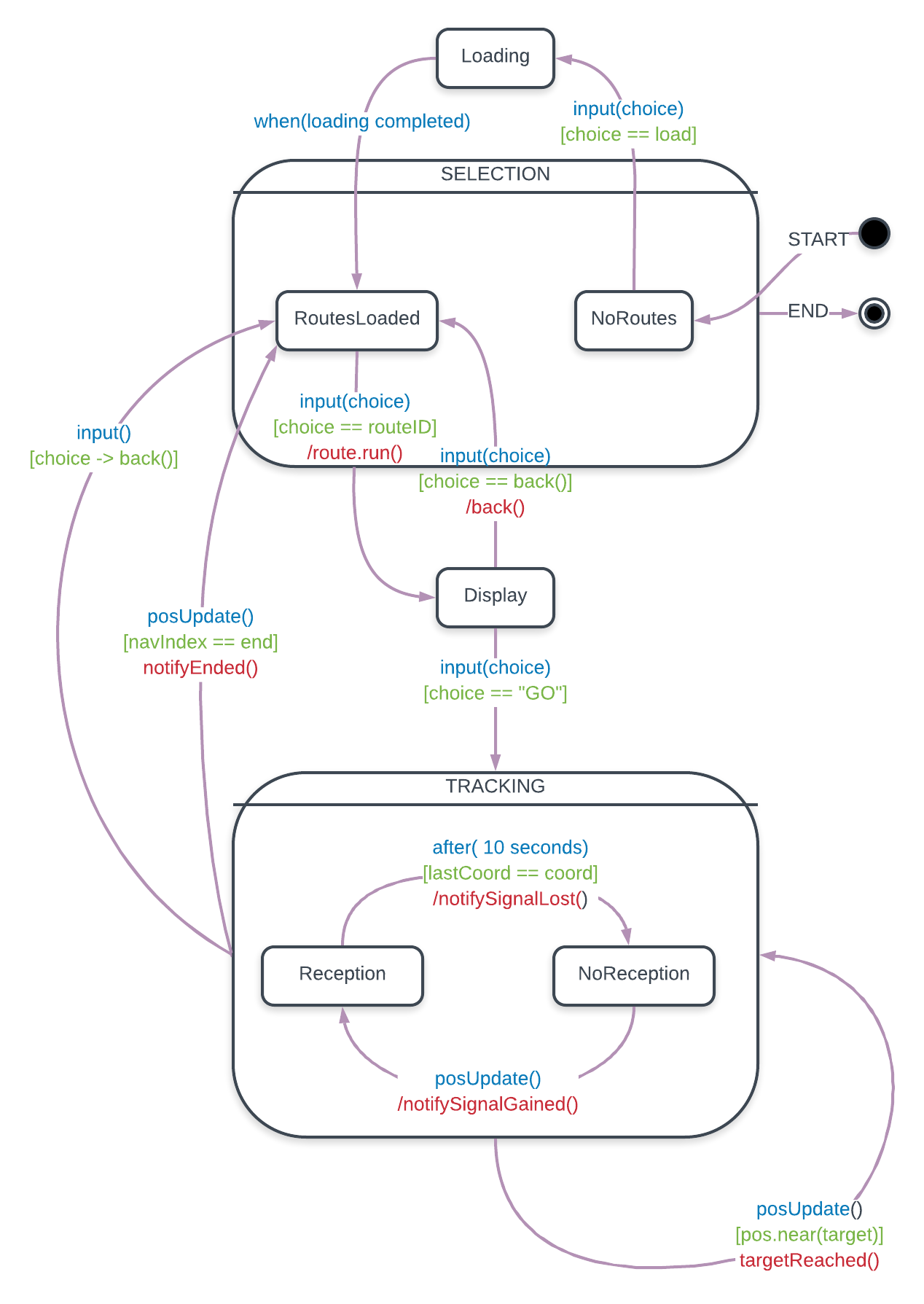
Date of 21/05/19

Signature: William Payne signature:

*(By submitting this form, you indicate that you agree with all the above text.)*

# UML Class diagram:

# Program UML State diagram:

­­­

# Problem A: Route Tracker

The purpose of this task was to design and implement a RouteTracker program that allowed trekkers and climbers to view possible routes and track their progress as a console application that was convertible into a mobile app. As a console application, the program needed to output a table of routes to the user, giving them the option to select a route. The user could then enter a tracking mode, which recorded them going from checkpoint to checkpoint, until the route was complete. The key focus when creating a solution was to ensure it had low coupling, high cohesion, used patterns to solve problems, and was testable.

## Architectural Design Pattern: MODEL-VIEW-CONTROLLER

The RouteTracker program has been developed to provide basic path tracking capabilities on a mobile platform. The program is currently implemented with a terminal UI but is intended to be converted to a mobile GUI. To achieve this, the program is been broken into to three main architectural separation of concerns: the Model, the View and the Controller. The low coupling of the MVC pattern provides high code reusability, accommodating simpler conversion to other platforms. As such, the next developer will only need to build the mobile GUI View which will interface with the pre-existing Model and Controller components.

As the MVC decouples code into separate self-contained parts, development was executed in stages (one for each concern). Developing in stages ensured that each component was self-contained and properly tested before moving on to the next, thus decreasing the overhead of debugging and increasing cohesion of the program.

This program has been developed in Java and as such, packages are used to implement the MVC.

## Other Packages:

## The program is being developed simultaneously and such will eventually need to interface with the classes GeoUtils and GpsLocator (presumably being developed by another person). To ensure the program will successfully interface with these classes, stubs have been created in the MVC to provide test input.

## The Model:

### The Model was created first, as its low coupling makes it self-contained and easiest to test. The Model stores and manages the data structures which are broken down into four smaller packages: route, point, coordinate and trekker.

### Route:

### Routes form a composite and chain-like structure comprising of waypoints, segments and sub-routes. To Model this structure, a modified Composite pattern was implemented that has subclasses forming an alternating linked list of waypoints and route links. The modification has the leaf class SubRoute wrap a single route which can potentially contain sub routes forming the composite object. To put it in other terms, a standard composite has the structure: A is-a B and A has-many B whereas the IRoute structure has: A is-a B, A has-a C and C has-a B. This allows a single route to be reused any number times as a route link without duplicating data. Each member of the Composite pattern implements the interface IRoute, providing the ability to polymorphically calculate the distance of route, check its validity, retrieve a list of points and build the route structure used when tracking. The alternating linked list is enforced by having waypoints and route links aggregate each other rather than the super class IRoute. This reduces polymorphism but provides a structural way to enforce the object structure alternates from point to link.

### Points:

### Points are the checkpoints being visited in a route and represented via a GPS coordinate. WayPoints form a part of the chain structure used to Model a route(as discussed earlier). These points are then converted to NavPoints which combine the point and link information into one object allowing for a linked list of a single data type. Both points are required to store a label and a GPS coordinate, but the aggregation changes from point-to-link to point-to-point. The similarities of these classes are defined in the class TemplatePoint which provides a basis for polymorphism and code reuse. NavPoint and WayPoint can extend the TemplatePoint to add their specific features while preserving the common fields.

### Coordinate:

Coordinates are required to store latitude, longitude and altitude information to accurately check if the user has reached a checkpoint. The algorithm to calculate the distance between these coordinates will be available at a later stage of the app development process, so to provide testability an ICoordinate Strategy pattern has been implemented. The interface provides methods to calculate horizontal and vertical distance between nodes, check if a point lies within a provide range, and retrieve the coordinate data. Cartesian coordinates trivialize the calculations required for this information, making a useful stub while testing the program. Geodesic coordinates will eventually replace the Cartesian coordinates and can be done trivially by implementing the ICoordinate interface. A mostly complete GeodesicCoordinate object has been created but requires the unimplemented method ‘calcMetresDistance()’ from the GeoUtils class. Coordinate factories are injected into the program to remove static dependencies on the creation of CartesianCoordinates. The factories CartesianCoordinateFactory and GeodesicCoordinateFactory inherit from the abstract factory CoordinateFactory, which will minimize the amount of changes needed when GeoUtils is ready for implementation. A single line of code can be altered to swap to a geodesic system.

### Trekker:

## The Trekker object represents previous and target checkpoints, the total distance remaining in a route, and the user’s current position and segment description. A simple container class is used to store this information and mutators are provided to allow the Controller to update the information. To communicate changes to the View (and parts of the Controller) the observer interface ITrekkerObserver can be used. This allows updating GPS coordinates to be communicated directly to the View polymorphically without needing to know anything about its implementation. A new Route object is created when starting any route, but the same observers need to be added. This is simplified via the use of a factory object that stores the observers and injects them into a new Trekker object upon creation.

## The Controller:

The Controller modifies the data in the Model and provides an interface for the View to perform operations on the Model. The Controller is broken up into sub-packages for the program handler, option system and geo utilities. The program handler package is broken up further, separating the ProgramHandler class (and its observers) from the its states. Some states however, have sub-states so another package is used to separate those as well.

### Program Handler:

The program handler provides methods for receiving input from the View and GPS, and dynamically modifies how said inputs are utilized based on its current state. The method ‘input()’ is used for all information passed in by the View, allowing the Controller to be naive of the Views implementation. GPS data is passed in through the ‘posUpdate()’ method and will either be discarded or used depending on if the program is in the tracking mode. Through the use of observers, the Controller can communicate to the View as an event source which again allows information to passed with knowing as little as possible.

### Program State:

The states of the program include route selection, displaying of routes, loading routes and the route tracking mode. A State pattern is used in the ProgramHandler to allow it to dynamically change its behaviour by delegating its responsibilities the current ProgramState object. The ProgramStates include SelectionState, LoadingState, DisplayState and TrackingState which provide different implementations for the ‘input()’ and ‘posUpdate()’ methods. The TrackingState is a super state that contains two states, ReceivingState and NoSignalState. This allows the user to manually enter checkpoints when GPS signal is lost and alerts the user when signal is gained. Using the State pattern, the program can dynamically change its behaviour depending on the current object state. This allows the program to become more extensible as when additional states are needed, a new State object can be created, rather than adding to the complexity of a pre-existing control structure, which may introduce bugs.

### Menu:

Menus are constructed using a Composite pattern of Option objects and SubMenu objects that both inherit the IMenuItem class. SubMenu classes hold a list of IMenuItem objects which can create a hierarchical structure of menus. Menus are not needed to this extent in the tracking program, but the foundation will make it simpler to extend and add more features later. The Option interface is used as a Strategy pattern to allow any kind of operation to be executed from the menu structure This will allow the application to add additional features to the menu by creating the new classes that inherit from Option rather than modifying pre-existing code which may introduce bugs.

## The View:

## The current View is a terminal based UI that takes user input from the command line and passes it to the Controller as integers. Each state of the program is displayed using a Canvas object that stores sets of Component objects to be displayed. The abstract Component class designs a Composite pattern of TextComponent, WindowComponent and Canvas object, making it simple to add and remove elements being displayed to the user and provides a means of creating intricate and interactive user displays. By programming to the Component interface it will allow for further extensibility to the UI design. To display information as it updates, the components observer the Controller and Model for events to occur such as when the GPS updates or the program state is switched. Using the Observer pattern will allow the program to be more flexible as the Inversion of Control (IoC) moves from just the GPS updates to an entire event driven interface.

# Alternative design choices:

## Simplifying routes using the Decorator Pattern:

## Before deciding on the current implementation, a few designs were considered on how to organize the Route class structure to make it as extendable as possible. All these designs involved Waypoints and RouteLinks aggregating each other in different ways using the IRoute as a common interface for calculating distance and building a traversable route. However, the final design was too complicated and did not utilize polymorphism to its full extent. The following UML diagram shows a newly proposed Decorator pattern that would greatly simplify the class structure by removing the need for multiple kinds of points and route links and thus making the structure more polymorphic. In this design, routes are formed by decorating an EndPoint with WayPoints and SubRoutes forming a linked list of points with the starting point being the outermost object. The SubRoute decoration allows a route to traverse another chain of routes before continuing with the main chain. This design allows the sub-route to start at any WayPoint up until the EndPoint, making it a more adaptable design. To identify a unique route, the outer WayPoint will store the route name in its name field, and each subsequent WayPoint will store an increasing integer. A disadvantage of this structure is that it requires the developer to ensure the outermost object is a WayPoint and that SubRoutes don’t aggregate another SubRoute or EndPoint as the start field as it adds redundancy to the structure. In general, changing to this class structure would remove six unneeded classes from the Model and greatly reduce code complexity making it the optimal solution.

## Controller managing the current View:

As it stands, the current Controller has no bearing over what information is displayed to the user. Instead, the View observes events in the controller and the Model for queues to switch displays and update UI components. Allowing the Controller to directly decide which View should be displayed will help give the program a more logical flow, making it easier to manage as a greater number of Views are added to the program. To achieve this the ProgramState class could aggregate a class from the View Strategy pattern, allowing any View object to be displayed when running ‘setup()’ on the State object. Using a Strategy pattern provides a basis for extensibility which will allow the controller to add and adapt Views for each state as the program is maintained. This method of changing Views is a more logical flow of design, making it easier to extend the View for each state, but it comes at the cost of needing to know more information about the View and reduces the flexibility of how and when the View represents information. Ultimately this is why the Controller was implement as an event source for the View, as it still provides extensibility by programming to an interface, has much lower coupling (as the Controller isn't required to know any information about the View), and lets the View decide on how information should be displayed to the user.