RoutePlanner UWA CITS5501

Callan Gray - 21341958

May 2019

RoutePlanner is an application for calculating optimal routes through a network of travel stops in a navigable network. The follow report outlines a testing strategy for verifying the state of the system during it's development.

1 JUnit Tests

1.1 A - Preconditions and Postconditions

Pre-conditions and post-conditions for class methods and functions are necessary in order for a developer to understand both how to use a class as well how to change a class without changing the existing behaviour and the authors original intent of of what a class is limited to.

1.1.1 Pre-Conditions

Statements that must be true before a method or function is executed in order for the correct execution of the code to occur. Examples of this include:

TravelStop Constructor: The street name and suburb must exist in a related on-disk file.

RoutePlanner Constructor: The caller must provide either a departure time (hour and minute), an arrival time (hour and minute) or neither to the constructor.

1.1.2 Post-Conditions

Statements that are true after a method or function has executed on the condition that the pre-conditions are met and the implementation is correct. Examples of this include:

TravelStop Accessors: Each of the accessors on a *TravelStop* instance will leave the object in the same state as at the start of the method. (Since the *TravelStop* instances do not have any mutators or other methods, it should be understood that it is an immutable class)

RoutePlanner testGetDirections: Will return a list of *TravelStop* representing a path of how to travel between the start point (at *getStartLatitude()* and *getStartLongitude()*) and the destination point (at *getDestinationLatitude()* and *getDestinationLongitude()*). The *description* parameter will contain a textual description of the route to take.

1.2 B - Identifying Test Cases

Test cases can be identified by analyzing a proposed system implementation and breaking down the types of testing into state testing and behaviour testing.

1.2.1 State Testing

Test cases for verifying system state can be created by identifying variables that are not invariant, such as numbers and enumerations that should only ever be between a predetermined range of values, and the state tests try various approaches to put these variables into an invalid state whist checking that the system is capable of identifying when it is in an invalid state. This is typically done by asserting that suitable exceptions and errors are being raised and returned, as well as a visual inspection that that the level of encapsulation is appropriate.

Further state testing can be performed by identifying which parts of a module should and should not mutate when particular operations are performed. Tests cases for verifying state under these circumstances can be written by performing complex operations and verifying that the expected state changes have occurred and that no unexpected state changes have occurred.

This type of testing is limited by the fact that in many cases only publicly visible state can be tested. A deeper level state testing can however be achieved by alternative techniques such as using package level encapsulation of member variables, checking for instance mutations by implementing a deep level instance comparison method and using pure functions and const method language features.

In most cases unit tests that perform state testing form a large part of a system's integration testing as they are easy to keep small and concise.

Examples of state testing performed in *RoutePlanner* includes tests that check the behaviour of accessors such as *getLeaveMinute* and *getArriveMinute* where the state is checked by comparing values passed in via the constructor and the accessor interface. It is then verified again in *teardown* to confirm that the accessors have not mutated any since construction and verify it is a immutable class.

1.2.2 Behaviour Testing

A simple set of test cases for verifying behaviour can be produced by testing the inputs and outputs of functions and methods and verifying that they behave correctly both when pre-conditions are met and not met.

Further test cases for verifying system behaviour can be formed by creating a scenario, performing a sequence of calls where multiple class instances interact, then verifying the outcome. This type of testing typically requires more knowledge about the system implementation and how to use it compared to other test cases. These tests tend to be large and are much easier to implement once a system works and are created to assist in preventing working behaviour from breaking and less useful for find hidden bugs. Behaviour tests run can be written as larger integration tests for a complex module in a system, and ultimately there will be a few that form a system test that covers a complete user use case.

Examples of behaviour testing performed in *RoutePlanner* includes the constructor and *testGetDirections* which tests a method that performs a complex calculation. This method tests the behaviour of different types of input arrays and verifies that an appropriate exception is thrown, followed by verifying the output travel stops start and end close to the desired values stored in the *routePlanner* state followed by additional state testing to ensure that the method does not unnessarily mutate the *routePlanner* instance.

1.2.3 Process

The process for identifying suitable test cases for classes is as follows:

- Write a test case for the constructor and perform behaviour testing on the constructor inputs
- Write a test setup method that creates correctly constructing class instances and any utility instances as member variables
- Write a test case per method and perform state testing with the constructor values against any accessor and mutator values
- Add state testing for testing any methods that do not mutate the instance
- Add behaviour testing for any methods with non trivial behaviour such as in the constructor, mutators and calculation methods.

1.3 C - Further Test Cases

1.3.1 testRoutePlanner

The *RoutePlanner* has a rule that either the leave time, arrive time or neither are passed into the constructor. At construction the instance will do a calculation to determine what the other time is, or in the case of neither passed in uses the current time for leave time, and throws an exception otherwise. In this case 16 different possibilities can occur and in this case can be tested to exhaustion for an exception.

1.3.2 testTravelStop

The *TravelStop* class has a rule that a created travel stop will be checked with data stored on a disk for validity and throw an exception otherwise. This test case can easily pick a value that is highly unlikely to exist on a disk, however the connection between the two from the constructor signature is not clear. It's likely that the validation occurs through some form of global state, such as with a singleton pattern or reading from a hard coded filename. This is manageable if the test author has access to the file data, however this means the test cases will be coupled to a production file and can break when the file changes and the code remains unchanged. A better way to handle this is to pass the file location via an interface (e.g. Service or File Pointer) and the test case implements a fake service.

1.3.3 testGetLeaveHour

The *RoutePlanner* class contains accessors for a leave time and an arrive time whereby one of the two is predicted. In this case it is important that a test case is created for at least 2 of the different ways that the *RoutePlanner* can be constructed.

1.4 D - Implementation

Please refer to the attach files located under "routeplanner/src/" and "routeplanner/test/"

1.5 E - Identifying Extra Tests

Further test cases will be necessary to test the system throughout the development lifecycle. In some cases it is viable to produce test cases once problem start appearing due to complicated algorithms, however large amounts of work may need to be performed to mock enough behaviour to find and test problematic classes. A better approach to this is to account and prepare time to write test code for new classes using the approach outlined in 1.2.3.

One of the complicated cases to handle with new testing is when the existing code that is tested needs to significantly change, e.g. the date must also be included with the planner in order to debug reasons why the arrive time might be behind the leave time. In this case the an extra method and a test case may be all that is required, however the test class will need to be revised in a way such as updating all cases that use the leave time or arrive time. Depending on which part of the system needs changing, a developer will need to:

- Verify that all related unit tests are passing
- Add a new test case that directly addresses the change needed
- Perform any implementation alterations and observe how many tests start to fail
- Read through the failing tests and develop an understanding of why they are failing
- Repair a few tests and observe which tests are intermittently failing and why
- Extend any intermittently failing tests and edge case tests with better reported messages if possible
- Perform any remaining updates and bug fixes to get all tests passing

An important series of remaining tests to create for this project are system tests which test the complete end-to-end functionality of a product that uses the code. This would be something like a suit that loads a backed-up production file of real world locations and creates several short routes where the shortest route is trivial and test it for performance, and then some longer routes that may or may not be trivial to determine shortest routes, but will be exposed to some potentially unforeseen complications in the map data and algorithm complexity.

2 Alloy Model

Refer to the model file located in "routeplanner/models/"