# Requirements

1. There are 6 named cities and 2 unnamed cities. The Inner cities are Mayfair, Akina, Stortford Lodge and Mahora. The outer cities are Napier, Flaxmere and Outside City (The 2 unnamed cities).
2. The driver can only exit out of the following named roads: Karamu Road, Havelock Road, Railway road and Omahu Road.
3. If a driver exits out of Karamu Road, then the program will display “The Driver has gone to Napier”
4. If a driver exits out of Omahu Road, then the program will display “The Driver has gone to Flaxmere”
5. If a driver exits out of Havelock Road or Railway Road, then the program will display “The Driver has gone to an Outside City”
6. There are 4 paths. The paths can consist of multiple streets and MUST link A city with an exit and then another City. The paths do not need to link streets between other paths.
7. A single city will only have 2 paths, and only link to 2 of the other 3 cities.
8. The simulation will use a random number generator for decision making. The chances for these decisions are provided below:
   1. The decision of what path to take is made by the random number generator, each path has a 50 percent chance to be chosen.
   2. Whether the driver uses an exit while driving to a city is decided by the random number generator, using an exit has a 20 percent chance. Continuing to the city has a 80 percent chance.
   3. The driver can start at any of the 4 inner Cities. This decision is made by the random number generator, each city has a 25 percent chance of being chosen.
9. When a driver uses an exit to reach an outside city (Including Napier and Flaxmere), that simulation will end. The end of the simulation will cause extra dialogue to be printed. (See requirements: 3 , 4 , 5 , 12)
10. The program will run through the simulation 5 times. Each iteration will have unique driver name, this will be “Driver” followed by the number of the simulation that the program is currently on. After the simulation is completed, it will print all information related to that simulation followed by “-----”
11. When the program loads it will prompt the user for a integer to be used for the random number generator’s seed.
12. After the driver simulation has ended (See requirement 9), the following additional dialogue will be displayed: (Related Requirements : 3, 4, 5)
    1. The amount of times that the driver has visited Akina. This display will be in this format: "Driver <Number> met with John Jamieson <Akina Count> time(s)."
    2. Extra text based on the number of times Akina was visited. The dialogue is as following:
       1. “This driver needed lots of help” - If the driver visited Akina 3 or more times
       2. “That passenger missed out!” - If the Driver never visited Akina
       3. If the driver visits Akina 1 or 2 times then this dialogue will be blank.
    3. In relation to requirements: 12.1 and 12.2: If a driver starts in Akina, it counts as visiting.
13. Every time the driver moves between inner cities (Requirement 1) it will display: “Driver <Number> heading from <Inner City name> to <Inner City name> via <a Path>”
14. In relation to Requirement 10. The information related to the simulation consists of: Requirements : 3, 4, 5, 9, 12, 13

# 

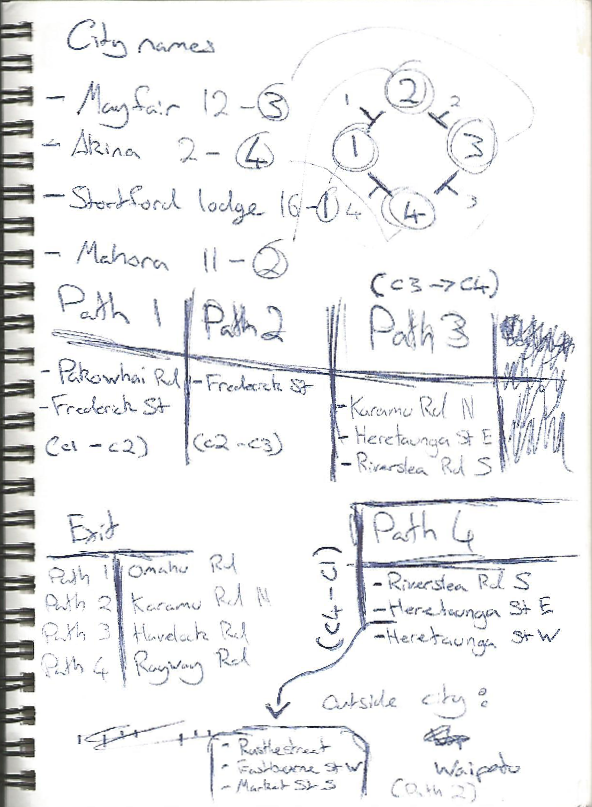
# Activity Log

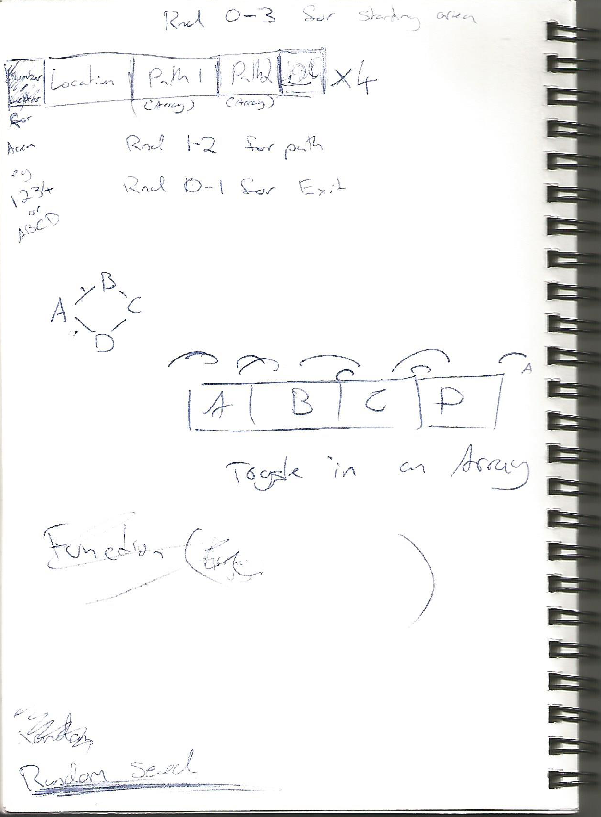
## Notes

During the brainstorming process, it was decided to use a array structure where, Arrays are in Arrays that are held by a single/master array. During the development/language introduction stages it was discovered that the Go language handles arrays in a way that made this structure difficult/odd to implement. The structure was then changed to 2-Dimensional arrays held in a single master array. This structure was then once again changes to a new stuct type (As stated in the Test plan revision history).

## Early Planning

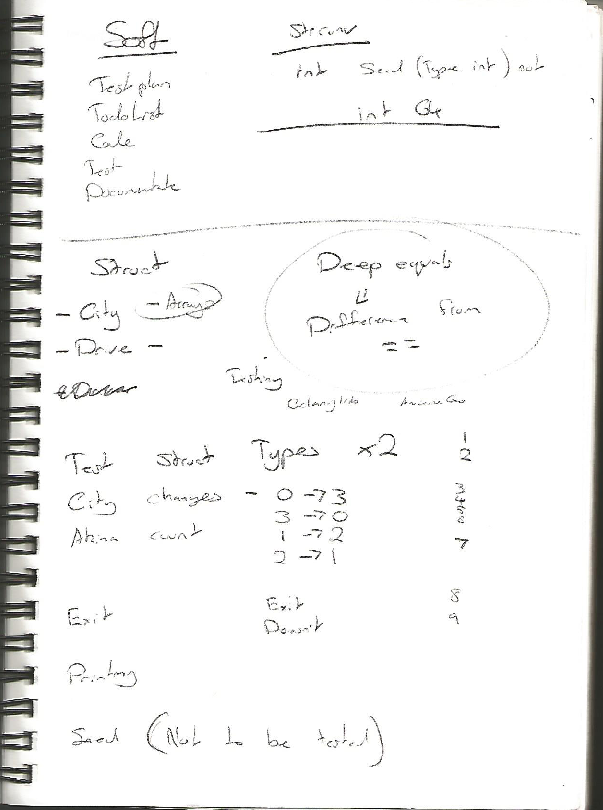
These notes were made after receiving the project brief and helped the creation of the interpreted requirements document. (ITPR6.590\_Interp\_Requirements\_Ver\_1.0). this was when the idea for the program was still Arrays in Arrays in an Array. When this was the plan, there were functions for reversing the array. (The idea was to reverse the array holding the paths)

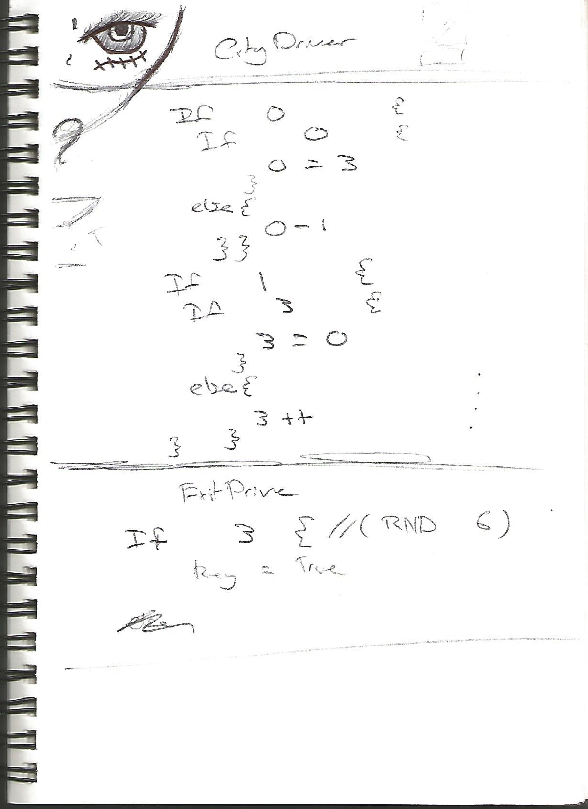


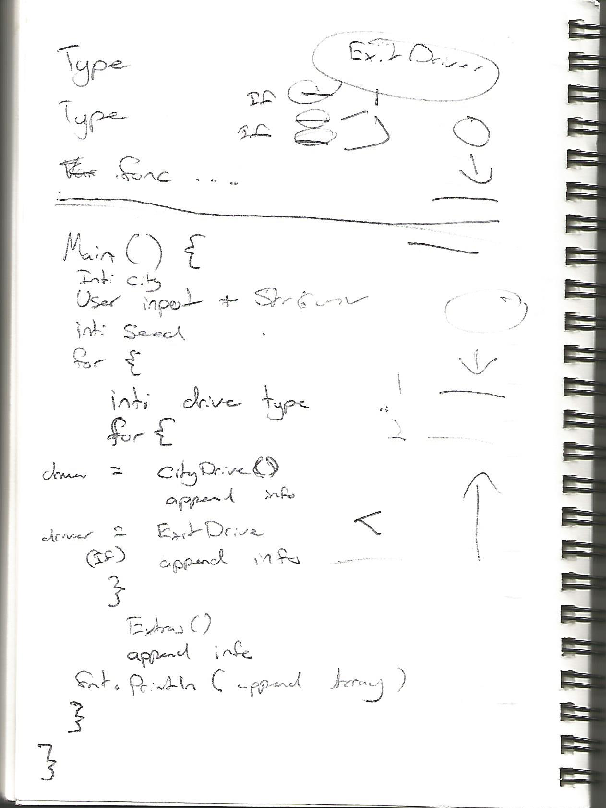


## Extra Planning

These notes were made during the development process.







# Test Plan

## Identifier

ITPR6.590\_Unit\_Plan\_Ver\_1.2

## Author

Cameron Anderson

https://github.com/WorxWrk/ITPR6.590

## Revision History

|  |  |  |
| --- | --- | --- |
| Revision Version | Description of main changes from the last revision | Change made by |
| 1.0 | Test Plan Created | Cameron Anderson |
| 1.1 | During development, changes were made to how the “driver” and “cities” work. Updated the test plan to replace the sections involving the old structures | Cameron Anderson |
| 1.2 | Small change. Changed the name of the (now) akinaCount function to match the change in the code. | Cameron Anderson |

## References

ITPR6.590\_Schedule.tdl

ITPR6.590\_Interp\_Requirements\_Ver\_1.0

ITPR6.590\_Testing\_Report\_No\_1

## Introduction

This Unit plan was created to help the development of the ITPR6.590 program. The plan supports the development by analyzing the requirements, and creating theoretical functions that meet them. The plan also details tests for these function, including the criteria for the test to pass. The test plan also lists potential risks that could occur during development and has notes to help the creation tests and test suite.

## Language Rationale

The development team’s experience with programming is extremely varied. One member has very limited experience programming where the other has experience programming in a few other languages. Due to lack of experience from a member, it was decided to avoid Object-Oriented languages. From the remaining languages “Go” was decided on to allow the development team to gain experience in a language new to them. The client also recommended this language and heavily praised the simplicity of it’s built in test framework.

## Go’s Testing framework, relies on a built in package that allows the test suite to be simply executed from windows command line. Due to the way tests fail and errors are logged, the tests can become very detailed. The framework can also use 3rd party packages to customize simplify the tests.

## Test Items

The program is being created in a single file named “Main.go” this will be the only package that requires testing. Due to the framework being used, the file that holds the tests will be called “Main\_test.go”

## Software Risk Issues

For the ITPR.6.590 program there are the following risks and possible cause:

### Software is not completed by the due date

Risk level: Medium

Possible Causes:

The developers are creating the program in a language that is new to them.

The allotted time for the programs creation is more than sufficient, but the developers also have other projects to complete during this time frame.

### Software does not meet the original specifications provided by the client

Risk level: Medium

Possible Causes:

The Developers and the client have scheduled meetings twice a week, which can be used to clear any misunderstandings.

If the developers do not notice any errors (inconsistent, ambiguous) in the client provided requirements.

Additional requirements are conveyed verbally from the client.

### Development slowed due to issues with the management platform

Risk level: Low

Possible Causes:

Some of the developers have never used “Git” before.

The chosen system “GitHub” is fully online which will cause the development to halt in cases of internet outages.

## Features to be tested

### Type City and Type Driver

A simple test for comparing the state of the new data types. The types themself have no have no functionality or features.

Related Requirements for the City type: 1 , 2 , 6 , 7.

Related Requirements for the Driver type: 10 , 12.1 , 12.2.

### Function akinaCount()

Related Requirements : 12.1 , 12.2

Related Variables : akinaCount , currentCityName

The function increases the akinaCount variable based on the currentCityName.

### Function driveExit()

Related Requirements : 8.2 , 9 ,

Related Variables : newNumber , key , nextCityName

This function changes the values of the key and nextCityName variables, based on the value of newNumber.

### Function driveCity()

Related Requirements : 8.1 , 2 ,

Related Variables : currentCityName , newNumber , pathName , exitName , currentCityNumber , currentCity

path1 , exit1 , path2 , exit2 , cityName , allCities

This function changes the currentCityName, currentCity and nextCityName variables. Based on the value of newNumber it will change the values of pathName and exitName variables. additionaly it will change value of currentCityNumber based on the value of the currentCityNumber.

### Function driveCityPrint()

Related Requirements : 2 , 3 , 4 , 5 , 10 , 14

Related Variables : driverNumber , currentCityName , nextCityName , pathName , totalOutput, exitname

### Function extras()

Related Requirements : 12 , 14

Related Variables : akinaCount , totalOutput , driverNumber

## Features not to be tested

The seed given to the Random Number Generator (Requirement: 11 and 8). This is a function provided by a package, which should contain its own tests.

The input from the user (Requirement: 11). Is easier to test manually.

## Item Pass/Fail Criteria

### TestCityType()

This test will check that 2 variables of type City are equal

### TestDriverType()

This test will check that 2 variables of type Driver are equal

### TestAkinaCountPlus()

This test will check that the variable, driver.akinaCount, will increase by 1 after calling the akinaCount() function.

### TestAkinaCountStay()

This test will check to see if the variable, driver.akinaCount, will stay the same after calling the akinaCount() function.

### TestDriverExits()

This test will check that the variables, key and nextCityName, have changed

### TestDriverExitFails()

This test will check that the variables, key and nextCityName, have not changed

### TestCityChangeUp()

This test will check if the variable, currentCityNumber , has increased by one

### TestCityChangeDown()

This test will check if the variable, currentCityNumber , has decreased by one

### TestCityChangeSkipUp()

This test will check if the variable, currentCityNumber , has been set to 0

### TestCityChangeSkipDown()

This test will check if the variable, currentCityNumber , has been set to 3

### TestAddingToPrintMain()

This test will check if the first string (index [0]) of the totalOutput variable is set to "Driver 1 heading from Akina to NextCity via the Road".

### TestAddingToPrintExtra()

This test will check if the second string (index [1]) of the totalOutput variable is set to "Driver 1 has gone to Napier".

### TestMeetingExtra()

This test will check if the first string (index [0]) of the totalOutput variable is set to "Driver 1 Met With John Jamieson 5 Times!"

## Test Deliverables

Requirement Interpretation

Example of testing

Testing Report

## Environmental Needs

### Multipart Functions

Multipart Features that are used commonly for output should be tested more thoroughly. These functions should contain some branch testing.

The driveCity() function is a good example, because it changes the variables that will be printed repeatedly until the simulation ends. If compiling errors (such as type errors) don't occur, then incorrect information could be used for the program’s output. In a large amount of output, spotting the incorrect information would be difficult without manually recreating the simulation.

Multipart variables that are involved in single/unique output will not require extensive branch testing as errors will be easier to catch manually.

The extras() function is a good example because the output it creates is only displayed once per simulation. By looking at the last few lines printed by a simulation (Requirements: 13 , 12.1 , 3 , 4 , 5) a developer can easily see what additional output should also printed (Requirement: 12.2 )

### Test Data

During the tests that involve editing data inside the “Driver” struct, the initialization of the variable (of type Driver) should be created using data relevant to that test. Variables that are closely related to each other, like the “currentCity” and “currentCityName” should be initialized to match each other. This will avoid errors in the test suite, that could never happen in the actual program.

### Supporting Software

No specific version of software will be required. Using the up to date version will be fine. The software being used is the Go Language, a package called “testifiy” and a text editor (Visual Studio Code, Atom, NotePad++).

## Staffing and Training needs

The development team currently has no experience using the Go language, which this program will be developed and tested in. The development and test testing will not require expert training. Instead, having each individual developer teach himself the basics using tools like: <https://tour.golang.org> and sites like <https://golang.org/pkg/testing/>.

## Schedule

See the referred file “ITPR6.590\_Schedule.tdl”

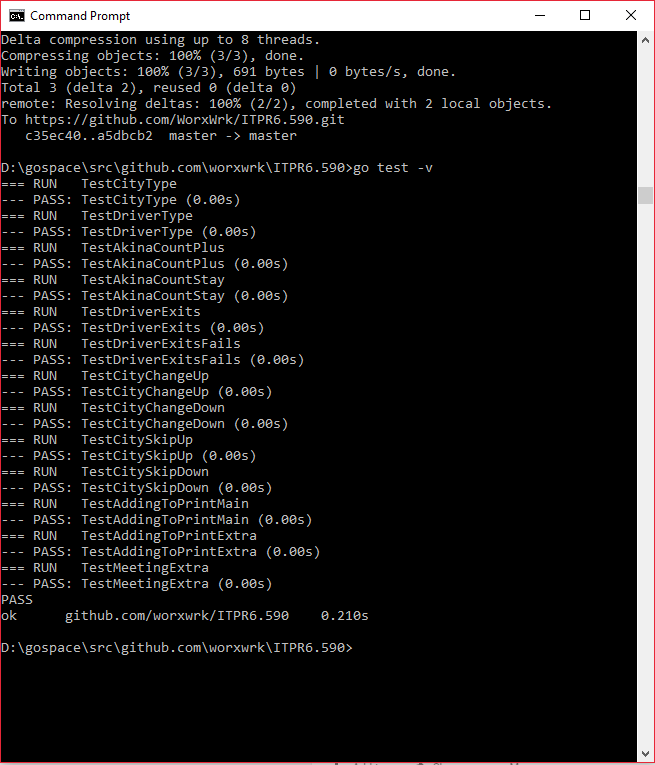
## Glossary

Simulation - Is when a single driver starts in a city, moves, and eventually leaves via an exit. (Requirement 10 shows a simulation in context to the whole program)

# Findings Report

The Test suite uses, and requires, the testify library to be installed.

## Proof



## Report

All 13 tests currently pass.

### Discoveries

There different ways of doing the comparison section of a test, each with their own advantages and disadvantages.

1. Using asserts (testify) provides good information for the error log, but only for single comparison. Standalone function
2. deep.Equals() (go-test) provides good information for the error log, but only for full comparison of objects.
3. DeepEqual() (reflect) is from an inbuilt package, but only give a true or false value for the comparison.
4. Simple if statements allow to the error log to be as detailed or simple as the coder desires. Using multiple if statements and error logs can make tests more thorough for functions with many logical decisions in them. Making the test detailed (Using multiple if statements on multiple variables) can catch unexpected branching in the function and unexpected results.

### Difficulties:

A few issues occurred while creating the tests that caused the test suite longer than expected to be completed. These issues are caused by my lack of knowledge of the Go language and it’s testing package/framework. The following are some of the issues that occurred and took time to fix/understand.

1. Downloading packages brought up an error with the Go WorkSpace. Using code like “go get github.com/go-test/deep” would cause a new file to be created within the workspace that had its own “src”, ”bin” and “pkg” folders. The package was installed into this new folder which caused the package to still be available in the test suite. (Caused :Lack of knowledge | Status: Fixed)
2. The deep.Equals() function always passed the test when used on the 2 created structs. The function worked as expected when used on example code. (Caused: Lack of knowledge | Status: Alternative found)
3. Was using the “go test” command, but was expecting output to be like the output from the “go test -v” command. This misunderstanding in commands caused me to assume that only one test was being tested. (Caused :Lack of knowledge | Status: Fixed)
4. The “akinaCount()” function was originally called “count()” in previous builds of the program. It was discovered during testing, that the function would not work due to a parameter type error. The name was changed under the impression that “count()” was a built-in function of the go language which was called instead of the function that was created. (Casued: Poor naming conventions | Status: Fixed )

### What could be better?

1. Using the “deep.Equals()” function, from the “go-test” package, for the 2 struct tests.
2. More Branch testing.
3. Test how the user input is handled.
4. Have a expected and actual test for all variables that are used in a function and give them unique Error logs (t.Error(<string>)). This would make errors easier to find. As explained in the “Environmental Needs” section of the test plan (ITPR6.590\_Unit\_Plan\_Ver\_1.0) adding these variable tests to the driveCity() function tests would make
5. Create test for when functions fail to meet the expected result
6. Combining If statements and for loops within the function tests would allow a single test to also be used under different variable states. Example for TestCityChangeUp(): the for loop would change driver.currentCityNumber variable. This would only work while the variable is 0, 1 or 2.
7. Some Error logs are poorly written.