SFT221 NGG – Milestone 2

Group 4

Test Plan

Devon Chan	***dchen80@myseneca.ca***
Jon Pablo	***japablo@myseneca.ca***
Hephzibah Nick-Oshomah	***hnick-oshomah@myseneca.ca***
Luckshihaa Krishnan	***lkrishnan1@myseneca.ca***
Shahidah Lala	***slala1@myseneca.ca***

Table of Contents

Introduction

This test plan will describe the strategies, processes, objectives, and methodologies used when testing our SFT221 Project.

Test Objectives

- Testing the software to ensure it meets the functional system requirements as outlined in the SFT221 project instructions.
- Testing the software to check for potential issues and bugs with the program.
- Ensure that the test results are properly documented and communicated to the team.

Scope

The test plan will cover the testing of all functions within the software. The plan will define the unit, integration, end to end, and regression testing plan. The plan will include testing of all system use case requirements outlined in the project instructions, as well as performance and security testing to ensure that the program is free of bugs.

Overview

The software we are testing is a program that will assist a local delivery company. This company has three different trucks that deliver on three different routes in the city. Our objective is to design a program that will calculate the shortest delivery path for the trucks and keep track of the truck's current and maximum load capacity, the number of shipments, and the shipment destination.

Glossary

SFT221	Software Testing course at Seneca College.
Euclidean Distance	It is the distance between two points. The formula is: 2√(a2+b2).
P Start T.	Planned Start Time. Abbreviation is used in the test schedule.
P End T.	Planned Finish Time. Abbreviation Is used in the test schedule

Test Strategy

We will be performing:

- Blackbox Unit Testing: Where we will divide the test into two categories: testing general use-case scenarios, and exploratory testing which tests the limits of the code. It may take roughly 1 hour to design the tests, and 30 mins to execute them.
- Whitebox Unit Testing: Where we will analyze the code for defects to find more test cases. It may take approximately 1 hour total to design and execute the Whitebox tests.
- Integration Testing: Make another set of test cases once the functions are integrated into the program and create general-use test cases as well as exploratory tests for the software. It should take around 2 to 2.5 hours to fully design and execute the integration tests.
- Acceptance Tests: These test cases will be designed after the software-requirements outlined in the project's pdf. It will take roughly 1 hour to prepare the tests, and 20 mins to execute them.

Approach

Paths Constraint

- The path's origin position is fixed to 1A.
- All paths (i.e. blue, green, and yellow) are fixed, unless a truck needs to make a detour in order to deliver a package outside of the path
    - Refer to "Path Selection/Selection Criteria for detour details.

Path Selection/Selection Criteria

- Truck select is based on which truck can reach the destination in the shortest path.
- Trucks cannot drive through buildings.
- If distances between 2 paths are equal, the truck with the lighter load is chosen.
- Distance is calculated using the Pythagorean theory.
- Trucks cannot move backwards during its' route.
- Destination must be within the delivery range, which is defined by the map's row and column.
- Each grid position is listed from (x axis, y axis), i.e.: A1.

Truck Specs

- Max load: 1000kg
- Max volume: $36m^3$

Box Specs

- Small Box: 0.25 m^3
- Medium Box: 0.50 m^3
- Large Box: 1.0 m^3
- All boxes are perfect cubes, with same dimensions on all sides.

Test Coverage

- upper and lower bounds of each function
- upper and lower bounds of each parameter type
- misuse of function, such as: incorrect datatype parameter, and passing in NULL
- display/printing errors with incorrect, correct parameters
- performance checks
- check for unused parameters, variables, and functions.
- check if calculated distance is correct, i.e., no miscalculation.

Software Component Validation

Getter Functions

- getNumRows()
- getNumCols()

Add Functions

- addRoute()
- addPtToRoute()
- addPointToRoute()
- addPointToRouteIfNot()

Algorithm + Helper Functions

- shortestPath()
- distance()
- getPossibleMoves()
- getClosestPoint()
- eqPt()

Business Process Validation

Customer must specify:

- weight of shipment in kilograms
- size of box in cubic meters
- customer must select from the 3 options (refer to Box Specs)
- shipment destination must follow the destination criteria (refer to “Path
- Selection/Selection Criteria)

Test Tools

Visual Studio Community’s Native Unit Testing Module will be utilized, along with JIRA, GitHub, and Microsoft Office. The full details of these software are found under the Environment section.

Test Types

Functional tests

- test each and all functions, initially with black box tests, followed up with white box
- Upper bounds of function specifications
- Lower bounds of function specifications
- Upper bounds of function parameter(s) datatype
- Lower bounds of function parameter(s) datatype

Unit tests

- Test the display related functions with black box tests, followed up with white box.
- White box testing used to formulate validation process.
- Validation of correct data (i.e., validate arguments passed into function)
- Black box and white box testing for all algorithm related functions.
- Black box testing to initially check the upper and lower bounds of algorithm specifications.
- White box testing to test limits of parameter datatype values.

End-to-End Tests

- test entire application, executing the executable produced by complier
- go through possible user stories (i.e. the sequence of how the user would use the application from start to end)
- including all the inputs the user could put input

Regression Test
After each code update, preceding tests should be performed again to ensure any changes do not break the code.

Test Data

The test data will be generated using following process:

- brainstorm possible use-case scenarios.
- break down each scenario and determine what type of function is needed to implement the user action.
- Develop Blackbox test data based on function prototypes, and use-case scenarios.
- Develop Whitebox test data based on the implemented functions.

Test Plan

Test Team

Devon Chan	Manage and distribute responsibilities, as well as help with editing, consulting. Also help with either the development team or the testing team. Create issues in Jira every time new work needs to be done and assigns work for himself through Jira.
Jon Pablo	Help with editing, consulting, and setting up test cases. Create issues in Jira every time new work needs to be done and assigns work for himself through Jira. Attend meetings and actively help team members.
Hephzibah Nick-Oshomah	Help with editing, consulting, and setting up test cases. Create issues in Jira every time new work needs to be done and assigns work for herself through Jira. Attend meetings and actively help team members.
Luckshihaa Krishnan	Help with editing, consulting, and setting up test cases. Create issues in Jira every time new work needs to be done and assigns work for herself through Jira. Attend meetings and actively help team members.
Shahidah Lala	Help with editing, consulting, and setting up test cases. Create issues in Jira every time new work needs to be done and assigns work for herself through Jira. Attend meetings and actively help team members.

Test Schedule

Milestone	Description	P Start T.	P End T.
MS3	Document a set of Blackbox tests for the functions	Mar 16	Mar 21
MS3	Blackbox test code	Mar 16	Mar 21
MS3	Set up the function-test matrix	Mar 16	Mar 21
MS4	Execute the Blackbox tests	Mar 23	Mar 28
MS4	Create Whitebox tests based on implemented functions	Mar 23	Mar 28
MS4	Execute the Whitebox tests	Mar 23	Mar 28
MS4	Debug and re-execute tests	Mar 23	Mar 28
MS5	Create integration tests	Mar 30	Apr 4
MS5	Create acceptance tests	Mar 30	Apr 4
MS5	Execute the integration tests	Mar 30	Apr 4
MS5	Debug and re-execute tests	Mar 30	Apr 4
MS5	Update test-matrix	Mar 30	Apr 4
MS6	Execute acceptance tests	Apr 6	Apr 11
MS6	Debug and re-execute tests	Apr 6	Apr 11
MS6	Create the final test report	Apr 6	Apr 11

Environment

To perform the tests, we will need a machine capable of running Visual Studio Community. This usually means a Windows machine is required. The following software are also required:

Visual Studio Community	This is required to run the Unit Testing framework that comes with Visual Studio Community. Since this software only supports the Windows operating system, a Windows machine is required.
Program's source code	Since we will be building and programming test cases on top of the source code, we will need to have the source code analyzed to properly test it. Knowledge in C and the VS Community's Native Unit Testing Module is required.
JIRA	Jira will be utilized to manage and keep track of our testing. Jira is a tool that a lot of teams use to organize projects, and for our test plan, we will use Jira to keep track of which software tests needs to be written, executed, and documented. Becoming familiar with how to use Jira is necessary, and can be done by reading the notes on: https://software-testing.sdds.ca
GitHub	For each milestone, we will deliver any deliverables onto GitHub. This includes final test documentation, reports, and source codes for this project.
Microsoft Suite	We are using Microsoft Suite to document the test data and results, and to communicate with the team.

Features

All features will be tested, such as: being able to calculate which truck should be selected to deliver a shipment. The software should also test all the constraints which are listed in the Approach section.

Since all functions will be tested, no functionality will be left out in this test plan.

Testing Procedure

The testing procedure is broken down into four steps: The Blackbox unit testing, Whitebox unit testing, Integration testing, and acceptance testing.

Blackbox Unit Test	The Blackbox unit test will be done through the Native Unit Testing module in Visual Studio Community. We will create Blackbox test cases using the function's prototype and create exploratory tests such as testing each variable in the parameter list. We will also create general-use test cases based on the program specifications outlined in the project's pdf. We will also test the minimum and maximum limits of each variable. The pass criteria would be for the program to be smart enough to provide a descriptive error message if the user's input is invalid, and to give the correct output if the user's input is valid. This may take roughly 2 days to complete.
Whitebox Unit Test	The Whitebox unit test will also be done through the Native Unit Testing module in Visual Studio Community. After having access to the function's code, we will be able to test the code more thoroughly by checking every possible path the function takes. The pass criteria would be for the program to be smart enough to provide a descriptive error message if the user's input is invalid, and to give the correct output if the user's input is valid. This may take roughly 2 days to complete.
Integration Test	The third procedure is the integration test. This is when we merge the new functions with the source code and see if the program can work together with all the given functions and give the correct output. This may take roughly 2 to 3 days to complete.
Acceptance Test	The acceptance test will verify that the code runs, and that the correct output is generated. We will analyze and test all requirements and program specifications to finalize and our software. This should take around 2 days to complete.

Test cases are broken down into two categories: exploratory test cases, and general-use test cases. General use follows all of the user-requirements for the program, and tests the program with input similar to what a user may use. The success rate for these tests should be 100%. Exploratory test cases test the bounds of the system, and the inputs can be irregular input. The success rate for these tests may be around 95%, because there are certain bugs that may have little to no impact on the usage of the program. This is further expanded on the defects management part of the test plan.

Defect Management & Risks

Any defects or bugs should be reported by creating a bug issue in Jira. The issue should have a description stating what the problem is, the filename that triggered the issue, and the line number.

The defects are ranked by importance from 1 to 5:

1. **Critical** — These are bugs that will greatly affect the system and cannot be worked around, thus should be the highest priority of the team.

   Critical defects risk potentially delaying development process. Workarounds may include developing other functions and other units so development can continue. However, critical defects must be addressed and fixed asap to properly finish the project.

2. **High** — These are bugs that will affect the base functionality of the software but still allows some features to function.

   Runs high risk of crashing the entire program. Possible workarounds include providing the correct, supported user-input, but high-severity bugs should be addressed asap.

3. **Medium** — These are bugs that do not necessarily affect the more important features of the software but run the potential of being discovered by the user and becoming a bigger issue as the software development progresses.

   The risk may not be so impactful to the system, but the bug should be eventually fixed, as it may lead to other problems down the line.

4. **Low** — These are bugs that are not very important. They run a low risk of being discovered by the user and occur very rarely. Even when the bug occurs, the user can workaround the bug easily, as these bugs have a low impact on the program's functionality. These bugs should still be fixed to ensure the complete workability of the program.

5. **Cosmetic or Informational** — These are bugs that make the user interface less readable or optimal but are not impactful enough to affect the functionality of the program.

Exit Criteria

Testing is complete when all test data has been executed, and at least 95% of tests pass. Also, there should be no critical, high, medium, or low-level defects remaining.

Risks

There are 5 types of risks that may occur during the implementation of this plan:

Schedule Risk: Tight deadlines and unavailability of teammates may result in missing meetings, deadlines, or important information that may affect and compromise the quality of this project.

Technical Risk: Technical issues with teammate's hardware may occur, but with GitHub, JIRA, and OneDrive, the risks are minimalized. If a component of the project becomes too complicated, this should be mentioned in the group asap to schedule a group meeting to work through any issues together.

Management Risk: May occur when leadership is not taken, and teammates feel confused with the direction of the project. Inaccurate time-estimation may also lead to compromises with the quality of the project.

Personnel Risk: Personnel should always be communicative and let teammates know if they cannot complete their part of the work in time, and to express any concerns they have with the project or material as soon as possible.

Requirements Risk: The requirements and deliverables for the project should be properly analyzed to ensure that this issue is minimalized and that teammates have a clear understanding on what the expectations are for each component of the project.

Documentation

The team will produce, and maintain the following documents for this project:

The Test Plan: It is this current document, and it keeps track of the initial plan we have for this project.

Blackbox Unit Testing Matrix: A document containing a detailed list of test data for Milestone 3.

Blackbox Unit Test Code: The code for the above tests written using C and the Native Unit Testing Module.

Whitebox Unit Testing Matrix: A document containing additional test data for Milestone 4.

Unit Testing Matrix: Blackbox and Whitebox test data should be executed and recorded.

Bug Fixes: Document capturing any bugs fixed during the development process should be recorded and maintained throughout the project.

Integration Testing Matrix: Document containing the test data and execution for the integration tests.
Acceptance Testing Matrix: Document containing the test data and execution for the acceptance tests.

Final Test Report: A final testing report should be conducted, which lists our conducted tests, bugs fixed, and the final testing results.

Test Approval

The testing results will be approved by everyone through a team meet-up. During the meet-up we will discuss the test results in-depth to ensure the testing was done correctly.

Business Requirements

R#	MODULE	DESCRIPTION
R001	Validate()	Post office clerk can verify that the customers package size is within the correct size specification. Result: size is small (0.25 m^3), medium (0.50 m^3), or large (1.00 m^3)
R002	Validate()	Post office clerk can verify that the customer's package weight is within the correct weight specification. Result: weight is between 1 to 1000kg.
R003	Validate()	Post office clerk can verify that the customer's package has a valid destination address, to prevent delivery to wrong address. Result: Address is within our delivery range of: {0, 0} to {25, 25}
R004	assignTruck()	When assigning packages to a truck, the truck should make the least number of diversions to get to the destination. Result: Minimizes fuel and time needed for truck to make all deliveries.
R005	assignTruck()	When assigning packages to a truck, the truck's current weight load should not exceed its maximum weight load capacity. Result: All trucks current load should be equal to or less than 1000kg.
R006	assignTruck()	When assign packages to a truck, the truck's current density load should not exceed its maximum density load capacity. Result: All truck's current density is equal to or less than 36 m^3.
R007	assignTruck()	When assigning packages to a truck, the distance takes the highest priority. Result: Packages closest to a Truck's route will be assigned to that truck.
R008	assignTruck()	When assigning packages to a truck, the truck's available weight takes second priority. Result: If two trucks have the same distance, then the truck carrying less weight shall be assigned to deliver the package.
R009	assignTruck()	When assigning packages to a truck, the truck's available volume takes third priority. Result: If two trucks have the same distance and available weight, then the truck carrying less volume shall be assigned to deliver the package.
R010	assignTruck()	When assigning packages to a truck, if distance, volume, and or density is all the same, then the truck is chosen by this priority: blue, green, and then yellow. Result: if two or more trucks have the same delivery efficiency, blue is prioritized first, then green, and lastly yellow.
R011	assignTruck()	When assigning packages to a truck, if all trucks are unavailable to make the delivery, then the package will be left in the post office to be delivered the next day. Result: If no trucks are available, no truck will be assigned that day.