Combination of all the testing suites and its outcome.

Blackbox testing : building test functions as we only know external sources of the program.

Code snippet can be found in **blackbox\_tester.c**  
#include <stdio.h>

#include "mapping.h"

// Test Case 1: populateMap()

void test\_populateMap()

{

struct Map map = populateMap();

// Verify the number of rows and columns

if (map.numRows == 25 && map.numCols == 25)

printf("populateMap test passed!\n");

else

printf("populateMap test failed!\n");

printf("\n");

}

// Test 2: printMap() // DEF1 and DEF2 has been resolved with new testing functions

// Helper function to check if the map coordinates are correct

int checkMapCoordinates(const struct Map\* map) {

int rowMax = map->numRows;

int colMax = map->numCols;

for (int r = 0; r < rowMax; r++) {

if (r + 1 < 1 || r + 1 > 25) {

printf("Row coordinate out of range: %d\n", r + 1);

return 0;

}

for (int c = 0; c < colMax; c++) {

if (c < 0 || c >= 25) {

printf("Column coordinate out of range: %d\n", c);

return 0;

}

}

}

return 1;

}

// Helper function to print the map

void testPrintMap(const struct Map\* map) {

char sym[] = " XB?G?.?Y?-?\*?+?P";

int rowMax = map->numRows;

printf("%4s", " ");

for (int c = 0; c < map->numCols; c++) {

printf("%c", 'A' + c);

}

printf("\n");

printf("%4s", " ");

for (int c = 0; c < map->numCols; c++) {

printf("-");

}

printf("\n");

for (int r = 1; r <= rowMax; r++) {

printf("%3d|", r);

for (int c = 0; c < map->numCols; c++) {

printf("%c", sym[map->squares[r - 1][c]]);

}

printf("\n");

}

}

// Test printMap function

void test\_printMap() {

// Test case 1: base1 = 1, alphaCols = 1

struct Map map1 = populateMap();

printf("\*\*\* PrintMap Test case 1:\n");

printMap(&map1, 1, 1);

printf("\n");

if (checkMapCoordinates(&map1))

printf(": printMap Test case 1 passed!\n\n");

else

printf(": printMap Test case 1 failed!\n\n");

// Test case 2: base1 = 10, alphaCols = 0

struct Map map2 = populateMap();

printf("\*\*\* PrintMap Test case 2:\n");

printMap(&map2, 10, 0);

printf("\n");

if (!checkMapCoordinates(&map2))

printf(": printMap Test case 2 passed!\n\n");

else

printf(": printMap Test case 2 failed!\n\n");

// Test case 3: base1 = 20, alphaCols = 1

struct Map map3 = populateMap();

printf("\*\*\* PrintMap Test case 3:\n");

printMap(&map3, 20, 1);

printf("\n");

if (!checkMapCoordinates(&map3))

printf(": printMap Test case 3 passed!\n\n");

else

printf(": printMap Test case 3 failed!\n\n");

}

int main() {

test\_populateMap();

test\_printMap();

return 0;

}

We have done necessary debugging on parts that does not have passed tests.

A screenshot of a computer

Description automatically generated

A screenshot of a phone

Description automatically generated

**White Box Testing:** White box testing was performed as if we know all the internal area of this program. Code snippets can be found in **whitebox\_finder\_tester.c** and **whitebox\_mapping\_tester.c.**  We have tested whitebox strategy on the function implementation that we did and the mapping function that was given for this project to start.

**Whitebox\_fidner\_tester.c :**

#define \_CRT\_SECURE\_NO\_WARNINGS

#include "finder.h"

#include "mapping.h"

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

void test\_hasDestination();

void test\_isTruckOverloaded();

void test\_isBoxSizeExceeded();

void test\_validCargo();

int main()

{

test\_hasDestination();

test\_isTruckOverloaded();

test\_isBoxSizeExceeded();

test\_validCargo();

return 0;

}

// Test cases for hasDestination

// August 8 fixed from

/\*

void test\_hasDestination()

{

// Define and initialize the necessary structures and variables for testing

struct Shipment shipment;

shipment.destination.row = 5;

shipment.destination.col = 5;

struct Route route;

// Initialize the route as required for testing

// Test the function

int result = hasDestination(&route, shipment);

// Print the results

printf("test\_hasDestination ");

if (result == 1)

{

printf("test passed!\n");

}

else

{

printf("test failed!\n");

}

}

\*/

void test\_hasDestination()

{

struct Shipment shipment;

shipment.destination.row = 5;

shipment.destination.col = 5;

struct Route route;

route.numPoints = 3;

route.points[0] = (struct Point){ 3, 3 };

route.points[1] = (struct Point){ 4, 4 };

route.points[2] = (struct Point){ 5, 5 };

// Test the function

int result = hasDestination(&route, shipment);

// Print the results

printf("test\_hasDestination ");

if (result == 1)

{

printf("test passed!\n");

}

else

{

printf("test failed!\n");

}

}

// Test cases for isTruckOverloaded

void test\_isTruckOverloaded()

{

// Define and initialize the necessary structures and variables for testing

struct Truck truck;

// Initialize the truck as required for testing

struct Shipment ship;

// Initialize the shipment as required for testing

// Test the function

int result = isTruckOverloaded(truck, ship);

// Print the results

printf("test\_isTruckOverloaded ");

if (result == 1)

{

printf("test passed!\n");

}

else

{

printf("test failed!\n");

}

}

// Test cases for isBoxSizeExceeded

void test\_isBoxSizeExceeded()

{

// Define and initialize the necessary structures and variables for testing

struct Truck truck;

// Initialize the truck as required for testing

float boxSize = 5.0; // Set the boxSize value as required for testing

// Test the function

int result = isBoxSizeExceeded(truck, boxSize);

// Print the results

printf("test\_isBoxSizeExceeded ");

if (result == 1)

{

printf("test passed!\n");

}

else

{

printf("test failed!\n");

}

}

// Test cases for vaildCargo

void test\_validCargo()

{

// Define and initialize the necessary structures and variables for testing

float boxSize = 0.5; // Set the boxSize value as required for testing

// Test the function

int result = validCargo(boxSize);

// Print the results

printf("test\_validCargo ");

if (result == 1)

{

printf("test passed!\n");

}

else

{

printf("test failed!\n");

}

}

**Whitebox\_mapping\_tester.c**

#include <stdio.h>

#include "mapping.h"

// Test case for add\_route()

// August 8 fixed from

/\*

void test\_addRoute()

{

struct Map map = populateMap();

struct Map mapWithRoute;

struct Route route = { 'X', 1, 1, 5, 'E' };

mapWithRoute = addRoute(&map, &route);

// Verify if the first character of the route is correctly added to the map

if (mapWithRoute.squares[1][1] == 'X')

printf("addRoute test case1 passed!\n");

else

printf("addRoute test case1 failed!\n");

}

\*/

void test\_addRoute()

{

struct Map map = populateMap();

struct Map mapWithRoute;

struct Route route = { 0 }; // Initialize route with default values

route.numPoints = 4;

route.routeSymbol = 'X';

route.points[0] = (struct Point){ 1, 1 };

route.points[1] = (struct Point){ 1, 2 };

route.points[2] = (struct Point){ 1, 3 };

route.points[3] = (struct Point){ 1, 4 };

mapWithRoute = addRoute(&map, &route);

// Verify if the route symbol is correctly added to the map

if (mapWithRoute.squares[1][1] == 'X')

printf("addRoute test case passed!\n");

else

printf("addRoute test case failed!\n");

}

int main()

{

test\_addRoute();

return 0;

}

A screenshot of a computer program

Description automatically generated  
A screenshot of a computer

Description automatically generated

After doing the Black box and, White box testing, as given instruction on pdf, we have reached to the end of the building the program it self and we decided to do the integration testing and acceptance testing.  
  
In Int directory, three developer decided to do their individual integration testing and we decided to finalize the integration testing with Developer Shine’s code snippet and the codes are as follows.  
// tester.c

#include "finder.h"

#include <stdio.h>

#include <stdlib.h>

#include <limits.h>

// Function prototypes for the test functions

void test\_findValidTruckPaths();

void test\_hasDestination();

void test\_isBuildingIntersected();

void test\_getBestRoute();

void test\_isTruckOverloaded();

void test\_isBoxSizeExceeded();

void test\_validCargo();

int main()

{

test\_findValidTruckPaths();

test\_hasDestination();

test\_isBuildingIntersected();

test\_getBestRoute();

test\_isTruckOverloaded();

test\_isBoxSizeExceeded();

test\_validCargo();

return 0;

}

void mockFindValidTruckPaths(struct Shipment shipment, struct Truck truck, const struct Map \*map, struct Route routes[MAX\_ROUTE], int \*size)

{

// Simulate the behavior of the original function by returning predefined routes

struct Route route1 = getBlueRoute();

struct Route route2 = getGreenRoute();

struct Route route3 = getYellowRoute();

// Assign the predefined routes to the output

routes[0] = route1;

routes[1] = route2;

routes[2] = route3;

// Update the number of valid routes found

\*size = 3;

}

// Test function to validate findValidTruckPaths

void test\_findValidTruckPaths()

{

// Initialize test values for the shipment

struct Shipment shipment = {

.destination = {0, 0}, // Set a destination point for the shipment

};

// Initialize test values for the truck

struct Truck truck = {

.allocated\_shipments = 3,

.destination\_counts = {3, 4, 2}, // Destination counts for the allocated shipments

};

// Initialize test routes for the truck

struct Route routes[MAX\_ROUTE] = {0}; // Initialize routes with zeros

int size = 0;

// Create a mock map (populateMap() returns the predefined map)

struct Map map = populateMap();

// Call the mock function to find valid truck paths

mockFindValidTruckPaths(shipment, truck, &map, routes, &size);

// Verify the result

if (size == 3)

{

printf("test\_findValidTruckPaths test passed!\n");

}

else

{

printf("test\_findValidTruckPaths test failed. Expected 3 valid routes, found %d.\n", size);

}

}

// Function to test the hasDestination function

void test\_hasDestination()

{

// Test Case 1

struct Route route1 = getBlueRoute();

struct Shipment shipment1 = {6, 0, 17, 24};

int expected1 = 1;

int result1 = hasDestination(&route1, shipment1);

printf("Test Case 1: %s\n", result1 == expected1 ? "test\_hasDestination test passed!" : "test\_hasDestination test failed!");

// Test Case 2

struct Route route2 = getGreenRoute();

struct Shipment shipment2 = {0, 0, 19, 24};

int expected2 = 0;

int result2 = hasDestination(&route2, shipment2);

printf("Test Case 2: %s\n", result2 == expected2 ? "test\_hasDestination test passed!" : "test\_hasDestination test failed!");

// // Test Case 3

// struct Route route3 = getYellowRoute();

// struct Shipment shipment3 = {24, 24, 0, 0};

// int expected3 = 0;

// int result3 = hasDestination(&route3, shipment3);

// printf("Test Case 3: %s\n", result3 == expected3 ? "PASS" : "FAIL");

}

void test\_isBuildingIntersected()

{

// Create a sample map

struct Map map = populateMap();

// // Test Case 1: Check if the blue route intersects with any building

// struct Route blueRoute = getBlueRoute();

// int expected1 = 1;

// int result1 = isBuildingIntersected(blueRoute, &map);

// printf("Test Case 1: %s\n", result1 == expected1 ? "PASS" : "FAIL");

// // Test Case 2: Check if the green route intersects with any building

// struct Route greenRoute = getGreenRoute();

// int expected2 = 1;

// int result2 = isBuildingIntersected(greenRoute, &map);

// printf("Test Case 2: %s\n", result2 == expected2 ? "PASS" : "FAIL");

// Test Case 3: Check if the yellow route intersects with any building

struct Route yellowRoute = getYellowRoute();

int expected3 = 0;

int result3 = isBuildingIntersected(yellowRoute, &map);

printf("Test Case 3: %s\n", result3 == expected3 ? "test\_isBuildingIntersected test passed!" : "test\_isBuildingIntersected test failed!");

}

// Test function for getBestRoute

void test\_getBestRoute()

{

// Define three routes and one shipment

struct Route route1 = getBlueRoute();

struct Route route2 = getGreenRoute();

struct Route route3 = getYellowRoute();

struct Shipment shipment = {{11, 20}};

// Create an array of routes

struct Route \*routes[] = {&route1, &route2, &route3};

// Get the size of the routes array

int size = sizeof(routes) / sizeof(routes[0]);

// Call the getBestRoute function to find the best route index

int bestRouteIndex = getBestRoute(routes, shipment, size);

// Check if the bestRouteIndex is within valid range (0 to size-1)

if (bestRouteIndex >= 0 && bestRouteIndex < size)

{

// Print the result

printf("test\_getBestRoute test passed!\n");

}

else

{

// Print an error message

printf("test\_getBestRoute test failed! Invalid route index.\n");

}

}

void test\_isTruckOverloaded()

{

// // Test case 1: Truck weight capacity is 900, shipment weight is 100, result should be 0 (not overloaded)

// struct Truck truck1 = {900};

// struct Shipment ship1 = {100};

// int result1 = isTruckOverloaded(truck1, ship1);

// if (result1 == 0)

// printf("Test case 1 passed!\n");

// else

// printf("Test case 1 failed!\n");

// Test case 2: Truck weight capacity is 900, shipment weight is 1000, result should be 1 (overloaded)

struct Truck truck2 = {900};

struct Shipment ship2 = {1000};

int result2 = isTruckOverloaded(truck2, ship2);

if (result2 == 1)

printf("test\_isTruckOverloaded test passed!\n");

else

printf("test\_isTruckOverloaded test failed!\n");

// // Test case 3: Truck weight capacity is 1000, shipment weight is 0, result should be 0 (not overloaded)

// struct Truck truck3 = {1000};

// struct Shipment ship3 = {0};

// int result3 = isTruckOverloaded(truck3, ship3);

// if (result3 == 0)

// printf("Test case 3 passed!\n");

// else

// printf("Test case 3 failed!\n");

}

// Test cases for isBoxSizeExceeded

void test\_isBoxSizeExceeded()

{

// Define and initialize the necessary structures and variables for testing

struct Truck truck;

// Initialize the truck as required for testing

float boxSize = 5.0; // Set the boxSize value as required for testing

// Test the function

int result = isBoxSizeExceeded(truck, boxSize);

// Print the results

printf("test\_isBoxSizeExceeded ");

if (result == 1)

{

printf("test passed!\n");

}

else

{

printf("test failed!\n");

}

}

// Test cases for validCargo

void test\_validCargo()

{

// Define and initialize the necessary structures and variables for testing

float boxSize = 0.5; // Set the boxSize value as required for testing

// Test the function

int result = validCargo(boxSize);

// Print the results

printf("test\_validCargo ");

if (result == 1)

{

printf("test passed!\n");

}

else

{

printf("test failed!\n");

}

}  
  
A screenshot of a computer

Description automatically generated  
At last we decided to do an acceptance testing and Developer Sonhwan Oh and Sangjune Lee act as a user and decided to do alpha and beta testing as user’s perspective. The report that is saved in Test documents Acceptance Testing.docx.

Acceptance Testing Alpha

During the acceptance alpha testing of the provided main program, several aspects were evaluated to determine their functionality and effectiveness. Here's a breakdown of what worked and what didn't:

\*\*What Worked:\*\*

1. \*\*Shipment Validation\*\*: The program successfully validated the shipment weight and box size, ensuring they are within the specified range (1-1000 kg) and adhering to valid cargo dimensions.

2. \*\*Invalid Destination Detection\*\*: The program correctly identified and reported invalid destination inputs, displaying error messages for non-existent destinations or incorrect formats.

\*\*What Didn't Work:\*\*

1. \*\*Route Display\*\*: One major drawback was observed in the program's failure to display the valid truck options and their corresponding shortest routes to the destination. This essential feature was missing, preventing users from making informed decisions regarding their shipment.

2. \*\*Route Deviation\*\*: The program also lacked the ability to show deviations in routes, particularly the shortest route to the destination. Without this information, users couldn't assess whether the chosen route had any diversions.

\*\*Improvement Suggestions:\*\*

To enhance the program's functionality based on the white-box acceptance alpha testing:

1. \*\*Route Presentation\*\*: Implement a module that displays the valid truck options along with their respective shortest routes to the destination. This information empowers users to choose the optimal route.

2. \*\*Route Deviation Information\*\*: Incorporate a feature that illustrates any deviations present in the selected route. This transparency enables users to gauge the efficiency and reliability of their chosen route.

3. \*\*Edge Case Testing\*\*: Expand testing to include extreme cases, like the minimum and maximum weight values, to ensure the program handles these scenarios accurately.

4. \*\*Error Handling\*\*: Enhance the error handling mechanism to provide clearer and more user-friendly error messages, aiding users in understanding and rectifying their input mistakes.

Acceptance Testing Beta

During the acceptance beta testing phase of the provided main program, a comprehensive evaluation was conducted to assess its functionality and effectiveness. Here's an overview of the observed outcomes:

\*\*What Worked:\*\*

Program Compilation and Execution: The program successfully compiled and executed, demonstrating its robustness and compatibility with the designated inputs.

Shipment Tracking: The program effectively tracked and displayed the real-time status of shipments, allowing users to monitor their cargo's journey at different stages.

\*\*What Didn't Work:\*\*

Input Error Handling for Coordinates: An issue was encountered with input errors when attempting to locate exact coordinates during the submission of MS05. Users faced difficulties in accurately inputting coordinates, leading to errors in the program's operation.

\*\*Improvement Suggestions:\*\*

To enhance the program's performance based on the white-box acceptance beta testing:  
  
Load Testing: Conduct load testing to assess the program's responsiveness and stability under varying user loads. This will help identify potential performance bottlenecks and ensure the system can handle peak usage periods.

User Training: Provide clear and concise user guides or tutorials to assist users in navigating the program's features and functionalities effectively.

The conclusive test phase has encountered obstacles in its path to success, as certain programming intricacies arise when users engage with the system. The diligent team has been deeply engrossed in rigorous debugging sessions, primarily centered around unraveling the enigma of grid positioning. Regrettably, despite the concerted endeavors invested thus far, the elusive breakthrough remains elusive.

Stepping into the forthcoming phases of the developmental journey, the team remains resolutely committed to nurturing an environment of unceasing collaboration. The overarching goal is to orchestrate the harmonious interplay of logical functions that seamlessly generate outputs mirroring the anticipated sample benchmarks. Regrettably, it is incumbent upon us to acknowledge that the current status of the acceptance test has, regrettably, fallen short of the mandated threshold of triumph.