Test Plan

Test Strategy

This document aims to explain our testing strategy and why each strategy was chosen. We will also record each test and the results of each in the same place.

For this stage of the assessment we are expected to adopt another project and expand it, everything we intend on adding to the project is required to be tested. We will reuse elements of the previous developer's testing strategy. This choice was made after carefully evaluating the previous team's testing strategy and documentation. After this process, we were confident that reusing certain elements would not hurt our test plan, as all the previous tests worked and successfully confirmed that the game and its components work.

Our testing strategy is split into four stages, unit testing, requirements testing, integration testing and alpha testing. Unit testing is when we test each module of the code to make sure they function as desired, and integration testing is when we will combine the modules together and test that they function as desired. Requirements testing is when we compare our program to the requirements, and alpha testing is when we simulate actual use to find bugs.

We plan to use an agile [3] development model, so testing for each module or functionality is carried out once we have developed that module or functionality. We then do this for each module communicating our ideas with our client after every few implementations.

In the testing phase for our original game, we gave the task of testing to a small subset of the team. For this assessment, Alex will be overlooking the testing, however each member will be responsible for writing unit tests for their own code and making sure their own code works.

We will be conducting regular code checks and reviews, whilst not necessarily a form of testing, these sessions will be used to make sure our software is meeting the requirements, that we are on schedule and that our programming skills are satisfactory. We can also check the clarity of the code and make sure it is readable.

Unit Testing

The first method of testing we will be using is unit testing, this is when we test each unit (in this case each class) to make sure each function works correctly. This is a gray box test, as the tests will be written and carried out without needing to look at the class' code but with knowledge of how they function and their context. We will use JUnit [1] to carry out our unit tests. This will allow us to create testing classes that we can specify inputs to the class' functions and see if the outputs match our expectations.

This method of testing was chosen as we used Java to program the game, which is an object oriented language meaning the code will be split into convenient classes. These classes can each be tested individually to make testing simple and less overwhelming. We then can expect less issues when each of the classes are integrated. This was also the method the original development team used, allowing us to easily expand on their tests. Upon reading through the previous developer's reports and test cases, we felt confident that the majority of their code was tested to our standards and our time should be focused on our own changes.

Unit Testing Procedure

In parallel to writing our additions we will construct JUnit test classes to test each class, where available we will expand existing test classes. Once the game classes have been completed, we will run the JUnit tests with these classes to ensure the game classes function as intended. The majority of our planned implementations should be easily tested using JUnit test cases.

Submitted with this document is the unit testing table which we will use to record each unit test, class under test, function under test and status of the test. This document is titled "Unit Testing".

Examples of JUnit test classes can be found in the appendix as items A and B..

Requirements Testing

It is important that our modified game continues to meet the original requirements of our client, ontop of these requirements the game has to meet new additional requirements. The previous team claims their software meets all of our clients requirements, when deciding on the project to adopt we analysed each requirement to make sure these claims were valid. We will use a similar table to the one we used to test that the produced software meets the additional requirements.

Requirements Testing Procedure

During the development of our improvements, we will regularly conduct code reviews and checks to compare the produced code with our requirements list.

Submitted with this document is a table we produced to relate each requirement with observations, example, related classes and related methods. This document is titled "Requirements Testing".

Integration Testing

The scale for our indented additions is much smaller than our originally produced game, so we expect the integration section of testing to be much smaller as the majority of our implementations should be testable mostly with other methods

Submitted alongside this document is the integration testing table which we will use to record each test, the expected and actual results of the test. This document is titled “Integration Testing”.

For each test, a description is provided which outlines the basic action that will be performed. These actions are designed to test game functionality, and each one requires the correct execution of multiple sections of code. The expected result of each test is what we expect (and want) to happen when the described action is performed. The actual result will describe the observed behaviour of the game when the action was tried. In a successful (passed) test, the expected and actual results should be essentially the same, indicating that the game is functioning as it should. In a failed test, the actual result will not match the expected result, and efforts will then have to be made through other testing methods to identify the problematic code.

It is also worth noting, that should a test fail, the test itself will have to be examined to ensure that the expected result is indeed what should happen when the action described by the test description field is carried out. It is possible that a misunderstanding of the game requirements and the way it should function could result in tests that cannot be passed, and this testing of the tests themselves should help to avoid this.

Alpha Testing

In a similar manner to how we tested our original game, we intend to carry out a stage of alpha testing. The purpose of this alpha testing is to make sure that the adopted game has no bugs the previous development team has missed and to make sure our modifications don’t cause any issues. The period of alpha testing will take place in the final week of the project, between the dates of 12th February to the 18th.

Alpha Testing Procedure

The game will be played many times over by different members of the team, this is to test different behaviours. Many bugs that are found during development or our alpha testing periods will be reported on the issues section of our GitHub [2] repository.

During this alpha testing, we will also attempt to get users (and testers) to perform undesirable, but possible in the real world, actions to ensure that the game handles them correctly. These sorts of actions are difficult to predict, and so can’t really be tested comprehensively in any other way. Alpha testing with a large enough group of testers (especially those who are actively trying to find bugs) should be able to find areas where the game does not perform as the user would expect, even though the code may be ‘technically’ correct according to our other testing methods. It will not be feasible to conduct an alpha test with a larger group of testers for this project due to our limited time and human resources.

Below is a table we produced to log a description of each bug encountered, the classes and methods we think are involved, the report date and the current status of the bug.

Problem Description	Date Found	Solution	Status
Aircraft only follow path when selected and in path mode.	10/02	Aircraft will now follow their flight path when not selected.	Fixed
Credits screen displays nothing.	12/02	The credits screen image was removed and has now been reintroduced and updated.	Fixed
Landing an aircraft with an aircraft ready to take off will cause the aircraft to crash.	14/02	Aircraft which are ready to take off can no longer collide with other aircraft - the runways for landing and taking off are considered separate.	Fixed
High Score does not update middle and low value is current score is less than top high score.	15/02	Checks added to ensure score is compared to all values	Fixed
High Score: Top score is replaced by current score. However the old high score is not compared to other values to give ranking.	16/02	When replaced, the old high score is stored and the list is looped through again, checking for the new position for the old score.Though inefficient, this method is acceptable for the small list of only 3 scores.	Fixed

References

[1] JUnit Team, JUnit 4, [Online]. Available: http://www.junit.org. [Accessed: Jan. 14, 2014]
[2] GitHub, GitHub inc. San Francisco, California.
[3] K. Beck et al. (2001) “Manifesto for Agile Software Development”, [Online]. Available: http://agilemanifesto.org. [Accessed: Feb. 14, 2014]

Appendix

Appendix Item A - Score Test Class

```
package unitTests;

import static org.junit.Assert.*;
import logicClasses.Score;
import org.junit.Test;

public class Score_Tests {

        // Test addTime Function
        @Test
        public void testAddTime() {
                Score testScore = new Score();
                testScore.addTime(100);
                testScore.addTime(50);
                assertTrue(150 == testScore.timePlayed());
        }
        // Test addManualTime function
        @Test
        public void testAddManualTime() {
                Score testScore = new Score();
                testScore.addTimeManual(100);
                testScore.addTimeManual(900);
                assertTrue(1000 == testScore.manualTime());
        }
        @Test
        // Test addSeparationViolated function
        public void testAddSeparationViolated() {
                Score testScore = new Score();
                testScore.addSeparationViolated(4);
                testScore.addSeparationViolated(10);
                assertTrue(14 == testScore.SeparationViolated());
        }
        @Test
        // Test addFlight function
        public void testAddFlight() {
                Score testScore = new Score();
                testScore.addFlight();
                testScore.addFlight();
                testScore.addFlight();
                assertTrue(3 == testScore.flightsSuccessful());
        }
        @Test
        // Test setDifficulty function 1
        public void testSetDifficulty1() {
                Score testScore = new Score();
                testScore.setDifficulty(3);
                testScore.addTime(400);
                int score = testScore.calculate();
                assertTrue(10 == score);
        }
        @Test
        // Test setDifficulty function 2
        public void testsetDifficulty2() {
                Score testScore = new Score();
                testScore.setDifficulty(1);
                testScore.addTime(300);
                int score = testScore.calculate();
                assertTrue(3 == score);
        }
        @Test
        // Test Calculate 1
        public void testCalculate1() {
                Score testScore = new Score();
                testScore.setDifficulty(1);
                testScore.addTime(900);
                testScore.addTimeManual(300);
                testScore.addSeparationViolated(300);
                testScore.addFlight();
                testScore.addFlight();
```

```
                        testScore.addFlight();
                        testScore.addFlight();
                        testScore.addFlight();
                        testScore.addFlight();

                        int score = testScore.calculate();
                        assertTrue(209 == score);
            }
            @Test
            // Test Calculate 2
            public void testCalculate2() {
                        Score testScore = new Score();
                        testScore.setDifficulty(2);
                        testScore.addTime(7000);
                        testScore.addSeparationViolated(10);
                        testScore.addTimeManual(10);
                        testScore.addFlight();
                        testScore.addFlight();
                        testScore.addFlight();

                        int score = testScore.calculate();
                        assertTrue(380 == score);
            }



}
```

Appendix Item B - Flight Landing Take Off Tests

```
package unitTests;

import static org.junit.Assert.*;
import logicClasses.*;

import org.junit.Test;
import org.junit.Before;

public class Flight_Landing_TakeOff_Tests {

            private Airspace airspaceTakeOff, airspaceLanding;
            private Flight flightTakeOff, flightLanding;

            @Before
            public void setUpTakeOff(){
                        airspaceTakeOff = new Airspace();
            //Waypoints
            airspaceTakeOff.newWaypoint(350, 150, "A");
            airspaceTakeOff.newWaypoint(400, 470, "B");
            airspaceTakeOff.newWaypoint(700, 60,  "C");
            airspaceTakeOff.newWaypoint(800, 320, "D");
            airspaceTakeOff.newWaypoint(600, 418, "E");
            airspaceTakeOff.newWaypoint(500, 220, "F");
            airspaceTakeOff.newWaypoint(950, 188, "G");
            airspaceTakeOff.newWaypoint(1050, 272,"H");
            airspaceTakeOff.newWaypoint(900, 420, "I");
            airspaceTakeOff.newWaypoint(240, 250, "J");
            //entry point - just the airport takeOff point.
            airspaceTakeOff.addEntryPoint(airspaceTakeOff.getAirport().getTakeOffPoint());
```

```
//exit points - just the ones which aren't a landing point.
        airspaceTakeOff.newExitPoint(800, 0, "1");
        airspaceTakeOff.newExitPoint(150, 250, "2");
        airspaceTakeOff.newExitPoint(1200, 350, "3");

        flightTakeOff = new Flight(airspaceTakeOff);
}

@Before
public void setUpLanding(){
        airspaceLanding = new Airspace();
        //Waypoints
        airspaceLanding.newWaypoint(350, 150, "A");
airspaceLanding.newWaypoint(400, 470, "B");
airspaceLanding.newWaypoint(700, 60,  "C");
airspaceLanding.newWaypoint(800, 320, "D");
airspaceLanding.newWaypoint(600, 418, "E");
airspaceLanding.newWaypoint(500, 220, "F");
airspaceLanding.newWaypoint(950, 188, "G");
airspaceLanding.newWaypoint(1050, 272,"H");
airspaceLanding.newWaypoint(900, 420, "I");
airspaceLanding.newWaypoint(240, 250, "J");
        //entry point - any which arent a take off point
airspaceLanding.newEntryPoint(150, 400);
        airspaceLanding.newEntryPoint(1200, 200);
        airspaceLanding.newEntryPoint(600, 0);

        //exit point - only the landing point
        airspaceLanding.addExitPoint(airspaceLanding.getAirport().getLandingPoint());

        flightLanding = new Flight(airspaceLanding);

}

@Test
public void initialTakeOffConditions(){
        //since only entry point is the airport, the flight MUST have generated on the runway
        //Testing inital conditions for a flight which must take off

        //Flight must be at ground level ie on the runway
        assertTrue(flightTakeOff.getAltitude() == 0);
        //Flight must be requesting to take off
        assertTrue(flightTakeOff.isRequestingToTakeOff() == true);
        //Flight must not yet be permitted to take off
        assertTrue(flightTakeOff.isPermittedToTakeOff() == false);
        //The Airport should have raised the flight on runway flag.
        assertTrue(airspaceTakeOff.getAirport().isFlightOnRunway() == true);
}

@Test
public void permitTakeOff(){
        double oldX = flightTakeOff.getX();
        double oldY = flightTakeOff.getY();
        //permit flight to take off
        flightTakeOff.permitTakeOff();
        //update flight once
        flightTakeOff.update();
        //test conditions after one update - flight allowed to take off, no longer requesting to take off
        // flight should have ascended and moved from initial position.
        assertTrue(flightTakeOff.getAltitude() == Flight.changeAltitudeRate);
        assertTrue(flightTakeOff.isRequestingToTakeOff() == false);
        assertTrue(flightTakeOff.isPermittedToTakeOff() == true);
        //there should no longer be a flight on the runway
        assertTrue(airspaceTakeOff.getAirport().isFlightOnRunway() == false);
```

```
		//update a few more times to move flight further
		for (int i = 0; i<=10; i++){
			flightTakeOff.update();
		}
		//ensure that a flight permitted to take off moves from the airport.
		assertTrue(oldX != flightTakeOff.getX() && oldY != flightTakeOff.getY());
	}

	@Test
	public void initialLandingConditions(){
		//since only exit point is the airport, flight MUST need to land at some point in it's plan.
		ExitPoint lastWaypoint = flightLanding.getFlightPlan().getExitPoint();
		double airportX = airspaceLanding.getAirport().getLandingPoint().getX();
		double airportY = airspaceLanding.getAirport().getLandingPoint().getY();

		//last waypoint in flight plan should be the airport for landing.
		assertTrue(lastWaypoint.getX() == airportX && lastWaypoint.getY() == airportY);
	}

	@Test
	public void checkRequestToLand(){
		//update until target waypoint is the landing exit point
		while(flightLanding.getFlightPlan().getPointByIndex(0) != flightLanding.getFlightPlan().getExitPoint()){
			flightLanding.update();
		}
		//update once further, so flight notices the next target is the landing exit point via updateRequiredToLand
		flightLanding.update();
		//since target is exit point, and target is the airport landing point, flight must be requesting to land
		assertTrue(flightLanding.isRequestingToLand() == true);
		//flight has not yet been permitted to land.
		assertTrue(flightLanding.isPermittedToLand() == false);
	}

	@Test
	public void checkPermitToLand(){
		//update until target waypoint is the landing exit point
		while(flightLanding.getFlightPlan().getPointByIndex(0) != flightLanding.getFlightPlan().getExitPoint()){
			flightLanding.update();
		}
		//update once further, so flight notices the next target is the landing exit point via updateRequiredToLand
		flightLanding.update();

		//permit flight to land.
		flightLanding.permitToLand();
		//flight is now permitted to land
		assertTrue(flightLanding.isPermittedToLand() == true);
		//flight should no longer be requesting to land
		assertTrue(flightLanding.isRequestingToLand() == false);
		//flight's target altitude should be zero.
		assertTrue(flightLanding.getTargetAltitude() == 0);

	}
}
```