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.

Our testing strategy is split into four stages, unit testing, integration testing, requirements 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 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.

Alex will be designated the task of overviewing the testing of the game, the people responsible for programming, Steven, Huw and Josh will have responsibility in testing their produced modules. Gareth will be overseeing the integration testing stage this is to spread the workload of the assessment evenly.

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.

Unit Testing Procedure

In parallel to writing the code we will construct JUnit test classes to test each class. Once the game classes have been completed, we will run the JUnit tests with these classes to ensure the game classes function as intended. Creating unit test classes may be time consuming for many classes as environments to simulate actual usage would have to be created, this may become time consuming.

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 1, 2 and 3.

Integration Testing

Once we are happy each of the modules of our code are working correctly we will combine them together using the “big bang” approach. In practice we expect our units will be developed alongside other units and this approach may be unrealistic.

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 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.

Acceptance Testing

We chose to use an agile development model, so regular contact with our client will be made. After each few features are implemented we intend to consult the client to make sure he is happy with the current state of the software. The client will be acting as a client for all teams, so our time to discuss our project will be limited.

Acceptance Testing Example

An example of the feedback from one of our client interaction sessions is shown here. This meeting took place on the 3rd December 2013. To check that the client was happy with the game, all of the currently implemented functions were demonstrated to him and his comments noted as shown below.

Feedback from the client:

- Make exit points visible so the user knows where the plane has to go
- A plane under manual control should still be able to pass through waypoints and move on to the next one
- Consider changes to game over screen
    - Make the text faster so that the user doesn't have to wait
- Make the game a little 'lighter'
    - Less gruesome
- Think about adding sound to the game
    - Although this is technically a non required feature, it can be added if time allows
- Consider showing crashes in some way
    - Animation
    - Sound

These comments were then discussed during a team meeting and the necessary changes implemented, evidence of this can be found in the game itself, or in the accompanying architecture and GUI reports.

Requirements Testing

Once the game has been written, we will need to check that the game meets all our clients requirements. We will relate each one of the requirements with evidence in the form of corresponding test cases. Some of the requirements will be hard to test, especially the non-functional requirements.

Requirements Testing Procedure

During the development of the game, 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”.

Alpha Testing

When the game is near completion, we will focus our testing on finding any bugs which we may have missed, alpha testing will be used to make sure no errors are encountered during regular usage. Certain requirements will be hard to give a definite pass, so we will conduct periods of alpha testing where we try and confirm we have met these requirements. Our period of Alpha testing will begin in the last couple of weeks of the project, and testing will mostly consist of playing the game many times.

Alpha Testing Procedure

The game will be played many times over by different members of the team, this is to test different behaviours. Any bugs found 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
Attempting to move a corner waypoint of an aircraft’s flight path will crash the game.	10/01/2014	Prevent the user from moving the corner waypoints in the aircraft’s flight path.	Fixed
Multiple parts of an aircraft’s flight path can all be moved to a single waypoint.	10/01/2014	It was decided not to fix this as it would take lots of effort to fix and it will not affect regular gameplay.	Non - Issue

Two planes may spawn in the same corner waypoint causing them to cause the game to end.	11/01/2014	This is a very rare occurrence, it was decided to devote time to more important bugs and feature additions.	Non - Issue
The game over screen tells the user to “press any key to continue” however the only key that continues is the space.	12/01/2014	The user is now able to press any key to start a new game.	Fixed
Manually controlling planes of the edge of the screen removes them from the game.	12/01/2014	The plane will now not be removed, but will not be displayed, manual control will be exited. The plane now will continue it’s flight path.	Fixed
Moving the plane using the compass interface will prevent the user from using the arrow keys for direction.	14/01/2014	The user will now be able to switch freely between control methods.	Fixed
Entering manual control whilst dragging a waypoint will make a plane invisible.	14/01/2014	It is no longer possible to modify the route of an aircraft under manual control, this prevents the route from being changed.	Fixed
Holding down a key when two aircraft crash will skip the game over screen.	15/01/2014	When the game ends, any held down keys will not interrupt the game.	Fixed
Planes are not selectable when transitioning between altitude layers.	16/01/2014	Clicking a plane at any altitude will now select it.	Fixed
Selecting credits after playing a game will crash the game.	17/01/2014	The game will no longer interfere with the credits interface.	Fixed
Occasionally the counter that displays the number of airspace in the game will be wrong.	17/01/2014	The counter will now accurately track the number of aircraft in the airspace.	Fixed
The user cannot change altitude of a plane with the mouse button, but scrolling still displays a message in the console.	20/01/2014	Scrolling with the mousewheel no longer prompts a console message.	Fixed

Alpha testing will be used to monitor the frames per second of the application, to make sure we could meet requirement NF5. And we will also make sure that the software has no issues with the lab computers for meeting requirement F9.

References

[1] JUnit Team, JUnit 4, [Online]. Available: http://www.junit.org. [Accessed: Jan. 14, 2014]
[2] GitHub, GitHub inc. San Francisco, California.

Appendix

Appendix Item A - Aircraft Test Class

```
package tst;

import static org.junit.Assert.*;

import org.junit.Test;

import cls.Aircraft;
import cls.Waypoint;
import cls.Vector;

public class AircraftTest {

		// Create test aircraft
		private Aircraft generateTestAircraft()
		{
				Waypoint[] waypointList = new Waypoint[]{new Waypoint(0, 0, true), new Waypoint(100, 100, true), new
Waypoint(25, 75, false), new Waypoint(75, 25, false), new Waypoint(50,50, false)};
				Aircraft testAircraft = new Aircraft("testAircraft", "Berlin", "Dublin", new Waypoint(100,100, true), new
Waypoint(0,0, true), null, 10.0, waypointList, 1);
				return testAircraft;
		}

		// Test get functions
		// Test getPosition function
		@Test
		public void testGePosition() {
				Aircraft testAircraft = generateTestAircraft();
				Vector resultPosition = testAircraft.position();
				assertTrue("x >= -128 and xy <= 27, y = 0, z = 28,000 or z = 30,000", ((0 == resultPosition.y()) && (128 >=
resultPosition.x()) && (-128 <= resultPosition.x()) && ((28000 == resultPosition.z()) || (30000 == resultPosition.z()))));
		}
		// Test getName function
		@Test
		public void testGetName() {
				Aircraft testAircraft = generateTestAircraft();
				String name = testAircraft.name();
				assertTrue("Name = testAircraft", "testAircraft" == name);
		}
		// Test getOriginName function
		@Test
		public void testGetOriginName(){
				Aircraft testAircraft = generateTestAircraft();
```

```
			String name = testAircraft.originName();
			assertTrue("Origin name = Dublin", "Dublin" == name);
		}
		// Test getDestinationName function
		@Test
		public void testGetDestinationName(){
			Aircraft testAircraft = generateTestAircraft();
			String name = testAircraft.destinationName();
			assertTrue("Destination name = Berlin", "Berlin" == name);
		}
		// Test getIsFinished function
		@Test
		public void testGetIsFinishedName(){
			Aircraft testAircraft = generateTestAircraft();
			boolean status = testAircraft.isFinished();
			assertTrue("Finished = false", false == status);
		}
		// Test getIsManuallyControlled function
		@Test
		public void testIsManuallyControlled(){
			Aircraft testAircraft = generateTestAircraft();
			boolean status = testAircraft.isManuallyControlled();
			assertTrue("Manually controlled = false", false == status);
		}
		// Test getSpeed function
		@Test
		public void testGetSpeed(){
			Aircraft testAircraft = generateTestAircraft();
			double speed = (int) (testAircraft.speed() + 0.5);
			assertTrue("Speed = 20", speed == 20.0);
		}
		// Test getAltitudeState
		@Test
		public void testAltitudeState(){
			Aircraft testAircraft = generateTestAircraft();
			testAircraft.setAltitudeState(1);
			int altState = testAircraft.altitudeState();
			assertTrue("Altitude State = 1", altState == 1);
		}

		// Test outOfBounds
		@Test
		public void testOutOfBounds(){
			Waypoint[] waypointList = new Waypoint[]{new Waypoint(0, 0, true), new Waypoint(100, 100, true), new
Waypoint(25, 75, false), new Waypoint(75, 25, false), new Waypoint(50,50, false)};
			Aircraft testAircraft = new Aircraft("testAircraft", "Berlin", "Dublin", new Waypoint(100,100, true), new
Waypoint(0,0, true), null, 10.0, waypointList, 1);
			boolean x = testAircraft.outOfBounds();
			assertTrue("Out of bounds = false", x == true);
		}

		// Test set methods
		// Test setAltitudeState
		@Test
		public void testSetAltitudeState(){
			Aircraft testAircraft = generateTestAircraft();
			testAircraft.setAltitudeState(1);
			int altState = testAircraft.altitudeState();
			assertTrue("Altitude State = 1", altState == 1);
		}

}
```

Appendix Item B - Vector Test Class

```
package tst;

import static org.junit.Assert.*;

import org.junit.Test;

import cls.Vector;

public class VectorTest {

		// Test get functions
		// Test getX function
		@Test
		public void testGetX() {
			Vector testVector = new Vector(1.0, 1.1, 1.2);
			assertTrue("x = 1.0", 1.0 == testVector.x());
		}

		// Test getY function
		@Test
		public void testGetY() {
			Vector testVector = new Vector(1.0, 1.1, 1.2);
			assertTrue("y = 1.1", 1.1 == testVector.y());

		}

		// Test getZ function
		@Test
		public void testGetZ() {
			Vector testVector = new Vector(1.0, 1.1, 1.2);
			assertTrue("z = 1.2", 1.2 == testVector.z());
		}

		// Test magnitude function
		@Test
		public void testMagnitude() {
			Vector testVector = new Vector(1.0, 2.0, 2.0);
			assertTrue("Magnitude = 3", 3.0 == testVector.magnitude());
		}
		@Test
		public void testMagnitude2() {
			Vector testVector = new Vector(12, 16, 21);
			assertTrue("Magnitude = 29", 29 == testVector.magnitude());
		}

		// Test magnitudeSquared function
		@Test
		public void testMagnitudeSquared() {
			Vector testVector = new Vector(1.0, 2.0, 2.0);
			assertTrue("Magnitude = 9", 9.0 == testVector.magnitudeSquared());
```

```
	}
	@Test
	public void testMagnitudeSquared2() {
		Vector testVector = new Vector(12, 16, 21);
		assertTrue("Magnitude = 841", 841 == testVector.magnitudeSquared());
	}

	// Test equals function
	@Test
	public void testEquals() {
		Vector testVector = new Vector(1.9, 2.2, 7.4);
		Vector testVector2 = new Vector(1.9, 2.2, 7.4);
		assertTrue("Equals = true", testVector.equals(testVector2));
	}
	@Test
	public void testEquals2() {
		Vector testVector = new Vector(9, 4.2, 5.1);
		Vector testVector2 = new Vector(9.0, 4.2, 5);
		assertTrue("Equals = false", !testVector.equals(testVector2));
	}

	// Test addition function
	@Test
	public void testAddition() {
		Vector testVector = new Vector(2.0, 2.0, 4.0);
		Vector testVector2 = new Vector(1.0, 3.0, 2.0);
		Vector resultVector = testVector.add(testVector2);
		assertTrue("Result =  3.0, 4.0, 6.0", (3.0 == resultVector.x()) && (5.0 == resultVector.y()) && (6.0 ==
resultVector.z()));
	}
	@Test
	public void testAddition2() {
		Vector testVector = new Vector(6.0, 8.1, 16);
		Vector testVector2 = new Vector(1.0, 2.0, 3.0);
		Vector resultVector = testVector.add(testVector2);
		assertTrue("Result =  7.0, 10.1, 19.0", (7.0 == resultVector.x()) && (10.1 == resultVector.y()) && (19.0 ==
resultVector.z()));
	}

	// Test subtraction function
	@Test
	public void testSubtraction() {
		Vector testVector = new Vector(2.0, 3.0, 4.0);
		Vector testVector2 = new Vector(1.0, 1.0, 2.0);
		Vector resultVector = testVector.sub(testVector2);
		assertTrue("Result = 1.0, 2.0, 2.0", (1.0 == resultVector.x()) && (2.0 == resultVector.y()) && (2.0 ==
resultVector.z()));
	}
	@Test
	public void testSubtraction2() {
		Vector testVector = new Vector(14.0, 6, 100);
		Vector testVector2 = new Vector(1.0, 6.0, 0);
		Vector resultVector = testVector.sub(testVector2);
		assertTrue("Result = 13.0, 0, 100.0", (13.0 == resultVector.x()) && (0 == resultVector.y()) && (100.0 ==
resultVector.z()));
	}

	// Test scaleBy function
	@Test
	public void testScaleBy(){
		Vector testVector = new Vector(1, 2, 3);
		Vector resultVector = testVector.scaleBy(1.0);
		assertTrue("ScaledBy = (1 , 2, 3)",  (1 == resultVector.x()) && (2 == resultVector.y()) && (3 ==
resultVector.z()));
	}
```

```
		@Test
		public void testScaleBy2(){
				Vector testVector = new Vector(1, 2, 3);
				Vector resultVector = testVector.scaleBy(-2.0);
				assertTrue("ScaledBy = (-2 , -4, -6)",  (-2 == resultVector.x()) && (-4 == resultVector.y()) && (-6 ==
resultVector.z()));
		}

		// Test normalise function
		@Test
		public void testNormalise() {
				Vector testVector = new Vector(1.0, 2.0, 2.0);
				Vector resultVector = testVector.normalise();
				assertTrue("Normalise = 1/3, 2/3, 2/3",  (1 == (resultVector.x()* 3)) && (2 == (resultVector.y()*3)) && (2 ==
(resultVector.z()*3)));

		}
		@Test
		public void testNormalise2() {
				Vector testVector = new Vector(1, 4, 8);
				Vector resultVector = testVector.normalise();
				assertTrue("Normalise = 1/9, 4/9, 8/9",  (1 == (resultVector.x()*9)) && (4 == (resultVector.y()*9)) && (8 ==
(resultVector.z()*9)));
		}


		// Test angle between function
		@Test
		public void testAngle() {
				Vector testVector = new Vector(1, 0, 0);
				Vector testVector2 = new Vector(0, 1, 0);
				double angle = Math.PI / 2;
				assertTrue("Angle = pi/2", angle  ==  testVector.angleBetween(testVector2));
		}
}
```

Appendix Item C - Waypoint Test Class

```
package tst;

import static org.junit.Assert.*;

import org.junit.Test;

import cls.Waypoint;

import cls.Vector;

public class WaypointTest {

		// Test Get Functions
		// Test get position function
		@Test
		public void testGetPosition() {
				Waypoint testWaypoint = new Waypoint(10,10, false);
				Vector resultVector = testWaypoint.position();
				assertTrue("Position = (10, 10, 0)", (10 == resultVector.x()) && (10 == resultVector.y()) && (0 ==
resultVector.z()));
		}
```

```
        // Test isEntryOrExit function
        @Test
        public void testIsEntryOrExit() {
                Waypoint testWaypoint = new Waypoint(10,10, false);
                assertTrue("Entry/Exit = false", false == testWaypoint.isEntryOrExit());
        }
        @Test
        public void testIsEntryOrExit2() {
                Waypoint testWaypoint = new Waypoint(0, 0, true);
                assertTrue("Entry/Exit = true", true == testWaypoint.isEntryOrExit());
        }

        // Test mouseOver checking
        @Test
        public void testIsMouseOver(){
                Waypoint testWaypoint = new Waypoint(5,5, true);
                assertTrue("Mouse over = true", true == testWaypoint.isMouseOver(10,10));
        }
        @Test
        public void testIsMouseOver2(){
                Waypoint testWaypoint = new Waypoint(25,25, true);
                assertTrue("Mouse over = false", false == testWaypoint.isMouseOver(10,10));
        }

        // Test getCost function
        @Test
        public void testGetCost(){
                Waypoint testWaypoint = new Waypoint(2, 4, false);
                Waypoint testWaypoint2 = new Waypoint(2, 2, true);
                double result = testWaypoint.getCost(testWaypoint2);
                assertTrue("Cost = 2", 2 == result);
        }@Test
        public void testGetCost2(){
                Waypoint testWaypoint = new Waypoint(6, 15, false);
                Waypoint testWaypoint2 = new Waypoint(15, 15, true);
                double result = testWaypoint.getCost(testWaypoint2);
                assertTrue("Cost = 9", 9 == result);
        }

        // Test getCostBetween function
        @Test
        public void testGetCostBetween(){
                Waypoint testWaypoint = new Waypoint(2, 4, false);
                Waypoint testWaypoint2 = new Waypoint(2, 2, true);
                double result = Waypoint.getCostBetween(testWaypoint, testWaypoint2);
                assertTrue("Cost = 2", 2 == result);
        }@Test
        public void testGetCostBetween2(){
                Waypoint testWaypoint = new Waypoint(6, 15, false);
                Waypoint testWaypoint2 = new Waypoint(15, 15, true);
                double result = Waypoint.getCostBetween(testWaypoint, testWaypoint2);
                assertTrue("Cost = 9", 9 == result);
        }


}
```