**System and Unit Tests for Next Generation Air Transportation Collision Avoidance System Software**

**Version 1.0**

**11/08/2015**

Joshua Dixie

Christi Kazakov

Shelley King

William King

Zach Kuhns

Lauren Malone

Adam Miller

Casey Monson

Prepared for

Software Engineering Principles

Instructor: Steve Beaty

Fall 2015

**Table of Contents**

**Overview 3**

**Unit Tests 4**

**Main 4**

**Aircraft 5**

**ADSBInterface 6**

**Collision 8**

**WarningLevelCalculator 11**

**TextCommunication 14**

**PeripheralInterface 16**

**System Tests 16**

**Collision Imminent 16**

**Parallel Vectors 17**

**Two Dimensional Collision with Different Altitude 19**

**Vector Intersection with No Imminent Collision 20**

**Collision Grazed 21**

**Other Aircraft with No Horizontal Movement 29**

**Other Aircraft with No Movement 31**

**Aircraft with Extreme Velocity 33**

**References 34**

**Overview**

1. **Scope**
   1. To ensure that the software identifies and reports potential collisions with absolute accuracy, a test driven development cycle has been chosen. By deciding on data that covers the spectrum of possible collision and noncollision events, we can determine if the final code is behaving as intended.
2. **Purpose**
   1. This document is to be used as a reference for the test data that will be used for the test driven development of the Next Generation Air Traffic Collision Avoidance System.
3. **Test Objectives**
   1. Unit tests are written to make sure each method that adds, updates, returns data, or does calculations are working as intended. Details for each test are located in the appropriate description section.
   2. System tests are written to ensure the software as a whole functions as intended. Collisions are checked for using combinations of two sets of potential aircraft activity. The first set involves 3 collision possibilities. These are: the aircraft’s safe zones will collide, the aircraft’s safe zones won’t collide, and the aircraft’s safe zones will just touch, but not overlap - which will be treated as the boundary case and as a non-collision. The second set will involves aircraft travel potential. This includes variables such as whether aircraft have differing speeds or the same speeds including one not moving and aircraft’s vertical movement possibilities.
   3. Testing will be done via flight simulators to show all possible situations.
4. **Assumptions**
   1. The current TCAS system will still be active and used as needed incase of failure of ADSB or any components of the system.
   2. FAA will have final say in testing and when the product is ready for production.

**Unit Tests**

1. **Main**
   1. The Main test tests the functionality of the Main class. The Main class handles declaring and instantiating the classes that control the aircraft data and calculate collisions. These tests assure that the functions inside main are instantiated any variables before they are used to avoid any runtime errors.

|  |
| --- |
| public class MainTests {  /\* This tests the functionality of the  \* start function in the Main class  \*/  @Test  public void startTest() {  Main main = new Main();  main.initialize();  main.start();    assertNotNull(main.thisAircraft);  assertNotNull(main.otherAircraft);  assertNotNull(main.input);  assertNotNull(main.calculator);  }    /\* This tests the functionality of the  \* reset function in the Main class  \*/  @Test  public void resetTest() {  Main main = new Main();  main.initialize();  main.reset();    assertNull(main.thisAircraft);  assertNull(main.otherAircraft);  assertNull(main.input);  assertNull(main.calculator);  }    /\* This tests the functionality of the  \* initialize function in the Main class  \*/  @Test  public void initializeTest() {  Main main = new Main();  main.initialize();    /\* Both thisAircraft and the otherAircraft  \* array should not be null after the initialize  \* function is called.  \*/  assertNotNull(main.thisAircraft);  assertNotNull(main.otherAircraft);  assertNotNull(main.input);  assertNotNull(main.calculator);  }    /\* This tests the functionality of the  \* getId function in the Main class  \*/  @Test  public void getIdTest() {  Main main = new Main();  main.initialize();    /\* The getId() function should never return a null  \* string id.  \*/  assertNotNull(main.getId());  }  } |

1. **Aicraft**
   1. The Aircraft tests verify the constructor and get methods. When the tests pass, we will know that the public constructor takes four parameters - a String, two double arrays, and an int. The value returned by getId is the first parameter. The value returned by getLocation is the second parameter. The value returned by getHeading is the third, and the value returned by getWarningLevel is the fourth.

|  |
| --- |
| public class AircraftTest {  @Test  public void testConstructor() {  String id = "A123";  double[] location = {0.0,0.0,0.0};  double[] heading = {0.0,0.0,0.0};  int warningLevel = 0;  Aircraft a = new Aircraft(id, location, heading, warningLevel);  assertTrue(id.equals(a.getId()));  assertTrue(Arrays.equals(a.getLocation(), location));  assertTrue(Arrays.equals(a.getHeading(), heading));  assertTrue(a.getWarningLevel() == warningLevel);  }  } |

1. **ADSBInterface**
   1. The ADSBInterface tests are designed to ensure that aircraft are added to the ArrayList if they do not exist, or update the correct aircraft if it does exist in the list.

|  |
| --- |
| public class ADSBInterfaceTest extends TestCase {  ArrayList<Aircraft> testNearList = new ArrayList<Aircraft>();  ADSBInterface testClass;  Random rng;  public static void main(String[] args) {  ADSBInterfaceTest it = new ADSBInterfaceTest();  }  public ADSBInterfaceTest() {  rng = new Random();  double[] l = new double[4];  double[] h = new double[4];  for (int i = 0; i < 40; i++) {  for(int j = 0; j < 4; j++) {  l[j] = 0;  h[j] = 0;  }  String id;  if(i < 10)  id = "TST100" + i;  else  id = "TST10" + i;  testNearList.add(new Aircraft(id, l, h));  }  testClass = new ADSBInterface(null);  try {  testAddAircraft();  } catch(Exception e) {  System.out.println("Error occurred: " + e.getMessage());  }  }  public void testAddAircraft() throws Exception {  int result;  //test the addition of 1 aircraft  result = testClass.addAircraft(testNearList.get(0));  System.out.println("Aircraft " + testNearList.get(0).getId() + " added. Result: " + determineResult(result));  //test the addition of multiple aircraft <= 20  for(int i = 1; i < 20; i++) {  result = testClass.addAircraft(testNearList.get(i));  System.out.println("Aircraft " + testNearList.get(i).getId() + " added. Result: " + determineResult(result));  }  //test the updating of aircraft  for(int i = 0; i < 20; i++) {  result = testClass.addAircraft(testNearList.get(i));  System.out.println("Aircraft " + testNearList.get(i).getId() + " added. Result: " + determineResult(result));  }  //test the addition of > 20 aircraft. This one //should always error out.  for(int i = 20; i < 40; i++) {  result = testClass.addAircraft(testNearList.get(i));  System.out.println("Aircraft " + testNearList.get(i).getId() + " added. Result: " + determineResult(result));  }  }  public String determineResult(int result) {  String report;  switch(result) {  case 1:  report = "passed as new add.";  break;  case 2:  report = "passed as updated aircraft.";  break;  case 0:  default:  report = "failed.";  break;  }  return report;  }  } |

1. **Collision**
   1. The CollisionTest class tests that two Collision objects can be accurately compared and ranked, so they can be used in a priority queue. It also tests that defensive copies are being returned when necessary, and that the predicted location of a collision matches what we expect based on the headings and locations of the planes.

|  |
| --- |
| // CollsionTest.java  import java.util.Calendar;  import java.util.Date;  import org.junit.Test;  import static org.junit.Assert.\*;  /\*\*  \* Test class for the Collision class. Simple accessor methods like  \* getOwnshipId() and getOthershipId() were not tested, and the constructor was  \* also not tested.  \*/  public class CollisionTest {  private final Collision testColl1;  private final Collision testColl2;  private static final int MINUTE\_OFFSET1 = 50;  private static final int MINUTE\_OFFSET2 = 10;  private static final double[] ALL\_ZEROS = {0.0,0.0,0.0};  private static final Aircraft SELF1 =  new Aircraft("thisone", ALL\_ZEROS, ALL\_ZEROS, 0);  private static final Aircraft OTHER1 =  new Aircraft("otherone", ALL\_ZEROS, ALL\_ZEROS, 0);  private static final Aircraft SELF2 =  new Aircraft("thistwo", ALL\_ZEROS, ALL\_ZEROS, 0);  private static final Aircraft OTHER2 =  new Aircraft("othertwo", ALL\_ZEROS, ALL\_ZEROS, 0);  /\*\* xyz velocities \*/  private static final double[] HEADING1 = {0.0, 550.0, 0.0};  /\*\* latitude, longitude, and altitude \*/  private static final double[] LOCATION1 = {0.0, 0.0, 1000.0};  /\*\* xyz velocities \*/  private static final double[] HEADING2 = {-550.0, 0.0, 0.0};  /\*\* latitude, longitude, and altitude \*/  private static final double[] LOCATION2 = {550.0, 550.0, 1000.0};  private static final Aircraft DOOMED\_PLANE1 =  new Aircraft("dp1", HEADING1, LOCATION1, 0);  private static final Aircraft DOOMED\_PLANE2 =  new Aircraft("dp2", HEADING2, LOCATION2, 0);  /\*\*  \* Set up the test objects  \*/  public CollisionTest() {  Calendar cal1 = Calendar.getInstance();  Calendar cal2 = Calendar.getInstance();  cal1.add(Calendar.MINUTE, MINUTE\_OFFSET1);  cal2.add(Calendar.MINUTE, MINUTE\_OFFSET2);  testColl1 = new Collision(SELF1, OTHER1, cal1.getTime());  testColl2 = new Collision(SELF2, OTHER2, cal2.getTime());  }  /\*\*  \* Test of compareTo method, of class Collision.  \*/  @Test  public void testCompareTo() {  System.out.println("compareTo");  int result = testColl1.compareTo(testColl2);  assertTrue(result > 0);  }  /\*\*  \* Test of getTime method, of class Collision.  \* Make sure it is making a new defensive copy each time it is called.  \*/  @Test  public void testGetTime() {  System.out.println("getTime");  Date testDate1 = testColl1.getTime();  Date testDate2 = testColl1.getTime();  assertTrue(testDate1 != testDate2);  }  /\*\*  \* Test of getLocation method, of class Collision.  \*/  @Test  public void testGetLocation() {  System.out.println("getLocation");  double expLatitude = 550.0;  double expLongitude = 550.0;  double expAltitude = 1000.0;  WarningLevelCalculator wlc = new WarningLevelCalculator();  double[] result =  wlc.detectCollision(DOOMED\_PLANE1, DOOMED\_PLANE2).getLocation();  assertEquals(expLatitude, result[0], Math.ulp(expLatitude));  assertEquals(expLongitude, result[1], Math.ulp(expLongitude));  assertEquals(expAltitude, result[2], Math.ulp(expAltitude));  }  /\*\*  \* Test of getRelativeAltitude method, of class Collision.  \*/  @Test  public void testGetRelativeAltitude() {  System.out.println("getRelativeAltitude");  double expResult = 0.0;  double result = testColl1.getRelativeAltitude();  assertEquals(expResult, result, Math.ulp(result));  }  } |

1. **WarningLevelCalculator**
   1. These tests are to assure the mathematics behind calculating the possible collisions are functioning correctly. There are many unit conversions that must be accounted for to assure the collisions data and subsequent instructions are correctly calculated.

|  |
| --- |
| public class WarningLevelCalculatorTest {  public WarningLevelCalculatorTest() {  }    @BeforeClass  public static void setUpClass() {  }    @AfterClass  public static void tearDownClass() {  }    @Before  public void setUp() {  }    @After  public void tearDown() {  }  /\*\*  \* Test of parseList method, of class \* WarningLevelCalculator.  \*/  @Test  public void testParseList() {  System.out.println("parseList");  WarningLevelCalculator instance = new WarningLevelCalculator();  instance.parseList();  fail("The test case is a prototype.");  }  /\*\*  \* Test of detectCollision method, of class \*  **\*** WarningLevelCalculator.  \*/  @Test  public void testDetectCollision() {  System.out.println("detectCollision");  Aircraft one = null;  Aircraft two = null;  WarningLevelCalculator instance = new WarningLevelCalculator();  Collision expResult = null;  Collision result = instance.detectCollision(one, two);  assertEquals(expResult, result);  fail("The test case is a prototype.");  }  /\*\*  \* Test of testCollisionPoint method, of class \* WarningLevelCalculator.  \*/  @Test  public void testTestCollisionPoint() {  System.out.println("testCollisionPoint");  Collision collision = null;  WarningLevelCalculator instance = new WarningLevelCalculator();  boolean expResult = false;  boolean result = instance.testCollisionPoint(collision);  assertEquals(expResult, result);  fail("The test case is a prototype.");  }  /\*\*  \* Test of setWarningLevelYellow method, of class \* WarningLevelCalculator.  \*/  @Test  public void testSetWarningLevelYellow() {  System.out.println("setWarningLevelYellow");  Aircraft plane = null;  WarningLevelCalculator instance = new WarningLevelCalculator();  instance.setWarningLevelYellow(plane);  fail("The test case is a prototype.");  }  /\*\*  \* Test of setWarningLevelOrange method, of class \* WarningLevelCalculator.  \*/  @Test  public void testSetWarningLevelOrange() {  System.out.println("setWarningLevelOrange");  Aircraft plane = null;  WarningLevelCalculator instance = new WarningLevelCalculator();  instance.setWarningLevelOrange(plane);  fail("The test case is a prototype.");  }  /\*\*  \* Test of setWarningLevelRed method, of class \*  **\*** WarningLevelCalculator.  \*/  @Test  public void testSetWarningLevelRed() {  System.out.println("setWarningLevelRed");  Aircraft plane = null;  WarningLevelCalculator instance = new WarningLevelCalculator();  instance.setWarningLevelRed(plane);  fail("The test case is a prototype.");  }  } |

1. **TextCommunication**
   1. These tests handle the TextCommunication class that handles translating the test information for the aircraft pilot and tower operators. These tests assure the the text is stored correctly and that text can be received and sent.

|  |
| --- |
| public class TextCommunicationIT {  public TextCommunicationIT() {  TextCommunication instance = new TextCommunication();  }  @BeforeClass  public static void setUpClass() {  }  @AfterClass  public static void tearDownClass() {  }  @Before  public void setUp() {  }  @After  public void tearDown() {  }  /\*\*  \* Test of listener method, of class TextCommunication.  \*/  @Test  public void testListener() {  System.out.println("listener");  TextCommunication instance = new TextCommunication();  instance.listener();  //fail("The test case is a prototype.");  }  /\*\*  \* Test of receiver method, of class TextCommunication.  \*/  @Test  public void testReceiver() {  System.out.println("receiver");  Object data = new Object();  TextCommunication instance = new TextCommunication();  String expResult = data.toString();  String result = instance.receiver(data);  assertEquals(expResult, result);  //fail("The test case is a prototype.");  }  /\*\*  \* Test of send method, of class TextCommunication.  \*/  @Test  public void testSend() {  System.out.println("send");  String message = " ";  String message2 = "Warning Level 1";  String message3 = "MayDay MayDay";  TextCommunication instance = new TextCommunication();  instance.send(message);  instance.send(message2);  instance.send(message3);  //fail("The test case did not go through");  }  /\*\*  \* Test of log method, of class TextCommunication.  \*/  @Test  public void testLog() {  Date date = new Date();  System.out.println("log");  String log = " ";  String logWarning = "Warning level 1 ";  TextCommunication instance = new TextCommunication();  instance.log(date.toString() + " : " + logWarning);  //fail("The test case is a prototype.");  }  } |

1. **PeripheralInterface**
   1. PeripheralInterface handles the interaction of the collision software with the audio and visual devices. As such, no specific test cases are deemed necessary.

|  |
| --- |
|  |

**System Tests**

1. **Collision Imminent**
   1. This tests ensures that a definite collision will produce the proper warning for the aircraft given.

|  |
| --- |
| import java.util.ArrayList;  import org.junit.Test;  public class DefiniteCollisionTest {  @Test  public void definiteCollisionTest() {  double[] vel1 = new double[] { 500, 0, 0};  double[] vel2 = new double[] { -500, 0, 0};  double [] c1Coords = new double [] {200, 0,  double [] c2Coords = new double[] {400, 0, 30000};  try {  runTest(c1Coords, vel1, c2Coords, vel2);  } catch(Exception e) {  System.out.println("Error: " + e.getMessage());  }  }    public void runTest(double[] craft1Coords, double[]  craft1Heading, double[] craft2Coords,  double[] craft2Heading) throws Exception {  ADSBInterface testADSB;  WarningLevelCalculator testWLC;  ArrayList<Aircraft> testNearList = new  ArrayList<Aircraft>();  //create aircraft objects  Aircraft us = new Aircraft("A1", craft1Coords,  craft1Heading, 0);  Aircraft them = new Aircraft("A2", craft2Coords,  craft2Heading, 0);  //initiate WarningLevelCalculator  testWLC = new WarningLevelCalculator(us,,  testNearList);  //first ADSB picks up aircraft data for us and them.  //ADSB.listen() will detect the aircraft and call  //addAircraft  testADSB = new ADSBInterface(testNearList);  testADSB.addAircraft(them);  //WarningLevelCalculator then checks for collisions  // and calls other relevant class methods.  testWLC.parseList();  }  } |

1. **Parallel Vectors**
   1. This test case compares to planes flying in parallel to make sure there are no problems with the math when no collision is possible. The collision calculation should result in a zero only from a parallel case making this non collision case unique from the other non collision cases.

|  |
| --- |
| //same altitudes parallel vectors opposite directions  public class ParallelTest {  @Test  public void testDetectCollision() {  System.out.println("detectCollision");  Aircraft one = new Aircraft("test1",  new double[] {10.0, 20.0, 5000.0},  new double[] {0.0, 400.0, 0.0}, 1);  Aircraft two = new Aircraft("test1",  new double[] {11.0, 21.0, 5000.0},  new double[] {0.0, -400.0, 0.0}, 1);  WarningLevelCalculator instance =  new WarningLevelCalculator();  Collision expResult = null;  Collision result = instance.detectCollision(one,  two);  assertEquals(expResult, result);  }  //different altitudes parallel vectors different opposite // direction  @Test  public void testDetectCollision() {  System.out.println("detectCollision");  Aircraft one = new Aircraft("test1",  new double[] {5.0, 10.0, 6000.0},  new double[] {0.0, 400.0, 0.0}, 1);  Aircraft two = new Aircraft("test1",  new double[] {7.0, 8.0, 10000.0},  new double[] {0.0, -400.0, 0.0}, 1);  WarningLevelCalculator instance =  new WarningLevelCalculator();  Collision expResult = null;  Collision result = instance.detectCollision(one,  two);  assertEquals(expResult, result);  }  //Same altitudes parallel vectors same direction  @Test  public void testDetectCollision() {  System.out.println("detectCollision");  Aircraft one = new Aircraft("test1",  new double[] {10.0, 20.0, 5000.0},  new double[] {0.0, 400.0, 0.0}, 1);  Aircraft two = new Aircraft("test1",  new double[] {11.0, 21.0, 5000.0},  new double[] {0.0, 400.0, 0.0}, 1);  WarningLevelCalculator instance =  new WarningLevelCalculator();  Collision expResult = null;  Collision result = instance.detectCollision(one,  two);  assertEquals(expResult, result);  }  //Different altitudes parallel vectors same direction  @Test  public void testDetectCollision() {  System.out.println("detectCollision");  Aircraft one = new Aircraft("test1",  new double[] {5.0, 10.0, 6000.0},  new double[] {0.0, 400.0, 0.0}, 1);  Aircraft two = new Aircraft("test1",  new double[] {7.0, 8.0, 10000.0},  new double[] {0.0, 400.0, 0.0}, 1);  WarningLevelCalculator instance =  new WarningLevelCalculator();  Collision expResult = null;  Collision result = instance.detectCollision(one,  two);  assertEquals(expResult, result);  }  } |

1. **Two dimensional collision with different altitude**
   1. This test case tests how the program will handle planes that will intersect laterally but not vertically. This should not generate any warning if the plane’s altitudes are far enough apart.

|  |
| --- |
| public class testPlanesPassSamePoint {  /\*\*  \* Default constructor for test class  \* testPlanesPassSamePoint  \*/  public testPlanesPassSamePoint() {  }  /\*\*  \* Test two planes in various directions of vector  \*/  @Test  public void testSameAltitude(){  ADSBInterface ADSB1 = new ADSBInterface({});  Aircraft plane1A = new  Aircraft("1A",{10.0,10.0,20.0},{0.0,0.0,-5.0});  Aircraft plane1B = new  Aircraft("1B",{0.0,0.0,10.0},{10.0,10.0,10.0});  ADSB1.addAircraft(plane1A);  ADSB1.addAircraft(plane1B);    // The design said in, so if it is called with this,  // it should put out the correct warning level  <Name of Our Program>(ADSB1);  }    /\*\*  \* Test two planes, one climbing through the path of the  \* other  \*/  @Test  public void testSameAltitude(){  ADSBInterface ADSB2 = new ADSBInterface({});  Aircraft plane2A = new  Aircraft("2A",{0.0,0.0,1000.0},{20.0,0.0,0.0});  Aircraft plane2B = new  Aircraft("2B",{0.0,0.0,500.0},{15.0,0,250.0});  ADSB2.addAircraft(plane2A);  ADSB2.addAircraft(plane2B);    // The design said in, so if it is called with this,  // it should put out the correct warning level  <Name of Our Program>(ADSB2);  }    /\*\*  \* Test two planes, on a level plane  \*/  @Test  public void testSameAltitude(){  ADSBInterface ADSB3 = new ADSBInterface({});  Aircraft plane3A = new  Aircraft("3A",{0.0,0.0,1000.0},{20.0,-5.0,0.0});  Aircraft plane3B = new  Aircraft("3B",{0.0,0.0,1000.0},{0.0,-10.0,0.0});  ADSB3.addAircraft(plane3A);  ADSB3.addAircraft(plane3B);    // The design said in, so if it is called with this,  // it should put out the correct warning level  <Name of Our Program>(ADSB3);  }  } |

1. **Vector intersection with no imminent collision**
   1. AltitudeNonCollisionTest makes sure a collision is not wrongly predicted when a craft will pass over another craft with adequate vertical distance separating them.

|  |
| --- |
| import org.junit.Test;  import static org.junit.Assert.\*;  /\*\*  \* Test class to make sure a collision is not wrongly predicted  \* when a craft will pass over another craft with adequate  \* vertical distance separating them.  \*/  public class AltitudeNonCollisionTest {  /\*\* xyz velocities \*/  private static final double[] HEADING1 = {0.0, 550.0,  0.0};  /\*\* latitude, longitude, and altitude \*/  private static final double[] LOCATION1 = {0.0, 0.0,  1000.0};  /\*\* xyz velocities \*/  private static final double[] HEADING2 = {-550.0, 0.0,  0.0};  /\*\* latitude, longitude, and altitude \*/  private static final double[] LOCATION2 = {550.0, 550.0,  2000.0};  /\*\* latitude, longitude, and altitude \*/  private static final double[] LOCATION3 = {550.0, 550.0,  1000.0};  /\*\* Value for a warning level corresponding to no warning  \*/  private static final int WARNING\_LEVEL\_GREEN = 0;  /\*\* Ownship test object \*/  private final Aircraft thisPlane = new  Aircraft("thisPlane", HEADING1,  LOCATION1, WARNING\_LEVEL\_GREEN);  /\*\* Other aircraft test object \*/  private final Aircraft otherPlaneA = new  Aircraft("otherPlane", HEADING2,  LOCATION2, WARNING\_LEVEL\_GREEN);  /\*\* Another aircraft test object \*/  private final Aircraft otherPlaneB = new  Aircraft("otherPlane", HEADING2,  LOCATION3, WARNING\_LEVEL\_GREEN);  /\*\*  \* Test that no collision is predicted when one plane will  \* pass over another with adequate vertical distance  \*/  @Test  public void testAltitudeSeparation() {  System.out.println("testAltitudeSeparation");  Collision expResult = null; // No collision  Collision result;  WarningLevelCalculator wlc = new  WarningLevelCalculator();  result = wlc.detectCollision(thisPlane,  otherPlaneA);  assertEquals(expResult, result);  }  /\*\*  \* Test that a collision is predicted when planes have no  \* vertical distance between them and are on a clear  \* collision vector with each other  \*/  @Test  public void testAltitudeMatch() {  System.out.println("testAltitudeMatch");  Collision result;  WarningLevelCalculator wlc = new  WarningLevelCalculator();  result = wlc.detectCollision(thisPlane,  otherPlaneB);  assertTrue(result != null);  }  } |

1. **Grazing Collision**
   1. This system test will check to ensure that aircraft that have their safe zones just come into contact but not overlap are read as a non-collision. The five potential cases for this are: one aircraft moving faster than the other, both aircraft moving at the same speed, and one aircraft not moving, the aircraft crossing the same latitude and longitude, but with exactly 900 feet of altitude difference, and the same test again, but with one stationary aircraft. This testing takes advantage of a program written to compute starting points of two aircraft given the point of first contact collision, velocities/headings of those craft, and the time desired between the starting points and collision.

|  |
| --- |
| public class GrazeTest extends TestCase {  ArrayList<Aircraft> testNearList = new  ArrayList<Aircraft>();  ADSBInterface testClass;  Random rng;  public static void main(String[] args) {  GrazeTest it = new GrazeTest();  }  public GrazeTest() {  Haversine haversine = new Haversine();  //create starting coordinates for grazes at  // specified collision points given times and  // headings degrees, feet, seconds, and ft/sec used  //near miss where one aircraft is moving faster than  //the other  double[] collisionPoint = new double[] {39.0,  -105.5, 35000};  double[] vel1 = new double[] { 513.33, 0, 0};  double[] vel2 = new double[] { 0, 366.66, 0};  double[] hResults1 =  haversine.calcStartPoints(collisionPoint,  480, vel1, vel2);  System.out.println("Start points lat, long, alt " +  " for 1 faster aircraft:");  System.out.println("Craft1 : " + hResults1[0] +  ", " + hResults1[1] + ", " + hResults1[2]);  System.out.println("Craft2 : " + hResults1[3] +  ", " + hResults1[4] + ", " + hResults1[5] +  "\n");  try {  double[] c1Coords = new double[] {hResults1[0],  hResults1[1], hResults1[2]};  double[] c2Coords = new double[] {hResults1[3],  hResults1[4], hResults1[5]};  runTest(c1Coords, vel1, c2Coords, vel2);  } catch(Exception e) {  System.out.println("Error: " + e.getMessage());  }  //near miss where both aircraft are moving equally  // fast  double[] collisionPoint2 = new double[] {39.0,  -105.5, 35000};  double[] vel1\_2 = new double[] {405.5, 0, 0};  double[] vel2\_2 = new double[] { 0, 405.5, 0};  double[] hResults2 =  haversine.calcStartPoints(collisionPoint2,  480, vel1\_2, vel2\_2);  System.out.println("Start points lat, long, alt " +  "for equal speed aircraft:");  System.out.println("Craft1 : " + hResults2[0] +  ", " + hResults2[1] + ", " + hResults2[2]);  System.out.println("Craft2 : " + hResults2[3] +  ", " + hResults2[4] + ", " + hResults2[5] +  "\n");  try {  double[] c1Coords = new double[] {hResults2[0],  hResults2[1], hResults2[2]};  double[] c2Coords = new double[] {hResults2[3],  hResults2[4], hResults2[5]};  runTest(c1Coords, vel1\_2, c2Coords, vel2\_2);  } catch(Exception e) {  System.out.println("Error: " + e.getMessage());  }  //near miss where one aircraft is not moving  double[] collisionPoint3 = new double[] {39.0,  -105.5, 35000};  double[] vel1\_3 = new double[] { 513.33, 0, 0};  double[] vel2\_3 = new double[] { 0, 0, 0};  double[] hResults3 =  haversine.calcStartPoints(collisionPoint3,  480, vel1\_3, vel2\_3);  System.out.println("Start points lat, long, alt " +  "for one non-moving craft:");  System.out.println("Craft1 : " + hResults3[0] +  ", " + hResults3[1] + ", " + hResults3[2]);  System.out.println("Craft2 : " +collisionPoint3[0] +  ", " + collisionPoint3[1] + ", " +  collisionPoint3[2] + "\n");  try {  double[] c1Coords = new double[] {hResults3[0],  hResults3[1], hResults3[2]};  double[] c2Coords = new double[] {hResults3[3],  hResults3[4], hResults3[5]};  runTest(c1Coords, vel1\_3, c2Coords, vel2\_3);  } catch(Exception e) {  System.out.println("Error: " + e.getMessage());  }  // near miss where both aircraft are moving equally  // fast but with a 900 ft altitude difference and  // crossing the same point at the same time  double[] overlapCenterLL1 = new double[] {39.0,  -105.5, 35000};  double[] overlapCenterLL2 = new double[] {39.0,  -105.5, 34100};  double[] vel1\_4 = new double[] { 405.5, 0, 0};  double[] vel2\_4 = new double[] { 0, 405.5, 0};  double[] c1Start =  haversine.calcSingleStartPoint(  overlapCenterLL1, 480, vel1\_4);  double[] c2Start =  haversine.calcSingleStartPoint(  overlapCenterLL2, 480, vel2\_4);  System.out.println("Start points lat, long, alt" +  " for equal speed aircraft with 900 ft " +  "altitude difference:");  System.out.println("Craft1 : " + c1Start[0] + ", " +  c1Start[1] + ", " + c1Start[2]);  System.out.println("Craft2 : " + c2Start[0] + ", " +  c2Start[1] + ", " + c2Start[2] + "\n");  try {  double[] c1Coords = new double[] {c1Start[0],  c1Start[1], c1Start[2]};  double[] c2Coords = new double[] {c2Start[0],  c2Start[1], c2Start[2]};  runTest(c1Coords, vel1\_4, c2Coords, vel2\_4);  } catch(Exception e) {  System.out.println("Error: " + e.getMessage());  }  //near miss where one aircraft is not moving but  // with a 900 ft altitude difference and crossing the  // same point at the same time  double[] overlapCenterLL1\_2 = new double[] {39.0,  -105.5, 35000};  double[] overlapCenterLL2\_2 = new double[] {39.0,  -105.5, 34100};  double[] vel1\_5 = new double[] { 405.5, 0, 0};  double[] vel2\_5 = new double[] { 0, 0, 0};  double[] c1Start\_2 =  haversine.calcSingleStartPoint(  overlapCenterLL1\_2, 480, vel1\_5);  System.out.println("Start points lat, long, alt" +  " for 1 non moving aircraft with 900ft " +  "altitude difference:");  System.out.println("Craft1 : " + c1Start\_2[0] +  ", " + c1Start\_2[1] + ", " + c1Start\_2[2]);  System.out.println("Craft2 : " +  overlapCenterLL2\_2[0] +  ", " + overlapCenterLL2\_2[1] + ",  " + overlapCenterLL2\_2[2] + "\n");  try {  double[] c1Coords = new double[] {c1Start[0],  c1Start[1], c1Start[2]};  double[] c2Coords = new double[]  {overlapCenterLL2\_2[0],  overlapCenterLL2\_2[1],  overlapCenterLL2\_2[2]};  runTest(c1Coords, vel1\_5, c2Coords, vel2\_5);  } catch(Exception e) {  System.out.println("Error: " + e.getMessage());  }  }  public void runTest(double[] craft1Coords, double[]  craft1Heading, double[] craft2Coords,  double[] craft2Heading) throws Exception {  ADSBInterface testADSB;  WarningLevelCalculator testWLC;  ArrayList<Aircraft> testNearList = new  ArrayList<Aircraft>();  //create aircraft objects  Aircraft us = new Aircraft("TST0001", craft1Coords,  craft1Heading, 0);  Aircraft them = new Aircraft("TST0002",  craft2Coords, craft2Heading, 0);  //initiate WarningLevelCalculator  testWLC = new WarningLevelCalculator(us,  testNearList);  //first ADSB picks up aircraft data for us and them.  //ADSB.listen() will detect the aircraft and call  //addAircraft  testADSB = new ADSBInterface(testNearList);  testADSB.addAircraft(them);  //WarningLevelCalculator then checks for collisions  // and calls other relevant class methods.  testWLC.parseList();  }  } |

1. **Other aircraft with no horizontal movement**
   1. This tests an aircraft with no horizontal movement with vertical movement like a helicopter slowly taking off. In this scenario a helicopter is taking off directly in Boulder and a jet airliner is heading from Denver to this helicopter’s lateral position at 40000 feet. This shoulder trigger a warning level 3 because the distance between these planes is within the 50 mile warning level 2 radius.

|  |
| --- |
| public class FullTest {  @Test  public void fullTest() {  Main main = new Main();  main.initialize();    assertNotNull(main.thisAircraft);  assertNotNull(main.otherAircraft);  assertNotNull(main.input);  assertNotNull(main.calculator);    // A plane directly over Denver  double[] location\_one = {39.7392, 104.9903, 40000};  double[] heading\_one = {-480.10, 435.88, 0};    main.thisAircraft = new Aircraft("ONE",  location\_one, heading\_one, 1);    assertEquals(main.thisAircraft.getId(), "ONE");  assertEquals(main.thisAircraft.getLocation(),  location\_one);  assertEquals(main.thisAircraft.getHeading(),  heading\_one);    // A helicopter directly over Boulder  double[] location\_two = {40.0274, 105.2519, 30000};  double[] heading\_two = {0, 0, 44.82};    main.input.addAircraft(new Aircraft("TWO",  location\_two, heading\_two, 1));    assertEquals(main.thisAircraft.getId(), "TWO");  assertEquals(main.thisAircraft.getLocation(),  location\_two);  assertEquals(main.thisAircraft.getHeading(),  heading\_two);    Collision testCollision =  main.calculator.detectCollision(  main.thisAircraft,  main.otherAircraft[0]);    // 548.049 miles per hour average jet liner velocity  // The collision point is 24.281 miles away from  // thisAircraft  // 19.924 x change 18.089 y change 1.86 z change  // Collision in 2.49 minutes  assertNotNull(testCollision);  assertEquals(testCollision, new  Collision(main.thisAircraft.getId(),  main.otherAircraft[0].getId(), new  Date(System.currentTimeMillis() + 149400)));    // The collision point is 24.281 miles away from  // thisAircraft  assertTrue(  main.calculator.testCollisionPoint(  testCollision));    // WarningLevel should be 2  assertEquals(main.thisAircraft.getWarningLevel(),  2);  assertEquals(  main.otherAircraft[0].getWarningLevel(), 2);  }  } |

1. **Other aircraft with no movement**
   1. This test will check to make sure that if an object that is still whill show up, such as a helicopter, tethered weather balloon which is hovering is shown and stationary objects like mountains, communication towers, and very tall buildings. The adsb will communicate the locations and the heights of the objects that are stationary while objects that are aircraft the transponder and adsb will report the position of the aircraft.

|  |
| --- |
| public class stationary {  @Test  public void stationary() {  Main main = new Main();  main.initialize();    assertNotNull(main.thisAircraft);  assertNotNull(main.otherAircraft);  assertNotNull(main.input);  assertNotNull(main.calculator);    // A plane over Denver  double[] location\_one = {39.7392, 104.9903, 15120};  double[] heading\_one = {-480.10, 435.88, 0};    main.thisAircraft = new Aircraft("ONE",  location\_one, heading\_one, 1);    assertEquals(main.thisAircraft.getId(), "ONE");  assertEquals(main.thisAircraft.getLocation(),  location\_one);  assertEquals(main.thisAircraft.getHeading(),  heading\_one);    // A helicopter not moving  double[] location\_two = {40.0274, 105.2519, 14440 };  double[] heading\_two = {0, 0, 0};    main.input.addAircraft(new Aircraft("TWO",  location\_two, heading\_two, 1));    assertEquals(main.thisAircraft.getId(), "TWO");  assertEquals(main.thisAircraft.getLocation(),  location\_two);  assertEquals(main.thisAircraft.getHeading(),  heading\_two);    Collision testCollision =  main.calculator.detectCollision(  main.thisAircraft, main.otherAircraft[0]);  // The collision point is 24.281 miles away from // thisAircraft  // 19.924 x change 18.089 y change 1.86 z change  // Collision in 2.49 minutes  assertNotNull(testCollision);  assertEquals(testCollision, new  Collision(main.thisAircraft.getId(),  main.otherAircraft[0].getId(), new  Date(System.currentTimeMillis() + 149400)));    // The collision point is 24.281 miles away from  // thisAircraft  assertTrue(  main.calculator.testCollisionPoint(  testCollision));    // WarningLevel should be 2  assertEquals(main.thisAircraft.getWarningLevel(),  2);  assertEquals(  main.otherAircraft[0].getWarningLevel(), 2);  assertTrue(  main.calculator.testCollisionPoint(  testCollision));  }  } |

1. **Aircraft with extreme velocity**
   1. This test case handles is a plane enters an aircraft’s airspace at an extreme velocity that would cause hardware errors with large data that results from unknown aircraft or data transfer errors.

|  |
| --- |
| public class FastClass {  /\*\* xyz velocities \*/  private static final double[] HEADING1 = {0.0, 1200.0,  0.0};  /\*\* latitude, longitude, and altitude \*/  private static final double[] LOCATION1 = {0.0, 0.0,  1000.0};  /\*\* xyz velocities \*/  private static final double[] HEADING2 = {-1200.0, 0.0,  0.0};  /\*\* latitude, longitude, and altitude \*/  private static final double[] LOCATION2 = {550.0, 550.0,  2000.0};  /\*\* xyz velocities \*/  private static final double[] HEADING3 = {900.0, 900.0,  900.0};  /\*\* latitude, longitude, and altitude \*/  private static final double[] LOCATION3 = {550.0, 550.0,  1000.0};  /\*\* Value for a warning level corresponding to no warning \*/  private static final int WARNING\_LEVEL\_GREEN = 0;  /\*\* Ownship test object \*/  private final Aircraft thisPlane = new  Aircraft("thisPlane", HEADING1,  LOCATION1, WARNING\_LEVEL\_GREEN);  /\*\* Other aircraft test object \*/  private final Aircraft otherPlaneA = new  Aircraft("otherPlane", HEADING2,  LOCATION2, WARNING\_LEVEL\_GREEN);  /\*\* Another aircraft test object \*/  private final Aircraft otherPlaneB = new  Aircraft("otherPlane", HEADING3,  LOCATION3, WARNING\_LEVEL\_GREEN);  }    @Test  public void testVelocity(){  System.out.println("Testing Fast Velocity");  Collision expResult = null;  Collision result;  WarningLevelCalculator warning = new  WarningLevelCalculator();  result = warning.detectCollision(thisPlane, otherPlaneA);  assertEquals(expResult, result);  System.out.println(expResult, result);  Collision result2;  result2 = warning.detectCollision(otherPlaneA,  otherPlaneB);  assertEquals(expResult, result2);  } |

**References**

1. <http://adsb-decode-guide.readthedocs.org/en/latest/introduction.html>
2. <http://www.movable-type.co.uk/scripts/latlong.html>