

Testing Log

Close to testing log:

Input	Expected Output	Actual output	Pass/fail	Reason
(0.0, 0.0), (0.0, 0.0)	True	True	Pass	
(0.0, 0.0) (0.0, 0.00015)	False	False	Pass	
(0.0, 0.0) (0.00015, 0.0)	False	False	Pass	
(0.0, 0.0) (0.0, 0.00014)	True	True	Pass	
(0.0, 0.0) (0.00014, 0.0)	True	True	Pass	
(0.0, 0.0) (0.00007, 0.00007)	True	True	Pass	
(0.0, 0.0) (12, 12)	False	False	Pass	
(-3.186874, 55.944494), (-3.186767933982822, 55.94460006601717)	True	True	Pass	
(-3.186874, 55.944494), (-3.1873, 55.9430)	False	False	Pass	
(-1, -1), (-1, -1.00014)	True	True	Pass	
(0,0)	Error	Unhandled error	Fail	Had not implemented error handling for this function

Delivery cost testing:

Input	Expected Output	Actual output	Pass/fail	Reason
Input 1 in test case doc	1890	1840	fail	Logic error, missing 50p delivery charge
Input 2	1570	1520	fail	As above
Input 3	1170	1120	fail	As above

After fixing logical error:

Input	Expected Output	Actual output	Pass/fail	Reason
Input 1 in test case doc	1890	1890	pass	
Input 2	1570	1570	pass	
Input 3	1170	1170	pass	

IsConfined testing:

Test	Input	Expected Output	Actual Output	Pass/Fail	Reason
IsConfined	appletonTower	True	True	Pass	
IsConfined	businessSchool	True	True	Pass	
isConfined	greyfriarsKirkyard	False	False	Pass	

Runtimes for whole system:

(Each number is an average of 5 runtimes)

117 seconds
94 seconds
102 seconds
85 seconds
92 seconds
78 seconds

Taking the average of these gives us 95.5 seconds runtime, this is over what was specified in the requirement however I decided that this requirement was secondary to functionality and so this isn't the end of the world.

During my testing process I didn't have time to maintain a complete testing log for all test cases, however my complete test suite with at least one example of each test for a given input is shown below, along with the results of running the entire suite

System can test whether a coordinate is within the confinement area:

```

Eoin Reid
@Test
public void testIsConfinedTrueA() { assertTrue(appletonTower.isConfined()); }

Eoin Reid
@Test
public void testIsConfinedTrueB() { assertTrue(businessSchool.isConfined()); }

Eoin Reid
@Test
public void testIsConfinedFalse() { assertFalse(greyfriarsKirkyard.isConfined()); }

```

System can calculate the distance between two longlat coordinates.

```

Eoin Reid
@Test
public void testDistanceTo(){
    double calculatedDistance = 0.0015535481968716011;
    assertTrue(approxEq(appletonTower.distanceTo(businessSchool), calculatedDistance));
}

```

System can test whether two longlat coordinates are within 0.00015 degrees of each other

```

Eoin Reid
@Test
public void testCloseToTrue(){
    LongLat alsoAppletonTower = new LongLat( longlat: -3.186767933982822, lat: 55.94460006601717);
    assertTrue(appletonTower.closeTo(alsoAppletonTower));
}

Eoin Reid
@Test
public void testCloseToFalse() { assertFalse(appletonTower.closeTo(businessSchool)); }

```

Drone can only fly in a direction that is a multiple of 10 from 0 to 350 (unit)

```

@Test
public void testAngle0(){
    LongLat nextPosition = appletonTower.nextPosition( angle: 0);
    LongLat calculatedPosition = new LongLat( longit: -3.186724, lat: 55.944494);
    assertTrue(approxEq(nextPosition, calculatedPosition));
}

± Eoin Reid
@Test
public void testAngle20(){
    LongLat nextPosition = appletonTower.nextPosition( angle: 20);
    LongLat calculatedPosition = new LongLat( longit: -3.186733046106882, lat: 55.9445453030215);
    assertTrue(approxEq(nextPosition, calculatedPosition));
}

± Eoin Reid
@Test
public void testAngle50(){
    LongLat nextPosition = appletonTower.nextPosition( angle: 50);
    LongLat calculatedPosition = new LongLat( longit: -3.186777581858547, lat: 55.94460890666647);
    assertTrue(approxEq(nextPosition, calculatedPosition));
}

± Eoin Reid
@Test
public void testAngle90(){
    LongLat nextPosition = appletonTower.nextPosition( angle: 90);
    LongLat calculatedPosition = new LongLat( longit: -3.186874, lat: 55.944644);
    assertTrue(approxEq(nextPosition, calculatedPosition));
}

```

System can calculate the cost of a given order

```

± Eoin Reid +1 *
@Test
public void testMenusTwo() {
    // The webserver must be running on port 9898 to run this test.
    Menu menu = new Menu( port: "8080");
    ArrayList<Menu> menuList = menu.getMenus();
    ArrayList<String> items = new ArrayList<>(Arrays.asList("Ham and mozzarella Italian roll", "Salami and Swiss Italian roll"));
    int totalCost = menu.getDeliveryCost(items);
    // Don't forget the standard delivery charge of 50p
    assertEquals( expected: 230 + 230 + 50, totalCost);
}

± Eoin Reid +1 *
@Test
public void testMenusThree() {
    // The webserver must be running on port 9898 to run this test.
    Menu menu = new Menu( port: "8080");
    ArrayList<Menu> menuList = menu.getMenus();
    ArrayList<String> items = new ArrayList<>(Arrays.asList(
        "Ham and mozzarella Italian roll",
        "Salami and Swiss Italian roll",
        "Flaming tiger latte"
    ));
    int totalCost = menu.getDeliveryCost(items);
    // Don't forget the standard delivery charge of 50p
    assertEquals( expected: 230 + 230 + 460 + 50, totalCost);
}

```

System can obtain items for a given order

```

└─ eoinpatrickreid
@Test
public void testOrderDetails() {
    // The database thingy must be running on port 9751 to run this test.
    OrderDetails orderDetails = new OrderDetails( dport: "9751");
    ArrayList<String> detailsList = orderDetails.getOrderDetails( number: "62b4f805");
    assertFalse(detailsList.isEmpty());
}

```

System can obtain orders and menus from the database server

```

└─ eoinpatrickreid
@Test
public void testGetOrders() {
    // The database thingy must be running on port 9751 to run this test.
    Date date = new GregorianCalendar( year: 2023, Calendar.DECEMBER, dayOfMonth: 31).getTime();
    OrderDetails orderDetails = new OrderDetails( dport: "9751");
    Menus menus = new Menus( port: "8080");
    menus.getMenus();
    String jdbcString = String.format("jdbc:derby://localhost:%s/derbyDB", "9751");
    Orders orders = new Orders(jdbcString, date, orderDetails, menus);
    ArrayList<Order> ordersList = orders.getOrders();

    assertFalse(ordersList.isEmpty());
}

└─ eoinpatrickreid +1
@Test
public void testGetMenus() {
    // The database thingy must be running on port 9751 to run this test.
    Menus menus = new Menus( port: "8080");
    ArrayList<Menu> menusList = menus.getMenus();
    assertFalse(menusList.isEmpty());
}

```

System can convert a what3words address into coordinates

```

└─ eoinpatrickreid
@Test
public void testLocationFromWords() {
    // The database thingy must be running on port 9751 to run this test.
    OrderDetails orderDetails = new OrderDetails( dport: "9751");
    Menus menus = new Menus( port: "8080");
    menus.getMenus();
    ArrayList<Menu> menusList = new ArrayList<>();
    menusList = menus.getMenus();
    Date date = new GregorianCalendar( year: 2023, Calendar.DECEMBER, dayOfMonth: 31).getTime();
    String jdbcString = String.format("jdbc:derby://localhost:%s/derbyDB", "9751");
    Orders orders = new Orders(jdbcString, date, orderDetails, menus);
    ArrayList<Order> ordersList = new ArrayList<Order>();
    ordersList = orders.getOrders();
    ordersList = (ArrayList<Order>) ordersList.stream().sorted(Comparator.comparing(Order::getValue).reversed()).collect(Collectors.toList());
    Words words = new Words( p: "8080");
    LongLat x = words.locationFromWords( input: "sketch.spill.puzzle");
    LongLat y = new LongLat( long: -3.191065, lat: 55.945626);
    assertTrue(approxEq(x, y));
}

```

System can obtain features and landmarks from the database

```

eoinpatrickreid
@Test
public void testGetFeatures() {
    // The database thingy must be running on port 9751 to run this test.
    Movement movement = new Movement( p: "8080");
    List<Polygon> features = movement.getFeatures();
    assertFalse(features.isEmpty());
}

eoinpatrickreid
@Test
public void testGetLandmarks() {
    // The database thingy must be running on port 9751 to run this test.
    Movement movement = new Movement( p: "8080");
    List<LongLat> landmarks = movement.getLandmarks();
    System.out.println(landmarks.get(1).latitude);
    System.out.println(landmarks.get(1).longitude);
    assertFalse(landmarks.isEmpty());
}

```

System can test whether a given route is possible without entering no-fly zones

```

eoinpatrickreid *
@Test
public void testPossible() {
    // The database must be running on port 9751 to run this test.
    Movement movement = new Movement( p: "8080");
    movement.getFeatures();
    movement.getLandmarks();
    //movement.getEdges();
    OrderDetails orderDetails = new OrderDetails( dport: "9751");
    Menu menu = new Menu( port: "8080");
    ArrayList<Menu> menuList = menu.getMenus();
    Date date = new GregorianCalendar( year: 2023, Calendar.DECEMBER, dayOfMonth: 31).getTime();
    String jdbcString = String.format("jdbc:derby://localhost:1527/derby08", "9751");
    Orders orders = new Orders(jdbcString, date, orderDetails, menu);
    ArrayList<Order> ordersList = new ArrayList<Order>();
    ordersList = orders.getOrders();
    ordersList = (ArrayList<Order>) ordersList.stream().sorted(Comparator.comparing(Order::getValue).reversed()).collect(Collectors.toList());
    Words words = new Words( p: "8080");
    LongLat x = words.locationFromWords( input: "sketch.spill.puzzle");
    LongLat y = new LongLat( longi: -3.186874, lat: 55.944494);
    //ArrayList<String> detailsList = orderDetails.getOrderDetails("62b4f805");
    assertFalse(movement.possible(x, y));
}

```

Given two coordinates the system can generate a series of moves moving the drone from one to another, avoiding no-fly zones and staying within the confinement area.

```

1 eoinpatrickreid
@Test
public void testMoveTo() {
    // The database must be running on port 9751 to run this test.
    Movement movement = new Movement( p: "8080");
    OrderDetails orderDetails = new OrderDetails( dport: "9751");
    Menus menus = new Menus( port: "8080");
    ArrayList<Menu> menusList = menus.getMenus();
    Date date = new GregorianCalendar( year: 2023, Calendar.DECEMBER, dayOfMonth: 31).getTime();
    String jdbcString = String.format("jdbc:derby://localhost:xs/derbyDB", "9751");
    Orders orders = new Orders(jdbcString, date, orderDetails, menus);
    ArrayList<Order> ordersList = new ArrayList<Order>();
    ordersList = orders.getOrders();
    ordersList = (ArrayList<Order>) ordersList.stream().sorted(Comparator.comparing(Order::getValue).reversed()).collect(Collectors.toList());
    Words words = new Words( p: "8080");
    LongLat x = words.locationFromWords( input: "surely.native.foal");
    LongLat y = new LongLat( longit: -3.186874, lat: 55.944494);
    //ArrayList<String> detailsList = orderDetails.getOrderDetails("62b4f805");
    Connection conn = null;
    try {
        conn = DriverManager.getConnection(jdbcString);
        App.createTables(conn);
    } catch (SQLException throwables) {
        System.out.println("wit");
        throwables.printStackTrace();
    }
    LongLat z = movement.moveTo(x, y, conn, orderNo: "example");
    assertTrue(z.closeTo(y));
}

```

System can calculate how many moves it would take to move from x to y

```

@Test
public void testHowMany() {
    // The database thingy must be running on port 9751 to run this test.
    String jdbcString = String.format("jdbc:derby://localhost:xs/derbyDB", "9751");
    Movement movement = new Movement( p: "8080");
    OrderDetails orderDetails = new OrderDetails( dport: "9751");
    Menus menus = new Menus( port: "8080");
    ArrayList<Menu> menusList = menus.getMenus();
    Date date = new GregorianCalendar( year: 2022, Calendar.SEPTEMBER, dayOfMonth: 15).getTime();
    Orders orders = new Orders(jdbcString, date, orderDetails, menus);
    ArrayList<Order> ordersList = orders.getOrders();
    Words words = new Words( p: "8080");
    LongLat x = new LongLat( longit: -3.191594, lat: 55.943658);
    LongLat y = new LongLat( longit: -3.191594, lat: 55.943658);
    //ArrayList<String> detailsList = orderDetails.getOrderDetails("62b4f805");

    Connection conn = null;
    try {
        conn = DriverManager.getConnection(jdbcString);
        App.createTables(conn);
    } catch (SQLException throwables) {
        System.out.println("wit");
        throwables.printStackTrace();
    }

    Order order = ordersList.get(6);
    int moves = movement.howManyMoves(x, words.locationFromWords(order.deliverTo));
    System.out.println(moves);
    assertTrue( condition: moves < 1500);
}

```

System can obtain shop details from webserver


```

public void testGetShops() {
    // The database thingy must be running on port 9751 to run this test.
    Movement movement = new Movement( p: "8080");
    OrderDetails orderDetails = new OrderDetails( dport: "9751");
    Menu menu = new Menu( port: "8080");
    ArrayList<Menu> menuList = menu.getMenus();
    Date date = new GregorianCalendar( year: 2023, Calendar.DECEMBER, dayOfMonth: 31).getTime();
    String jdbcString = String.format("jdbc:derby://localhost:%s/derbyDB", "9751");
    Orders orders = new Orders(jdbcString, date, orderDetails, menu);
    ArrayList<Order> ordersList = new ArrayList<>();
    ordersList = orders.getOrders();
    ordersList = (ArrayList<Order>) ordersList.stream().sorted(Comparator.comparing(Order::getValue).reversed()).collect(Collectors.toList());
    Words words = new Words( p: "8080");
    LongLat x = words.locationFromWords( input: "sketch.spill.puzzle");
    LongLat y = new LongLat( long: -3.186874, lat: 55.944494);
    //ArrayList<String> detailsList = orderDetails.getOrderDetails("62b4f805");
    Connection conn = null;
    try {
        conn = DriverManager.getConnection(jdbcString);
        App.createTables(conn);
    } catch (SQLException throwables) {
        System.out.println("wit");
        throwables.printStackTrace();
    }
    ArrayList<String> details = new ArrayList<String>();
    details = orderDetails.getOrderDetails( number: "lad5fiff");
    List<String> shops = new ArrayList<String>();
    shops = orderDetails.getShops( orderNo: "lad5fiff", menu);
    assertEquals(shops.isEmpty());
}

```

System can calculate how many moves a proposed order would take to deliver

```

public void testHowManyForOrder() {
    // The database thingy must be running on port 9751 to run this test.
    String jdbcString = String.format("jdbc:derby://localhost:%s/derbyDB", "9751");
    Date date = new GregorianCalendar( year: 2023, Calendar.DECEMBER, dayOfMonth: 31).getTime();
    LongLat y = new LongLat( long: -3.186874, lat: 55.944494);
    //LongLat x = words.locationFromWords("sketch.spill.puzzle");
    OrderDetails orderDetails = new OrderDetails( dport: "9751");
    Menu menu = new Menu( port: "8080");
    ArrayList<Menu> menuList = menu.getMenus();
    Orders orders = new Orders(jdbcString, date, orderDetails, menu);
    ArrayList<Order> ordersList = orders.getOrders();
    Words words = new Words( p: "8080");
    Movement movement = new Movement( p: "8080");
    movement.getLandmarks();
    movement.getFeatures();
    Connection conn = null;
    try {
        conn = DriverManager.getConnection(jdbcString);
        App.createTables(conn);
    } catch (SQLException throwables) {
        System.out.println("wit");
        throwables.printStackTrace();
    }
    int totalMoves = 0;
    for (Order ord : ordersList) {
        int moves = movement.movesForOrder(y, ord, orderDetails, words, menu);
        System.out.println(moves);
        totalMoves = totalMoves + moves;
    }
    System.out.println(totalMoves);
    assertTrue( condition: totalMoves<1500);
}

```

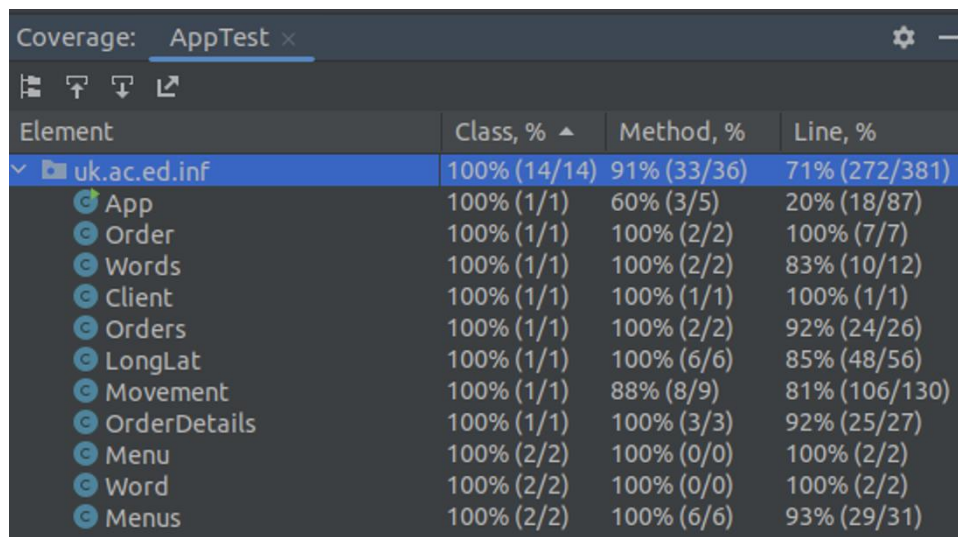

Final System test

```
@Test
public void testApp() {
    String[] args = {"12", "7", "2023", "8080", "9751"};
    App.main(args);
    assertFalse(1==2);
}/*
```

Adequacy of Testing

As I used a range of techniques in various areas of the system it is hard to use all the adequacy criteria and apply it to different sections. The strategy I am therefore going to go with is to focus on structural testing adequacy criterion specifically class, method and line coverage criterion. My goal is to have complete class and method coverage and high line coverage. This will ensure that most areas that are likely to be used in the final system are tested. It is however not entirely complete, therefore allowing the possibility that some untested lines of code may cause rare errors. With my budget though I am happy to concede this.

Initial coverage of test suite.



Coverage: AppTest x			
Element	Class, % ^	Method, %	Line, %
uk.ac.ed.inf	100% (14/14)	91% (33/36)	71% (272/381)
App	100% (1/1)	60% (3/5)	20% (18/87)
Order	100% (1/1)	100% (2/2)	100% (7/7)
Words	100% (1/1)	100% (2/2)	83% (10/12)
Client	100% (1/1)	100% (1/1)	100% (1/1)
Orders	100% (1/1)	100% (2/2)	92% (24/26)
LongLat	100% (1/1)	100% (6/6)	85% (48/56)
Movement	100% (1/1)	88% (8/9)	81% (106/130)
OrderDetails	100% (1/1)	100% (3/3)	92% (25/27)
Menu	100% (2/2)	100% (0/0)	100% (2/2)
Word	100% (2/2)	100% (0/0)	100% (2/2)
Menus	100% (2/2)	100% (6/6)	93% (29/31)

Above we can see the initial coverage of my test suite, this was fairly good coverage however I was unhappy with not having run each method in at least one test. Therefore, I implemented a further whole system test (running as the program was intended)

Coverage after implementing this test

Coverage: AppTest x			
Element	Class, % ▲	Method, %	Line, %
uk.ac.ed.inf	100% (14/14)	100% (36/36)	89% (342/381)
App	100% (1/1)	100% (5/5)	85% (74/87)
Order	100% (1/1)	100% (2/2)	100% (7/7)
Words	100% (1/1)	100% (2/2)	83% (10/12)
Client	100% (1/1)	100% (1/1)	100% (1/1)
Orders	100% (1/1)	100% (2/2)	92% (24/26)
LongLat	100% (1/1)	100% (6/6)	91% (51/56)
Movement	100% (1/1)	100% (9/9)	90% (117/130)
OrderDetails	100% (1/1)	100% (3/3)	92% (25/27)
Menu	100% (2/2)	100% (0/0)	100% (2/2)
Word	100% (2/2)	100% (0/0)	100% (2/2)
Menus	100% (2/2)	100% (6/6)	93% (29/31)

As we can see this improved the coverage, extending the test suite to run each class and method, and to run 89% of all lines of code in the system.