

This living document provides the specifications for the remainder of the project. Get started with your team on understanding and planning. We will use GitHub for source control, but for now, you can build the skeleton and work on parts separately without it.

Part 1: Navigation Overlay Builder

The goal of Part 1 is to read a single text file that collectively defines the navigational aids (navaids) in the world, build the corresponding navaids with the components provided in the architecture, and add them to a navigation overlay for the radar display.

Definitions

There are five kinds of navaids: fix, NDB, VOR, ILS, and airway. The function of each is irrelevant here. Each has its own section in the definition file, always in this order. Entries appear on separate lines. A blank line ends a section. Ignore whitespace.

Italic text refers to these field definitions:

<u>Field</u>	<u>Definition</u>	<u>Description</u>	<u>Example</u>	<u>Datatype</u>
<i>altitude</i>	double	altitude (feet)	1000	Altitude
<i>azimuth</i>	double	degrees (compass)	270	AngleNavigational
<i>beacon</i>	double,double	distance (miles), altitude (feet)	10,11	NavaidILSBeaconDescriptor
<i>id</i>	String	arbitrary nonempty string	LOLKI	String
<i>latitude</i>	int,int,double	latitude (degrees, minutes, seconds)	49,39,32	Latitude
<i>longitude</i>	int,int,double	longitude (degrees, minutes, seconds)	117,25,30	Longitude
<i>position</i>	<i>latitude,longitude,altitude</i>	latitude, longitude, altitude	49,39,32, 117,25,30, 1950	CoordinateWorld3D
<i>uhf_frequency</i>	int	frequency (kilohertz)	320	UHFFrequency
<i>vhf_frequency</i>	int,int	frequency (megahertz and kilohertz)	118,1	VHFFrequency

Fix Navaid

A fix navaid is an arbitrary point in the world that defines something of interest at that location, but it does not need to refer to anything physically there. In other words, it is simply an × on the chart.

This section is defined as follows:

```
[NAVAID:FIX]
id, position
```

For example (minus the section header):

```
fix1, 48,38,31, 116,24,29, 1949
```

Blue is latitude, green is longitude, and orange is altitude.

Create a `ComponentNavaidFix` and add it to the shared navigation overlay `OverlayNavigation` (see [The Story](#)) with `addNavaid()`.

NDB Navaid

An NDB (nondirectional beacon) navaid defines a simple radio beacon at a position in three-dimensional space.

This section is defined as follows:

```
[NAVAID:NDB]
id, uhf_frequency, position
```

For example:

```
ndb1, 320, 48,38,31, 116,24,29, 1949
```

Purple is frequency.

Create a ComponentNavaidNDB.

VOR Navaid

A VOR (very-high-frequency omnidirectional range) navaid defines a complex radio beacon at a position in three-dimensional space.

This section is defined as follows:

```
[NAVAID:VOR]
id, vhf_frequency, position
```

For example:

```
vor1, 118,1, 51,41,34, 119,27,32, 1952
```

Create a ComponentNavaidVOR.

ILS Navaid

An ILS (instrument landing system) navaid defines a transmitter for landing. It consists of a reference point for the runway in three-dimensional space, plus three distance beacons at an azimuth from the reference point.

This section is defined as follows:

```
[NAVAID:ILS]
id, vhf_frequency, position, azimuth, beacon, beacon, beacon
```

For example:

```
ils1, 118,325, 49,39,32, 117,25,30, 1950, 135, 14,13, 12,11, 10,9
```

Azimuth is gold. The order of the beacons in pink is outer, middle, inner.

Create a ComponentNavaidILS.

Airway

An airway is a straight connection between two navaids. It has three forms, which can appear in any order in this section provided that navaids are defined before they are used.

This section is defined as follows:

```
[NAVAID:AIRWAY]
```

The first type of airway is coordinate₁ to coordinate₂:

```
id, CC, position1, position2
```

For example:

v1, CC, 49,39,32, 117,25,30, 1950, 50,40,33, 118,26,31, 1951

The second type is navaid₁ to coordinate:

id, NC, id₁, position

For example:

v2, NC, vor1, 50,40,33, 118,26,31, 1951

The third type is navaid₁ to navaid₂:

id, NN, id₁, id₂

For example:

v3, NN, ndb1, ils1

Navaid identifiers are in teal.

Create a ComponentNavaidAirway.

Setup

Eclipse is the standard IDE in the CS curriculum. Setup is similar for other IDEs, but it is your responsibility to figure out how because we cannot support every configuration.

Put the latest JAR file somewhere in your project. The exact location does not matter.

Create class `atcsim.loader.NavigationOverlayBuilder` in a new project. It must reside in the correct place as specified by the package.

In the constructor, add a print statement that says this is your code executing.

From Eclipse Package Explorer, right-click your project, select *Build Path*, then select *Configure Build Path*. In the *Libraries* tab, highlight *Classpath*, click *Add External JARs*, and add the JAR.

From a tester class of your choice (with a `main` method), instantiate `NavigationOverlayBuilder`. When you run this tester, it should print your statement. If it says “this feature is locked; execution is disabled”, then you are still accessing my solution instead of substituting yours.

Implementation

Implement the Java classes in `atcsim.loader` and `atcsim.loader.navaid`.

Use a `Scanner`. Build helper methods for the fields in the table.

A test file is provided. There is no graphical output for this part. You need to verify your implementation by printing the contents of objects of interest. Be careful with your test values because they must be valid for the architecture to accept them.

The Story

Class `NavigationOverlayBuilder` is the entry point for execution. Call `loadDefinition()` with the definition filename. Create an `InputStream` from a `FileInputStream` for the scanner. Also create the `OverlayNavigation`. Then call `LoaderFix`, `LoaderNDB`, `LoaderVOR`, `LoaderILS`, and `LoaderAirway` to have them add their contribution to the overlay and the collection of nav aids in the hashmap, which airways need.

Each loader is slightly different, but the interaction is the same. Scan each line of the section, extract its elements, and assemble them into the parameters that the corresponding ComponentNavaidX needs, then add it to the overlay.

Deliverables

For Pre-Task P1, submit your questions as a team in plain text format with a `.txt` extension and a blank line between questions. Only one team member needs to do this.

For the actual task, submit your source files zipped up in their correct folders.

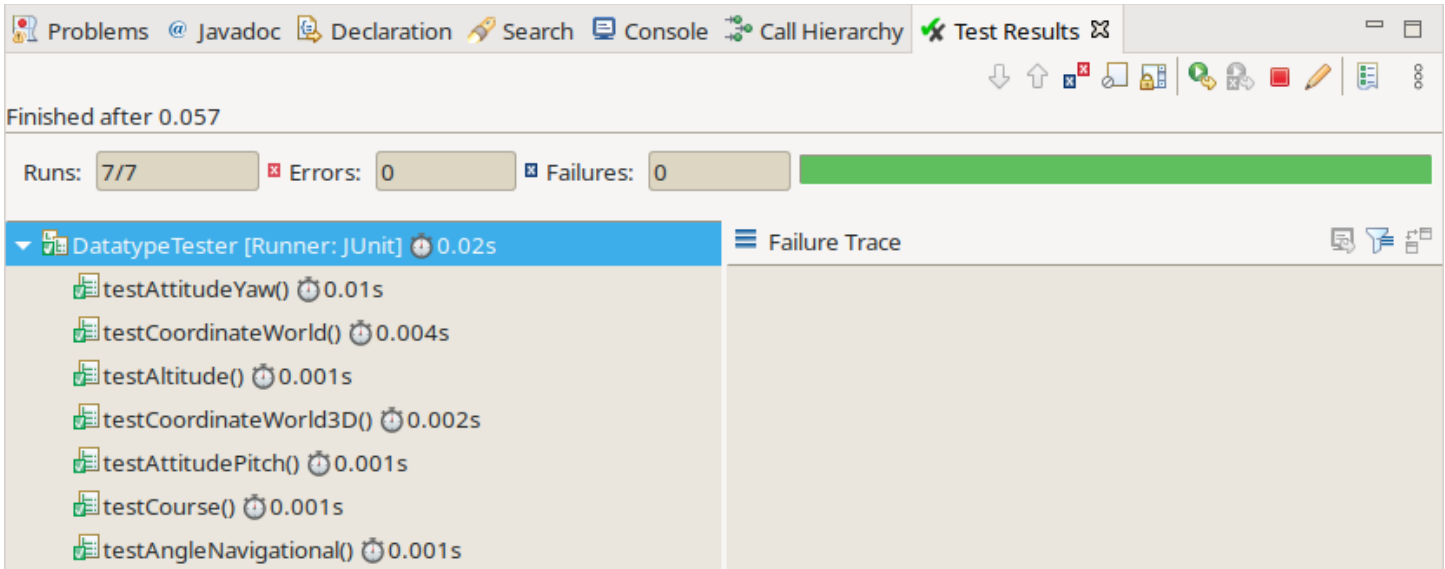
Part 2: JUnit Datatype Tests

The goal of Part 2 is to demonstrate basic unit testing on a representative subset of the datatypes provided by the architecture.

Setup

You can add the tester source file to your Part 1 project or create a new project and connect to the JAR as described in Part 1.

In Eclipse, create a new run configuration for *JUnit Test*. When you run your test suite, you should see something similar to this.



If a test fails, the green bar will be red. The failure trace will indicate which assertion failed.

Implementation

Implement class `atcsim.part2.DatatypeTester` with the following JUnit tests. Each method has the annotation `@Test`. Within the method, use either variant of `assertEquals()` as appropriate to verify the results.

Method `testAltitude`

Create altitudes $a_1=1000$ and $a_2=200$.

Verify the following:

- $a_1 + a_2$ is correct. Use `getValue_()`.
- $a_2 + a_1$ is correct.
- $a_1 - a_2$ is correct.
- $a_2 - a_1$ is correct.
- $a_1 = a_1$ is correct. Use `compareTo()`.
- $a_1 < a_2$ is correct.
- $a_2 > a_1$ is correct.

Method testAngleNavigational

Create angles $a_1=90$ and $a_2=180$.

Verify the following:

- a_1 reciprocate is correct.
- a_2 reciprocate is correct.
- a_1 interpolate a_2 is correct. Use scaler 50% (0.5).
- a_2 interpolate a_1 is correct.

Method testAttitudePitch

Create pitch $p=10$.

Verify the following:

- $0 + p$ is correct.
- $90 + p$ is correct.
- $175 + p$ is correct.

Method testAttitudeYaw

Create yaw $y=10$.

Verify the following:

- $-5 + y$ is correct.
- $175 + y$ is correct.
- $5 - y$ is correct.
- $-175 - y$ is correct.

Method testCourse

Create course $c=10$

Verify the following:

- $0 + c$ is correct.
- $355 + c$ is correct.
- $0 - c$ is correct.
- $355 - c$ is correct.

In the comments of your method, explain how AttitudeYaw and Course differ.

Method testCoordinateWorld

Create positions $p_1=\text{CoordinateWorld.KSFF}$ and $p_2=(\text{latitude}(1^\circ 2' 3'') \text{ longitude}=(3^\circ 2' 1''))$

Verify the following:

- $p_1 = p_1$ is correct. Use `compareTo()`.
- $p_1 + p_2$ is correct.

Method testCoordinateWorld3D

Create position $p = \text{CoordinateWorld.KSFF}$

Verify the following:

- p calculateBearing p is correct for angle. Use `getAngle().getValue_()`.
- p calculateBearing p is correct for distance. Use `getRadiusNauticalMiles().getValue_()`.
- p calculateBearing KSFF_N is correct for angle and distance.
- p calculateBearing KSFF_S is correct for angle and distance.
- p calculateBearing KSFF_E is correct for angle and distance.
- p calculateBearing KSFF_W is correct for angle and distance.

Deliverables

Submit `DatatypeTester.java`. Only one team member needs to do this.

Part 3: System Test and Evaluation

This final part of the project addresses testing and evaluation of the provided solution. It focuses predominantly on breadth, not depth. The goal is to demonstrate that each feature reasonably works for at least one representative scenario. A real test plan for a project of this relatively small size could easily expand to hundreds or even thousands of times the size of this document.

Lectures 49 onward cover this form of testing.

Setup

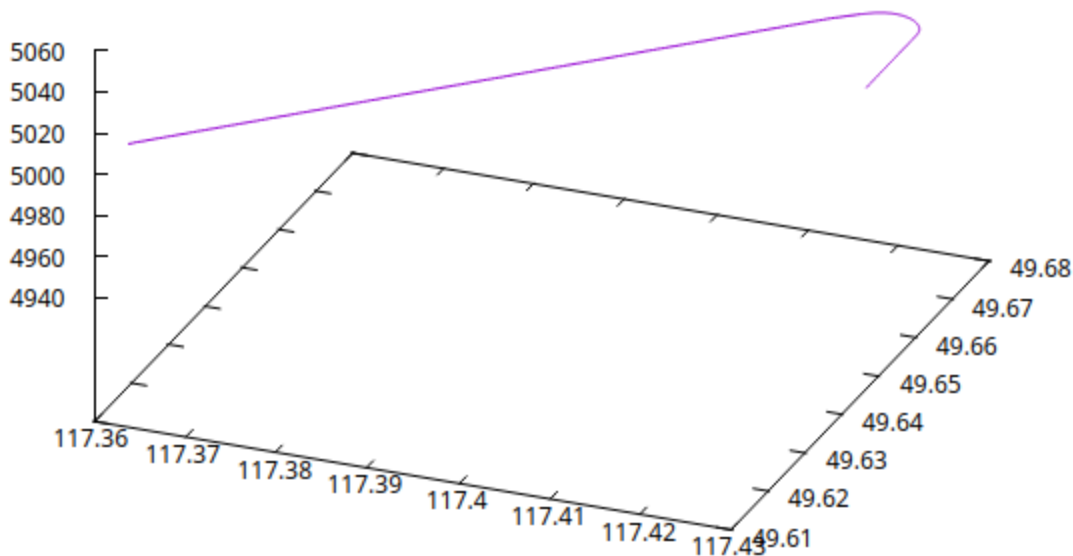
The project jar (0.2 or later) supports Part III as an executable program. It is recommended to run the JAR from the command line even on Windows because some error messages go to standard output, which is not visible if you double-click the JAR. To execute it on Windows or Linux, enter: `java -jar cs350-project-v02.jar`. You should not need it, but the main method is in `atcsim.gui.MasterGUI`.

Install gnuplot from www.gnuplot.info for the visualizations. For Windows it looks like you need to download `gp542-win64-mingw.exe`. When you run the simulator, it generates file `atc.gnu` in your home path. It consists of space-delimited longitude, latitude, and altitude values of the test aircraft at each update (much like in Task 3).

The gnuplot interface varies according to your platform. You need to load `atc.gnu` and plot it in 3D:

```
plot 'atc.gnu' with lines
```

The sample log file produces this output:



You can drag the perspective around with the mouse and zoom to some degree. Produce the images for the test from screenshots. Be sure to label the axes and indicate where north is somehow.

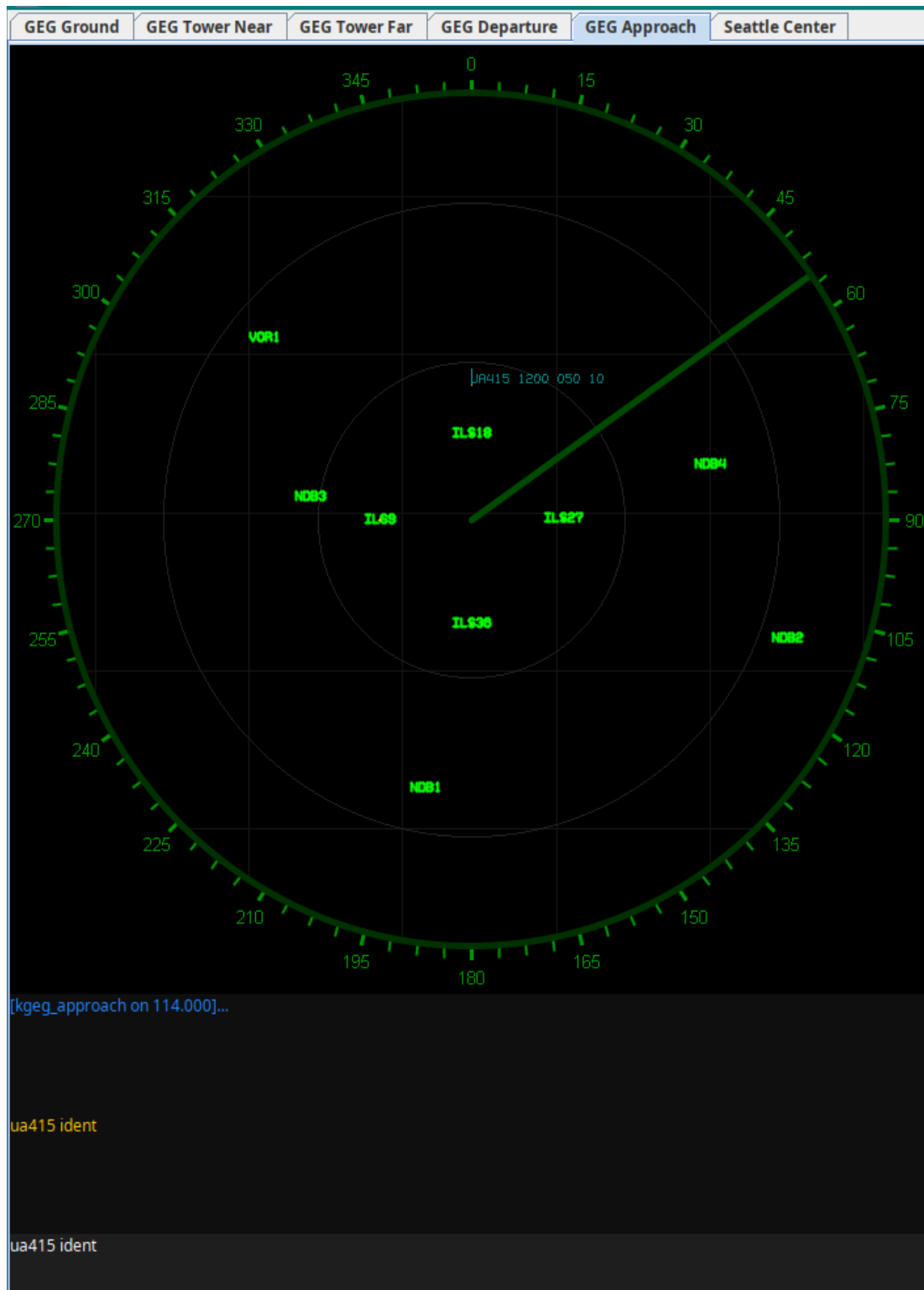
Usage

The command line interface is the architecture controller that interacts with the predefined world. Most of the world, like the airport environment, is not considered. All tests use the GEG Departure and Approach and maybe the Seattle Center tabs. (KGEK is Spokane International.)

The radar displays are all different views of the same model, which reflect the ATC controller roles of ground, tower, departure, enroute, and approach. All commands function on all displays, but the scope may not be appropriate to observe the results.

The text interface is on the bottom. White is your input as a controller. Orange is previous input. Ignore the blue, which we are not using.

The keep logging clean, there is only one aircraft: UA415. Furthermore, to keep the tests manageable, we can use unrealistically slow speeds like 20 knots. The bright green elements are fixes. The radar sweep effect is disabled because it makes screenshots very difficult.



As in Part I, the format consists of rules with fields. The fields are different, however, because this is a format for humans, not loaders.

Field	Definition	Example
altitude	digit digit digit [digit] [digit] flight level digit digit digit	below 18000 feet: 800, 8500, 11500; otherwise: flight level 180, flight level 320
callsign	(letter [letter] digit digit digit) (N char char char char char)	AA429, UA007, N4290Q
course	digit digit digit	030, 315
fix	letter letter letter letter letter	PUPPY, SHLBY, ESTRN
frequency	digit digit digit point digit [digit] [digit]	118.1, 124.35, 128.625
gate	gate space letter digit [digit]	gate A3, gate C17
runway	runway space digit [digit]	runway 3, runway 27
speed	[digit] digit digit	80, 250
squawk	squawk digit digit digit digit	squawk 1412
taxiway	letter [digit]	A, B2, C
digit	0...9	
letter	A...Z	
char	digit letter	

All the controller commands are listed here, although many are not used for the specified tests. Error handling is rudimentary. If you enter an invalid command, you should get a dialog with an error message. Generally, execution should be unaffected, so you may be able to correct the error and continue without restarting the test.

Unused commands may lack a configuration of the world to function properly.

I. Ground Commands

1. “**callsign** taxi to **runway** via **taxiway**+”

Controller instructs aircraft to taxi up to runway threshold via one or more taxiways.

2. “**callsign** taxi to **taxiway**+”

Controller instructs aircraft to taxi via one or more taxiways.

3. “**callsign** stop”

Controller instructs aircraft to stop taxiing.

4. “**callsign** proceed”

Controller instructs aircraft to continue taxiing as previously instructed.

5. “**callsign** taxi to **gate**”

Controller instructs aircraft to taxi to gate from current taxiway threshold, which must be nearest the gate.

II. Tower Commands

1. “**callsign** cleared for takeoff **runway**”

Controller instructs aircraft at runway threshold to enter runway and take off.

2. “**callsign** position and hold **runway**”

Controller instructs aircraft at runway threshold to enter runway, position itself for take off, then wait.

3. “**callsign** contact departure on **frequency**”

Controller instructs aircraft to contact Departure on frequency.

4. “**callsign** cleared to land **runway**”

Controller instructs aircraft to land on runway.

5. “**callsign** execute missed approach to **fix** at **altitude** contact approach on **frequency**”

Controller instructs aircraft to abort landing, head to missed-approach fix, and contact Approach on frequency.

6. “**callsign** contact ground on **frequency**”

Controller instructs aircraft to contact Ground on frequency.

III. Departure Commands

1. “**callsign** contact center on **frequency**”

Controller instructs aircraft to contact Enroute on frequency.

IV. Approach Commands

1. “**callsign** contact tower on **frequency**”

Controller instructs aircraft to contact Tower on frequency.

2. “**callsign** cleared to **fix** for procedure turn **runway**”

Controller instructs aircraft to proceed to initial approach fix and execute standard procedure turn to align inbound for runway.

3. “**callsign** cleared for the approach ILS **runway**”

Controller instructs aircraft that it is cleared for ILS approach to outer marker then middle marker.

V. Enroute Commands

1. “**callsign** {climb,descend} and maintain **altitude**”

Controller instructs aircraft to climb or descend to altitude and maintain it.

2. “**callsign** turn [{left,right}] to **course**”

Controller instructs aircraft to turn left, right, or in the shortest way to course.

3. “**callsign** {reduce,increase} speed to **speed**”

Controller instructs aircraft to reduce or increase its speed.

4. “**callsign** (proceed to **fix** [cross at **altitude**])+”

Controller instructs aircraft to fly to one or more fixes with optional altitudes.

5. “**callsign** (proceed to **fix** [cross at **altitude**])+ and hold”

Controller instructs aircraft to fly to one or more fixes and hold according to standard hold procedures.

6. “**callsign** contact approach on **frequency**”

Controller instructs aircraft to contact Approach on frequency.

VI. General Commands

1. “**callsign squawk**”

Controller instructs aircraft to set its transponder code.

2. “**callsign ident**”

Controller instructs aircraft to trigger an ident.

Deliverable

You need to produce one document with all your tests. Tests are stated in the form similar to requirements. Unless otherwise specified, you may satisfy each however you want. Each must address the following in exactly this form, including the number, in a separate section:

The test number and title, as indicated below.

1. The rationale behind the test; i.e., what is it testing and why we care.
2. A general English description of the conditions of the test.
3. The commands for (2), which must appear in a standalone form that could be directly copied into a text file to reproduce the test without manual intervention. Do not cross-reference tests; instead, repeat duplicated code.
4. An English narrative of the expected results of executing the test. Consider this *before* running the test.
5. The actual results with at least one screen shot of the view.
6. A brief discussion on how the actual results differ from the expected results, if they differ.
7. A suggestion for how to extend this test to cover related aspects not required here.

Your document must be formatted professionally. It must be consistent in all respects across all team members. Code references must be in a monospace font.

Tests

Test 0 is an example of an expected report entry from a previous project with similar requirements. It demonstrates more than you may need to do for any single test.

Example Test 0: @wait command

1.

This test verifies that the `wait` command introduces the appropriate time delay between command servicing. It is necessary to automate subsequent tests, which may require dwell for actions to carry out.

2.

MY_SHIP1 starts north of MY_SHIP2, facing west, with no speed. It is equipped with two missiles that have a radar sensor and an acoustic fuze. MY_SHIP2 starts facing south with speed 10.

3.

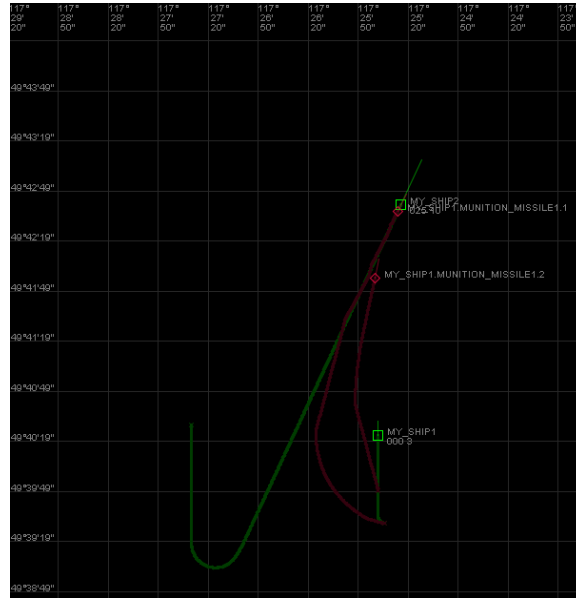
```
define sensor radar FUZE_RADAR1 with field of view 30 power 50 sensitivity 10
define sensor acoustic FUZE_ACOUSTIC1 with sensitivity 10
define munition missile MUNITION_MISSILE1 with sensor FUZE_RADAR1 fuze FUZE_ACOUSTIC1 arming distance 0.5
define ship ACTOR_SHIP1 with munition (MUNITION_MISSILE1)
create actor MY_SHIP1 from ACTOR_SHIP1 at 49*39'31#/117*25'34#/0 with course 270 speed 0
create actor MY_SHIP2 from ACTOR_SHIP1 at 49*40'30#/117*27'30#/0 with course 180 speed 10
set MY_SHIP1 load munition MUNITION_MISSILE1
set MY_SHIP1 load munition MUNITION_MISSILE1
@wait 5
set MY_SHIP2 course 25
@wait 6
set MY_SHIP1 deploy munition MY_SHIP1.MUNITION_MISSILE1.1
set MY_SHIP1 course 360; set MY_SHIP1 speed 3
@wait 5
set MY_SHIP1 deploy munition MY_SHIP1.MUNITION_MISSILE1.2
@wait 10
set MY_SHIP1 course 225
set MY_SHIP1 speed 20
```

4.

After 5 seconds, MY_SHIP2 will change its course to 25 degrees and continue for 6 seconds. MY_SHIP1 will then fire missile MY_SHIP1.MUNITION_MISSILE1.1 and change to course 360 at speed 3. It will wait 5 seconds and fire missile MY_SHIP1.MUNITION_MISSILE1.2. It will wait 10 seconds and change course to 225 at speed 20.

The first missile should chase MY_SHIP1. The second missile should do the same, except from a slightly more northerly starting position. Both missiles should strike MY_SHIP1 in order.

5.




6.

The actual results are consistent with the expected results.

7.

The wait command could be introduced in front of every other command to verify that it applies. (Currently not all commands are affected by it.)

The state of the aircraft from the controller's perspective is indicated on the display as follows:

A digital display with a black background and green text showing the aircraft state: UA415 0365 050 10.

The fields are the callsign, transponder (squawk) code, altitude (in hundreds of feet) and speed (in tens of knots).

All tests assume the initial conditions of UA415 when the radar display appears.

The following tests demonstrate basic maneuvering.

Test 1: Transponder

Instruct UA415 to change its transponder code to 0324 and ident.

The *IDENT* notification on the aircraft state appears very briefly. You need to be creative to get a picture of it.

Test 2: Simple Straight Climb

Instruct UA415 to climb straight ahead and maintain flight level 240.

Test 3: Compound Straight Climb

Instruct UA415 to climb straight ahead and maintain flight level 320, then upon arriving descend to 15,000 feet.

Test 4: Simple Level Turn

Instruct UA415 to turn right to 135 degrees.

Test 5: Compound Level Turn

Instruct UA415 to turn left to 270 degrees, then upon arriving right to 090.

Test 6: Simple Speed Change

Instruct UA415 to increase speed to 200 knots (entered as 20).

Test 7: Compound Speed Change

Instruct UA415 to increase speed to 250 knots, then upon arriving reduce speed to 100.

Test 8: Descending Turn

Instruct UA415 to climb to flight level 240 to start the test (ignore this setup in the results). Upon arriving, descend to 8,000 feet while turning left to 010.

Test 9: Turn Radius Comparison

This involves three independent tests with combined results. Use Excel for a top-view plot.

9a. Instruct UA415 to turn right to 270.

9b. Instruct UA415 to turn right to 270 and increase speed to 30.

9c. Instruct UA415 to turn right to 270 and increase speed to 50.

The following tests demonstrate basic navigation.

Test 10: Proceed to Fix, Level

Instruct UA415 to proceed to fix NDB4.

Test 11: Proceed to Fix, Climbing

Instruct UA415 to proceed to fix NDB4 and cross it at flight level 200.

Test 12: Proceed to Fixes, Altitude Changes

Instruct UA415 to proceed to fix NDB4 and cross it at flight level 200, NDB2 at 100, NDB1 and 240.

Test 13: Proceed to Fix, Climbing and Hold 1

Instruct UA415 to proceed to fix NDB3 and hold.

Test 14: Proceed to Fix, Climbing and Hold 2

Instruct UA415 to proceed to fix NDB4 and hold.

Explain how Tests 13 and 14 differ.

Test 15: Proceed to Fix, Procedure Turn

Instruct UA415 to proceed to fix NDB4 and execute the procedure turn for runway 22.