CS 350 Project Specification v0.3

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:

Field	Definition	Description	Example	Datatype
altitude	`double`	altitude (feet)	`1000`	`Altitude`
azimuth	`double`	degrees (compass)	`270`	`AngleNavigational`
beacon	`double,double`	distance (miles), altitude (feet)	`10,11`	`NavaidILSBeaconDescriptor`
id	`String`	arbitrary nonempty string	`LOLKI`	`String`
latitude	`int,int,double`	latitude (degrees, minutes, seconds)	`49,39,32`	`Latitude`
longitude	`int,int,double`	longitude (degrees, minutes, seconds)	`117,25,30`	`Longitude`
position	*latitude*, *longitude*, *altitude*	latitude, longitude, altitude	`49,39,32, 117,25,30, 1950`	`CoordinateWorld3D`
uhf_frequency	`int`	frequency (kilohertz)	`320`	`UHFFrequency`
vhf_frequency	`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 coordinate_1 to coordinate_2:

id, CC, $\textit{position}_1$, $\textit{position}_2$

For example:

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

The second type is $navaid_1$ to coordinate:

`id``, NC, `*`id`*$_1$`, `*`position`*

For example:

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

The third type is $navaid_1$ to $navaid_2$:

`id``, NN, `*`id`*$_1$`, `*`id`*$_2$

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 navaids 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.

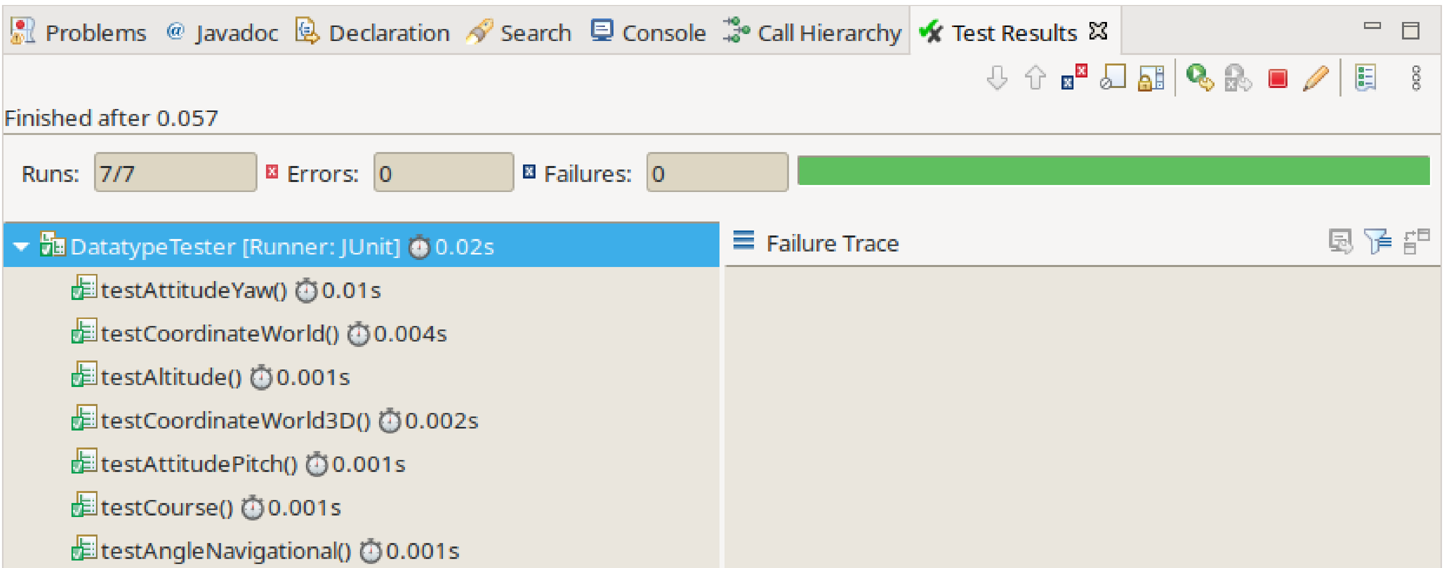
Part 2: JUnit Datatype Tests

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 `atssim.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.
- $a_1 > a_2$ 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:

- 0 + y is correct.
- 355 + y is correct.
- 0 − y is correct.
- 355 − 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=`CoordinateWorld.KSFF` and p_2=(latitude(1°2'3") longitude=(3°2'1"))

Verify the following:

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

Method `testCoordinateWorld3D`

Create position *p*=`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.