CS 350 Project Specifications v0.6

Part I: Command Parser

This living document specifies how to build the communication interface that interprets English-like text commands from the theater controller (you) to configure and manipulate the model, view, and simulation controller. The goal is to parse and process the commands and hand them off to the architecture for execution.

This documentation guides you in the role of programmer for development and quasi end user for evaluation. Everything here derives from decisions made throughout the development process this quarter. Most of it was not your responsibility, but it did happen. Lectures will cover the decisions I made, which affect the decisions you have to make now.

Use our development process to make sense of this document. Ask questions.

Specifications

You will need to link your code to my jar file to use the architectural components I provide. Instructions are coming.

Your classes must reside in package `cs350s21project.cli`.

Create class `CommandInterpreter` with a single `void` method `evaluate(String)` that takes a command string, builds the corresponding command object, and submits it to the `schedule` method in `CommandManagers`. Throw a `RuntimeException` for invalid commands.

Use `CommandManagers.getInstance()` to access `schedule()`. The command object to pass is indicated for each rule below.

Use the most recent jar. To connect to it in Eclipse, right click on the project and select Build Path→Configure Build Path→Libraries→Add External JARs.

To run the solution, execute `cs350s21project.controller.command.test.Part3Startup`. It should report:

```
ERROR: You are accessing the CommandInterpreter in the solution, not your own code. Verify your package.
```

When you configure your `CommandInterpreter` with the correct packaging, it will replace mine and execute your code. Start with a *Hello World* test to verify that everything connects and runs properly before implementing your solution.

Command Protocols

The following fields formally define the variable elements of the command protocols. Subscripts are for clarity only; they are not part of the fields.

Field	Definition	Examples	Datatype
altitude	signed integer (feet)	1000, 9500, -100	Altitude
azimuth	unsigned real (navigational degrees)	10, 45	AttitudeYaw
coordinates	*latitude/longitude/altitude*	45*30'15"/110*30'10"/200	CoordinateWorld3D
course	unsigned 3-digit integer (navigational degrees)	090, 270	Course
distance	unsigned real (nautical miles)	10, 25.3	DistanceNauticalMiles
elevation	unsigned real (mathematical degrees)	10, 25	AttitudePitch
filename	'filename' with normal filename punctuation	'/stu/myfilename.txt'	String
fov	unsigned real (navigational degrees)	10, 45	FieldOfView
id	alphanumeric string, plus underscore and dot	dog, cat32, dog.cat	AgentID
latitude	*degrees***minutes*'*seconds*"	45*30'15"	Latitude
longitude	*degrees***minutes*'*seconds*"	110*30'10.3"	Longitude
power	unsigned real (decibels)	10, 20.5	Power
sensitivity	unsigned real (decibels)	10, 20.5	Sensitivity
size	unsigned integer (pixels)	300	int
speed	unsigned integer (knots)	25	Groundspeed
time	unsigned time (seconds)	5, 10.8	Time

Minutes is an integer on [0,60); degrees is an integer on [0,90] for latitude and [0,180] for longitude; seconds is a double on [0,60).

Angle brackets, and square brackets are not part of commands. Vertical bar indicates logical or; asterisk indicates zero or more instances of the preceding term; plus indicates one or more. Square brackets indicate an optional group.

Whitespace, except in literals, does not matter. All text except identifiers is case insensitive.

Commands may appear on the same line if they are separated by a semicolon.

A comment prefixed with `//` may follow a command or be on its own line.

All identifier definitions must be unique.

All reasonable failure modes must be accounted for with appropriate error handling. The messages and delivery mechanism need not be elaborate or particularly user-friendly. Remember that you are working in an architecture now, which may do some of the work for you.

Use only standard Java (11 or higher), no external tools, libraries, grammar builders, etc. Ask if you are unsure.

Implement all the commands in blue. Be sure to plan carefully to create support functionality that the team can share.

Teams of two do only the commands with an asterisk.

For all teams, implement only the command protocols for the commands you are assigned.

I. VIEWS

View commands govern the creation, destruction, and usage of views.

1. `create window` ***id*** `top view with` ***size*** `(`***latitude₁ latitude₂ latitude₃***`) (`***longitude₁ longitude₂ longitude₃***`)`

   Creates a square window called ***id*** of ***size*** pixels with a top-down view anchored in the center at ***latitude₁*** with vertical extent ***latitude₂*** and grid spacing ***latitude₃***, and at ***longitude₁*** with horizontal extent ***longitude₂*** and grid spacing ***longitude₃***.

   Use `CommandViewCreateWindowTop`.

2. `create window` ***id*** `front view with` ***size*** `(`***longitude₁ longitude₂ longitude₃***`) (`***altitude₁ altitude₂ altitude₃ altitude₄***`)`

   Creates a square window called ***id*** of ***size*** pixels with a north-looking view anchored in the center at ***longitude₁*** and ***altitude₁*** with horizontal extent ***longitude₂*** and grid spacing ***longitude₃***, and vertical extent ***altitude₂***, above-water grid spacing ***altitude₃***, and below-water spacing ***altitude₄***.

   Use `CommandViewCreateWindowFront`.

3. `create window` ***id*** `side view with` ***size*** `(`***latitude₁ latitude₂ latitude₃***`) (`***altitude₁ altitude₂ altitude₃ altitude₄***`)`

   Creates a square window called ***id*** of ***size*** pixels with a west-looking view anchored in the center at ***latitude₁*** and ***altitude₁*** with horizontal extent ***latitude₂*** and grid spacing ***latitude₃***, and vertical extent ***altitude₂***, above-water grid spacing ***altitude₃***, and below-water spacing ***altitude₄***.

   Use `CommandViewCreateWindowSide`.

4.* `delete window` ***id***

   Deletes a window called ***id***.

   Use `CommandViewDeleteWindow`.

5. `lock window` ***id₁*** `on` ***id₂***

   Instructs window ***id₁*** to stay centered on agent ***id₂***, where the agent is an actor or munition.

   Use `CommandViewLockWindow`.

   For Part III, note that my solution does not support this command.

6. `unlock window` ***id***

   Instructs window ***id*** to stop centering on an agent.

Use `CommandViewUnlockWindow`.

II. ACTORS

Actor commands govern the definition, creation, configuration, and usage of airplanes, ships, and submarines.

1.* `define ship` id_1 `with munition[s] (`id_n`+)`

Defines a ship family called id_1 with munitions id_n.

Use `CommandActorDefineShip`.

2. `define airplane` id_1 `with munition[s] (`id_n`+)`

Defines an airplane family called id_1 with munitions id_n.

Use `CommandActorDefineAirplane`.

3. `define submarine` id_1 `with munition[s] (`id_n`+)`

Defines a submarine family called id_1 with munitions id_n.

Use `CommandActorDefineSubmarine`.

4. `undefine actor` *`id`*

Undefines an actor family called *`id`*.

Use `CommandActorUndefineActor`.

5.* `create actor` id_1 `from` id_2 `at` *`coordinates`* `with course` *`course`* `speed` *`speed`*

Creates an instance of actor family id_2 called actor id_1 at *`coordinates`* with azimuth *`course`* and speed *`speed`*.

Use `CommandActorCreateActor`.

6.* `set` *`id`* `course` *`course`*

Instructs actor *`id`* to change its course to *`course`*.

Use `CommandActorSetCourse`.

7.* `set` *`id`* `speed` *`speed`*

Instructs actor *`id`* to change its speed to *`speed`*.

Use `CommandActorSetSpeed`.

8.* `set` *`id`* `altitude|depth` *`altitude`*

Instructs actor *`id`* to change its altitude or depth to *`altitude`*.

Use `CommandActorSetAltitudeDepth`.

9. `set` id_1 `execute maneuver` id_2 `with (`id_n`+)`

Instructs actor id_1 to execute maneuver id_2, where id_n binds to placeholder fields $*n* in VI.1, starting at 1.

For example, `set mysub execute maneuver surface_and_stop with (mysub)`

Use `CommandActorExecuteManeuver`.

III. MUNITIONS

Munition commands govern the creation, configuration, and usage of munitions, which are carried aboard actors.

1.* `define munition bomb` *`id`*

Defines a bomb family called *`id`*.

Use `CommandMunitionDefineBomb`.

2.* `define munition shell id`

Defines a battery-gun-shell family with called *id*.

Use `CommandMunitionDefineShell`.

3.* `define munition depth_charge` id_1 `with fuze` id_2

Defines a depth-charge family called id_1 with depth-sensor fuze id_2.

Use `CommandMunitionDefineDepthCharge`.

4.* `define munition torpedo` id_1 `with sensor` id_2 `fuze` id_3 `arming time` *time*

Defines a torpedo family called id_1 with sensor id_2, fuze id_3, and arming time *time*.

Use `CommandMunitionDefineTorpedo`.

5.* `define munition missile` id_1 `with sensor` id_2 `fuze` id_3 `arming distance` *distance*

Defines a missile family called id_1 with sensor id_2, fuze id_3, and arming distance *distance*.

Use `CommandMunitionDefineMissile`.

6. `undefine munition id`

Undefines a munition family called *id*.

Use `CommandMunitionUndefineMunition`.

7.* `set` id_1 `load munition` id_2

Instructs actor id_1 to create an instance of munition family id_2, activate its sensor (if applicable), and add it to the live-munition scoreboard, which shows its synthetic id and signal strength with respect to all actors (if applicable).

Use `CommandActorLoadMunition`.

8.* `set` id_1 `deploy munition` id_2

Instructs actor id_1 to deploy loaded munition id_2. This is not applicable to battery-gun munitions.

Use `CommandActorDeployMunition`.

9. `set` id_1 `deploy munition` id_2 `at azimuth` *azimuth* `elevation` *elevation*

Instructs actor id_1 to deploy loaded battery-gun munition id_2 laid at azimuth *azimuth* and elevation *elevation*.

Use `CommandActorDeployMunitionShell`.

IV. SENSORS/FUZES

Sensor commands govern the definition and configuration of sensors, which are carried aboard munitions. Sensors may also serve as fuzes.

1.* `define sensor radar id with field of view fov power power sensitivity sensitivity`

Defines a radar-sensor family called *id* with field of view *fov*, transmission *power*, and sensitivity *sensitivity*. This sensor is actually a transmitter and receiver.

Use `CommandSensorDefineRadar`.

2. `define sensor thermal id with field of view fov sensitivity sensitivity`

Defines a thermal-sensor family called *id* with field of view *fov* and thermal sensitivity *sensitivity*.

Use `CommandSensorDefineThermal`.

3. `define sensor acoustic id with sensitivity sensitivity`

   Defines an acoustic-sensor family called *id* with acoustic sensitivity *sensitivity*.

   Use `CommandSensorDefineAcoustic`.

4.* `define sensor sonar active id with power power sensitivity sensitivity`

   Defines an active-sonar-sensor family called *id* with power *power* and sensitivity *sensitivity*. This sensor is actually a transmitter and receiver.

   Use `CommandSensorDefineSonarActive`.

5. `define sensor sonar passive id with sensitivity sensitivity`

   Defines a passive-sonar-sensor family called *id* with sensitivity *sensitivity*. It receives on the same frequency that the active sensor transmits on.

   Use `CommandSensorDefineSonarPassive`.

6. `define sensor depth id with trigger depth altitude`

   Defines a depth-sensor family called *id* with fuzing depth *altitude*.

   Use `CommandSensorDefineDepth`.

7.* `define sensor distance id with trigger distance distance`

   Defines a distance-sensor family called *id* with fuzing distance *distance*, which fuzes after having traveled this distance.

   Use `CommandSensorDefineDistance`.

8. `define sensor time id with trigger time time`

   Defines a time-sensor family called *id* with fuzing time *time*, which fuzes after having traveled this amount of time.

   Use `CommandSensorDefineTime`.

9. `undefine sensor id`

   Undefines a sensor family called *id*.

   Use `CommandSensorUndefineSensor`.

VI. MANEUVERS

Maneuver commands govern the definition of maneuvers, which are ordered collections of commands.

1. `define maneuver id as ("command`$_n$`"+)`

   Defines a maneuver called *id* based on $command_n$, which are expected to be II.6 through II.8, but actually any command is valid. Identifiers in commands are replaced with placeholders of the form $\$x$, where x is a running number starting with 1 that corresponds to the argument list passed in by II.9.

   For example, `define maneuver surface_and_stop as ("set $1 depth 0" "set $1 speed 0")`

   Use `CommandManeuverDefineManeuver`.

   For Part III, note that my solution does not support this command.

2. `undefine maneuver id`

   Undefines a maneuver called *id*.

   Use `CommandManeuverUndefineManeuver`.

VII. MISCELLANEOUS

Miscellaneous commands govern actions that do not fit into the categories above.

1.* `@load filename`

Loads and executes file *filename* as if the commands were typed at the command line. Each command must be on a separate line.

Use `CommandMiscLoad`.

2.* `@pause`

Pauses the simulation.

Use `CommandMiscPause`.

3. `@resume`

Resume the simulation after a pause.

Use `CommandMiscResume`.

4. `@set update time`

Changes the update rate of the simulation to time *time* per time slice.

Use `CommandMiscSetUpdate`.

5. `@wait time`

Dwells with no action for time *time*.

Use `CommandMiscWait`.

6.* `@force id state to coordinates with course course speed speed`

Forces agent *id* to change its position to *coordinates*, its course to *course*, and its speed to *speed*. This command is for testing only. It applies to actors and munitions.

Use `CommandActorSetState`.

7.* `@exit`

Exits the application.

Use `CommandMiscExit`.

Part II: Unit and Behavioral Testing

This part addresses low and mid-level testing of a standalone component that could define our bomb. There are two aspects:

1. Implement simple unit tests to verify that the boilerplate aspects of the code are correct.
2. Demonstrate the behavior of the bomb under various conditions.

The solution is provided in class `Bomb`.

Description

The bomb falls from an initial position at a speed under wind conditions with error conditions. This is the input. The output is where the bomb hits the ground. The analysis is the error as measured by the horizontal distance from the release position to the impact position.

Creational Elements

The bomb is created at a three-dimensional position defined in terms of (*x*,*y*,*z*), where *z* is altitude, all in meters. The coordinate system is the same as in Task 3.

The bomb is subject to wind from a direction (in standard compass degrees) and speed (in meters per second), which is constant from the release position to the ground (0 meters). Assume the full effect of the wind is imparted onto the bomb.

The bomb has an inherent targeting inaccuracy in terms of a type of error and its severity:

None: There is no error.
Uniform: There is uniform random error.
Gaussian: There is a bell-curve random error.

Uniform and Gaussian errors are on the interval [0, *error_range*) in meters.

Structural Elements

There are no structural elements. This is a bad design because each bomb defines its own wind conditions, which may be inconsistent if multiple bombs are dropped in a single load. More appropriate would be to have a world object that defines the shared wind conditions that act on all bombs equally.

Behavioral Elements

The story is to create a bomb and drop it under various conditions. There are six combinations of configurations: {*no_wind*, *wind*} × {*no_error*, *uniform_error*, *gaussian_error*}.

Unit Tests

Create a class called `BombTest` with three public methods. Import `org.junit.jupiter.api.Assertions.assertEquals`. This should be JUnit 5.*x*.

Method `testWind()` creates a bomb with wind direction 45 and wind speed 10. Use `assertEquals()` to verify that `getWindDirection()` and `getWindSpeed()` return these values. Remember, these values are doubles.

Method `testRelease()` creates a bomb at release position (100,200,300) with descent speed 10. Verify that `getReleaseCoordinates()` is correct for `getX()` and `getY()` and that `getReleaseAltitude()` and `getDescentSpeed()` are correct.

Method `testError()` creates three bombs, each with one of the error types, and all with an error range of 10. Verify `getErrorType()` and `getErrorRange()`.

When you run the test class, it reports the results of each test method or assert (depending on your IDE).

Behavioral Tests

Unit testing is a mechanical process that compares actual results to expected results. It only works when the expected results are known. Since the bomb has nondeterministic (randomized) behavior, we need to verify it in a different way. Here we visually and statistically evaluate the results of dropping represenative bombs with all six configurations by plotting their errors. For each, drop 100 bombs independently. Transfer the 100 (*x*,*y*) impact positions to a spreadsheet and plot them in properly formatted scatterplots. Also report the average error and standard deviation. Finally, indicate whether the results are consistent with expectations. Present everything in a single PDF document.

All configurations release from (200,300,1500) with a descent speed of 100. Configure the rest of the parameters as follows:

Test Name	Error Type	Error Range	Wind Direction	Wind Speed
No Error No Wind	None	0	0	0
No Error With Wind	None	0	60	25
Uniform Error No Wind	Uniform	150	0	0
GaussianError No Wind	Gaussian	150	0	0
Uniform Error With Wind	Uniform	150	60	25
Gaussian Error With Wind	Gaussian	150	60	25

Deliverables

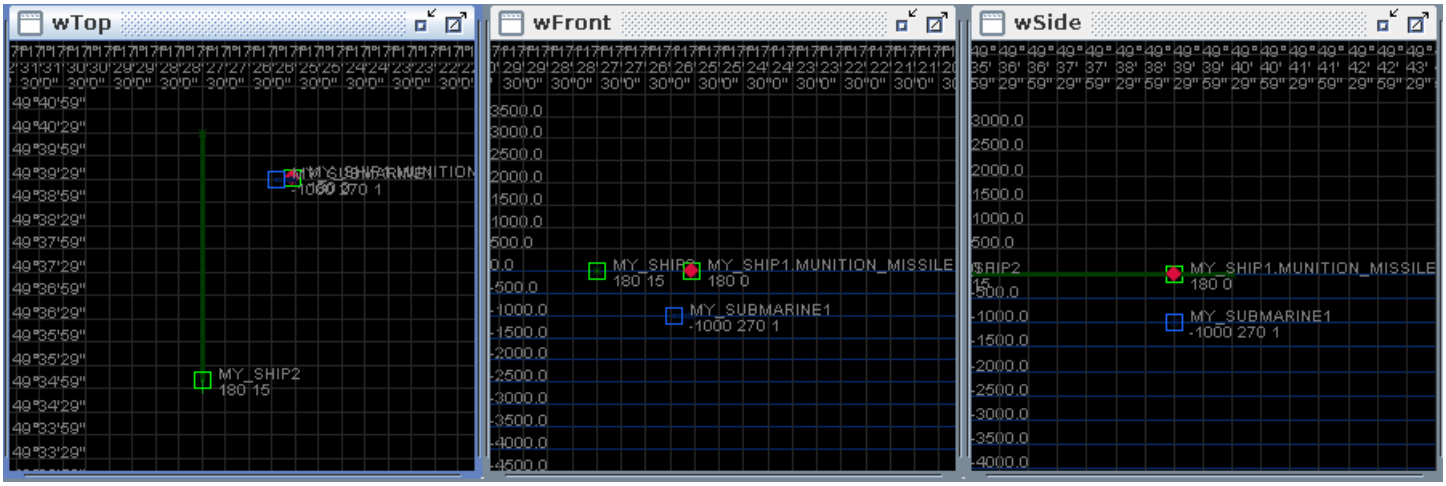
Submit `BombTest.java` and the PDF report.

Part III: 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 unit of functionality 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.

Setup

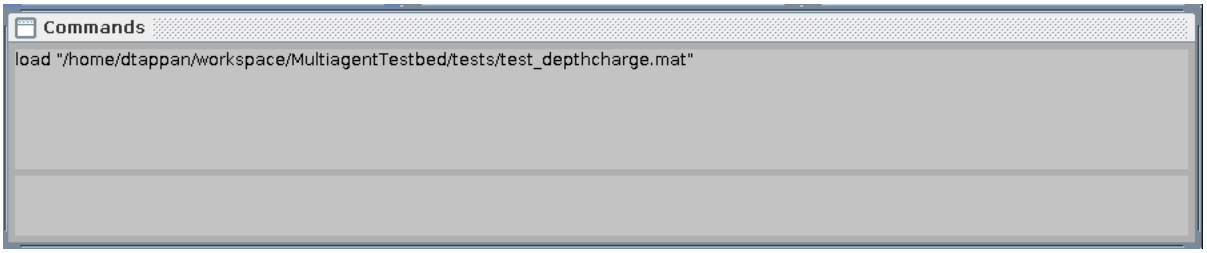
The project jar (0.3 or later) supports Part III as an executable program. When you run it, it will display three views (top, front, and side) over a physical range that is expected to be consistent with all your tests. Since the views are interactive, you can adjust them dynamically with the mouse. It should not be necessary to issue any "create window" commands.



The glyphs represent agents in the world as follows:

- All actors are squares, where airplanes, ships, and submarines are colored orange, green, and blue, respectively. Each indicates its identifier, altitude/depth (except ships), course, and speed.
- Munition glyphs vary. Bombs and depth charges are inverted triangles, colored orange and yellow, respectively. Missiles and torpedoes are diamonds, respectively colored red and teal, and are solid until armed. Shells are not supported.
- All agents have a barb that shows course and projected destination in a few seconds.

In addition to the views, there is a command line interface. The lower pane is for input; the upper is a history of the input. You are expected to write your tests in separate text files, which are executed with the `@load` command.



The log file with execution output is written to `output.csv` wherever you execute the jar. The definition of most fields should be self-explanatory: `command`, `event_num`, `event_group`, `time`, `agent_type`, `agent_id`, `latitude`, `longitude`, `altitude`, `course`, `speed_horizontal`, `speed_vertical`, `deployed`, `armed`, `target_id`, `power_raw`, `power_attenuated`, `distance_elapsed`, `time_elapsed`, `target_bearing`. Raw power is the signal strength without considering how it is attenuated over distance. Aspect angle is discussed below.

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 initial 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 testwithout manual intervention. Do not cross-reference tests; instead, repeat duplicated code.
4. An English narrative of the expected results of executing the test.
5. The actual results with at least one screen shot of the view.
6. A snippet of the actual results from the log file with a supporting explanation. Use a spreadsheet to generate statistics and graphs, if appropriate. For log output, use a clean column format, not the raw CSV. As long as the text is readable when zoomed in, small size is not an issue.
7. A brief discussion on how the actual results differ from the expected results, if they differ.
8. 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 monospace font.

Tests

Test 0 is an example of an expected report entry. It demonstrates more than you may need to do for any single test. My answers do not exactly match the test because I wanted to provide some variety.

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.

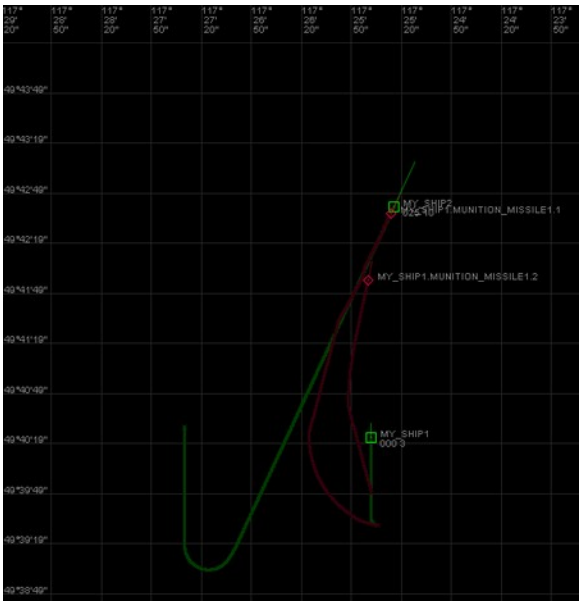
```
define sensor radar FUZE_RADAR1 with field of view 30 power 50 sensitivity 10
define sensor acoustic FUZE_ACOUSTIC1 with sensitivity 10

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.

Log entry 9 at time 8.745 shows the change of course to 25. Entry 13 at time 16.0 shows the munition deploying, which is six seconds later and consistent with the `@wait` command.

event_num	event_group	time	agent_type	agent_id	latitude	longitude	altitude	course	speed_horizontal	speed_vertical	deployed	armed	target_id	power_raw	power_attenuated	distance_elapsed	time_elapsed	target_bearing
8	106	3.71	acoustic	SUB1.TORPEDO1.2.FUZE_ACOUSTIC1.4	49.6586111111111	117.426111111111	0	270	0	0			SHIP1		0	0		
9	107	3.745	ship	SHIP1	49.6748333333333	117.472222222222	0	180	12	0								
10	107	3.745	submarine	SUB1	49.6586111111111	117.426111111111	0	270	0	0								
11	107	3.745	missile	SUB1.TORPEDO1.1	49.6586111111111	117.426111111111	0	270	0	0	false	false						
12	107	3.745	missile	SUB1.TORPEDO1.2	49.6586111111111	117.426111111111	0	270	0	0	false	false						
13	107	3.745	radar	SUB1.TORPEDO1.1.FUZE_RADAR1.1	49.6586111111111	117.426111111111	0	270	0	0			SHIP1	47.1662784671449	12.5772757712328			289.382240592817
14	107	3.745	radar	SUB1.TORPEDO1.2.FUZE_RADAR1.3	49.6586111111111	117.426111111111	0	270	0	0			SHIP1	47.1662784671449	12.5772757712328			289.382240592817

7.

The actual results are consistent with the expected results.

8.

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.)

Munition Deployment

Part II of the project was originally expected to investigate creating, loading, and deploying munitions. Since we did not do it this way, we did not talk about it. However, you need to have a basic understanding of how this process works for Part III.

The example above works as follows:

```
define ship ACTOR_SHIP1 with munition (MUNITION_MISSILE1)
```

This creates ship instance `ACTOR_SHIP1` with munition family `MUNITION_MISSILE1` on board. This is an unlimited supply. Think of it as a Java class.

```
set MY_SHIP1 load munition MUNITION_MISSILE1
```

This makes munition instance `MY_SHIP1.MUNITION_MISSILE1.1` from munition family `MUNITION_MISSILE1`. Each subsequent load increments the final digit to differentiate the instances. It is possible to create any number of instances. They activate at this point, so sensors and fuzes report their state to the log file. Think of this as instantiating the Java class.

```
set MY_SHIP1 deploy munition MY_SHIP1.MUNITION_MISSILE1.1
```

This fires munition instance `MY_SHIP1.MUNITION_MISSILE1.1` from ship instance `MY_SHIP1`. In the object-oriented compositional view, this is instance `1` of `MUNITION_MISSILE1` from `MY_SHIP1`.

Miscellaneous

Only active sonar is supported. Acoustic and thermal signatures increase based on the speed of the target. Radar performance depends on the aspect angle, which is the angle between the sensor or fuze and the target: broadside is strongest; inline is weakest. Attenuation is the degradation of signal strength over distance.

Expect to use values in the 1 to 20 range for power and sensitivity. Power is the output from a sensor or fuze, and sensitivity is the threshold on the reflected signal for determining whether the sensor or fuze sees the target.

Speeds seem to be most manageable between 0 and 10.

None of these numbers have a legitimate correspondence to their real-world counterparts, so do not be concerned with realistically grounded performance. Your tests should just look reasonable. We do this because real-world values are too complicated and messy, and components move too fast and far to keep track of in our windows reasonably.

Execute the jar on Linux with `java -jar cs350-project-v0_x.jar`, where *x* is the latest version number. On Windows, you can double-click it directly. However, there is no text output this way. It is safer to open a Windows command line and execute the same statement. You can also execute it from your IDE as in Part I.

Choose three tests from each group for a total of 15.

For Part III only, the command protocols are slightly different from Part I:

Field	Definition	Examples	Datatype
coordinates	*latitude/longitude/altitude*	45*30'15#/110*30'10#/200	CoordinateWorld3D
filename	"filename" with normal filename punctuation	"/stu/myfilename.txt"	String
latitude	*degrees*minutes'seconds#*	45*30'15#	Latitude
longitude	*degrees*minutes'seconds#*	110*30'10.3#	Longitude

Airplane Tests

Test 1: Airplane Straight-and-Level Flight

Fly an airplane on a constant course at a constant altitude.

Test 2: Airplane Climbing

Fly an airplane on a constant course in a climb.

Test 3: Airplane 360-Degree Turn

Fly an airplane in a 360-degree clockwise turn approximated by an octagon.

Test 4: Airplane Climbing 360-Degree Turn, Maximum-Performance

Fly an airplane in a 360-degree clockwise turn approximated by an octagon in a climb. Excel plots are required.

Test 5: Airplane Climbing 360-Degree Turn, Stepped

Fly an airplane in a 360-degree clockwise turn approximated by an octagon in a climb where each leg of the octagon alternates between level flight and climbing. Excel plots are required.

Test 6: Airplane 180-Degree Turn, Slow-Speed

Fly an airplane in a 180-degree left turn at a slow speed.

Test 7: Airplane 180-Degree Turn, High-Speed

Fly an airplane in a 180-degree left turn at a high speed.

Bomb Tests

Test 8: Bomb Drop, High Speed

Drop a bomb from a high-speed airplane at 8,000 feet onto a ship.

Test 9: Bomb Drop, Low Speed, Hit

Drop a bomb from a low-speed airplane at 8,000 feet onto a ship.

Test 10: Bomb Drop, Low Speed, Miss

Drop a bomb from a low-speed airplane at 8,000 feet into the water.

Depth-Charge Tests

All depth charges are dropped by a ship.

Test 11: Depth Charge, Acoustic Fuze, Hit

Drop a depth charge with an acoustic fuze onto a submarine.

Test 12: Depth Charge, Acoustic Fuze, Miss

Drop a depth charge with an acoustic fuze near a submarine, but miss.

Test 13: Depth Charge, Depth Fuze

Drop a depth charge with a depth fuze.

Test 14: Depth Charge, Sonar Fuze

Drop a depth charge with a sonar fuze.

Test 15: Depth Charge, Time Fuze

Drop a depth charge with a time fuze.

Missile Tests

Test 16: Missile, Radar Sensor, Depth Fuze

Fire a missile with a radar sensor and depth (altitude) fuze from a ship at an airplane, detonating near the airplane.

Test 17: Missile, Radar Sensor, Distance Fuze

Fire a missile with a radar sensor and distance fuze from a ship at an airplane, detonating near the airplane.

Test 18: Missile, Radar Sensor, Radar Fuze

Fire a missile with a radar sensor and radar fuze from a ship at an airplane.

Test 19: Missile, Radar Sensor, Thermal Fuze

Fire a missile with a radar sensor and thermal fuze from a ship at an airplane.

Test 20: Missile, Radar Sensor, Time Fuze

Fire a missile with a radar sensor and time fuze from a ship at an airplane, detonating near the airplane.

Test 21: Missile, Thermal Sensor, Radar Fuze

Fire a missile with a thermal sensor and radar fuze from an airplane at a ship.

Test 22: Missile, Thermal Sensor, Radar Fuze, Field-of-View Miss

Fire a missile with a thermal sensor and radar fuze from an airplane at a ship. Redirect the ship to move it out of the field of view of the missile and break lock.

Test 23: Missile, Radar Sensor, Radar Fuze, Aspect Angle

Fire a missile with a radar sensor and radar fuze from an airplane at a ship. Move the ship in such a way that the radar signal strength goes from maximum to minimum and back to maximum as a function of aspect angle.

Test 24: Missile, Thermal Sensor, Thermal Fuze

Fire a missile with a thermal sensor and thermal fuze from an airplane at a ship.

Torpedo Tests

Test 25: Torpedo, Sonar Sensor, Acoustic Fuze, Fast Target

Fire a torpedo with a sonar sensor and acoustic fuze from a submarine at a fast ship.

Test 26: Torpedo, Sonar Sensor, Acoustic Fuze, Slow-Target Miss

Fire a torpedo with a sonar sensor and acoustic fuze from a submarine at a slow ship. Miss the ship.

Test 27: Torpedo, Sonar Sensor, Sonar Fuze

Fire a torpedo with a sonar sensor and sonar fuze from a submarine at a fast ship.

Test 28: Torpedo, Acoustic Sensor, Acoustic Fuze, Fast Target

Fire a torpedo with an acoustic sensor and acoustic fuze from a submarine at a fast ship.

Test 29: Torpedo, Acoustic Sensor, Acoustic Fuze, Slow-Target Miss

Fire a torpedo with an acoustic sensor and acoustic fuze from a submarine at a slow ship. Miss the ship.

Test 30: Torpedo, Acoustic Sensor, Thermal Fuze

Fire a torpedo with an acoustic sensor and thermal fuze from a submarine at a fast ship.