CS 350 Project Specifications v0.2

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`.