CS 350 Project Specifications, Version 0.3 (Parts 1 and 2)

Description

This living document describes the team elements of the project:

- Part 1 is to build the parser for the command line interface that allows a user to create, connect, and manipulate a subset of the aspects of the project. This is the main programming effort.
- Part 2 is to demonstrate through system testing how well the complete, provided solution works. This is a written submission. It will be defined later.

Our architecture will be covered in detail in lecture. No changes are expected, but always make sure that you are using the most current version of these documents and the code.

This document serves as both design and usage instructions. As such, you must determine who the audience is from context. For example, a lot of error checking is already handled by the architecture. When the instructions state that something must or must not happen, it may or may not be necessary for you as the programmer to enforce this. You need to check by running basic tests. In any case, the user (and tester) has to know and follow the correct usage instructions. Eventually this document would be separated by audience into a development manual and a users manuals, but since you are playing all roles, the combined form is more appropriate here.

Remember to read, understand, plan, execute, verify, and reflect. Do not throw code at the problem or architecture.

PART 1

See the spreadsheet that accompanies this document for the commands that play the creational, structural, and behavioral roles in the project.

Definitions

The commands reference the following grammatical fields, the type of each being a Java primitive or a class in `cs350f20project.datatype`. Single quotes enclose literals but are not part of the input.

Field	Description	Datatype
`angle`	`number`	`Angle`
`coordinates_delta`	`number ':' number`	`CoordinatesDelta`
`coordinates_world`	`latitude '/' longitude`	`CoordinatesWorld`
`id`	*standard Java variable name, underscore included*	`String`
`integer`	`[-|+]` *integer value*	`int`
`latitude`	`integer '*' integer ''' number '"'`	`Latitude`
`string`	*ordinary Java string delimited by single quotes, no escape characters*	`String`
`longitude`	`integer '*' integer ''' number '"'`	`Longitude`
`number`	`( integer | real )`	`double`
`real`	`[-|+]` *real value, leading zero required*	`double`

Commands

Parentheses, square brackets, and angle brackets are not part of commands. Vertical bar indicates logical *or*; asterisk indicates zero or more instances of the preceding term or parenthetical group; plus indicates one or more. Question mark or square brackets indicate an optional group.

White space (spaces and tabs), except in literals, does not matter. Carriage returns are not allowed because all commands reside on a single line.

All text except identifiers is case insensitive.

All identifiers definitions must be unique.

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

All failure modes not already handled must be accounted for appropriately. The messages and delivery mechanism need not be elaborate or particularly user-friendly. But do not allow a crash.

Architecture

Setup

The architecture is provided in a JAR file on the task link. Always use the most recent version (which may differ from this document version). Before doing anything with your part, ensure that the architecture starts and runs correctly. From the command line of your operating system, execute: `java -jar cs350-project-v01.jar`

You should see this on standard output:

```
[CONTROLLER] initiated
I> 
```

The `I>` prompt is the command-line input interface for the system. It accepts any command from the spreadsheet. At this point, however, you do not know how to use these commands appropriately (i.e., their semantics and pragmatics, which we will discuss). As a simple test, enter `@WAIT 5 ; @EXIT`

It should wait five seconds, then output the following and terminate:

```
O> good-bye
```

The `O>` prompt indicates output from the architecture.

Now configure Eclipse to reference the JAR. (You may use another IDE, but you are on your own to determine how to do this part because Eclipse is the standard IDE in the curriculum.) Right-click on your project in the Package Explorer pane, select *Build Path* → *Configure Build Path*, and click *Add External JARs*.

Implementation

Your job is to build a parser that accepts the subset of the commands in blue from the spreadsheet. (Teams of two omit the ones with an asterisk.) Recognize each rule from its definition, extract any field contents, create the specified command object from `cs350f20project.controller.command`, and submit it to the action processor for execution as specified below. Rule 1 is the entry point.

Part 1 addresses syntax; Part 2 will address semantics and pragmatics. In other words, for Part 1 your input must be syntactically valid, but it need not make sense in terms of performing a reasonable action the way the user would to solve an actual problem with our product. Your commands will likely crash the architecture (if you submit them, as defined below) because they are probably semantically invalid. As long as it is not your parser that is crashing, this behavior is acceptable.

Define your parser as at least class `cs350f20project.controller.cli.parser.CommandParser` with constructor `CommandParser(MyParserHelper parserHelper, String commandText)`. The command text is the string representation of any command-line input from the rules. The parser helper is a combination of the supplied `A_ParserHelper` and whatever you might need stored in your version of subclass `MyParserHelper`. You are not required to use this class yourself, but it must be passed into your parser because it contains the `ActionProcessor` to which you submit your command through `schedule(A_Command command)`.

Instantiate class `cs350f20project.Startup` and call its `go()` method to start the system. Your parser should replace mine. If you ever see `### ACCESSING MY CommandParser ###`, something is wrong with your project configuration.

Instantiation of `CommandParser` builds and configures a parser (data and control), but it does not execute it (behavior). For this you must implement and call public method `void parse()`. No other methods from the Javadoc are necessary here. Example execution looks like this from caller's perspective:

```
MyParserHelper parserHelper = new MyParserHelper();
String commandText = "@exit";
CommandParser parser = new CommandParser(parserHelper, commandText);
parser.parse();
// the command action occurs
```

Within your `CommandParser`, execution would look something like this:

```
public void parse()
{
   if (this.commandText.equalsIgnoreCase("@exit"))
   {
      A_Command command = new CommandMetaDoExit();
      this.parserHelper.getActionProcessor().schedule(command);
   }
}
```

Do not use any other resources from the JAR or elsewhere except those specified here.

This solution must be in standard Java (any recent version). No external libraries or packages or tools are allowed.

PART 2

This final part of the project addresses system testing and evaluation of a subset of the provided solution. It focuses on breadth of coverage, 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 of times the size of this document.

Deliverable

You need to produce one document with all your tests. Tests are stated in the form of 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 designator and title in bold; e.g., **Test B.1: Power network**

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 test without manual intervention. Do not crossreference other tests.
4. A brief English narrative of the expected results of executing the test. (Proper testing discipline demands that you do this *before* running the test.)
5. At least one graphical representation of the actual results. The form is your choice. See the log output.
6. A brief discussion on how the actual results differ from the expected results.
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 monospaced font.

Tests

Each test is independent. Not all tests may work as expected. Teams of two omit A.3, A.4, B.1, C.12, and C.14.

A. Track Tests

Test A.1: Oval layout

Build an oval track.

Test A.2: Oval layout with passing loop

Build an oval track with a passing loop.

Test A.3: Oval layout with dead-end siding

Build an oval track with a dead-end siding.

Test A.4: Oval layout with roundhouse siding

Build an oval track with a roundhouse siding that has three engine houses.

B. Infrastructure Tests

Test B.1: Power network

Create a power network with one power station connected to two substations, the second of which supplies a catenary of at least four poles.

Test B.2: Semaphore

Demonstrate that the stop, caution, and proceed states of a semaphore work. train's response.

Test B.3: Signal light

Demonstrate that the stop and proceed states of a signal light work. response.

C. Stock Tests

Test C.1: Speed validation

Accelerate a diesel train up to 60 kph, reduce its speed to 30 kph, then brake.

Test C.2: Collision 1

Force two trains to collide head-on. Make it cool.

Test C.3: Collision 2

Force a train to collide with another from behind.

Test C.4: Derailment 1

Show that a train cannot enter a switch that is out of position.

Test C.5: Derailment 2

Derail a train that is partially over a switch.

Test C.6: Derailment 3

Derail a train by running it off an open drawbridge.

Test C.7: Direction changing 1

Bring a train running at 20 kph to a stop, then reverse its direction to 30 kph.

Test C.8: Direction changing 2

Reverse a train running at 20 kph without stopping it.

Test C.9: Uncoupling 1

Bring a train running at 20 kph to a stop, then uncouple it. Use one of each car type.

Test C.10: Uncoupling 2

Uncouple a train running at 20 kph without stopping it. Use one of each car type.

Test C.11: Power loss.

Demonstrate that an electric engine must be on a track segment with a catenary.

Test C.12: Roundhouse origination

Start an engine in the roundhouse in A.4 and bring it into operation on the oval.

Test C.13: Passing operations 1

Run two trains of six cars each on A.2 in opposite directions without colliding.

Test C.14: Passing operations 2

Run two trains of six cars each on A.3 in opposite directions without colliding.

D. View Tests

Test D.1: Orient to north

Show that a layout is oriented north with respect to an engine.

Test D.2: Orient to track

Show that a layout is oriented forward with respect to an engine.