

This living document specifies the communication interface that interprets English-like text commands to configure and manipulate the architecture. It guides you in the role of programmer for development in Part I and quasi end user for test and evaluation in Part II (previously referred to as Part III). 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 some of the decisions.

The Javadoc will be posted shortly. Initially it documents only what is needed here, but the full documentation should be available soon to help understand what the solution consists of. We already know what we started with in April:

Mechatronics Virtual Testbed

The customer wants a simulation toolkit with a basic user interface to create, configure, and manipulate a hierarchical network of virtual components comprising actuators for semi-realistic movement from one position to another and sensors for monitoring and reporting their states to support conceptual performance analysis of simplistic real-world engineering systems.

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

Part I: Command Parser

Your team is implementing most of the parser that takes text input from the user and instructs the architecture to perform corresponding actions. Before you do anything with your solution, make sure the provided solution runs correctly in your environment.

The first step is to connect the JAR to your own new project. Save the JAR somewhere related to your project. To connect to it in Eclipse, right-click on the project, then select Build Path→Configure Build Path. In the Libraries tab, select Classpath, click Add External JARs, and select the project JAR. In IntelliJ, select File→Project Structure. In the Modules tab, Dependencies tab, click the + symbol, select JARs or Directories, and select the project JAR. If a new JAR is posted, remove the current one from your project and repeat this process with the new one.

When you run your project (with no other contents), you are executing the Startup class in the JAR. You should see this statement appear to standard output:

```
STARTUP
PARSE> @CONFIGURE LOG "a.txt" DOT SEQUENCE "b.txt" NETWORK "c.txt" XML "d.txt"
*** RUNNING SOLUTION PARSER IN ARCHITECTURE ***
PARSE> @exit
@EXIT
```

Now add Startup.java to your project (from the task link). This is how to start the project for Part I. You can also put tests here or anything else you want.

Run the project with Startup as the main(). It should use the parser from the JAR still, and the asterisk statement above should still appear.

The next step is to create your own parser class as defined here. Build the skeleton and make sure it works before adding functionality. The class is called Parser in package cs350s22.component.ui.parser.

The constructor takes an A_ParserHelper and the string containing the command to parse:

```
public Parser(A_ParserHelper parserHelper, String commandText)
```

The parse method parses the command text and performs the actions specified in Architecture Binding for the command:

```
public void parse() throws IOException
```

Class Startup shows how to set up and use this code. There is no interactivity in this form.

Run the project again. It should use your parser now, and the asterisk statement should not appear. If it still does, then something is not configured properly.

Each command instantiates a new parser. Therefore, you cannot store anything in your parser that needs to be retained across commands. Class `ParserHelper` already provides most of what your solution needs for this storage. If you need anything else, add the code to `ParserHelper.java` provided and make sure it is in the correct place in your project for the package.

This class provides symbol tables for all components you need to keep track of. A symbol table is simply a hash map that uses the identifier of a component as the key and the component itself as the value. In general, `CREATE` commands put something into a symbol table, and `id` fields within commands get something out of one or more symbol tables.

Description

Commands are presented here in the form of uppercase keywords (although they are not case sensitive) and lowercase fields that require some entry from the user.

- `id` is a standard Java identifier, including underscore (e.g., `myID1`). Put its string contents into an `Identifier` object.
- `value` is a standard signed real or integer value (e.g., `1`, `3.14`, `-5`). Put its numerical contents into a Java double or integer as necessary.
- `string` is a standard string delimited by double quotes. No escape characters are supported. Avoid the backslash character in file paths because it may be misinterpreted by your operating system.

Except for command `B3`, punctuation is 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. Parentheses show grouping. Subscripts appear for reference in the text.

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

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

All identifiers must be unique. The symbol tables already enforce this constraint.

Commands that specify the recipient or recipients of an action have two options based on two lists of identifiers.

- `ids` is defined as `ID[S] id+`
- `groups` is defined as `GROUP[S] id+`

All components that can receive messages have an identifier. If this identifier is specified in `ids`, the component will be called directly (think email address). These components can also be a member of one or more groups, which are also specified by identifiers. If the group is called by its identifier, all members of the group are subsequently called automatically (think mailing list).

Wherever `[ids]` and `[groups]` appear together, at least one identifier is required in at least one of the two lists.

Keywords with singular and plural forms, like `COMPONENT[S]`, need not align with the list count grammatically.

The architecture does most of the error checking on the results of a command that your parser parses. Your parser needs to throw a `RuntimeException` if it cannot parse the command. The messages and delivery mechanism need not be elaborate or particularly user-friendly.

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

All tests must start with the `@CONFIGURE` command. See class `Startup`.

Implement the following commands: `A1`, `C1`, `C2`, `C3`, `C4`, `D1`, `D2`, `D3`, `E1`, `E2`, `E3`, `E4`, `E5`, `E6`, `F1`, `G1`, `G2`, `H1`, `I1`, `I2`, `I3`. Teams of two omit all `E` and `F` commands.

A. Actuator Commands

The actuator command is responsible for creating an actuator and optionally connecting sensors to it. This component is very similar to the proof of concept in Task 4.

```
1. CREATE ACTUATOR (LINEAR | ROTARY) id1 [groups] [SENSOR[S] id2+] ACCELERATION LEADIN value1 LEADOUT value2 RELAX value3 VELOCITY LIMIT value4 VALUE MIN value5 MAX value6 INITIAL value7 JERK LIMIT value8
```

Creates an actuator with identifier id_1 with optional membership in groups and optional embedded sensors id_2 . The other arguments are:

- value₁ The acceleration when the actuator starts moving
- value₂ The deceleration when the actuator stops moving as it arrives normally at the target value
- value₃ The deceleration when the actuator stops moving when it is commanded to relax and come to a stop now
- value₄ The maximum velocity in either direction
- value₅ The minimum value the actuator can assume
- value₆ The maximum value the actuator can assume
- value₇ The initial value the actuator assumes
- value₈ The acceptable abruptness of changes in velocity as a result of being commanded to a new target value

Example:

```
CREATE ACTUATOR LINEAR myActuator0 ACCELERATION LEADIN 0.1 LEADOUT -0.2 RELAX 0.3 VELOCITY LIMIT 5  
VALUE MIN 1 MAX 10 INITIAL 2 JERK LIMIT 3  
CREATE ACTUATOR ROTARY myActuator8 SENSORS mySensor3 ACCELERATION LEADIN 0.1 LEADOUT -0.2 RELAX 0.3  
VELOCITY LIMIT 5 VALUE MIN 1 MAX 10 INITIAL 2 JERK LIMIT 3
```

Architecture Binding:

Get the sensors by calling `get()` with `id` on `SymbolTable<A_Sensor>`, then create an `ActuatorPrototype` object with the arguments and add it to `SymbolTable<A_Actuator>`.

B. Controller Commands

Controller commands are responsible for creating a subnetwork.

```
1. CREATE CONTROLLER (FORWARDING | NONFORWARDING) id1 [groups] [DEPENDENCY SEQUENCER id2]
   WITH COMPONENT[S] id3+
```

Creates controller id_1 and connects components id_3 and an optional dependency sequencer id_2 . A forwarding controller automatically forwards incoming messages to its components. A nonforwarding controller does not and therefore needs specific Java code to define its behavior. We are not doing this.

Example:

```
CREATE CONTROLLER FORWARDING myController1 WITH COMPONENTS myActuator1
CREATE CONTROLLER NONFORWARDING myController2 GROUPS myControllerGroup1 DEPENDENCY SEQUENCER
  myDependencySequencer2 WITH COMPONENTS myActuator8
```

Architecture Binding:

Get the components by calling `get()` with `id` and the `true` argument on `SymbolTable<A_Controller>`, `SymbolTable<A_Actuator>`, and `SymbolTable<A_Sensor>`, then add the components to a `MyControllerSlaveForwarding` or `MyControllerSlaveNonforwarding` object with its `addComponents()` method.

```
2. CREATE DEPENDENCY SEQUENCER id1 SEQUENCE[S] id2+
```

Creates dependency sequencer id_1 from dependency sequences id_2 . A sequencer is a collection of logical statements (dependency sequences) that are evaluated in order. For example, if the first sequence of two is waiting on a message with identifier `myMessage1`, it does nothing until `myMessage1` is received. The second sequence then becomes active. When all sequences have been processed, the first becomes active again.

Example:

```
CREATE DEPENDENCY SEQUENCER myDependencySequencer2 SEQUENCES myDependencySequence2
  myDependencySequence3
```

Architecture Binding:

Do not implement this command. It needs B.3, which we are not implementing.

```
3. CREATE DEPENDENCY SEQUENCE id1 EXPRESSION expression
   where expression is ( '(' expression (AND | OR) expression ')' ) | id2
```

Creates dependency sequence id_1 out of the recursive logical expression `expression`. This evaluates to true when a message with identifier id_2 is received by the controller.

The parentheses in quotes are required (but not the quotes). There does not need to be whitespace around them.

Example:

```
CREATE DEPENDENCY SEQUENCE myDependencySequence1 EXPRESSION myMessage1
CREATE DEPENDENCY SEQUENCE myDependencySequence4 EXPRESSION ((myMessage1 AND myMessage2) OR
  (myMessage3 AND (myMessage4 OR myMessage5)))
```

Architecture Binding:

Do not implement this command. It requires an understanding of context-free grammars, which is beyond the scope of this project or course.

C. Mapper Commands

Mapper commands are responsible for creating mappers that modify the raw value reported directly by a sensor.

1. CREATE MAPPER id EQUATION PASSTHROUGH

Creates mapper id that does not remap the raw value. This is equivalent to `CREATE MAPPER id EQUATION SCALE 1`.

Example:

```
CREATE MAPPER myMapper EQUATION PASSTHROUGH
```

Architecture Binding:

Create a new `EquationPassthrough` object, provide it to a new `MapperEquation` object, and add the latter to `SymbolTable<A_Mapper>`.

2. CREATE MAPPER id EQUATION SCALE value

Creates mapper id that remaps the raw value by the linear coefficient value.

Example:

```
CREATE MAPPER myMapper EQUATION SCALE 10
```

Architecture Binding:

Create a new `EquationScaled` object with value, provide it to a new `MapperEquation` object, and add the latter to `SymbolTable<A_Mapper>`.

3. CREATE MAPPER id EQUATION NORMALIZE value₁ value₂

Creates mapper id that remaps the raw value onto a percentage scale as defined by the lower limit value₁ and upper limit value₂.

Example:

```
CREATE MAPPER myMapper EQUATION NORMALIZE 10 20
```

Architecture Binding:

Create a new `EquationNormalized` object with value₁ and value₂, provide it to a new `MapperEquation` object, and add the latter to `SymbolTable<A_Mapper>`.

4. CREATE MAPPER id INTERPOLATION (LINEAR | SPLINE) DEFINITION string

Creates mapper id that remaps the raw value based on the comma-delimited interpolation table defined in string. Each row defines a point in the two-dimensional graph. The first value is the raw sensor value; the second is its mapped value. LINEAR mode does linear interpolation; SPLINE does a smoother nonlinear interpolation.

Examples:

```
CREATE MAPPER myMapper INTERPOLATION LINEAR DEFINITION "mapfile.map"
```

```
CREATE MAPPER myMapper INTERPOLATION SPLINE DEFINITION "C:/temp/definition.map"
```

Architecture Binding:

Create a new `MapLoader` object with filename, call its `load()` method to get an `InterpolationMap`, provide the latter to a new `InterpolatorLinear` or `InterpolatorSpline` object, provide the latter to a new `MapperInterpolation` object, and add the latter to `SymbolTable<A_Mapper>`.

D. Message Commands

Message commands are responsible for sending messages from the master controller at the top-level network to its components or components in subnetworks.

1. SEND MESSAGE PING

Sends a ping message to the master controller, which propagates it recursively throughout the network regardless of whether controllers are forwarding or nonforwarding.

Example:

```
SEND MESSAGE PING
```

Architecture Binding:

TBD

2. SEND MESSAGE [ids] [groups] POSITION REQUEST value

Sends a request message to the recipients such that their position is expected to go to value, if possible. Inappropriate requests (for example, to a sensor) are ignored.

Example:

```
SEND MESSAGE ID myActuator1 POSITION REQUEST 10
```

```
SEND MESSAGE GROUPS myActuators1 myActuators2 POSITION REQUEST 20
```

```
SEND MESSAGE ID myActuator1 GROUPS myActuators1 myActuators2 POSITION REQUEST 30
```

Architecture Binding:

TBD

3. SEND MESSAGE [ids] [groups] POSITION REPORT

Sends a request message to the recipients such that they report their current value, if possible. Inappropriate requests (for example, to a controller) are ignored.

Example:

```
SEND MESSAGE ID myActuator1 POSITION REPORT
```

```
SEND MESSAGE GROUPS myActuators1 myActuators2 POSITION REPORT
```

```
SEND MESSAGE ID myActuator1 GROUPS myActuators1 myActuators2 POSITION REPORT
```

Architecture Binding:

TBD

E. Meta Commands

Meta commands are responsible for configuring and manipulating the architecture itself, not the network it is executing.

1. @CLOCK (PAUSE | RESUME)

Pauses or resumes automated updating by the clock.

Example:

```
@CLOCK PAUSE
```

```
@CLOCK RESUME
```

Architecture Binding:

Get the system clock through `Clock.getInstance()` and call its `isActive()` method with the specified state.

2. @CLOCK ONESTEP [count]

Updates the clock manually either once or optionally count times. This is valid only while the clock is paused.

Example:

```
@CLOCK ONESTEP
```

```
@CLOCK ONESTEP 5
```

Architecture Binding:

Get the system clock through `Clock.getInstance()` and call its appropriate `onestep()` method with count.

3. @CLOCK SET RATE value

Sets the clock rate `value` in milliseconds per update.

Example:

```
@CLOCK SET RATE 20
```

Architecture Binding:

Get the system clock through `Clock.getInstance()` and call its `setRate()` method with `value`.

4. @EXIT

Exits the system. This must be the last statement; otherwise, log files may not be complete.

Example:

```
@EXIT
```

Architecture Binding:

Call `exit()` in `ParserHelper`.

5. @RUN string

Loads and runs the script in fully qualified filename `string`.

Example:

```
@RUN "myfilename.mvt"
```

Architecture Binding:

Call `run()` in `ParserHelper` with `string`.

```
6. @CONFIGURE LOG string1 DOT SEQUENCE string2 NETWORK string3 XML string4
```

Defines where the output goes for logging and reporting. This must be the first command issued.

Example:

```
@CONFIGURE LOG "tbd" DOT SEQUENCE "tbd" NETWORK "tbd" XML "tbd"
```

Architecture Binding:

TBD

F. Network Commands

The network command is responsible for creating the top-level network from actuators, sensors, and/or controllers. The network is the top-level network. It automatically provides in ParserHelper its own forwarding controller MyControllerMaster called myControllerMaster.

1. BUILD NETWORK WITH COMPONENT[S] id+

Creates the network with components id.

Example:

```
BUILD NETWORK WITH COMPONENT myController
BUILD NETWORK WITH COMPONENTS myController myActuator
```

Architecture Binding:

Get the components by calling `get()` with `id` and the `true` argument on `SymbolTable<A_Controller>`, `SymbolTable<A_Actuator>`, and `SymbolTable<A_Sensor>`, then add the components to the master controller from `getControllerMaster()` in `ParserHelper` with its `addComponents()` method.

G. Reporter Commands

Reporter commands are responsible for creating reporters. The job of a reporter is to inform recipients of the value of a sensor based on a trigger event.

1. CREATE REPORTER CHANGE id NOTIFY [ids] [groups] DELTA value

Creates reporter id that informs the recipients when the sensor value has changed by at least value.

Example:

```
CREATE REPORTER CHANGE myReporter1 NOTIFY IDS myActuator1 myActuator2 DELTA 3
```

Architecture Binding:

Create a new ReporterChange object with ids, groups, and value and add it to SymbolTable<A_Reporter>.

2. CREATE REPORTER FREQUENCY id NOTIFY [ids] [groups] FREQUENCY value

Creates reporter id that informs the recipients every value updates.

Example:

```
CREATE REPORTER FREQUENCY myReporter6 NOTIFY IDS myActuator1 myActuator2 GROUPS myGroup3 FREQUENCY 4
```

Architecture Binding:

Create a new ReporterFrequency object with ids, groups, and value and add it to SymbolTable<A_Reporter>.

H. Sensor Commands

The sensor command is responsible for creating a sensor with optional reporters, optional watchdogs, and an optional mapper.

```
1. CREATE SENSOR (SPEED | POSITION) id1 [groups] [REPORTER[S] id2+] [WATCHDOG[S] id3+] [MAPPER id4]
```

Creates a sensor that reports either the speed or position of the actuator it monitors. Position comes from the actuator. Speed is based on change in position with respect to change in time.

An optional mapper maps the raw value of the sensor to its mapped value. If no mapper is provided, the mapped value is the same as the raw value.

Optional reporters send messages at specified times with the mapped value of the sensor.

Optional watchdogs verify that the mapped value of the sensor is in compliance.

Example:

```
CREATE SENSOR POSITION mySensor8 GROUP myGroup1 REPORTERS myReporter1 MAPPER myMapper1
CREATE SENSOR POSITION mySensor16 GROUP myGroup1 REPORTERS myReporter1 WATCHDOGS myWatchdog1
    myWatchdog2 MAPPER myMapper1
```

Architecture Binding:

If reporters, watchdogs, or a mapper are specified, get them from `SymbolTable<A_Reporter>`, `SymbolTable<A_Watchdog>`, or `SymbolTable<A_Mapper>`, respectively. Create a new **MySensor** object with the arguments provided and add it to `SymbolTable<A_Sensor>`.

I. Watchdog Commands

Watchdog commands are responsible for monitoring the value of a sensor and reporting whether constraints are violated. All commands support these modes for mode:

MODE (INSTANTANEOUS | (AVERAGE [value₁]) | (STANDARD DEVIATION [value₂]))

- INSTANTANEOUS uses the current value. Use WatchdogModeInstantaneous.
- AVERAGE uses the average of the last value₁ values or all values if value₁ is omitted. Use WatchdogModeAverage.
- STANDARD DEVIATION uses the standard deviation of the last value₂ values or all values if value₂ is omitted. Use WatchdogModeStandardDeviation.

The optional grace value defines how many continuous violations are allowed before reporting. Omitting it reports the first violation.

```
1. CREATE WATCHDOG ACCELERATION id mode THRESHOLD LOW value1 HIGH value2 [GRACE value3]
```

Creates a watchdog that monitors the acceleration of a value. The watchdog triggers if the acceleration is less than value₁ or greater than value₂.

Example:

```
CREATE WATCHDOG ACCELERATION myWatchdog1 MODE INSTANTANEOUS THRESHOLD LOW 1 HIGH 3
CREATE WATCHDOG ACCELERATION myWatchdog2 MODE AVERAGE THRESHOLD LOW 1 HIGH 3 GRACE 4
```

Architecture Binding:

Create a WatchdogAcceleration object with value₁, value₂, mode, and optionally value₃ and add it to SymbolTable<A_Watchdog> with id.

```
2. CREATE WATCHDOG (BAND | NOTCH) id mode THRESHOLD LOW value1 HIGH value2 [GRACE value3]
```

Creates a watchdog that monitors a value. For BAND, the watchdog triggers if the value is less than value₁ or greater than value₂. For NOTCH, the watchdog triggers if the value is greater than value₁ and less than value₂.

Example:

```
CREATE WATCHDOG BAND myWatchdog1 MODE INSTANTANEOUS THRESHOLD LOW 1 HIGH 3
CREATE WATCHDOG NOTCH myWatchdog2 MODE AVERAGE 10 THRESHOLD LOW 1 HIGH 3 GRACE 4
```

Architecture Binding:

Create a WatchdogBand or WatchdogNotch object with value₁, value₂, mode, and optionally value₃ and add it to SymbolTable<A_Watchdog> with id.

```
3. CREATE WATCHDOG (LOW | HIGH) id mode THRESHOLD value1 [GRACE value2]
```

Creates a watchdog that monitors a value. For LOW, the watchdog triggers if the value is less than value₁. For HIGH, the watchdog triggers if the value is greater than value₁.

Example:

```
CREATE WATCHDOG LOW myWatchdog1 MODE STANDARD DEVIATION THRESHOLD 3 GRACE 4
CREATE WATCHDOG HIGH myWatchdog2 MODE STANDARD DEVIATION 10 THRESHOLD 3 GRACE 4
```

Architecture Binding:

Create a WatchdogLow or WatchdogHigh object with value₁, mode, and optionally value₂ and add it to SymbolTable<A_Watchdog> with id.

Part II: 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 part.

This will be provided later next week.