Assignment 3

IMPORTANT: In this assignment you will fully model and implement your system. The idea is that you improve your UML models and Java implementation by (i) applying a subset of the studied design patterns and (ii) adding any relevant implementation-specific details (e.g., classes with "technical purposes" which are not part of the domain of the system). The goal here is to improve the system in terms of maintainability, readability, evolvability, etc.

Do NOT describe the obvious in the report (e.g., we know what a `name` or `id` attribute means), focus more on the key design decisions and their "why".

Format: establish formatting conventions when describing your models in this document. For example, you style the name of each class in bold, whereas the attributes, operations, and associations as underlined text, objects are in italic, etc.

Summary of changes of Assignment 2

Provide a bullet list summarizing all the changes you performed in Assignment 2 for addressing our feedback.

- Cleaning code
- Adding explanations for our design choices into our diagram summaries
- Implementing the full calculator
- Thinking deeply about where to implement further modularity

Maximum number of pages for this section: 1

Application of design patterns

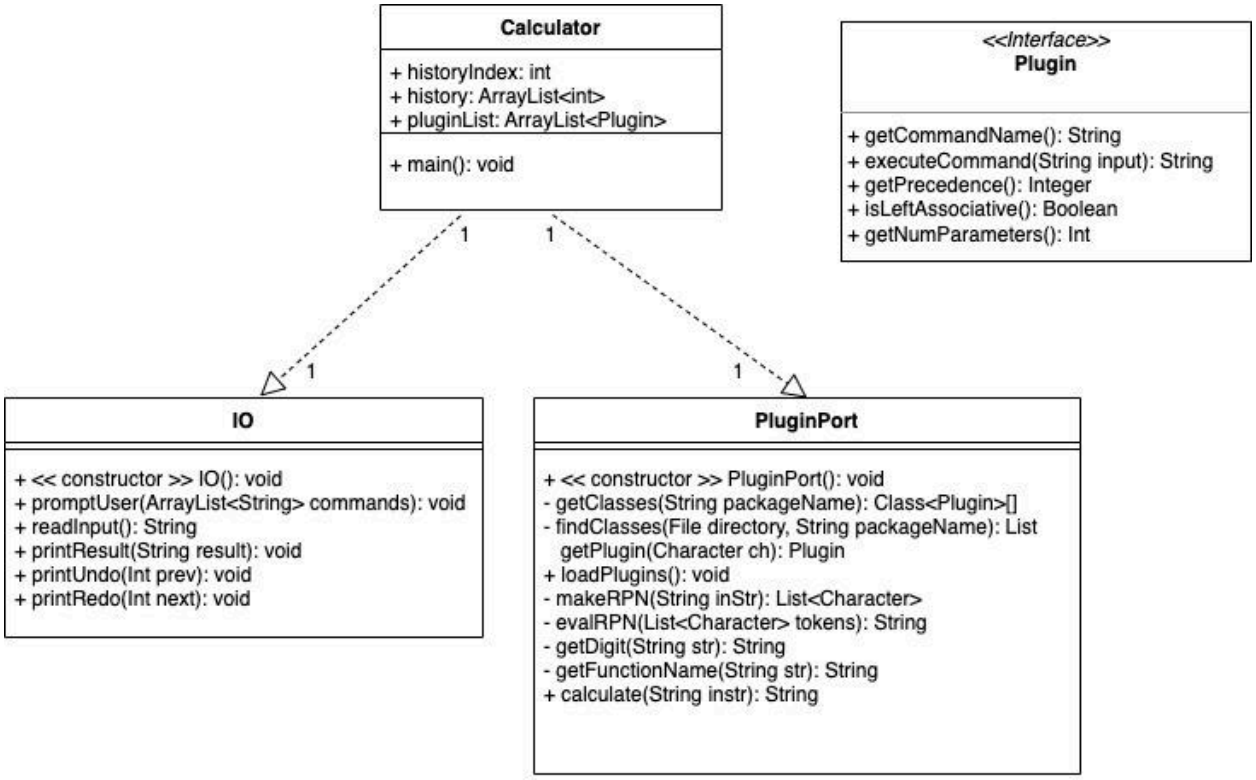
<Figure representing the UML class diagram in which all the applied design patterns are highlighted graphically (for example with a red rectangle/circle with a reference to the ID of the applied design pattern>

For each application of any design pattern you have to provide a table conforming to the template below.

	DP
Design pattern	Singleton
Problem	To ensure that only one instance of a IO and PluginPort classes is allowed within the calculator
Solution	1. Make constructor private 2. Make a static field containing its only instance 3. A static factory method for obtaining the instance
Intended use	The Calculator object uses the getInstance method of IO and PluginPort classes
Constraints	Any additional constraints that the application of the design pattern is imposing, if any
Additional remarks	none

Maximum number of pages for this section: 4

Class diagram



Our class diagram shows the 3 modules we outlined in Assignment 1.

I. **Calculator** (main module)
   A. This module is responsible for running the program. It maintains a list of all uploaded plugins. The other fields keep track of the current answer index in the list (which is initialized to 0) and a running list of all previous answers. This is to implement the undo/redo functionality. We keep a list of where everything is so that we can always get back to what was previously done without having to sacrifice our current position. The main module builds the PluginPort object and calls on it to load all plugins, then builds the IO object and calls on it to prompt the user and accepts input in a continuous loop that exits when the user enters “quit”.

II. **IO** (responsible for sending output/receiving input from the user)
   A. The functions in this module are called by *Calculator*, and they are as follows:
      1. promptUser: lists the available commands and activated plugins and prompts the user to enter a command
      2. readInput: returns the line of user input as a string
      3. printResult: prints the result of a calculation after the command has been executed
      4. printUndo: prints a message along with the previous result
      5. printRedo: prints a message along with the next result

III. **PluginPort** (responsible for all calculator functionalities and plugin management)
   A. The public functions in this module are only called within the *Calculator* module and are as follows:

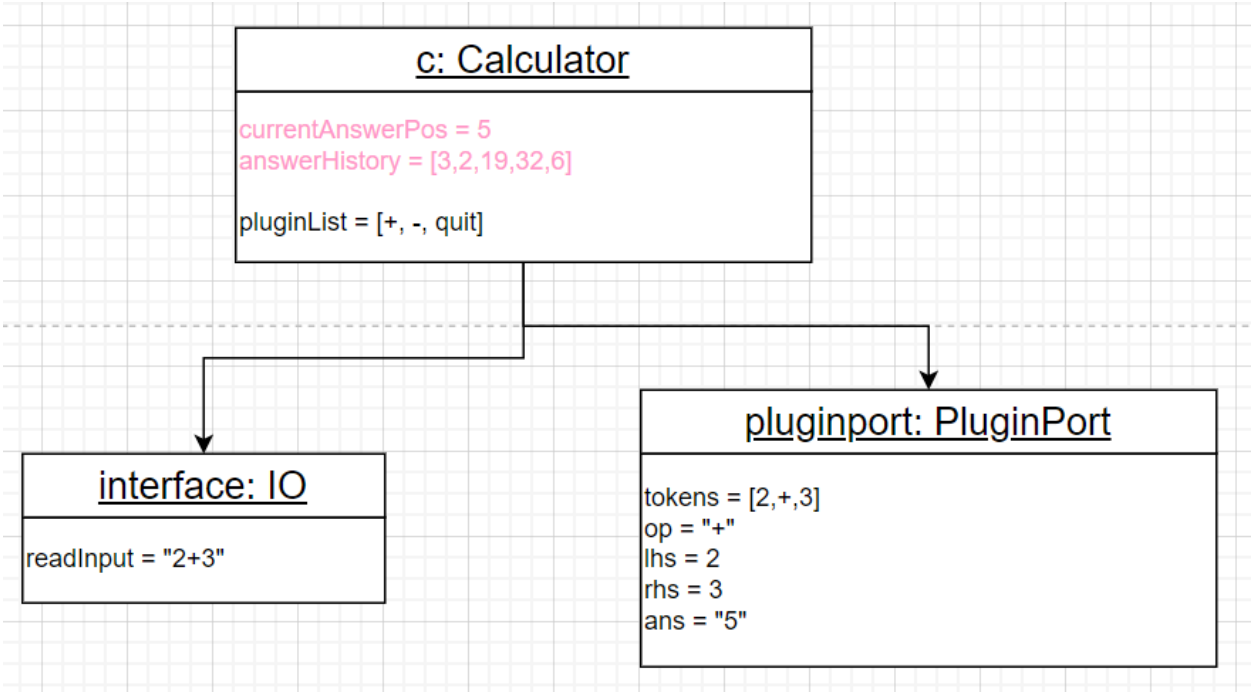
1. getClasses: Scans all classes accessible from the context class loader which belong to the given package and subpackages. Source
2. findClasses: Recursive method used to find all classes in a given directory and subdirs. Source
3. loadPlugins: loads all plugins at the start of the program
4. makeRPN: transforms the input into a reverse polish notation equation
5. evalRPN: evaluates the equation
6. getPlugin: returns the plugin to be used
7. getDigit: returns the digit as a string
8. getFunctionName: returns the function name
9. Calculate: performs the calculation

Additionally, we have defined a **Plugin** interface which all plugins must implement. This constrains 3rd party developers to write plugins that will be compatible with our calculator program. You can see this with our examples addition.java and subtraction.java, among others, which are found in the plugin folder.

Hopefully it is clear from the diagram and description of the modules, but the general flow of the program will be as follows: The *Calculator* module starts the program, and via the command loop calls on the *IO* module to interact with the user. The command is then provided back to the *Calculator* module, which passes it to the *PluginPort* module to be transformed into RPN format and subsequently evaluated. If the command is improperly formatted, the *PluginPort* module will throw an error that is caught in the *Calculator* module, which prints a message to the user indicating improper formatting. If the input can be executed, the *PluginPort* returns the result to the *Calculator* module, which then sends it to the *IO* module to be sent to the user. This process continues on loop until the user inputs “quit” which exits the calculator program.

As you can see below in our object diagram, we have one *Calculator* which holds the information about the program’s history and is responsible for running the program. A downside of this decision is that undo/redo will have to call upon the fields in the *Calculator* rather than hold that knowledge directly, but we felt this was the correct design choice as the *Calculator* should hold that information, not the *PluginPort*. Similarly, we have one *IO* which is responsible for sending output and receiving input. While it adds more switching of responsibilities in the program, we think this nicely divides the labor so as to avoid the *Calculator* module doing all the work and becoming overwhelming to the developer.

Object diagram

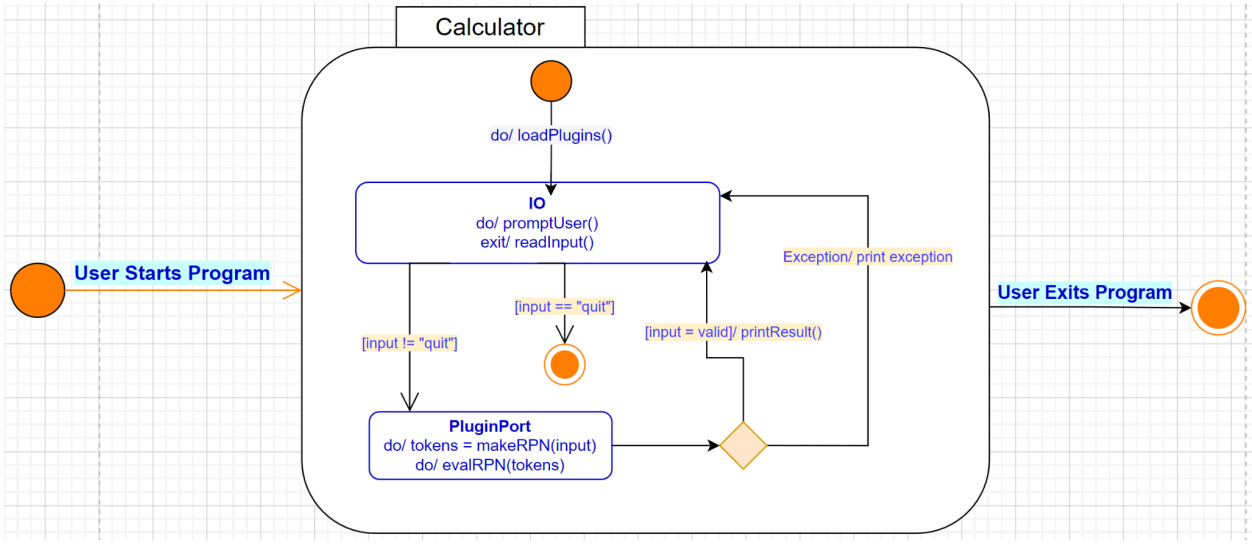


The above object diagram depicts a snapshot scenario where a user has just inputted the string “2+3” into the program. The Calculator object *c* represents the current instance of the program that the user is running. The attribute currentAnswerPos shows the results of the previously inputted string of “2+3”. Thus the current value of currentAnswerPos is “5”. The answerHistory attribute shows an array of the previous answers returned by the program. The two attributes are in pink because they have yet to be implemented.

The **IO** object interface has the attribute of readInput which has been registered as the user’s inputted string “2+3”. The **PluginPort** object *pluginport* contains the attributes that parse through the inputted string, separating it into values required by the plugins for calculation. In this snapshot example, the attribute “tokens” depicts the inputted string as a list of individual characters. The op attribute represents the operator (aka plugin command) found in the inputted string — in this case, “+”. The lhs and rhs attributes represent the left side of the inputted expression and the right side of the inputted expression, respectively. The ans attribute is a string representation of the answer of the expression as derived by the specified plugin.

Having only three objects, representing each of the main classes in the program, simplifies the calculator’s implementation and operation. While the *pluginport* object may seem disproportionately large, its task of parsing through the input is relatively streamlined. The input is tokenized, split, and calculated in only a few steps, which aids to the operation’s runtime.

State machine diagrams

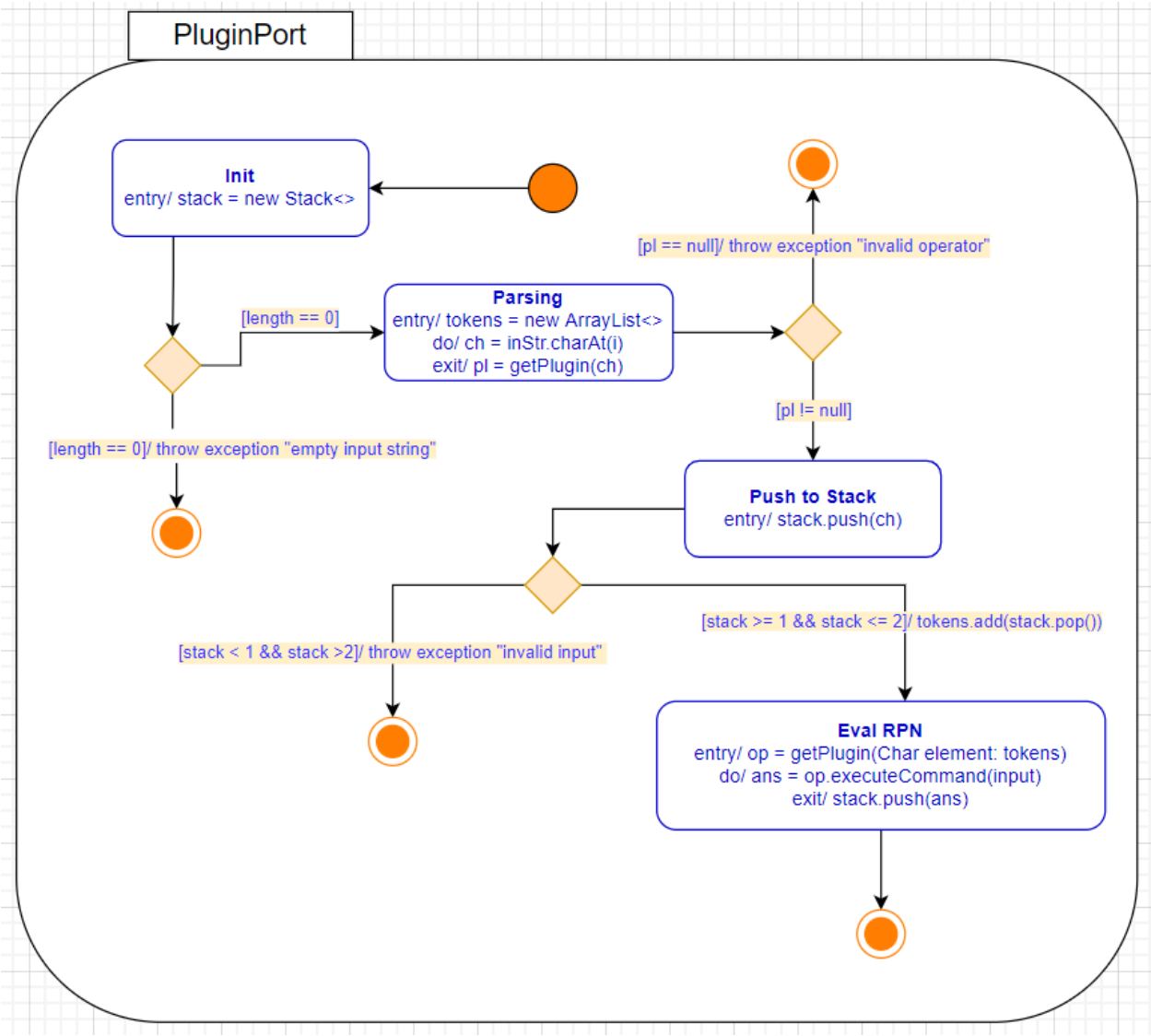


This state machine diagram depicts a composite state — *Calculator* — with several nestled substates that show the flow of interactions within the program. The *Calculator* composite state corresponds with the **Calculator** class, the nestled substate *IO* corresponds with the **IO** class, and the nestled substate *PluginPort* corresponds with the **PluginPort** class.

The user starts the program and transitions into the *Calculator* composite state immediately. Then the program executes the loadPlugins() method and transitions into the *IO* substate. There, the promptUser() method is run as an internal activity to await the user's input. When the user's input is registered, the *IO* substate's exit activity runs readInput() before reaching a decision node. If the input is the quit command, the program transitions to a destroy node, exiting the compost state and the program at large. If the read input is not the quite command, the program transitions into the *PluginPort* substate. In the *PluginPort* substate, the list of characters "tokens" is set to equal makeRPN(input). The evalRPN command is also executed before the program exits and transitions to another decision node.

If the makeRPN() method did not throw an exception, then the input must have been valid. As such, the program would transition back to the *IO* substate, executing the printResult() method along the way. This will display the result of the calculation to the user and prompt them to enter another expression, restarting the cycle until the the "quit" command is inputted. However, if the makePRN() method did throw an exception, then the alternative transition path would be taken as the input must have been invalid. The program would return to the *IO* substate, printing the results of the exception along the way. Still, the user would be prompted to enter another expression and the loop would continue until the user quits.

The philosophy for the composite state is that the program is only even stalling when awaiting the user's input in the *IO* substrate. The principle is that the loop should occur as quickly as possible, returning the correct calculation, so that the user can input their next command.



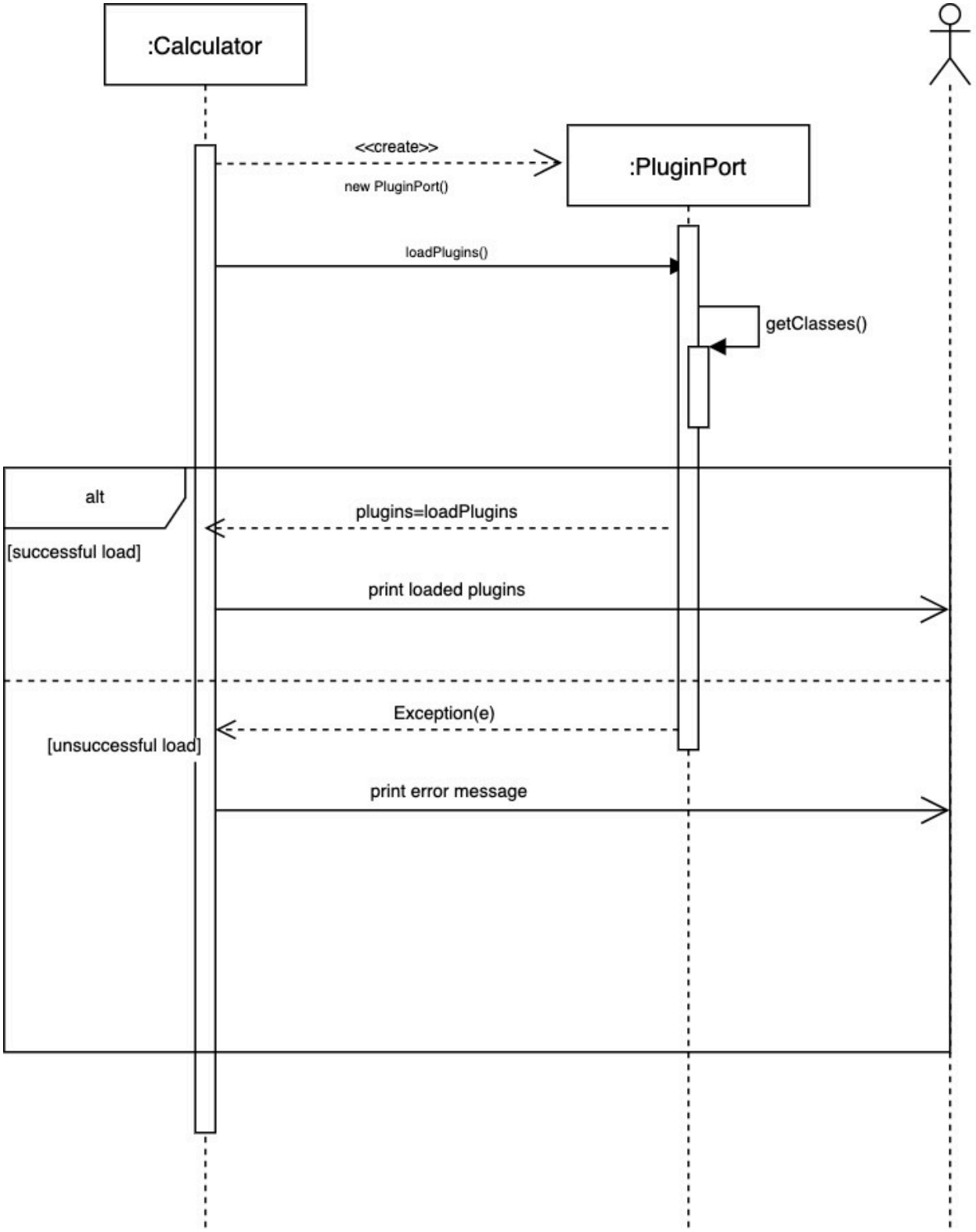
The above state machine diagram depicts the internal substates of the *PluginPort* composite state. This composite state is entered following the user inputting a command other than “quit.”

Upon starting, the program immediately transitions into the Init substate, creating a new stack as the entry activity, before transitioning to a decision node. If the inputted string was empty, then the program transition to a destroy node, throwing an “empty input string” exception along the way. If the length is greater than zero, the program transitions to a Parsing substate that creating an ArrayList “token” upon entry. The substate executes charAt() and getPlugin() for each character of the input string before exiting the substate to a decision node. If the inputted string did not contain a valid plugin command, the program transitions to an end node, throwing an “invalid operator” exception along the way. If the plugin command was valid, the program transitions to a Push to Stack substate, executing stack.push() for each character in the inputted string. The program immediately exits to another decision node. If the size of the stack is less

than one or greater than two, the program transitions to a destroy node, throwing an “invalid input” command along the way. If the size of the stack is one or two, the program transitions to the Eval RPN substate, executing tokens.add() for each of the contents of the stack. Upon entry to the Eval RPN substate, the Plugin op is set to equal the result of the getPlugin() method, which determines which plugin the user wishes to use. The the internal activity of the substate sets the string “ans” the the results of the executeCommand(input) method. Finally, the program exits the substate, executing stack.push(ans), and transitions to a destroy node, leaving the *PluginPort* composite state.

With the program having exited the *PluginPort* composite state, the user will be presented with the results: either the inputted expression’s answer or the exception informing them of an syntax error.

Sequence diagrams



sequence diagram for loading and initializing plugins:
The sequence diagram above follows the initialization processes that occur after the program is started.

To load the plugins, the **Calculator** module first creates a new instance of **PluginPort** class and invokes its loadPlugins() method. We chose to do this so that all of the plugins are loaded by the time the user interact with the program, so it will not cause a long response time during the run of the program and also does not invite user error in loading plugins. A downside to this is that, in its current implementation, the program will automatically load all plugins even if the user does not want to use all of them. The internal initialization steps performed by this method are abstracted away in the sequence diagram. This method uses java.lang.ClassLoader to load plugins from the plugins directory and add them to the list of plugins.
If the initialization phase is not successful, *Calculator* will print an error message and will end the program.

sequence diagram for post-initialization:
To interact with the user, *Calculator* creates a new instance of **IO** class and invokes the promptUser() method to print the list of available operators. The user input is then read from the command line using the readInput() method. Although the two methods are always called together, we kept them separate for modularity purposes as they are two separate tasks.
The *Calculator* then uses the makeRPN() method of *PluginPort* to convert the user input to a list of tokens in postfix notation. This method uses a variation of the shunting yard algorithm, and its purpose is to make parsing user calculator input easier. While this method took more effort to implement fully, it allows for a more seamless calculator experience for the user, which we believe is important.
If the user input is not valid, an exception will be thrown, and an error message will be printed. The program then awaits valid input. It would be inappropriate for the program to end or crash here, as it is easy to make input mistakes, so our method allows the user to try again without the need to restart the program.
In case of valid user input, the returned list of tokens will be evaluated using the evalRPN() method. It finds the plugin instance representing each operator in the list of tokens and uses the plugin's executeCommand method to calculate the results.
The returned result will be printed using the printResult() method, which once again is included for modularity purposes.

Implementation

In this chapter you will describe the following aspects of your project:

- the strategy that you followed when moving from the UML models to the implementation code;
- the key solutions that you applied when implementing your system (for example, how you implemented the movement of the cursor in in the Cyberpunk minigame, how you manage Exploding Kittens matches, etc.);
- the location of the main Java class needed for executing your system in your source code;
- the location of the Jar file for directly executing your system;

- the 30-seconds video showing the execution of your system (you are encouraged to put the video on YouTube and just link it here).

IMPORTANT: remember that your implementation must be consistent with your UML models. Also, your implementation must run without the need of any other external software or tool. Failing to meet this requirement means 0 points for the implementation part of your project.

Maximum number of pages for this section: 2

When transferring our UML models to implementation, we followed a heavily agile strategy. This was critical to our success, as we did not know precisely how to implement everything at the time we created our models, so as we learned how to write implementation for things like supporting the use of plugins, we had to go back and adjust our models to match the implementation. The models acted as a framework for the code we wrote, which in turn allowed us to specify our models even further.

The most critical development in our implementation was figuring out how to load the plugins. We went through many ideas of how to load plugins including; adding them at runtime, using JSON files, or loading all plugins at the start of the program. Ultimately, we determined that loading the plugins at the start of the program would be most efficient for the user, as it eliminates possible crashes or time-constraints from occurring in the middle of running the program. We found a source (cited) from which we pulled functions used for loading all classes from a package and edited those functions to load those classes as *Plugin* objects. The *Calculator* stores a list of all *Plugins*, so the *Plugins* and their underlying functions will be accessible throughout the entire execution of the program.

Another important solution in our implementation is the **Plugin** interface. This constrains 3rd party developers to write plugins that will work with our implementation. This is another example of something that did not exist in the first draft of our UML model, since we had to do further reading to understand why the interface is necessary. The interface contains function declarations for retrieving the command name, executing the command, and clarifying calculational precedence and left association. Our implementation also supports undo/redo functionality in addition to calculations.

For Assignment 3, we focused on addressing feedback from Assignment 2, adding functionalities that did not exist at that time, and modularizing our code. We made the calculate() method to hide the RPN algorithms from the client object (in this case the client is Calculator).

The main java class is called Calculator, as such you can find main in the file Calculator.java. The Jar file is located under the file path from the repository: software-design-vu/out/artifacts/software_design_vu_2020_jar/software-design-vu-2020.jar.
This link shows a brief demonstration of our program:
https://youtu.be/5j_Di5w7s4o