Assignment 3

Team number: 19
Team members

Name	Student Nr.	Email
Hung Hoang Duy	2711030	h.hoangduy@student.vu.nl
Toyesh Chakravorty	2689157	t.chakravorty@student.vu.nl
Dora Kementzey	2714287	d.kementzey@student.vu.nl
Yudai Nakazaki	2707442	y.nakazaki@student.vu.nl

Source of Diagrams : https://app.diagrams.net/#G18Q1qIswAYXYAmiZzuCVpjDBJzkJVOo1i

Summary of changes of Assignment 2

Author(s): Yudai

Modified parts from Assignment 2:

General:

- Explained why we chose the design or structure for our system in each diagram

Class Diagram:

- Added an interface class for Plugin. We and the third party developers implement each Plugin class based on the interface
- Made all descriptions about roles of classes clear
- Removed unnecessary classes
- Made public/private status for each method and attribute more precise.
- Made associates(composition, multiplicity, etc.) among classes more precise
- Changed names of attributes and methods which are originally unclear
- Made data types for attributes and method more precise

State machine:

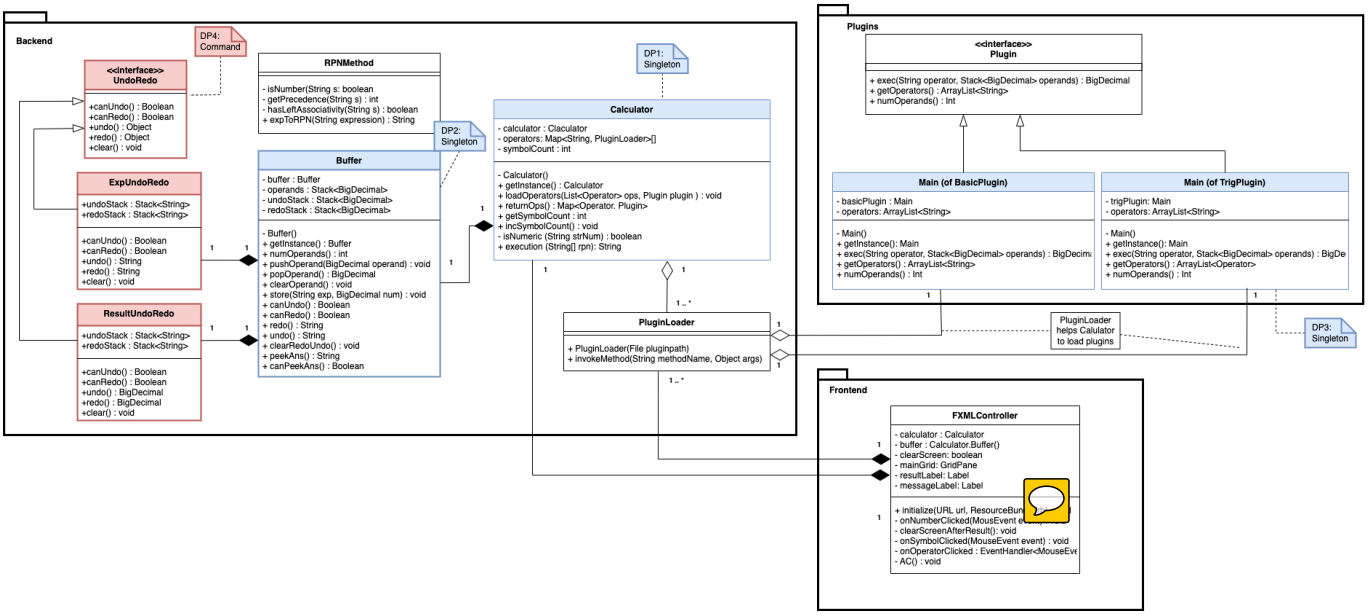
- Change state names to noun and activity names to verbs respectively
- Made state machines more like state machines

Implementation:

- Leave useful and brief comment
- Removed uncommented codes
- Make sure jar file is executable

Application of design patterns

Author(s): Yudai&Hugo



	Design Pattern 1
Design pattern	Singleton
Problem	In our system, the Calculator object holds a lot of crucial information about calculation such as previous results in Buffer and always needs to correct information corresponding to each timing at runtime. So we have to make sure to have one single Calculator object at runtime.
Solution	By applying the singleton design pattern, we can avoid the Calculator class to be instantiated externally and having more than one object instantiated for the class.
Intended use	The Calculator object instantiated from Calculator class using singleton gets the list of available operators and corresponding execution methods from Plugins. It also stores inputs and calculation results, and provides a required execution method to execute a calculation. In short, this object is the core part of this system.
Constraints	The class with this design pattern cannot be instantiated outside, which makes some process of implementation of the system less flexible, while this design pattern makes our system more static and robust.
Additional remarks	We could use the flyweight design pattern instead of Singleton. However, we need to have extra data structure such as an object pool and it might make our source code more redundant.

	Design Pattern 2
Design pattern	Singleton

Problem	In the Calculator object, we need to make sure that we have one single Buffer object in it, since the Buffer object holds important information such as calculation results which needs to be consistent with the user operation and user interface.
Solution	By applying the singleton design pattern, we can avoid instantiating the Buffer class externally and having more than one Buffer object instantiated for the class.
Intended use	The Buffer object store previous calculation results in a few stacks. As results are ready, this object gets data from the Calculator object and returns data in turn as Undo and Redo operations are called.
Constraints	Refer to DP 1.
Additional remarks	Refer to DP 1.

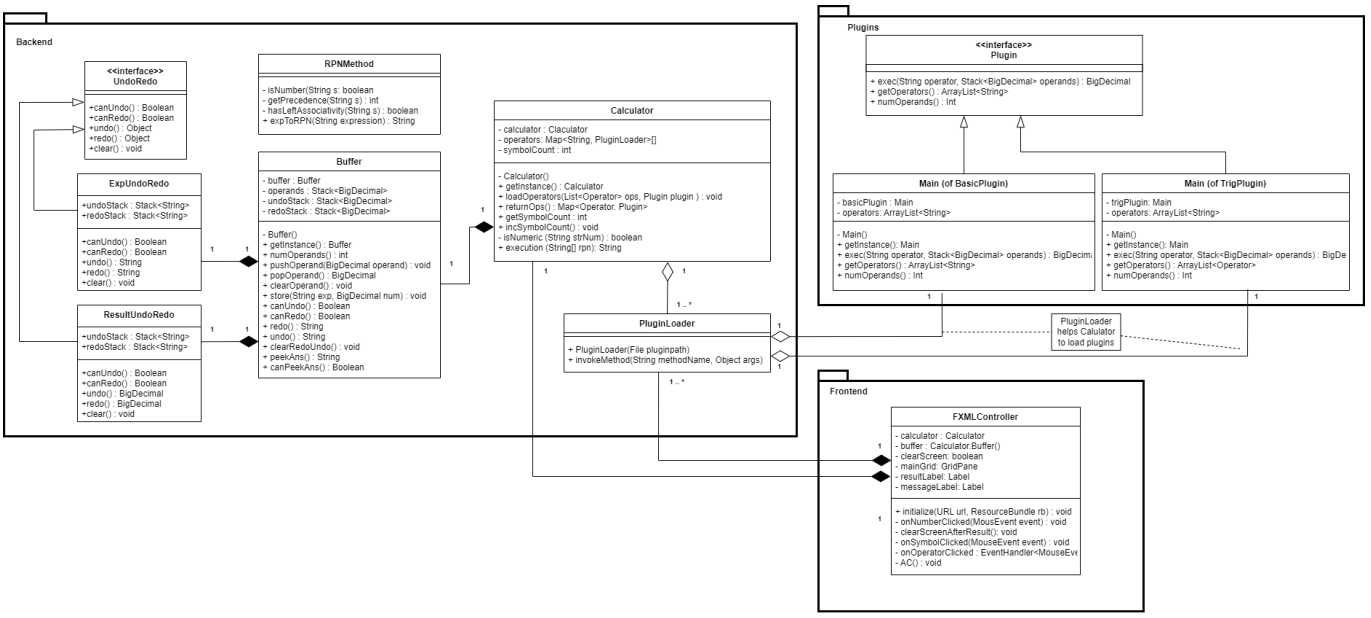
	Design Pattern 3
Design pattern	Singleton
Problem	We need to confirm that we instantiate each Plugin class once at a runtime because if a Plugin class is instantiated more than once by mistake, the calculator object might have duplicated operators and this situation is quite error-prone.
Solution	By applying the singleton design pattern, we can avoid instantiating a class with a Plugin interface externally and having more than one object instantiated for the class.
Intended use	Each plugin class is instantiated in the main method and the Calculator object obtains the object and operators to store them. If a button is clicked (ex/ “+”), a corresponding Plugin’s exec() method is called.
Constraints	Refer to DP 1.
Additional remarks	Refer to DP 1.

	Design Pattern 4
Design pattern	command
Problem	We have to have two similar operations for redoing and undoing calculation results and corresponding expressions. We might lose the consistency in the code, if we manage them separately.
Solution	By applying the command design pattern, we make an interface for all the operations on redo and undo, and we define concrete classes specific to each purpose, expressions and results, based on the interface.
Intended use	Two concrete objects based on the interface, **expRedoUndo** and **resultRedoUndo**, are instantiated and used by a Buffer object to manage operations of redo and undo.

Constraints	By having this design pattern, we need to increase the number of classes and corresponding files, which makes our system complicated.
Additional remarks	In terms of making the interface less complicated, this design pattern works pretty well.

Class diagram

Author(s): Yudai



<Backend : Package>

Plugins : Package

In the package of Plugins, this system holds all the plugins which can be applied to extend the function of this calculator. All the plugins need to be implemented to follow **<<interface>> Plugin.** In this class diagram, we have only the BasicCalculator plugin in the package. Plugins can be added to this package as needed.

Plugin : interface

Plugin interface is the model of all the plugins which can be added to this system. Applying this design to the system, we can extend the functionality of this JCalculator without modifying the source code of the main frame of the calculator. Furthermore, the third-party developers can implement their own plugins by following this interface. This is the core design of extensible systems.

Operations

- **exec(String operator, Stack<Bigdecimal> operands) : String** — This operation is called by **Calculator** and executes a calculation corresponding to **operator** with **operands.**
- **getOperators() : ArrayList<Operator>** — This operation is called in the **initialize** method to pass operators in the Plugin to **Calculator**.
- **numOperands() : Int** — This operation is called in **Calculator** when a calculation is executed to confirm that the number of operands is valid for the operation to be executed.

Associations

- **Inheritance with Plugins** — It has relations of inheritance with plugins with this interface.

BasicCalculator

BasicCalculator is a default plugin class in the Plugins package implemented based on the Plugin interface.

Attributes

- **operators : ArrayList<Operator>** — This holds all the operators in the plugin in an ArrayList. This attribute is private so that it is not exposed to outside.

Operations -> refer to Plugins

Associations

- **Shared aggregation w/ Calculator** — This class has a shared aggregation association with one calculator.

Operator

Operator class defines a model for operators in each plugin. The reason we made this class is that we need to hold a symbol and name for an operator to use for showing on the GUI and for id names in a fxml file respectively. Although getting a class for this small amount of information could make our source code redundant, it removes the complexity and reduces the number of operations required.

Attributes

- **name : String** — It represents the name of the operator such as “Multiply”. It is private.
- **Symbol : Character** — It represents the symbol of the operator such as ‘*’. This is also private.

Operations

- **Operator(String name, Character symbol)** — This is a constructor of the class.
- **getName() : String** — This method returns the name of the operator.
- **getSymbol() : Character** — This method returns the name of the operator.

Association

- **Composite relation w/ a Plugin** — Each operator has a plugin class as a composite object.

Calculator

Calculator class is the main frame of the JCalculator and manages all the information about calculation and the process of the calculation. The most important role of this class is connecting plugins and the main controller which directly interacts with users. Without this class, the system has to call **exec()** method separately for different plugins.

Attributes

- **operators: Map<String, PluginLoader>[]** — This map object stores **privately** all the operators as symbols and corresponding plugin objects which are currently plugged into the system.
- **buffer : Buffer** — One single Buffer class is contained in a Calculator class. Description about Buffer follows.
- **symbolCount : int** — This counts the number of operators currently loaded.

Operations

- **getInstance() : Calculator** — This operation returns the privately constructed Calculator object.
- **loadOperators(ArrayList<Operator> operators, Plugin plugin) : void** — This operation is called in FXMLController method as a plugin initialized and stores operators and plugin in **operators.**
- **getSymbolCount() : int** — This operation returns the **symbolCount**.
- **incSymbolCount : void** — This operation increments the **symbolCount**.
- **getOperators() : Map<String, PluginLoader>** — This operation returns **operators**.
- **execution(String[] rpn) : String** — This operation takes an expression in a reverse polish notation format as a list of String and executes a calculation on it, then returns the result in String.

Associations

- **Shared aggregation relationship with Plugins –**- Calculator object has a shared aggregation relationship with each plugin in the Plugins package.
- **Composite relationship with a Buffer** –- Calculator is a composite object of a Buffer class.
- **Composite relationship with a FXML Controller** — Calculator has a FXMLController as a composite object.

Buffer

A Buffer class is designed to manage and store data about calculation. This class is contained in a Calculation class and this removes the complexity of the implementation of a Calculator class by separating data management roles from the main part of a Calculator class.

Attributes

- **operands : Stack<BigDecimal> –**- This Stack data structure stores operands of the current calculation. This stack is passed to **exec()** method of each plugin to be calculated.
- **resultUndoRedo : ResultUndoRedo** –- This object manages stacks to redo and undo results and all the commands needed to execute operations.
- **expUndoRedo : ExpUndoRedo** –- This object manages stacks to redo and undo expressions and all the commands needed to execute operations.

Operations

- **getInstance() : Buffer** — This operation is called at an initialization step and returns the privately constructed Buffer object.
- **numOperands() : int** — This operation returns the size of current **operands**. The returned value is used to check whether the number of operands is valid for an operation being executed.
- **pushOperand(BigDecimal operand) : void** — This operation pushes an operand on **operands**.
- **popOperand() : BigDecimal** — This operation pops an element on top and returns it.
- **clearOperand() : void** — This operation clears all the content of **operands**.
- **store(String exp, BigDecimal num) : void** — This operation pushes a calculation result and corresponding expressions on **undoStack** in **resultUndoRedo** and **expUndoRedo**.
- **canUndo() : boolean** — This operation checks whether undoStacks both in **resultUndoRedo** and **expUndoRedo** are undoable.
- **canRedo() : boolean** — This operation checks whether redoStacks both in **resultUndoRedo** and **expUndoRedo** are redoable.
- **undo() : String** — This operation executes undo operations both for **resultUndoRedo** and **expUndoRedo** and returns a value which shows the result and expression.
- **redo() : String** — This operation executes redo operations both for **resultUndoRedo** and **expUndoRedo** and returns a value which shows the result and expression.
- **clearRedoUndo() : void** — This operation is called to clear redoStack and undoStack both in **resultUndoRedo** and **expUndoRedo.**

Associations

- **Composite relationship with a Calculator** — Buffer class has Calculator class as a composite object.

RPNMethod

This class is an independent method class to support the Calculator class to execute calculations. The main role of this class is to convert the normal expression format into the reverse polish notation format. All the operations in this class are static.

Operations

- **letterOrDigit(char c) : boolean** — This operation is called in expToRPN and tells whether the input char is a digit or not.
- **getPrecedence(char c) : int** — This operation is called in expToRPN and tells the precedence of the input operation.
- **hasLeftAssociativity(char c) : boolean** — This operation is called in expToRPN and tells whether the input operation has the left -> right associativity.
- **expToRPN (String expression) : String** — This method converts the normal expression to the reverse polish notation.

UndoRedo : interface

This class is an interface for redo and undo commands. Since the redo and undo commands for calculation results and expressions are almost the same, we managed these commands as a class following the **command design pattern**.

Operations

- **undo() : Object** — This operation executes undo operation and returns a value. The concrete data type is defined in a concrete class.
- **redo() : Object** — This operation executes redo operation and returns a value. The concrete data type is defined in a concrete class.
- **canUndo() : Boolean** — This operation returns the empty status of **undoStack**.
- **canRedo() : Boolean** — This operation returns the empty status of **redoStack**.
- **clear() : void** — This operation is supposed to clear **undoStack** and **redoStack** in a concrete class.

ExpUndoRedo

This class is a concrete class based on **UndoRedo** interface. It has **undoStack** and **redoStack** for expressions of each calculation and it overrides operations in UndoRedo.

Attributes

- **undoStack : Stack<String>** — This Stack stores calculation expressions for **Undo** operation.
- **redoStack : Stack<String>** — This Stack stores calculation expressions for **Redo** operation.

Operations -> All the operations follow the **UndoRedo** interface. For **undo()** and **redo()**, it overrides the data type as String.

Associations

- **Composite relationship with a Buffer** –- This class has Buffer class as a composite object.

ResUndoRedo

This class is a concrete class based on **UndoRedo** interface. It has **undoStack** and **redoStack** for calculation results and it overrides operations in UndoRedo.

Attributes

- **undoStack : Stack<BigDecimal>** — This Stack stores calculation results for **Undo** operation.
- **redoStack : Stack<BigDecimal>** — This Stack stores calculation results for **Redo** operation.

Operations -> All the operations follow the **UndoRedo** interface. For **undo()** and **redo()**, it overrides the data type as BigDecimal.

Associations

- **Composite relationship with a Buffer** –- This class has Buffer class as a composite object.

PluginLoader

This class is used to load plugins from outside. It allows users to choose a plugin jar file by choosing a directory and instantiates the Plugin class to hold the instance. After the PluginLoader gets the plugin instance, the calculator retrieves the reference to the instance and operators.

Operations

- **PluginLoader(File pluginPath)** — This operation lets a user choose a path of a plugin jar file and instantiate it to hold the instance.

- **invokeMethod(String methodName, Object … args) : Object** — This operation is called to use the operations and attributes inherited from the Plugin through the pluginLoader object.

<Frontend : Package>

FXMLController

FXMLController class is the main contact point between the user interface and backend program. All the operations interacting with the GUI are executed in this class.

Attributes

- **calculator : Calculator** — The calculator class is instantiated in the controller class.
- **buffer : Calculator.Buffer()** — The buffer class is also instantiated in the controller class.

Operations

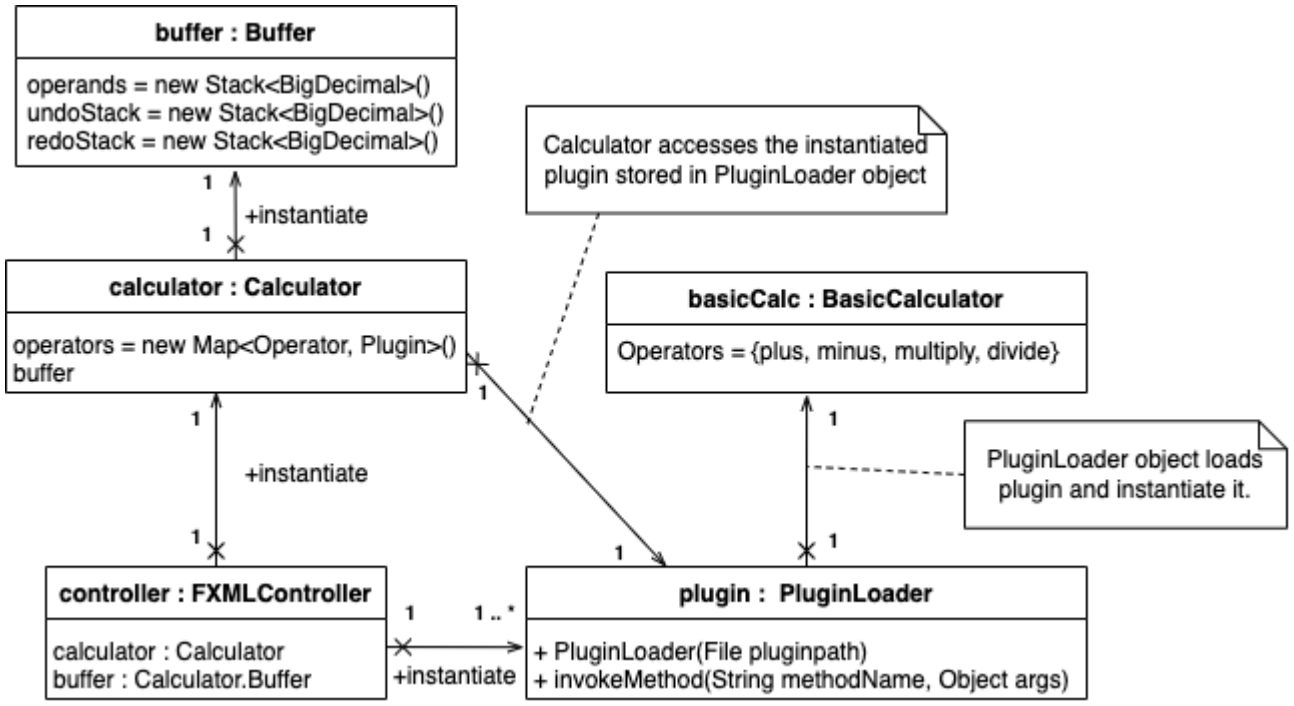
- **initialize(URL url, ResourceBundle rb) : void** — This method is called at the initialization phase of the system.
- **onNumberClicked(MouseEvent event) : void** — This method handles click events on numbers.
- **onSymbolClicked(MouseEvent event) : void** — This method handles click events on symbols other than operators.
- **onOperatorClicked : void** — This method handles click events on operators.
- **AC() : void** — This operation is called when AC operation is required and reset the calculator's state.

Associations

- **Composite relation w/ Calculator** — FXMLController class is a composite object of Calculator.

Object diagram

Author(s): Yudai



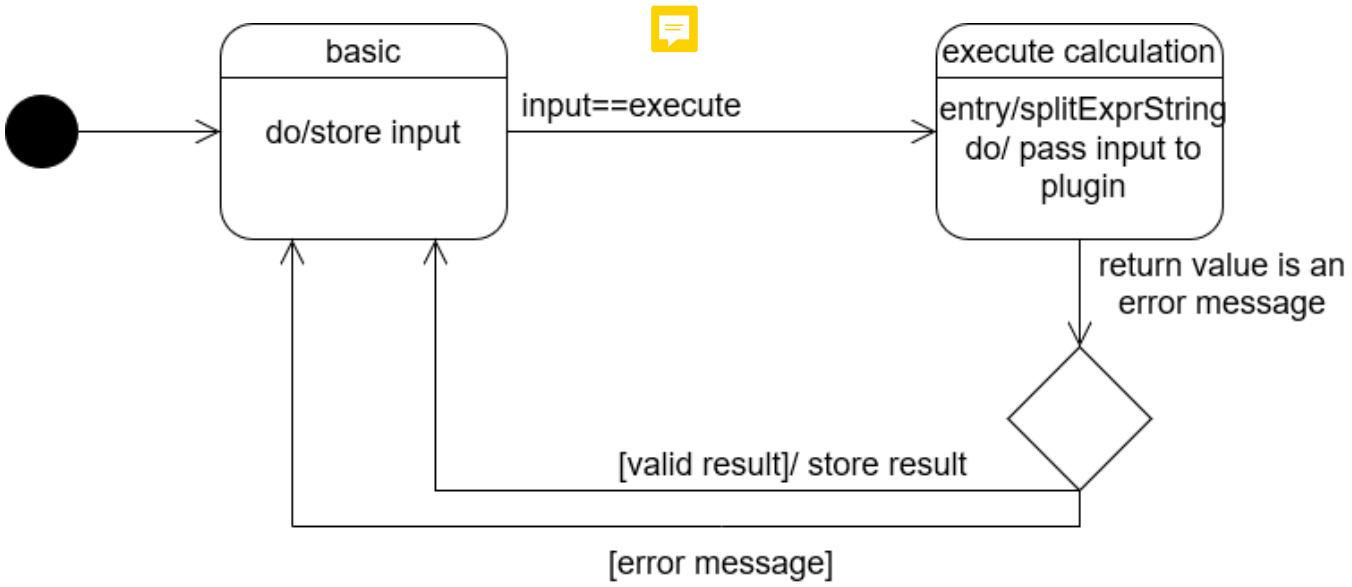
The object diagram is a snapshot of a scenario where the system is initialized and the calculator system is loading the BasicCalculator plugin from outside of the system.

- **controller x—> calculator** — At this phase, **controller** instantiates the Calculator class as a calculator and the calculator holds an empty Map object, **operators**.
- **calculator x—> buffer** — At this stage, **calculator** instantiates the Buffer class as **buffer**.
- **controller x—> plugin** — Here, **controller** instantiates a **PluginLoader** to load a plugin.
- **Plugin x—> basicCalc** — **plugin** loads the BasicCalculator class and stores its instance.
- **Calculator x—> plugin** — In this step, **calculator** retrieves information about the BasicCalculator through the instance of it in **plugin**.

State machine diagrams

Author(s): Dora & Toyesh

Calculator class

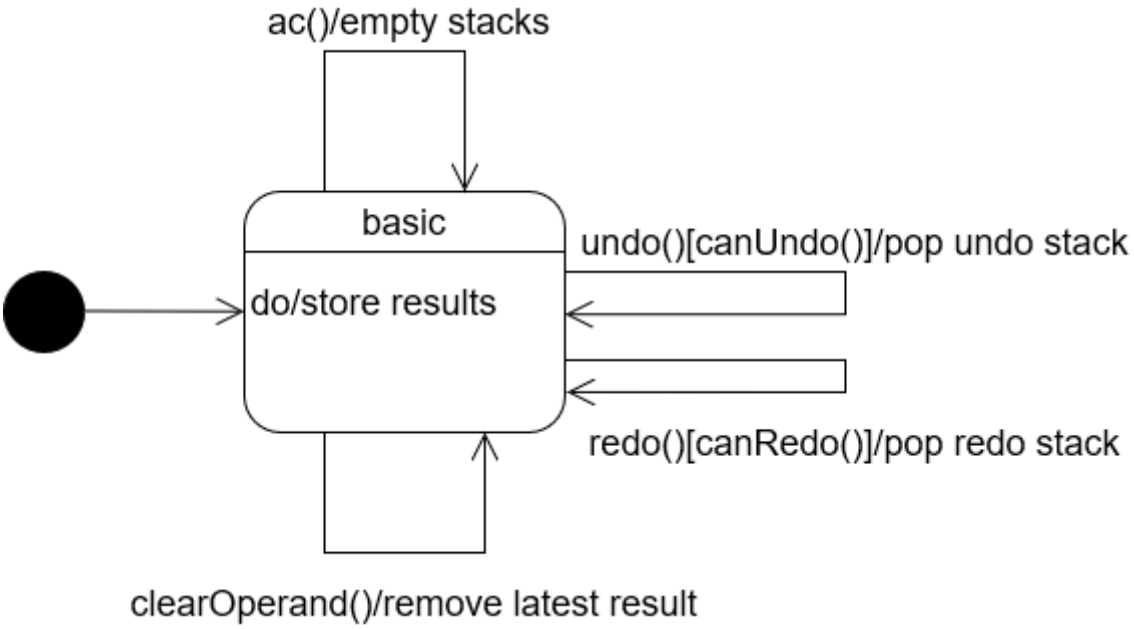


This state diagram is used to describe the main behavior of the Calculator class and the corresponding results that it needs to wait for. After the initial state where the user enters a combination of operators/operands and clicks on the execute button, the Calculator class splits the string and transforms it to a reverse polish notation, and consequently sends it to the Plugin class to take in the string as an input. The Plugin class in turn sends a corresponding result back to the Calculator class. This repeats for every subsection of an expression (for ex. 3+2*4) until the final result is received (evaluation==false). If the result is not an error message this is sent to the Buffer which stores the result. A detailed description of the calculation process can be seen in the sequence diagrams, for clarity this has been omitted here (And also because it does not happen in the actual Calculator). Finally, the Calculator returns to its original state: waiting for the input.

We opted for the reverse polish notation as it is faster and easier to calculate expressions, particularly the more complex ones. It also removes the need for parentheses that are otherwise required by infix notation. The recursive manner in which the calculations are done makes sure we stop immediately after encountering an error, as well as making it more efficient since we always only do basic calculations.

In accordance with the feedback we received, the diagram was modified to mainly have states that are nouns. We also got comments about rather having reasons for design decisions than a description, however as the exercise states to add a description ‘in a narrative manner’ we decided to include both.

Buffer class



This diagram shows the states of the buffer class. When the calculator is started, the first and only state called ‘basic’ is activated. Here, the final results of each expression is stored in a stack called undoStack. If the undo button is clicked,and the boolean canUndo is evaluated as true, the buffer will get the previous result (pop the item from the stack), and move it to the redoStack. After this the buffer returns to waiting for results. If the redo button is clicked, similarly to undo, an item is popped from the redoStack and added back to the undoStack. After this the buffer returns to waiting for results. When ac is clicked, both stacks are emptied, and the buffer returns to its basic state. If the button C (so clearOperand) is clicked the latest result is deleted.

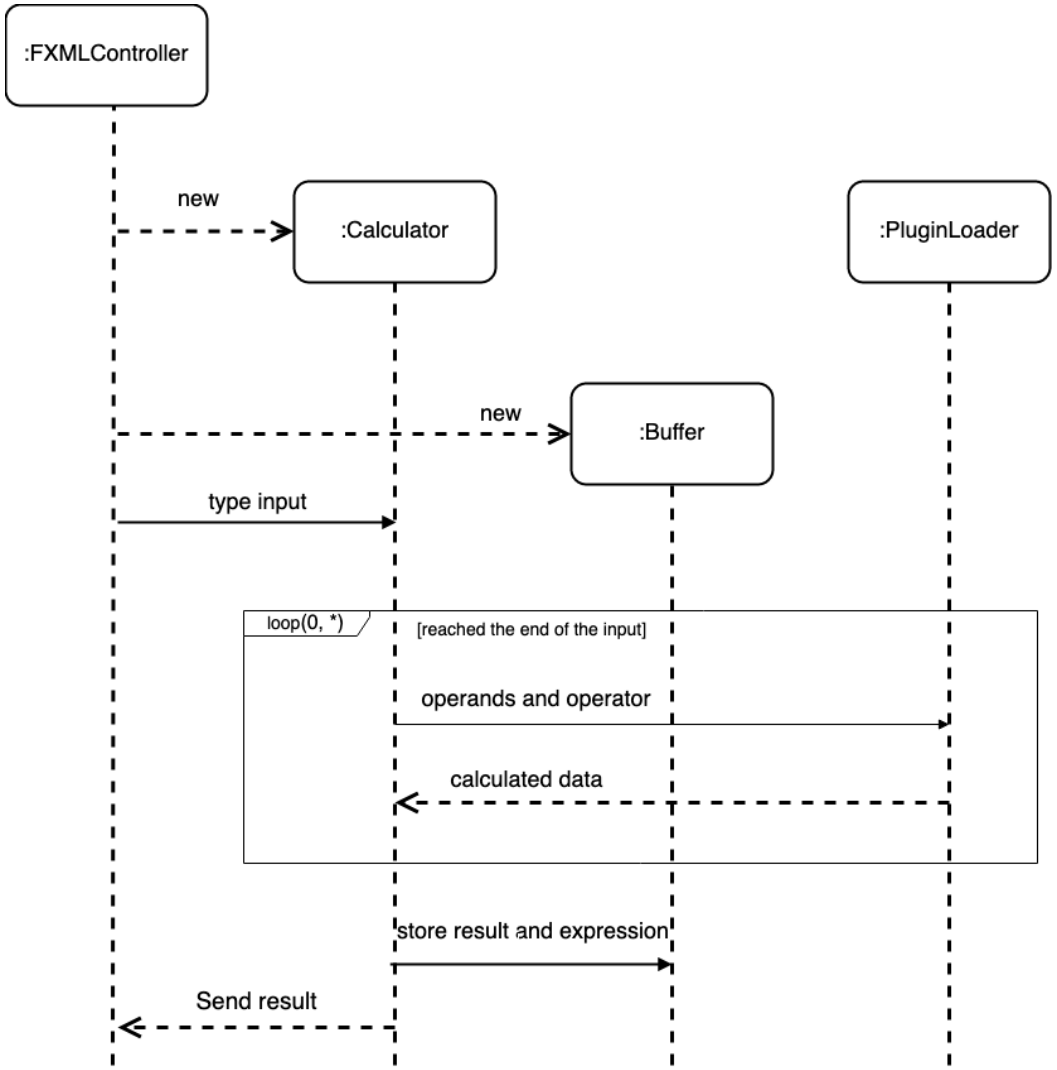
We decided to add a separate clear next to the all clear, as we felt it would be useful when performing longer calculations, for example if one makes a mistake. The undo/redo implementation made the most sense this way, stacks are the most convenient way of storing and manipulating data for our purposes. When designing this part we took user friendliness into consideration, and added features that we feel would be useful and logical to people using our program.

Reflecting on our feedback from assignment 2 we updated the diagram to not only fit our updated implementation, but we also removed state names that were verbs, and opted for a smaller number of states. This is more in line with what a state diagram should be, removing states that have intermediate actions. A simpler diagram also makes it easier to understand what is going on in the implementation.

Sequence diagrams

Author(s): Dora & Toyesh

Calculating an Expression



This sequence diagram paints a clear picture of the sequence of steps that our system undergoes when a user inputs a few operands, chooses the desired operation to be performed and then chooses to execute it, that is click on '='.

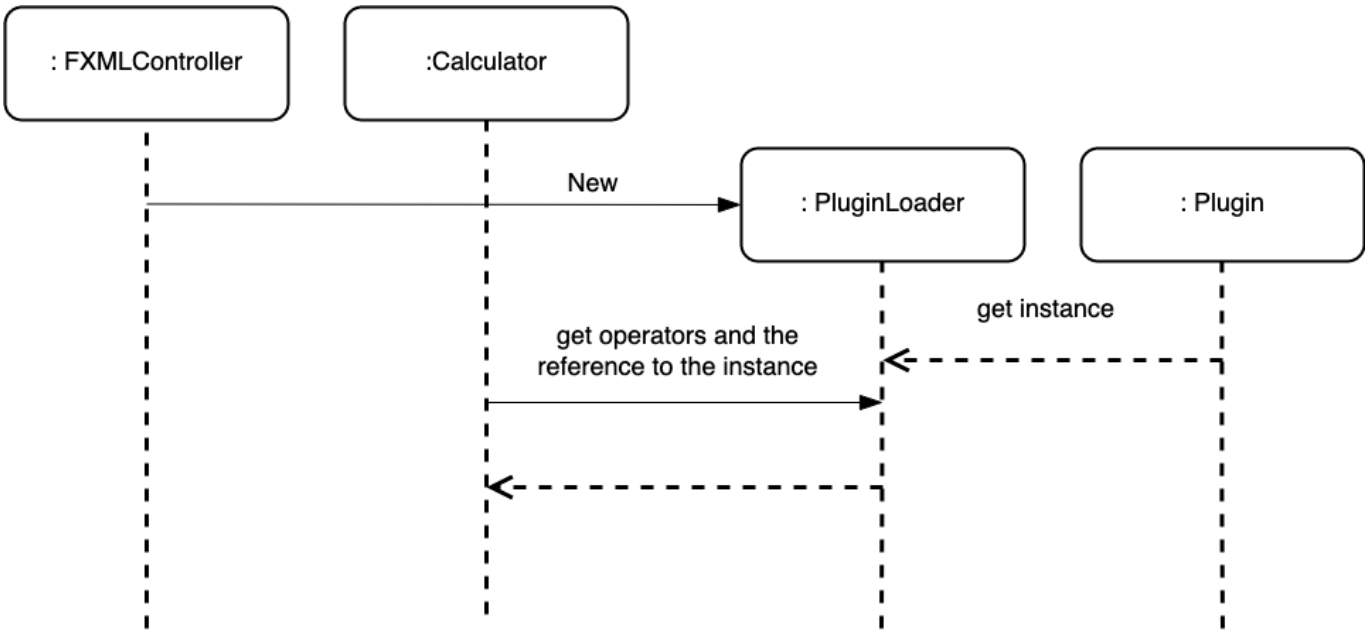
In the beginning, the class FXMLController creates a new class, named Calculator. Next, it creates another new class named Buffer. When the user clicks on the buttons (combination of operands and operators), the input is sent to the Calculator class and displayed to the user. After the user finishes typing in the required expression here, it is executed by pressing the '=' button.

Now, the program enters the stage of evaluating the expression. A loop is initiated and it runs until the entire input line is read. During each iteration, a subsection of the expression (consisting of the operands and the operators) is sent to the PluginLoader class. Here, the result is computed and the calculated

data is sent back to the Calculator class. After the final iteration has taken place, the calculated result sent back to the Calculator class is the actual answer desired by the user.

Once the final result has been calculated, it is sent to the Buffer class. Here, it is stored until the undo or redo button is clicked. Finally, the Calculator class sends the computed result to the FXMLController class. However, if there was an error in the input, an error message is displayed as the final result.

Adding a Plugin



This sequence diagram is used to describe the steps that our system undergoes when a new plugin is added to the calculator.

In the beginning, the FXMLController class created the new class PluginLoader. This class then accepts as input an instance of the Plugin class. Next, the Calculator class sends a request to receive these instances and the references to the corresponding operators to the PluginLoader class. Consequently, it receives the appropriate response and the buttons get initialized.

Implementation

Author(s): name of the team member(s) responsible for this section

Strategy when moving from the UML models to Java code

Our main strategy in this project was that we followed the **agile model** to implement our system. Basically, we follow the following steps;

- Document functionalities and make a pseudo diagrams on google slides
- Make/modify a tentative version of class diagram
- Make/modify a tentative version of class diagram
- Implement Java programs based on the class diagram and object diagram

The good point of this plan is that we don't have to change or modify all our original diagrams when we realize that the modeled system is not realistic or not doable to implement in the context of coding Java.

Key solutions applied when implementing our system

- **Calculation using Reverse Polish Notation**

  Our calculator is designed to get the entire expression, such as "3*(3-2.5)/0.3", as input and return the result of calculation. To enable this calculation to be executed, we had to consider the operator precedence to execute a calculation properly. To solve this problem, we use the following step;

  - ☐ Get an input as in a normal expression
  - ☐ Convert the normal expression input into the reverse polish notation format
  - ☐ Execute a calculation on the RPN format data

  By implementing a method class called RPNMethod, we could execute the ideal calculation. Our design based on this calculation makes our system user-friendly, because the input string is visible and users do not lose what they put as input during the input phase.

- **Implement Plugin Loader**

  To make our calculator extensible with plugins, we made a PluginLoader class. With this class and its methods, users can choose plugins which are stored in a plugin package in a specific path. This class instantiates the plugin selected by the user and makes it accessible by other core systems such as the Calculator object. This is the crucial technology which supports our system to be extensible.

Location of the main Java class for executing our system in the source code

-> src/main/java/backend/Main.java

Location of the Jar file for directly executing your system

For mac users -> out/artifacts/software_design_vu_2020_jar/Calculator-1.0_Mac.jar
For windows users -> out/artifacts/software_design_vu_2020_jar/Calculator-1.0_Windows.jar

30-seconds video showing the execution of your system

- **Demonstration**
  https://youtu.be/NqCsG9ngQrU

- **Instruction in case you can not launch our system through jar files**
  https://youtu.be/YAox4mjsLgA

Instruction about launching our project

In this version of the project, we support **BasicPlugin**, which supports basic operations such as “+”, “-”, etc, and **TrigPlugin**, which supports trigonometry operations such as “sin”, “cos”, “tan”. In the path of **plugins** in our project, you can find the plugin jar files and corresponding implementation zip files. To launch and run our system, you do not need to open the zip files but you can check how they are implemented. As it is shown in the demonstration video, you can follow the steps below to use our system.

- Download our source code from Github
- Run a jar file (**Mac :** Calculator-1.0_Mac.jar / **Windows :** Calculator-1.0_Windows.jar)
- Click the **load** button on the bottom left
- Choose **BasicPlugin** or/and **TrigPlugin** in **plugins**.
- Now you see some operators are installed on the calculator. Enjoy it

If these jar files do not work for you, you can launch your own jar file from

gradle(on top right) -> **build** -> **build**

Or you can run our program, from **src/main/java/backend/Main.java**.

Time logs

Team number	19		
Member	Activity	Week number	Hours
Group	Reviewing Assignment 2	6	2
Hugo & Yudai	Design Pattern	7	2
Hugo & Yudai	Class diagram	7&8	6
Hugo & Yudai	Object diagram	7&8	2
Toyesh & Dora	Sequence diagrams	7&8	6
Toyesh &Dora	State machine diagrams	7&8	6
Group	Implementation	7&8	30
Group	Meeting	6	0.5
Group	Meeting	7	0.5
Group	Finishing document& formatting	8	3
		TOTAL	58