Assignment 3

Team number: 3
Team members

Name	Student Nr.	Email
Isaac Lewis	2687776	i.lewis@student.vu.nl
Martynas Vaznonis	2701013	m.vaznonis@student.vu.nl
Bekir Altun	2694301	b.e.altun@student.vu.nl
Matthijs Hulsebos	2681382	m.c.hulsebos@student.vu.nl

Format: classes start with uppercase letters while methods, fields, and objects start with lowercase. When a name is comprised of multiple words, we use thisNotation instead of this_notation.

Implemented features

ID	Short name	Description
F1	Plugins	The functionality of the calculator can be augmented with 3rd party extensions
F2	CLI	The user can enter operands and operators from any of the modules he has and the interface will show the result dynamically
F3	Arithmetic	The base version of the calculator comes prepackaged with arithmetic modules
F4	Undo	The user can undo their computations one by one
F5	Store	Online plugin store where users can browse and download plugin

Used modeling tool: diagrams.net

Summary of changes of Assignment 2

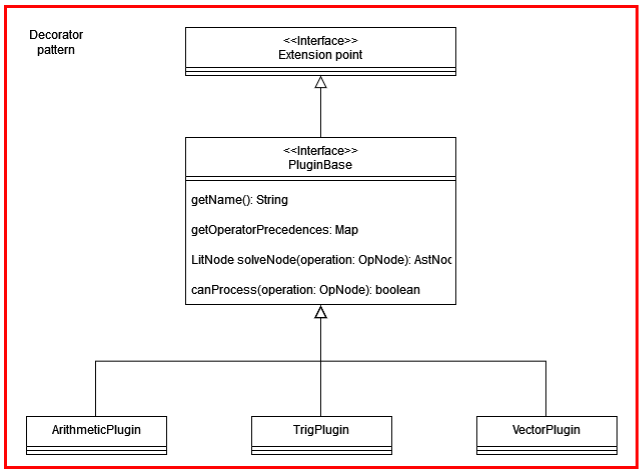
Author(s): name of the team member(s) responsible for this section

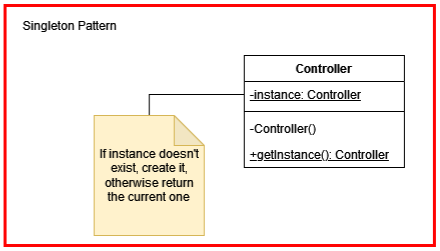
- Added design patterns section.
- Changed the optional fragment into an alternative fragment in the first sequence diagram.
- Changed the class diagram section to include a short overview as well as a detailed description of the diagram.
- Added end state to state machine diagrams and minor improvements in naming
- Object diagram minor update on both the diagram and the description

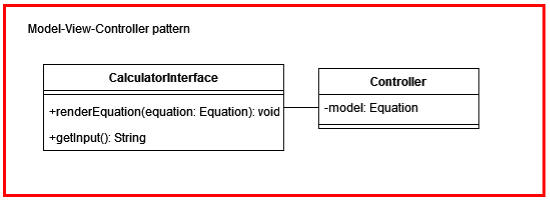
Application of design patterns

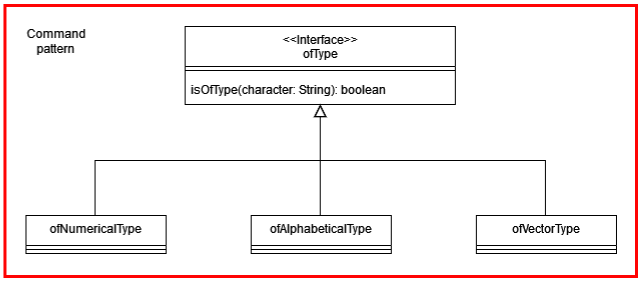
Author(s): Martynas

[Design patterns]









	Decorator
Design pattern	Decorator
Problem	The plugins used in our system inherit from an external library but must also contain a handful of fields and methods specified for our implementation
Solution	The decorator design pattern describes a wrapper class which can be "decorated" with additional fields
Intended use	We create a decorator interface pluginBase which allows us to modify the external interface with the methods we require

	Singleton
Design pattern	Singleton
Problem	There can be at most one controlling object responsible for delegating responsibilities
Solution	The singleton design pattern makes it impossible to have multiple objects of the singleton class by making the constructor private
Intended use	We use the singleton design pattern on our controller class to limit it to one controller which holds needed data and sends requests to other parts of the system

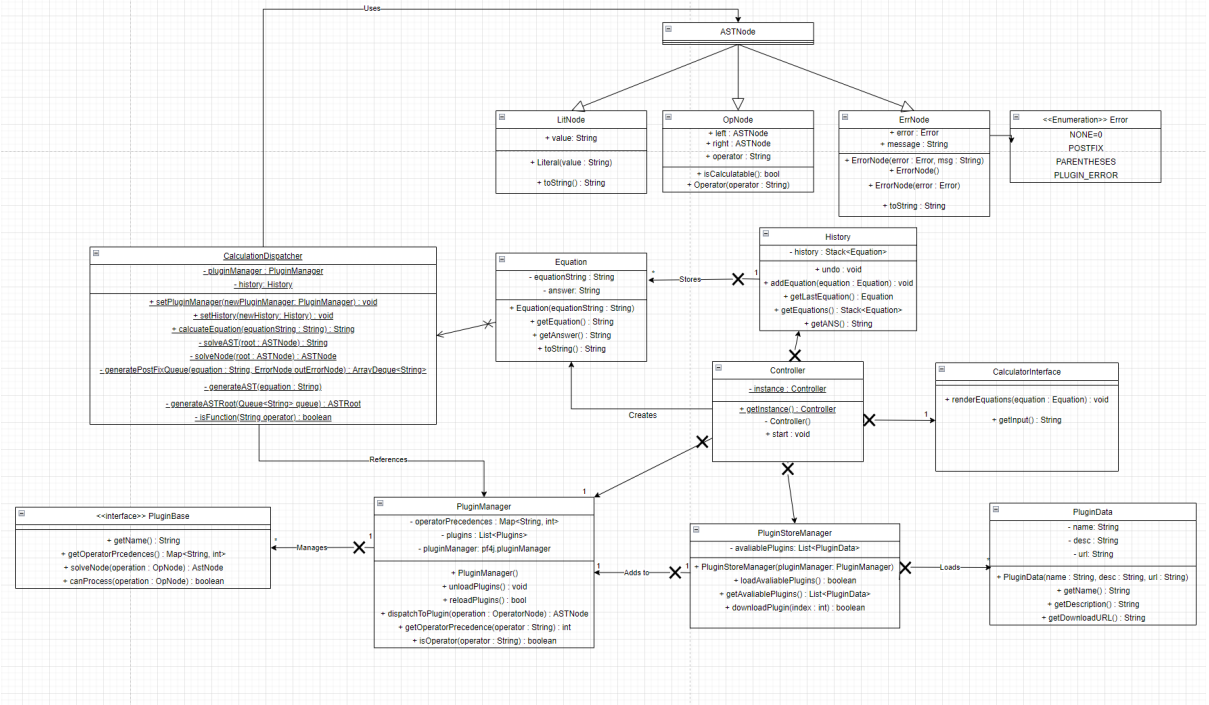
	Command
Design pattern	Command
Problem	One method is responsible for checking if a certain condition is met but the condition is undefined at compile time so a command needs to be passed as an argument
Solution	Using the command design pattern we define anonymous classes and an interface which allows us to send command objects as parameters to the method
Intended use	We use the command objects in a parses which turns the expression string into an array list of tokens which can then be processed
Constraints	The method signature is the same for all command objects which means that we cannot pass arguments for these objects
Additional remarks	This design pattern is also used in the history class to represent undo commands

	Model-View-Controller
Design pattern	Model-View-Controller
Problem	Our current implementation uses the command line as the interface but there is no reason why it should be responsible for managing what happens. As well as that, if we chose to implement a GUI, having the logic code be in the the interface makes it very difficult to transition
Solution	The Model-View-Controller design pattern is a broad pattern expressing that the view which the user sees and interacts with should be separated from the controls of the system
Intended use	We have the controller class be responsible for the inner workings of our system, it also stores the model as the whole model in our project is merely one equation long. The interface is separated into a separate class so that if we chose to transition to a GUI, we would only need to create a GUI class and replace the command line class with it
Constraints	The interface cannot manage anything and an extra dependency is created

Class diagram

Author(s): Isaac

[Class Diagram]



The equation class represents the equation that is entered by the user. It stores the equation string and the answer to that equation. It is connected to the controller which creates it, and it calls on the calculationDispatcher on construction to calculate the answer.

The Controller singleton class takes inputs and gives outputs to the calculatorInterface class. It is the starting point of the whole system, and contains references to all other objects, but it delegates all of its operations avoiding the god class antipattern; it exists only to initialize the other objects, and pass input through. It stores instances of the History, CalculatorInterface, and PluginManager classes. It has the start method which begins the whole program. It also manages the menu navigation, typing quit for example, exits the program

The Calculation dispatcher class is responsible for calculating the result of an equation. It splits the equation up and sends each part to the relevant plugin. The calculateEquation function is passed an equation string. The string is then split into postfix using the generatePostFixQueue function. The generateASTRoot function transforms this queue into an abstract syntax tree using the ASTNode family. It then recursively solves the AST by traversing the tree until it finds an operator node with 2 literals. It then sends the node to the PluginManager to dispatch to the correct plugin. At any point an ErrorNode could be created, which consumes the tree. At the end, there should be one node left, and the string value of that node is returned to the calling equation. If it's a value the value is returned, if it's an error message, the error message is returned.

The ASTNode class is an abstract class meant to be the polymorphic base type of all AST elements. The first child is the LiteralNode, which simply stores a value found to be a literal. The next child is OperatorNode, which stores a string containing an operator, as well as up to two children. The final node is the error node which stores an enum representing an error. These are small data structures so all attributes are public to allow for simple manipulation.

The History class stores the previously computed equations and is used for undoing operations. It contains an array field with every equation and has a method for undos.

The CalculatorInterface class is used to interact with the user. It takes in inputs, sends them to the controller and gains outputs from the controller which are displayed for the user.

The PluginBase is an interface used for inheritance for creating plugins which calculate the needed formulas. It has a method getOperatorPrecedences() which is essential for the shunting yard algorithm to work as it informs it of the operator precedences. The canProcess() function takes in an operator, and returns a boolean representing if the plugin knows how to handle the operator. This is a really flexible system as it allows for plugins to define what a value and operator is; a number isn't really hard coded, so plugins can choose to accept arbitrary values, such as representing vectors with [1,2,3] or imaginary numbers with i. the plugin sees if it understands these values. The final function is the solveNode() function, which takes an OperatorNode and returns a LiteralNode containing the solution to the operator.

The PluginManager() is used for managing plugins. It registers the operators from the plugins, dispatches operations to plugins and so on. It serves as a bridge between the plugins and the rest of the systems. The reloadPlugins() function reloads the plugins from disk, as well as populating the operatorPrecedences dictionary. The getOperatorPrecedence() function is used to retrieve a value from the dictionary using an operator string as a key. Finally, the dispatchToPlugin() takes an OperatorNode, goes through each of the plugins to find one that knows how to solve the operation, returning a Literal from the plugin that knows how to solve the Operation.

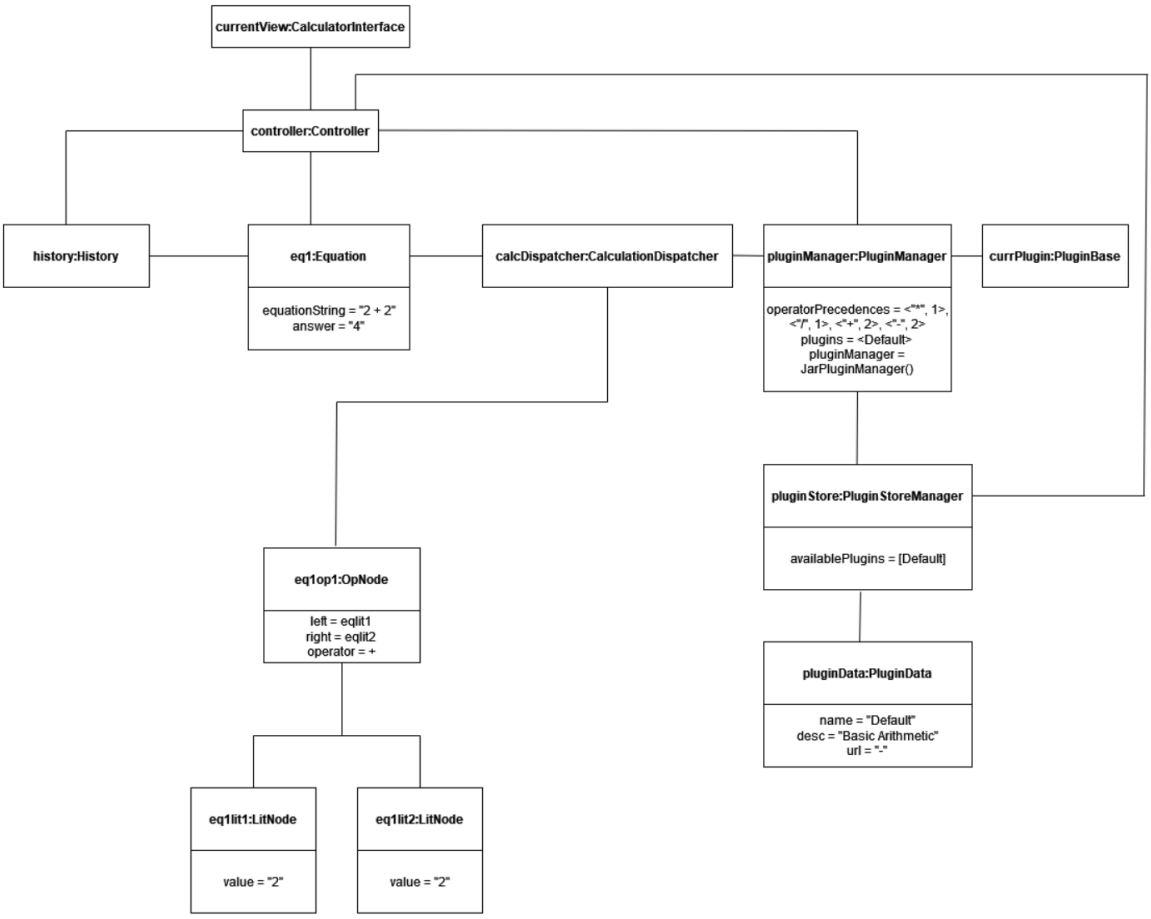
The PluginStoreManager is the object responsible for contacting the PluginStore server, retrieving a list of available plugins, and downloading one. The PluginData class is an immutable helper class that simply stores the information about a plugin on the remote server.

Most relevant attributes:
-Calculation dispatcher is responsible for managing the equation and necessary plugins to calculate the equation.
-Interface displays whatever is received from the controller.
Most relevant operations:
-Calculating input
-Downloading additional plugins from the PluginStoreManager class.
-Undo commands.
Most relevant associations:

-Controller acts as a link between the user interface and the needed functionality.
-PluginData acts to store data from online plugins client-side through the PluginStoreManager.

Object diagram

Author(s): Bekir



The interface accepts input which is sent to the controller. The controller creates an instance of the equation class based on the input. The equation gets handled by the CalculationDispatcher, which asks the precedence of operators to the PluginManager. Based on the precedence of operators an AST tree is made out of the equation. If the nodes of the tree contain only two literals, they are sent as a simple equation to the PluginManager so that an appropriate plugin as an instance of PluginBase can solve that part of the problem. In this way the whole tree is “solved” recursively. The final answer is returned to the Equation class. With this, the history class is updated with this solved instance of Equation, and the answer is displayed on the CalculatorInterface through the controller.

The PluginStoreManager is only used to download more plugins, and PluginData stores data about them on the client side.

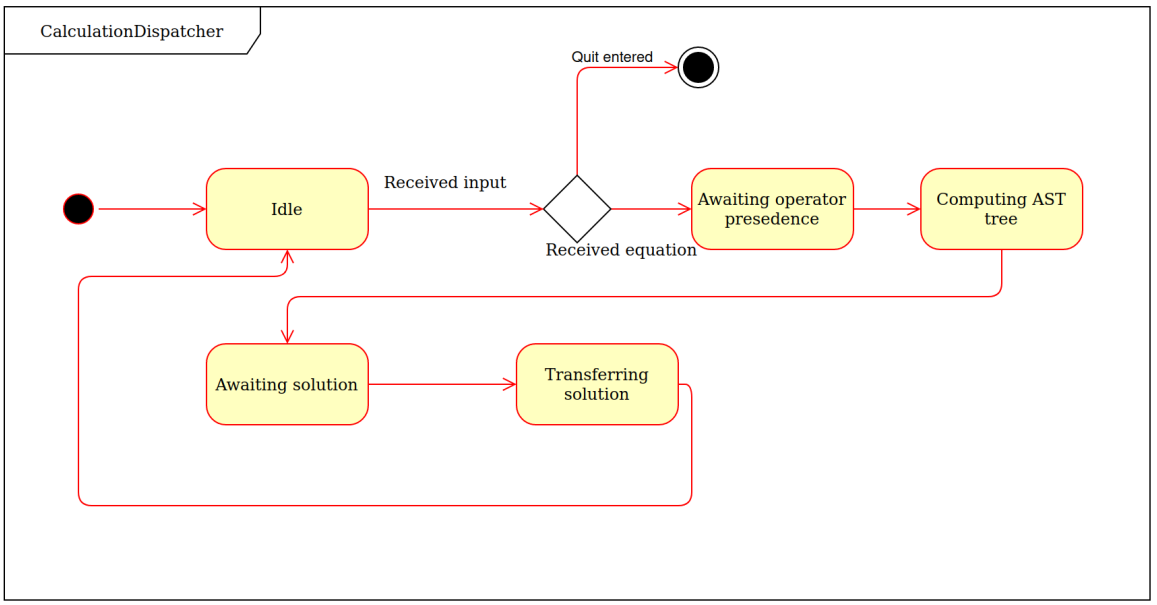
State machine diagrams

Author(s): Matthijs

CalculationDispatcher

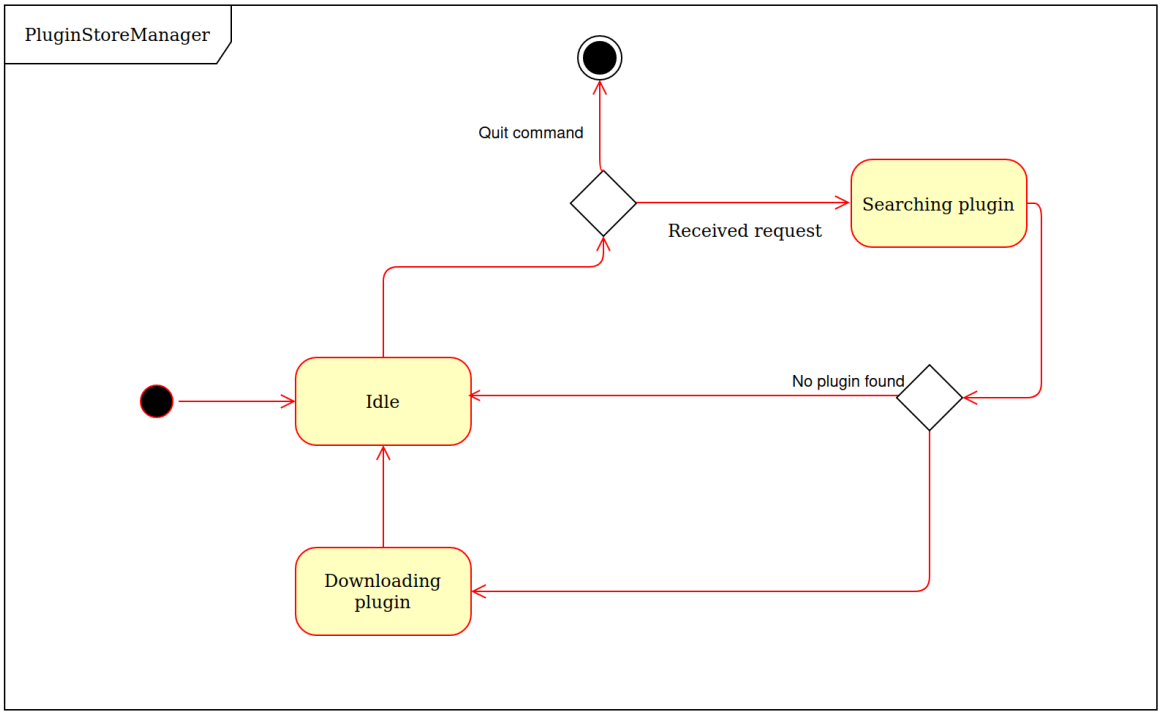
The calculationDispatcher class consists of five states that interact with objects of the Equation, PluginManager, and OperatorNode classes. All the existing states listed are: Idle, Awaiting operator precedence, Computing AST tree, Awaiting solution, and Transferring solution.

Initially, the dispatcher receives an Equation object for which an AST-tree must be created so that a plugin can compute the outcome or a quit command in which case the final state is reached. When the equation input is received by the dispatcher, it leaves the idling state and requests the operator precedences so that it can create nodes for the AST-tree. When the AST-tree has been established it leaves the state to send a request to the pluginManager in order to have the correct corresponding plugin(s) solve the AST-tree and return a solution. Finally, when the solution has been computed and returned to the dispatcher by the pluginManager, the CalculationDispatcher object sends this back to the Equation object. After successful completion of the calculation flow the CalculationDispatcher returns to its Idle state waiting for the next equation input that requires computation.



PluginStoreManager

The PluginStoreManager class is concerned with retrieving plugins from the ‘store’ and downloading one of them. The class can arrive in three states: Idle, Searching plugins, and Downloading plugin. When the PluginStoreManager receives a request to retrieve available plugins it leaves its Idling state and transitions to the Searching plugin state or quits the process if a quit command is received. If no available plugins are found the PluginStoreManager returns to the Idle state. Otherwise a plugin is downloaded before returning to the idling state.



Sequence diagrams

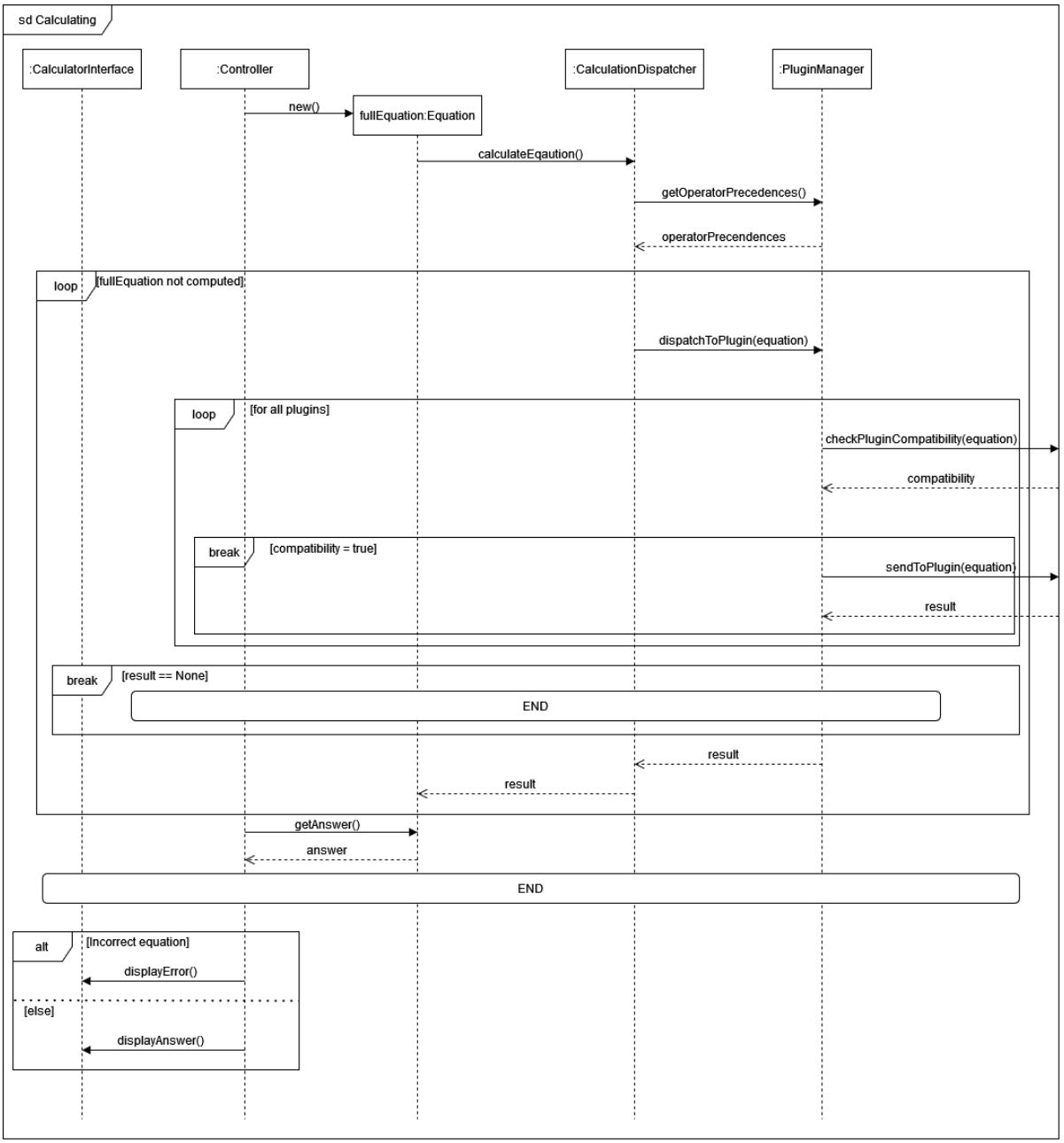
Author(s): Martynas

The "Calculating" sequence diagram shows the calculation steps taken between entering an equation and displaying the answer for the user to see. When the user enters something, a new full equation object gets created. That object then requests the precedents of the operators available in the currently loaded plugins and based on those priorities break down the equation into a tree such that the equation can be computed stepwise. Every operator node is then sent to the plugin manager object with postorder traversal. The plugin manager attempts to find a plugin which accepts the current equation. If none are found, the calculations end and an error message is displayed. If a module which accepts the equation is found, the equation is sent to it, the results are computed, and the loop is ended prematurely as there is no point in further looking for a module to accomplish the computation. The result for that partial equation is used to update the tree and inserted into subsequent partial equations. When every partial equation has been computed, the answer to the full equation is known and the outer loop is finished. The answer to the full equation is then the answer to the last partial equation. This answer is sent to the view controller which displays the answer for the user using the calculator interface.

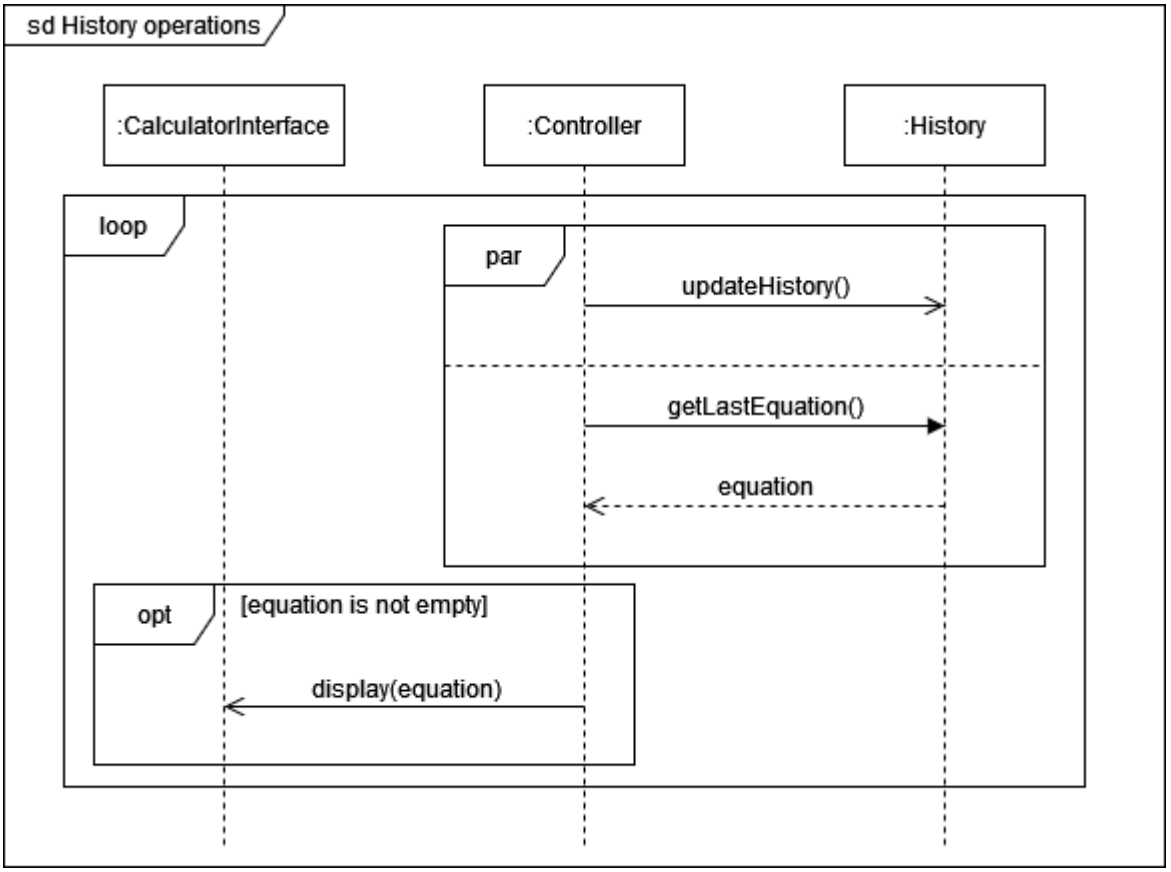
The plugins to which the equations are sent are not modeled here because they are variable and unknown. Instead gates are used to show that the sequences are going to some third party module.

The "History operations" sequence diagram shows how the history class gets updated and the use it provides for undo operations. The diagram is encased in a perpetual loop as it always updates the history class with the most recent new equation and there is no limit to how many equations can be entered in any particular session. When the user requests to undo, the history class pops and returns the latest previous equation. If it is not empty the old equation is displayed for the user to adjust and change. If the equation is empty (e.g. the user undos before entering any equation), the controller simply does not ask the interface to display it thus defining errors out of existence and adhering to the principles of the book "A Philosophy of Software Design". The sequence diagram is also open for easy modifications if we decide that such modifications are necessary or otherwise desirable. We can extend the parallelization combined fragment to support redo operations. This would also have only minimal effects on the implementation such as instead of using a stack and pop operations, we use a linked list and indexes for moving through history. Furthermore, updateHistory() method would also need to be changed slightly to support overriding future history once an edit has been made in a previous equation but this wouldn't change the sequence diagram.

Calculating



History operations



Implementation

Author(s): Isaac and Martynas

VIDEO OF PROGRAM:
https://drive.google.com/file/d/1ZvFQtITjS0lsQO2bm4naNJII5gvi6_63/view?usp=sharing

We started the implementation by transcribing the class diagram into a skeletal code foundation with every class, their methods and fields. A few things became clear while doing this and some refactoring was undertaken both in code and in the class diagram. After adding everything described in the class diagram, we began looking at the object diagram to make sure that the correct state with the right object instances can be reached. To implement functionality, we looked at the sequence diagrams and modeled the necessary methods based on them. It took a few iterations of implementing and refactoring to get everything to work and then to split up large methods and move them to more appropriate locations.

We implemented a few design patterns useful for our program. We used the model-view-controller pattern for displaying and managing information, we used the singleton design pattern for the controller class. Furthermore, we defined errors out of existence and instead for every improper operation the program prints the error message for the user to see and allow him to continue using the calculator.

Implementation came with numerous issues and problems such as reading inputs from the console which gradle brushes over unless specified otherwise in the gradle.build file. Furthermore, we initially could not load the PF4J library which we need for the plugin implementation. A large issue was the architecture of the gradle system, as there needed to be the main application, the common API, and the plugins, all separately built but linked together. How the gradle worked together was a big issue throughout the implementation, and took hours of debugging to get to to work

Alongside the gradle issues, we encountered a number of problems with the computation of the answer. Ultimately, most of the issues were resolved by using an established 15-point priority system for different types of operators and the shunting yard algorithm for transforming the equation into postfix notation.

The main java file is in app\src\main\java\softwaredesign

The Jar file is built to app\build\libs upon running gradlew build. The plugins are automatically built and put into app\build\libs\plugins. The fat jar that can be runned immediately after cloning is in the out folder.

We have added 3 plugins to showcase both how they are implemented and that the calculator works. The libraries are for arithmetic, which has basic operations such as addition, multiplication, and so on, trigonometry, which has the basic trigonometric functions, and vectors, which supports vector addition and scalar multiplication of vectors.

To see how to set up and run the program, see the primary readme file in branch “Assignment-2” of our GitHub repository. A jar has also been put in out\, as instructed.