PostFix Calculator

Design Document

Version 2.0

Team 2

Chauhan, Yash
Jin, Hui
Khan, Akbar
Martintsov, Vladimir
Nguyen, Andrew
Podimov, Michael

Prepared for
EECS 2311 - Software Development Project
Instructor: Gunnar Gotshalks
Winter 2016

Contents

1. Introduction

This document provides a detailed description of the design that was chosen for the calculator as well as rejected alternatives.

The chosen design architecture for this calculator is Model-View-Controller (MVC). This design architecture is perfect for any project that uses a Graphical User Interface (GUI), as it separates the different components and allows for easy implementation, testing and maintenance of each component individually.

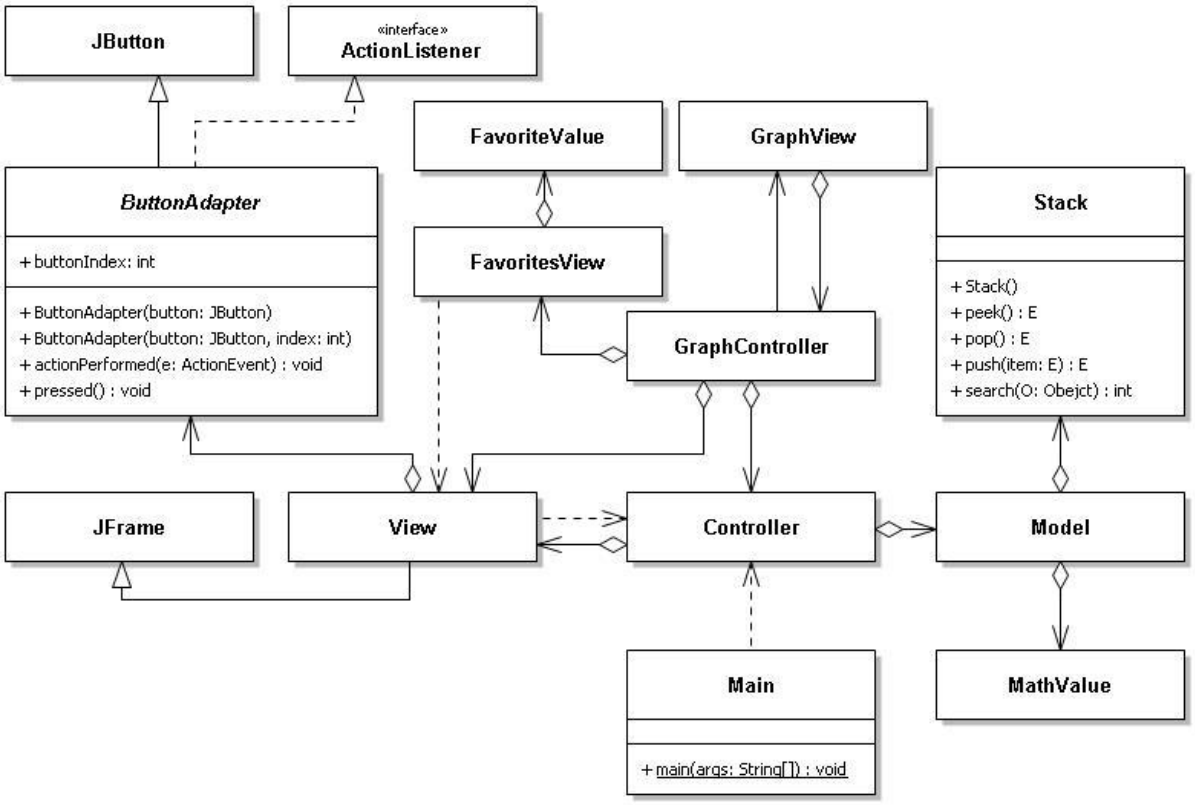


Figure 1 – General UML diagram of the calculator

As can be seen in the UML diagram above, the MVC architecture is flexible, and allows for additional classes to exist in the system, as long as these classes do not violate the MVC design principles. Additional classes, like in this project, are often used to simplify the implementation of the MVC architecture.

The program itself does not require a lot of space: the program package consists only from 15 classes, 5 of which are Test Classes, a separate Button Adapter class, Model View Controller structure classes main, favorites and graphing functionalities(8 classes), and an extra class that stores the points to be graphed.

2. The Model Object

The Model object in the MVC architecture is responsible for processing the user input, data storage and providing the desired output. This is achieved using different operations.

In this calculator, input comes in 2 forms, numeric input and operations:

- Numeric input is stored in **input_value : StringBuilder** while the user is typing.
- An operation is an input that tells the Model to do something with the numeric input, and/or the stored data.

Since this is a postfix calculator (refer to the Requirements Document for additional information), this means that the latest input will be used for any operation. Because of this, a Stack data type is used to store the calculator data. The Stack data type fits perfectly since it uses the "last in, first out" (LIFO) design. This means that whenever an operation is required, the data is taken from the top of the stack.

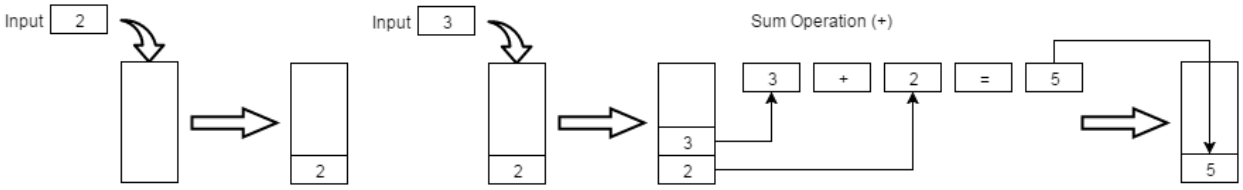


Figure 2 – General operation of the stack in the calculator

The Model uses the following stacks to store relevant data:

- **Calculation_Stack : Stack<MathValue>** – holds the numeric data used for mathematical operations, such as input and the results of mathematical operations.
- **Previous_calculations_Stack : Stack< MathValue >** – holds the values used to produce the results of the mathematical operations that are stored in the **Calculation_Stack**. This is needed for the **undo()** operation (see below).
- **History_Stack : Stack<String>** – holds the string representation of input and mathematic operations (E.g. an addition operation will stored as a "+").
- **Print_Stack : Stack<String>** – a temporary stack used by the **printHistory()** operation (see below), to convert the postfix input, to infix output.
- **Comma_Stack : Stack<Integer>** – Another stack used for convenience to keep track of the right most comma to help with the postfix to infix conversion.
- **precedenceStack : Stack<Integer> -** Holds precedence values for operators and number. Used in the **printHistory()** to minimize the number of parenthesis in the **history_value** (see the Requirements document).

The calculator displays 4 forms of output:

- **history_value : StringBuilder** - The calculation history so far. This includes all the input that was accepted by the calculator and is stored in the **History_Stack**, as well as any operations that might have occurred. This output is generated by converting the postfix data in the **History_Stack** to infix notation.
- **input_value : StringBuilder** - the current input typed by the user, or the result of the last mathematical operation if it occurred, as long as neither of the operands are variables.
- An error message will be printed in **input_value** if a mathematical error has happened (refer to the User Manual for more information).
- Graph of the top expression in the **Calculation_Stack : Stack<MathValue>** stack.

2.1 MathValue Data Type

MathValue is the data type that is used in this project. It is used to store the numeric data of the input and operation results in **Calculation_Stack** and **Previous_calculations_Stack.**

Each value of this data type is stored in an array of type double**, value : double[].** The array represents the points on the y-axis of the function that would represent this value. For example, an input of 5 will be stored as an array, where for each index in the array a value of 5 is stored. This correspond to the function y = 5.

The value can also represent a variable **X** (y = x), for a certain domain. The domain chosen for this calculator is from -50.0 to 50.0 with 1000 points in this interval, giving an accuracy of 0.1.

To distinguish between values that represent a variable and values that represent a constant, a Boolean variable is used, that when set to **true**, the value is a variable and **false** otherwise.

Operations on this data type, involve the use of a *"for loop"*, where the values at each index of the array are operated on.

Results of operations that involve variables are also variables, and results that involve only constants are also constants.

In this implementation, one assumption that was made, is that a **MathValue**, (whether it is an input, or an operation result) will not be modified once created, and so it is immutable. After being created, a value can be used for output, operations, can be stored in **Calculation_Stack** or **Previous_calculations_Stack** or destroyed. This means that an operations result is in fact a new value.

2.2 Operations

The following is a description of different types of operations that are performed by the calculator.

Input

- **numeric_input(ButtonName : String) : void**
  - **ButtonName** a number 0 – 9 or a decimal point.
  - **ButtonName** is appended to **input_value,** ensuring proper format for a real number.

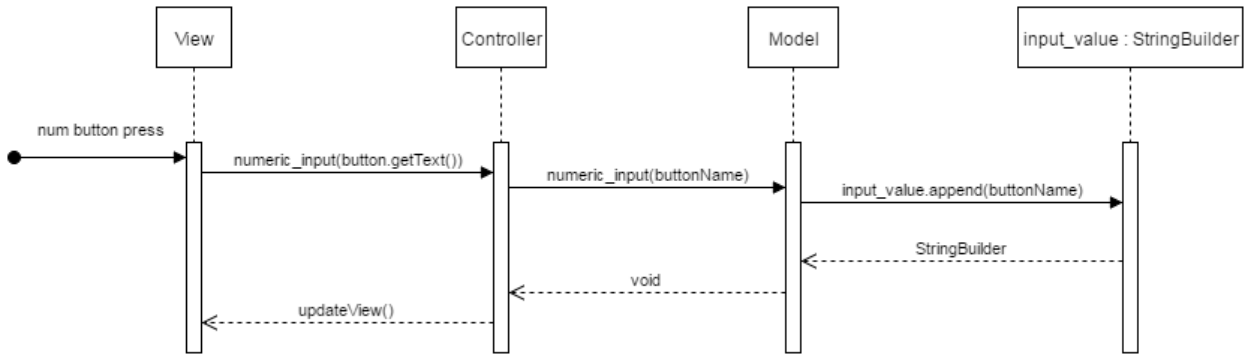


Figure 3 – Sequence diagram of the **numeric_input(ButtonName : String) : void** operation.

- **changeSign() : void** – Changes the sign of the **input_value** currently being input.
- **pi() : void** – A mathematical constant. The numeric value is pushed into the **Calculation_Stack** and the symbol **π** is pushed into **History_Stack**.
- **x() : void** - A variable type representing the **y = x** function. Pushed into the **Calculation_Stack** and the symbol **x** is pushed into **History_Stack**.
- **enter() : void** – Pushes the numeric value of **input_value** to the top of the **Calculation_Stack** and the string representation into **History_Stack**.

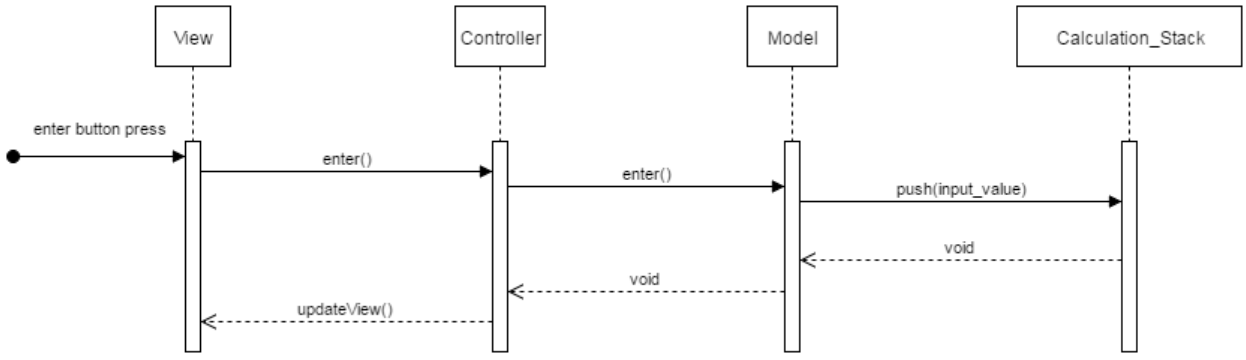


Figure 4 – Sequence diagram of the **enter() : void** operation.

Mathematical

- **sum() : void**– Pops 2 values from the **Calculation_Stack**, performs the sum operation and pushes the result back to the **Calculation_Stack**. The 2 values are pushed in the same order that they were into the **Previous_calculations_Stack**. The symbol **" + "** the operation is pushed into the **History_Stack**.

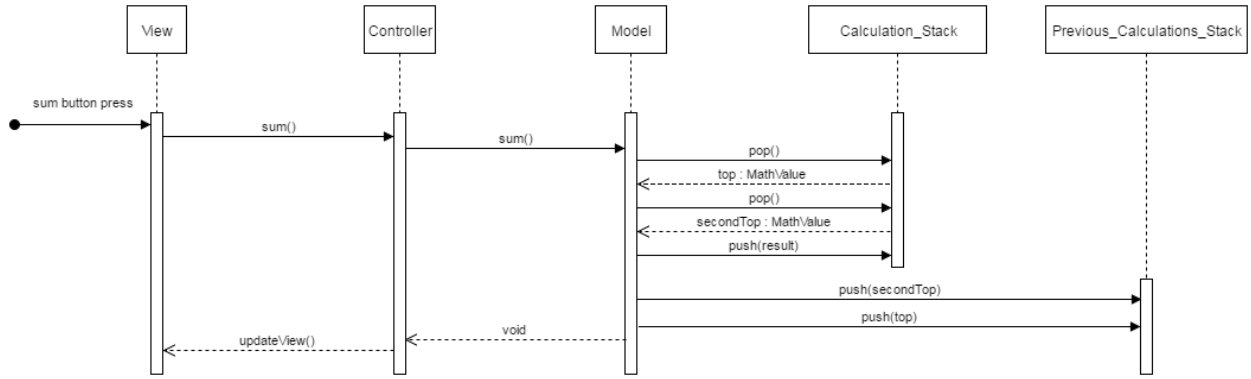


Figure 5 – Sequence diagram of the **sum() : void** operation. All the other mathematical operations perform in a similar manner.

- **subtract() : void**– Pops 2 values from the **Calculation_Stack**, performs the subtract operation and pushes the result back to the **Calculation_Stack**. The 2 values are pushed in the same order that they were into the **Previous_calculations_Stack**. The symbol **" - "** the operation is pushed into the **History_Stack**.
- **multiply() : void**– Pops 2 values from the **Calculation_Stack**, performs the multiplication operation and pushes the result back to the **Calculation_Stack**. The 2 values are pushed in the same order that they were into the **Previous_calculations_Stack**. The symbol **" × "** the operation is pushed into the **History_Stack**.
- **divide() : void**– Pops 2 values from the **Calculation_Stack**, performs the division operation and pushes the result back to the **Calculation_Stack**. The 2 values are pushed in the same order that they were into the **Previous_calculations_Stack**. The symbol **" ÷ "** the operation is pushed into the **History_Stack**. Division by zero should present an error to the user and reset the calculator using **clear() : void.**
- **sine() : void** – Pops the top value from the **Calculation_Stack**, performs the Sine operation and pushes the result back to the **Calculation_Stack**. The value is pushed into the **Previous_calculations_Stack**. The string **"sin"** is pushed into the **History_Stack**.
- **cosine() : void** – Pops the top value from the **Calculation_Stack**, performs the Cosine operation and pushes the result back to the **Calculation_Stack**. The value is pushed into the **Previous_calculations_Stack**. The string **"cos"** is pushed into the **History_Stack**.
- **factorial() : void** - Pops the top value from the **Calculation_Stack**, performs the factorial operation and pushes the result back to the **Calculation_Stack**. The value is pushed into the **Previous_calculations_Stack**. The "**!**" symbol is pushed into the **History_Stack**. Performing the factorial operation on anything but a whole number (0, 1, 2, …) will present an error to the user and reset the calculator using **clear() : void.**

Utility

- **clear() : void** – Resets the calculator back to the initial state. This clears all the data stored in all the stacks.
- **undo() : void** – Reverts the previous operation. If the operation was typing the input, the last digit typed in **input_value** is deleted until the **input_value** is empty. After the **input_value** is empty, reverts the previous mathematical or **enter()** operation, and shows the result in **input_value**. If the previous operation was mathematical, the top value in the **Calculation_Stack** is removed, and either 2 or 1 values are pulled from the **Previous_calculations_Stack**, based on whether the mathematical operation was binary or unary, and pushed into the **Calculation_Stack**, to show the values used to get the result of the mathematical operation.
- **changeSign() : void** – When no input is being typed, changes the sign of the last expression in **History_Stack**.
- **enoughOperandsBinary() : void**– If the Calculation Stack does not currently have enough operands for a binary mathematical operation, push 0 (zero) into the **Calculation_Stack** and "0" into the **History_Stack** as a substitution for missing operands (see the Requirements Document).
- **enoughOperandsUnary() : void**– If the Calculation Stack does not currently have enough operands for a unary mathematical operation, push 0 (zero) into the **Calculation_Stack** and "0" into **History_Stack** as a substitution for missing operands (see the Requirements Document).
- **checkPrecedence(precedenceStack : Stack<Integer>, value : String) : int - c**hecks the precedence of the current binary operator, compared to previous ones.
    - **precedenceStack** - Holds precedence values for operators and number.
    - **value –** the string representation of the current binary operation.

  Returns 0 - If the current operator does not have precedence
  1 - If the current operator has precedence over the left expression.
  2 - If the current operator has precedence over the right expression.
  3 - If the current operator has precedence over both expressions.

Output

- **getInputValue() : String** – Returns the string currently stored in **input_value**.
- **getHistoryValue() : String** – Returns the string currently stored in **history_value.**
- **printHistory() : void** – Generates the infix representation of the data stored in **History_Stack** and stores it in **history_value**.
- **updateOperationValue(result : MathValue) : void** – Checks the result of the last mathematical operation. If the result is a variable, **input_value** will displays **0.** If the **result** is not a variable and is a whole number it is stored in **input_value** in integer representation, otherwise it is stored as a real number representation.
- **getFunction() : double[]** – Returns the **value : double[]** of the top **MathValue** in **Calculation_Stack.**
- **getEquation() : String –** Returns the right most equation of the **history_value.**

2.3 Operations interaction

- The **enter()** operation is invoked either by the user, or by any mathematical operation if the user was in the middle typing an input, when the mathematical operation was called.
- The **pi()** operation invokes the **multiply()** operation if the user was typing an input. This improves user experience by allowing the user to type expressions such as 3π, which mathematically mean 3 x π.
- Every binary mathematical operation invokes the **enoughOperandsBinary()** operation before performing the mathematical operation.
- Every unary mathematical operation invokes the **enoughOperandsUnary()** operation before performing the mathematical operation.
- The **clear()** operation is invoked either by the user, when a mathematical error occurs during a mathematical operation or when the **undo()** operation has nothing left to undo.
- The **printHistory()** operation is invoked after every **enter()**, **undo(), changeSign()** or mathematical operation.
- The **updateOperationValue(result : Double)** is invoked after every **enter()**, **undo()**, **changeSign()** or mathematical operation.
- The **printHistory()** operation invokes **checkPrecedence(precedenceStack : Stack<Integer>, value : String)** whenever it tries to print a binary operator.
- Performing the **changeSign()** operations twice in a row, returns the expression or input to its original unaltered form.

2.4 Converting postfix into infix

The data stored in **History_Stack**, must be converted from postfix to infix. In truth, the data is never actually converted from one notation to another, but instead it is represented in a different way using **printHistory().** The general operation of **printHistory()** is illustrated in the following diagram.

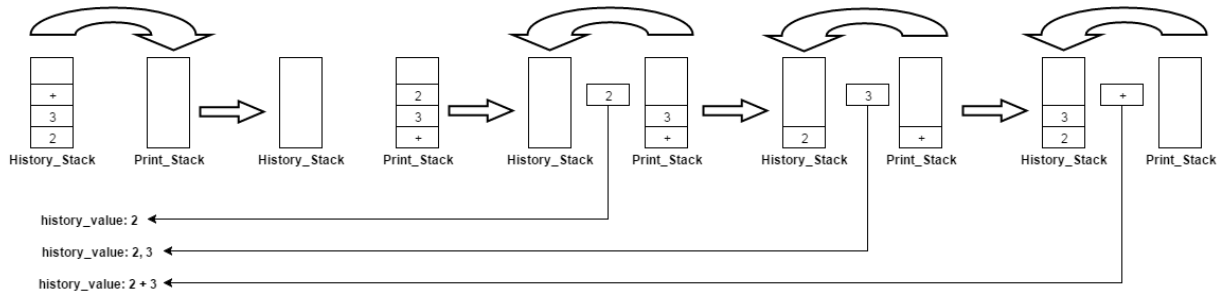


Figure 6 – Abstract representation of how the **printHistory()** operation works

If an operand is to be printed, a comma is appended to the **history_value,** followed by the operand. The comma position in the **history_value** is then stored in the **Comma_Stack.**

If an operator is to be printed, it is printed at the same position as the rightmost comma, overwriting the comma **history_value**. Once overwritten, this comma position is discarded. The rightmost comma position is always at the top of **Comma_Stack.**

It is now also easy to see how the **undo()** operation works. The **undo()** operation does not manipulate the **history_value**, instead it just pops the top of **History_Stack**, and then invokes the **printHistory().**

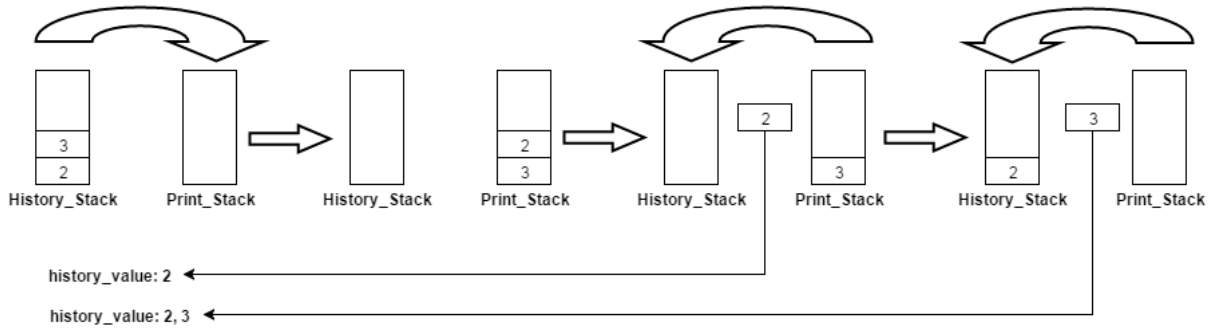


Figure 7 – Same as figure 6, but the top most element of the **History_Stack** was removed.

2.5 Alternative Design

It should be noted that it is a viable option to store the results of the **printHistory()** operation in **History_Stack**, instead of storing the elements need to create the results each time.

This option requires a different approach to input handling, so instead of simply pushing the input into the **History_Stack**, a copy of the latest history string has to be made and changed based on the current input, and then pushed into **History_Stack** as the latest history entry.

The tradeoff between this design and the chosen one, is that, the chosen design has trivial input handling, but requires some work to present the output in proper form, while this alternative design allows for trivial output handling, while requiring extra work in input handling.

2.6 Calculating the points of a function expression

To be able to graph a function expression such as **cos(x),** the y – axis points of the function have to be calculated. This is achieved by creating a variable x, which represents the function **x**, with points over a certain interval (-50.0 to 50.0 in this case), and calculating the cosine of each point. The resulting set of points will correspond to the y - axis points of the function **cos(x)**. This resulting set of points is saved as a new **MathValue,** and pushed to the top of the **Calculations_Stack**.

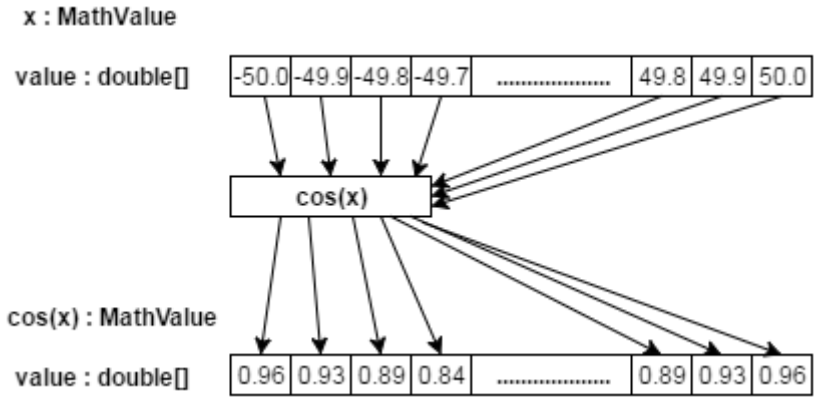


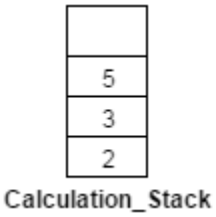
Figure 8 – Calculation of all the points of **x** to get all the points of **cos(x)**.

The resulting points can then be used to calculate more functions, such as **x * cos(x)**, by creating another x variable and multiplying all of its points with all the points of **cos(x)** that have been calculated before.

2.7 Rejected Designs

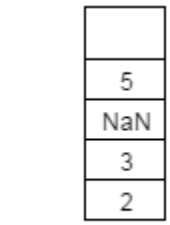
In order to try and minimize the number of **Stacks** used, some ideas were proposed to try and eliminate the need for **Previous_calculations_Stack.**

The first idea was to store both the values used in a mathematical operation and the result all in the same stack, such that the result is always on top of the values used to create it.



The problem is immediately evident, since we cannot distinguish between input and results, such that if the **sum()** operation is invoked on the **Calculation_Stack** in its current state, we will sum 5 and 3 instead of 5 and a substituted 0.

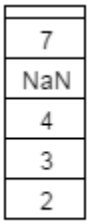
A solution to this design problem was presented, by providing a “space” between the result and the value. This “space” can be a special number used only to indicate a space. The special number **Double.NaN** was used.



Calculation_Stack

At a first glace this solution seems to work, since we can check for the space, and if it exists we would treat it as the bottom of the stack for the current mathematical operation.

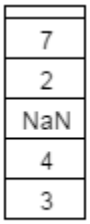
The problem occurs in the following scenario, where the user input was 2, 3, 4, +.



Calculation_Stack

If a binary operation such as **sum()** is invoked on the **Calculation_Stack** in its current state, instead of adding 2 and 7, we will add 7 and a substituted 0. Essentially all the values that were possibly below the top values needed for a mathematical operation, are cut off from future operations.

A solution to this design problem was to carry any values below the top values needed for a mathematical operation over the space, followed by the result.



Calculation_Stack

This solution was deemed too complicated for such a simple task and was rejected in favor of having **Previous_calculations_Stack**.

3. The View Object

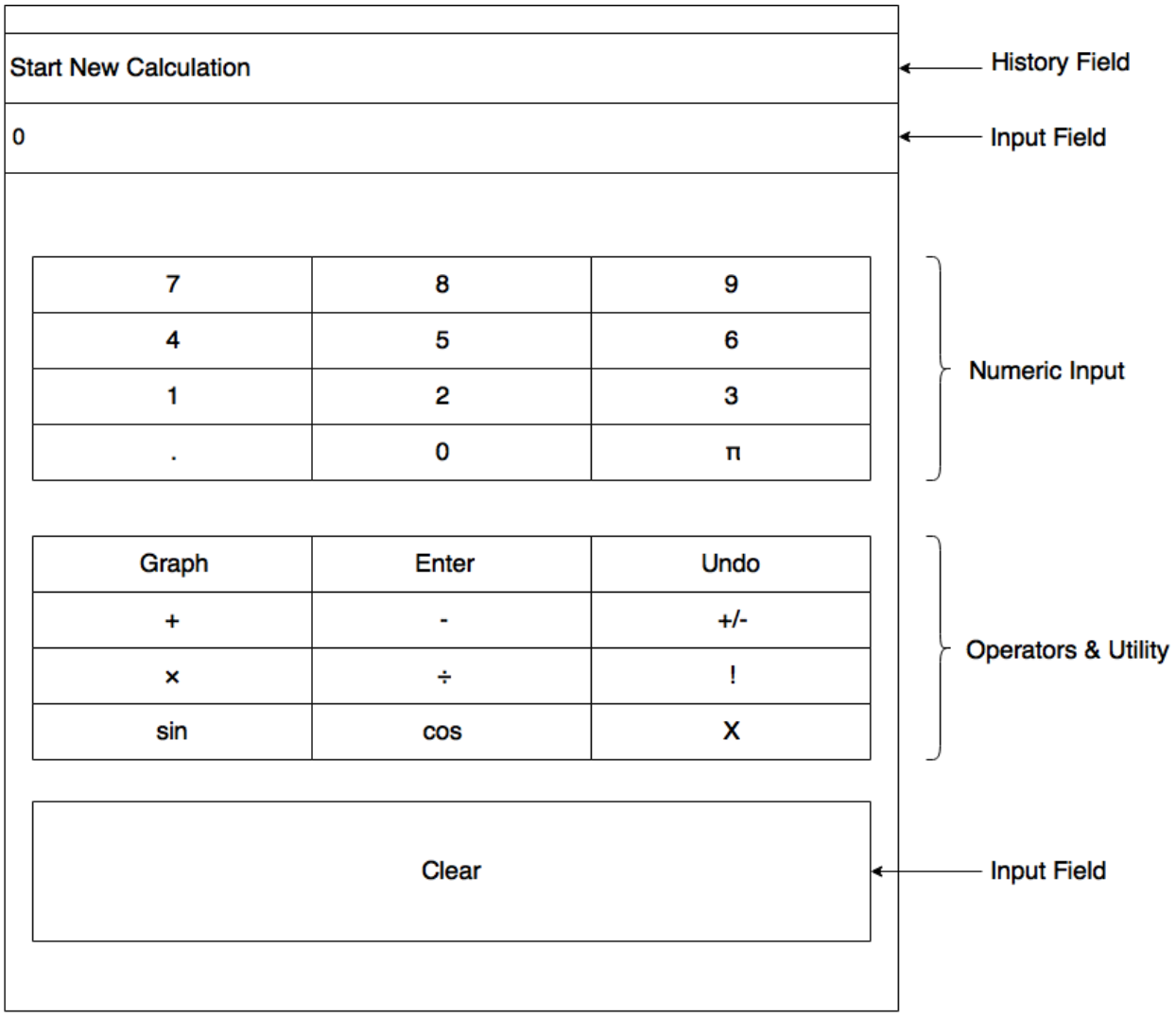


Figure 9 – The general layout of the Calculator UI

The View object in the MVC architecture is responsible for interaction with the User through the graphical interface, and sending the input to the Controller object. Then, the Controller object will send the information to the Model object, which will do the data processing. This is achieved using an extra simple class called ButtonAdapter, which allows to create an ActionListener and attach it to JButton object. As a reminder, ActionListener - an abstract class in Java - allows the buttons to be become functional and execute some operation upon a mouse click. Please refer to the sections below for additional information.

3.1 CalcView Class

This class implements the graphical interface of the program. The picture above shows the layout of the user interface. So called Main View, it has buttons and ButtonAdapter's for them and is

implemented in the this class as a constructor, where all the necessary planes and buttons were declared and allocated in the window as shown in the Figure 9. In the constructor of the CalcVIew class there are some additional methods that are called to reassure flawless program execution as well as programmer-friendly maintenance opportunities. Those methods are described below.

- **betterViewDesing(): void -** Allocates certain colors to the background of the panels and textFields of the View object. The colours of the **historyDisplay** and **display -** the textFields that hold **history_value** and **input_value** respectively, are also defined in this method. Note that the colours are picked according to the design.
- **buttonColor(): void -** Allocates specific background colours to the buttons grouped by certain functionality. The colour of the buttons' text is also set up here. Again, the colours are picked according to the design of the program.

The two methods mentioned above safely configure the colours of the elements: by calling those methods in the main constructor of the View, the main UI gets its color theme.

3.1.1 Additional Operations

For the CalcController object to update the data processed the Model on the CalcView objects, there are two additional methods implemented in the CalcView class. They are as follows:

- **setDisplayText(String): void -** Updates the textField that holds the **input_value** of the Model object.
- **setHistoryText(String): void -** Updates the textField that holds the **history_value** of the Model object.

3.2 Graph View (DrawFunction Class)

The GraphView - a UI that allows user to see the plot of the graph - is implemented in DrawFunction class of the program. Using additional methods described later in this section, this class's constructor produces a new separate JPanel with a graph of the function taken from the Model object, exactly whenever the "Graph" button is clicked. This allows the program to "keep in memory" the panel with the current graph when switching between the Main View and Favorites View (CalcFavorites), until a new function is graphed, at which point the previous instance of the Graph View object is destroyed. To be created, the DrawFunction constructor requires GraphController object, array of points and equation name that will be taken from Model class, and, most importantly the CalcView object. Since the CalcView creates the so-called Main View(JFrame) with calculator pad(Main Panel), and GraphController is the one that creates a DrawFunction object(which is basically a JPanel with the graph) upon pressing the Graph button, DrawFunction object will set the Main View's panel to the panel that it will create once it's called. Notice that even though the panels are changed, Main Panel is still exists in memory, and it may be accessed at all times: for instance, when the user will press the Back button form Graph View, Main Frame will display the Main Panel.

In addition the constructor of the DrawFunction class creates a panel with buttons designed to also switch to Favorites View(by pressing the Favorites and Add To Favorites buttons in Graph View), this

class requires additional functions that would call GraphController, which would, in its turn, invoke proper functions to switch between the Views.

Before going into the detail of additional functions of DrawFunction class, the picture below provides the main UI of Graph View.

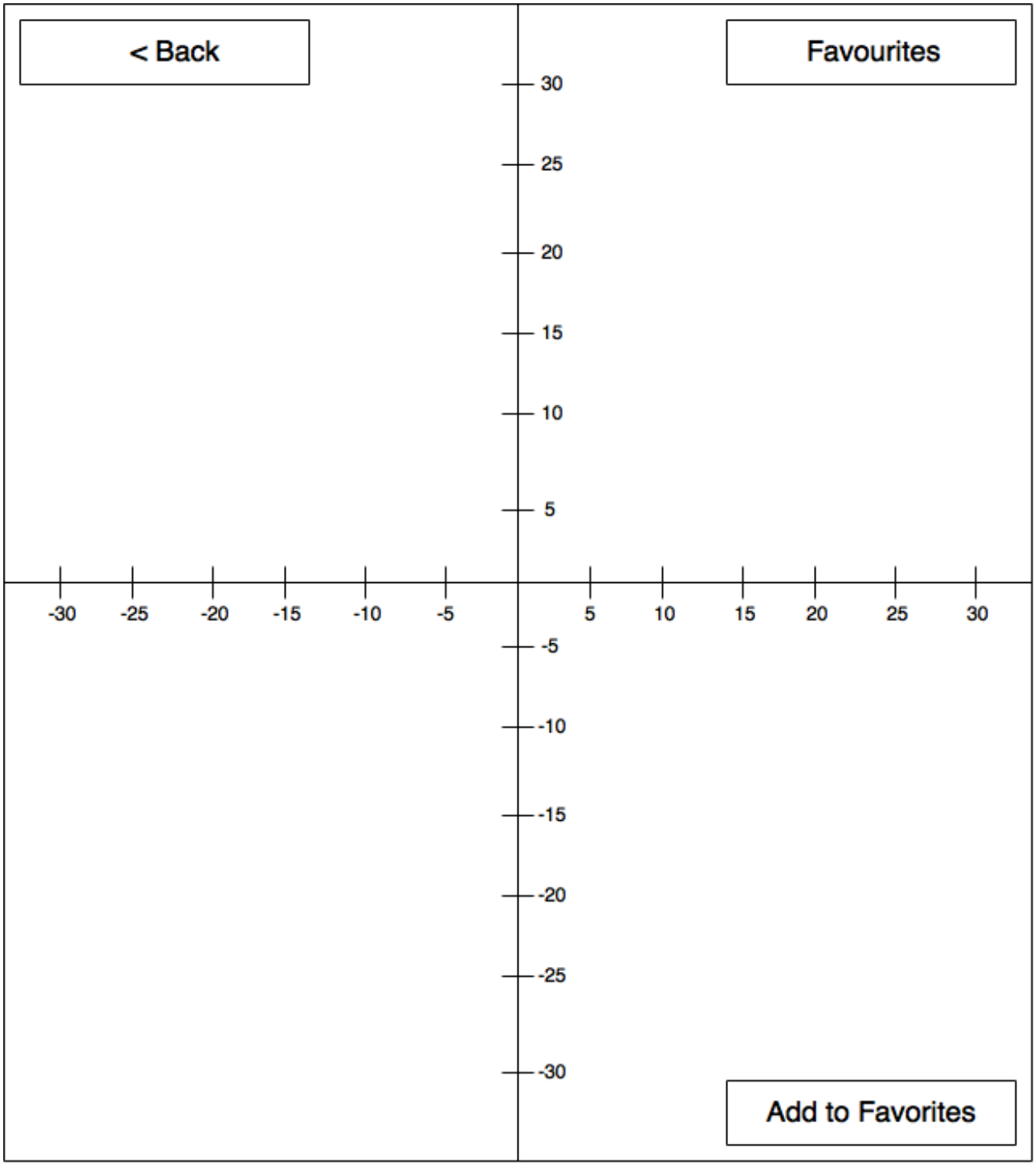


Figure 10 –Layout of the Graph UI

3.2.1 Additional Operations

For the purposes of good programming practice, the DrawFunction object(JPanel) is created and properly configured by invoking several inner-class methods described below.

- **protected Back(): void** - sets the contents of the Main JFrame to the Main View (exits the Graph View).
- **protected paintComponent(Graphics g): void** - gives this JPanel appropriate axis and draws the function.
- **private setYAxis(int w, int h, Graphics g): void** - draws the scale on the Y-Axis of the Graph View.
- **private setXAxis(int w, int h, Graphics g): void** - draws the scale on the X-Axis of the Graph View.
- **public addButtonsToGraph(JPanel graph): void -** properly positions buttons and the text field with the graph's name on this Graph View.
- **public graphPanelButtonColor(): void -** properly configures the look of the buttons and the text field's text .
- **public openFavourites(): void -** calls GraphController to switch to Favorites View.
- **public addToFavourites(): void -** calls GraphController to add this graph to the list of Favorites.
- **public getGraphController(): GraphController -** helper method to access the GraphController of this DrawFunction object.

3.3 Favorites View (CalcFavorites Class)

This class creates a Panel for Favorites - in the same style as DrawFunction class - the constructor of this class, which only requires CalcView object, creates a panel with a "Back" button (which takes the user back to the Graph View).

The favorites are stored in an ArrayList of type FavoriteValue (discussed below).

In case that a graph was added to favorites, this class has an additional functionality of creating and displaying up to 15 little subpanels with graphs' names "Graph" and "Delete" buttons - to either graph or delete the favorite graph. The above mentioned functionalities are defined in this class as separate methods as follows.

3.3.1 Additional Operations

- **protected back(): void -** a routine that takes the user back to Graph View, by resetting the contents of the main JFrame to Graph View.
- **public displayFavorites(DrawFunction theGraph): void -** sets the contents of the main JFrame to Favorites View.
- **public addToFavourites(DrawFunction theGraph, String name, double[] yPoints): void -** adds the current graph(its set of points and name) to the list of the Favorites
- **public showFavoritesList(): void -** a method that generates a little panel with a name of the graph, properly placed , configured and formatted "Delete" and "Graph" buttons for each graph added to Favorites.

- **public delete(int index): void -** a method that allows to remove a specific graph from Favorites. An index passed - index in the List of Favorites of the element to be removed.
- **public graph(int index): void -** a method that allows to graph a certain favorite graph. An index passed - index in the List of Favorites of the element to be graphed.

The Favorites View is shown on the picture below.

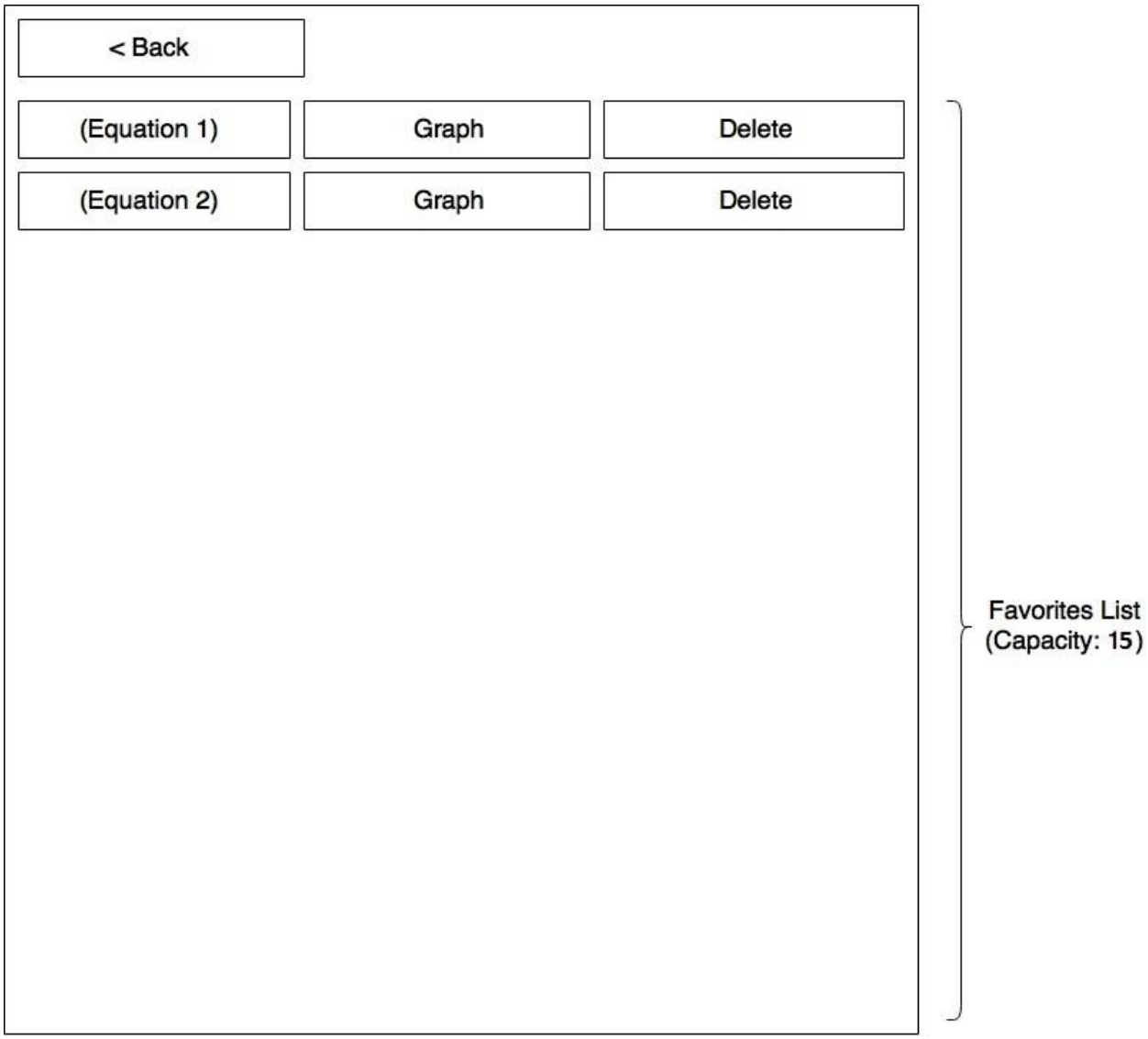


Figure 11 –Layout of the Favorites UI

3.4 FavoriteValue Class

This simple class allows to store together the graph name and its set of points. The objects of this class are passed to CalcFavorites class as Favorites objects that could be stored.

3.4.1 Additional Operations

- **public getEquation(): String** - Gets the function name of the stored favorite value.
- **public getPoints(): double[]** - Gets the y - axis data points of the stored favorite value.

3.5 ButtonAdapter Class

The ButtonAdapter Class, which is also included in the default package of the Project, implements an ActionListener interface. The ButtonAdapter class takes the JButton as an object and attaches an ActionListener to it. This makes the implementation of the View very easy: instead of creating and adding the ActionListener for every single button, ButtonAdapter does it on its own. Also, inside the abstract method of the ActionListener interface, it is possible to pass a function that must be executed upon the action on the button being performed, which is a mouse click in this case. In particular, in the event of clicking a particular button, the CalcController object performs the function designated for this button.

An additional constructor in the ButtonAdapter class, works similarly to the first constructor in the sense that it adds an ActionListener to the JButton, but in addition it also stores an integer index associated with the JButton. This allows for easy removal of favorites from the favorite list, since this stored index corresponds to the index at which the favorite value is stored, in the favorites list.

3.6 Rejected Designs

The View is such an interface that must be very user friendly and therefore has to be designed carefully. There were a lot of different propositions of the design of the program. One of them was having all the utility buttons to the right of the numbers and operators; another one was having operators to the right of the numbers and utility buttons below the numbers. After several simple testing of pressing the buttons and seeing how long it would take the user to reach the common buttons, the Design mentioned in Figure 5 was finally accepted. Generally, calculators would have the Enter(Equals) button at the very bottom of the window. Since the calculator is postfix and requires the user to press Enter after pressing any of number or operator sign buttons, this idea was rejected, and the Enter button was put above all the other utility buttons. Design of the Version 1.0 prevents the user from wasting time reaching the Enter button during the operation of the calculator.

In Version 2.0 of the program, a new functionality of graphing functions with variable X has been introduced. An initial design of this Version has offered having all the graph related buttons at the bottom of the Main View, thus separating the calculating and graphing UI. However, after several tests, it was decided that having such an interface is not user friendly, and, thus, the current design was introduced.

The graphing functionality has also had some changes; in the color of the graph line, in particular. Initially, it was offered to have blue lined graphs, but due to the fact that the Graph View has black

background, the blue colored graph line turned out to be way too dark to see and therefore had to be changed. Currently, the graphs have a cyan color.

As for Favorites, initially, it was offered to have user friendly big buttons on the subpanels with graph name, "Delete" and "Graph" buttons. But because of the fact that even some simple graphs could take a lot of space in writing, the design of the Favorites View was changed in a way that allows to fit visibly longer equations.

4. The Controller Object

The Controller object in the MVC architecture is the mediator between the View and the Model. It controls the response to user input done through the View and the output generated by the Model.

Each button pressed in the View, requests a certain operation to be performed by the Controller. The Controller in turn calls for the appropriate operation in the Model, and then updates the View with the output generated by the Model.

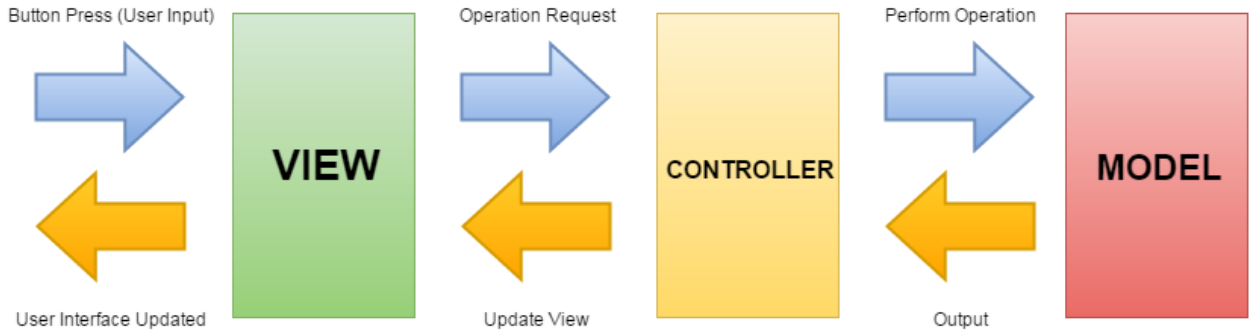


Figure 12 – The work of the Controller in the calculator

4.1 Operations

- **updateView() : Void –** Takes the output of the Model and updates the View with it. This operation is a helper operation that is invoked at the end of each of the following operations. It is never is called on by itself.
- **sum() : Void –** Calls for the **sum()** operation of the Model.
- **subtract() : Void –** Calls for the **subtract()** operation of the Model.
- **multiply() : Void –** Calls for the **multiply()** operation of the Model.
- **divide() : Void –** Calls for the **divide()** operation of the Model.
- **clear() : Void –** Calls for the **clear()** operation of the Model.
- **undo() : Void –** Calls for the **undo()** operation of the Model.
- **enter() : Void –** Calls for the **enter()** operation of the Model.
- **sine() : Void –** Calls for the **sine()** operation of the Model.
- **cosine() : Void –** Calls for the **cosine()** operation of the Model.
- **pi() : Void –** Calls for the **pi()** operation of the Model.
- **x() : Void** – Call for the **x()** operation of the Model.
- **Graph() : Void –** Call the **graph()** operation of Graph Controller.
- **factorial() : Void –** Calls for the **factorial()** operation of the Model.
- **changeSign() : Void –** Calls for the **changeSign()** operation of the Model.
- **numericButton(buttonName : String) : Void –** Calls for the **numericButton(buttonName : String)** operation of the Model.
  buttonName - The name of the button that was pressed in the View (“0” – “9” or “.”).

5. The Graph Controller Object

In a properly implemented MVC architecture, every view should have its own controller. In this case the Graph Controller performs in a similar way to the Controller object above.

The Graph Controller is the mediator between the View, the Graph View and the Favorites View. It connects the three views together, which allows the views to switch from one to another.

Since the three views are never displayed at the same time, the Graph Controller stores an instance of the Favorites View, so that the data stored in the Favorites View persists when switching away to a different View.

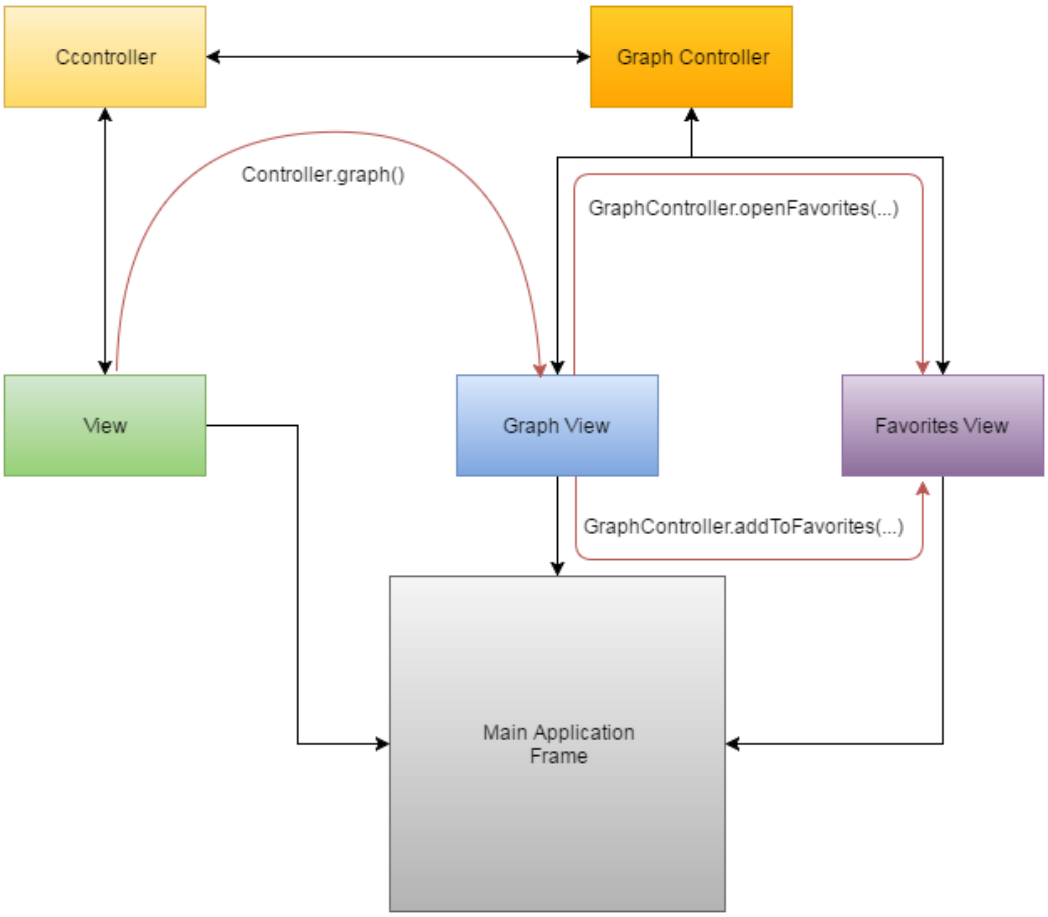


Figure 13 – Abstract representation of how the different views are connected

The reason behind using only one controller for two views (Graph View and Favorites View), is that the Graph View is not permanent. A new Graph View object is created whenever a function is graphed. Since no instance of the Graph View object is saved anywhere except for when it is displayed, an additional separate controller for it is not needed.

5.1 Operations

- **graph() : Void** – creates a new instance of the Graph View, and displays it in the main application frame.
- **openFavorites(GraphView graph) : Void** – displays the Favorites View in the main application frame.
- **addToFavorites(GraphView graph, String functionName, double[] yPoints) : Void –** displays the Favorites View in the main application frame, with the new Favorite Value with parameters functionName and yPoints, added to the favorites list.

6. Maintenance Scenarios

PostFix Calculator UI has been programmed in the programmer-friendly way. Every class related to the design of the user interface has helpful comments. From programming point of view, the variables used in the code are related to their functionality; some of the generalized routines like coloring the buttons are combined in separate methods for reliability. Changing the theme colors will be easy, as it takes only to change a Color field related to the certain functionality or panel only once. Adding extra buttons should not be challenging too due to clear step-by-step setups of all the Panels.

6.1 Adding additional buttons to the calculator

Adding additional operators or constants to this calculator is a simple matter. For any additional mathematical operations (such as additional trigonometric operations, logarithms, exponents, etc.), the following modifications to the program are required:

1. A method that will perform the new mathematical operation or input the new constant has to be added to the Model.
2. A unique symbol to represent the new mathematical operation or constant has to be added in the Model.
3. The **printHistory()** operation will have to be modified to handle printing the new operation or constant.
4. The precedence of the new mathematical operation has to be defined. Constants have the same precedence as any number.
5. A method in the Controller that will mediate between the call in the View and the execution in the Model.
6. The **JButton** in the main View with the appropriate name and **actionListener** added to it with the **buttonAdapter** class that will call the corresponding method in the Controller when pressed.