**Design Document**

Michael Podimov

EECS 2311

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 Graphic User Interface (GUI), as it separates the different components and allows for easy implementation, testing and maintenance of each component individually.

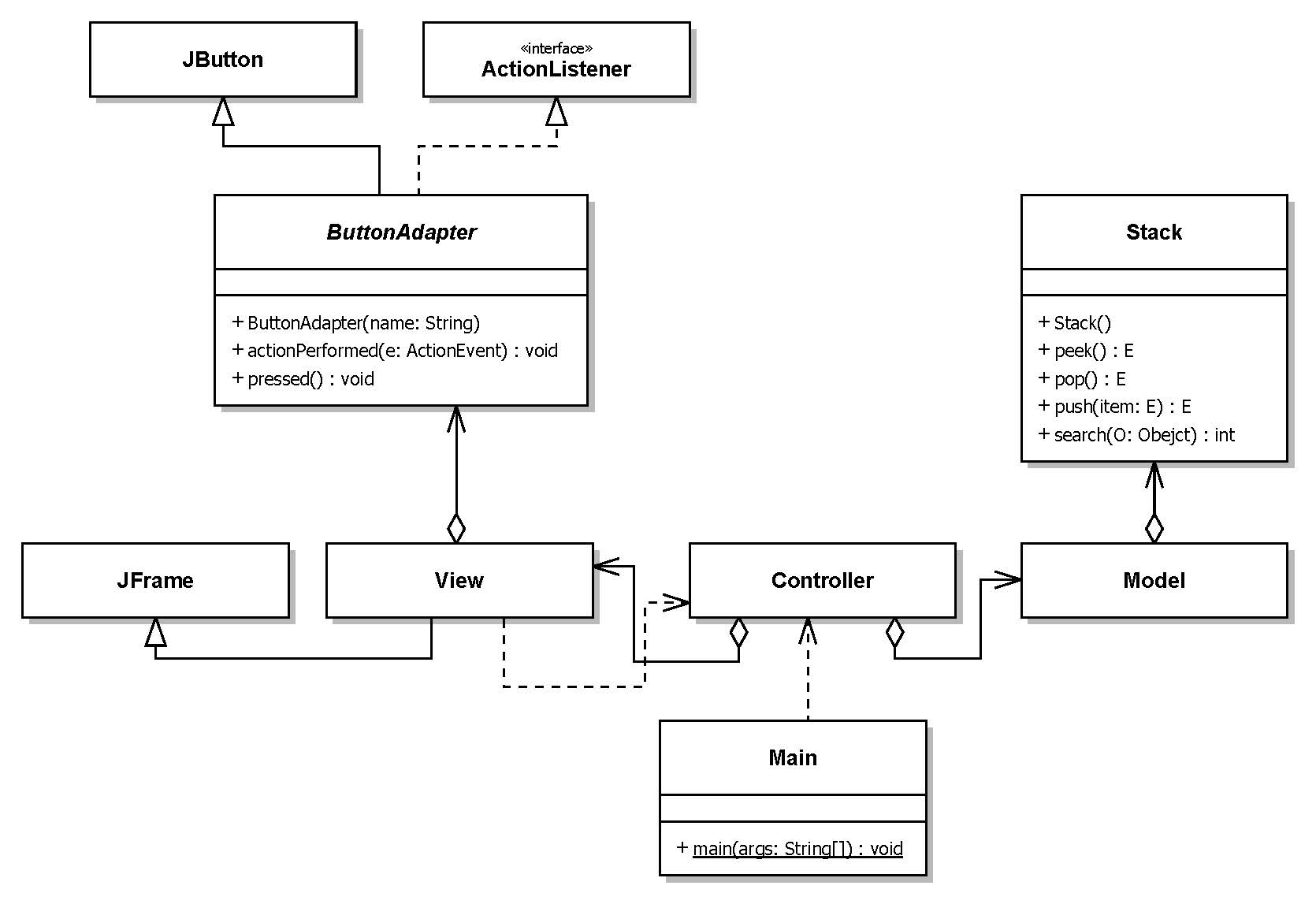


Diagram 1 – General UML diagram of the calculator

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.

1. **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.



Diagram 2 – General operation of the stack in the calculator

The Model uses the following stacks to store relevant data:

* **Calculation\_Stack : Stack<Double>** – holds the numeric data used for mathematical operations, such as input and the results of mathematical operations.
* **Previous\_calculations\_Stack : Stack<Double>** – holds the values used to produce the results of the mathematical operations that are stored in the Calculations 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 3 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 so far, or the result of the last mathematical operation if it occurred.
* An error message will be printed in **input\_value** if a mathematical error has happened (refer to the User Manual for more information).
  1. **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.
* **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**.
* **enter() : void** – Pushes the numeric value of **input\_value** to the top of the **Calculation\_Stack** and the string representation into **History\_Stack**.

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**.
* **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**. If division by 0 occurs, an error message is given.
* **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**.

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.
* **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 : Double) : void** – Checks the result of the last Mathematical operation. If the **result** is a whole number it is stored in **input\_value** in integer representation, otherwise it is stored as a real number representation.
  1. **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 Mathematical operation invokes the **enoughOperandsBinary()** operation or **enoughOperandsUnary()** 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 wants to print a binary operator.

**2.3 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.



Diagram 3 – Abstract representation of how the **printHistory()** operation works

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().**

****

Diagram 4 – Same as diagram 3, but the top most element of the **History\_Stack** was removed.

**2.4 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.

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.

**2.5 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 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.



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 Mathematical operations.

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



If a binary operation such as **sum()** was invoked in this case, 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 cutoff from future operation.

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.



This solution was deemed too complicated for such a simple task and hence was rejected in favor of having **Previous\_calculations\_Stack**.