UNIVERSITÉ Concordia UNIVERSITY

COMP 354 Software Engineering

ASSIGNMENT 2: LEARNING CODING LIKE 1,2,3

v1.9

FALL 2021

GROUP J
Annika Timermanis
George Mavroeidis
Jahrel Stewart
Axel Solano
Phuong Anh Trinh
Jordan Chan Kum Sang

25 October 2021

Dr. Rajagopalan Jayakumar
Farbod Farhour

Revision History

Version	Change	Author(s)	Date
v1.0	Creation of Document, Section Headings	Annika	2021-10-20
v1.1	Filled Introduction	Annika, Anh	2021-10-20
v1.2	Filled Software Constraint Requirements	George	2021-10-23
v1.3	UML diagram	George, Anh, Annika, Jahrel, Axel, Jordan	2021-10-23
v1.4	Filled Section 2 and 4	Annika, Anh	2021-10-23
v1.5	Added UI Design and Explanation	Anh, Jahrel	2021-10-24
v1.6	Added Algorithm and Interpreter classes explanation and design	George	2021-10-24
v1.7	Added MVC Overview and References	Anh	2021-10-25
v1.8	Completed Composition in System Architecture	George	2021-10-25
v1.9	Finished Architecture Strategies	George	2021-10-25

1. Introduction

Purpose and Overview

This document is intended as a description of detailed requirements for [editor's name]. The purpose of developing this software is to implement and augment the code for a text editor, documented by S. Saxena, with the ability of a calculator to allow children to explore algorithms without having knowledge of programming.

As we have previously established our detailed requirements, we will now focus on how our software design should meet the needs of our stakeholders. We are planning to use an object-oriented design architecture, in which our code will be split up into distinct classes. Apart from focusing on our chosen architectural pattern, we want to ensure our code has very low coupling, whilst having very high cohesion. Our goal is to incorporate these two principles into our software design.

The first section of this document will outline the considerations that influence the decisions on architecture and design. The second section will discuss the architectural strategies, including architectural pattern, coupling, and cohesion. The third section will be an in-depth explanation of the architecture. The final section will be the design for the user interface.

Definition, Acronyms, Abbreviations

Definition	**Acronym/ Abbreviation**
Learning Coding like 1, 2, 3	LCL123
User Interface	UI
Graphic User Interface	GUI

2. Design Considerations

Assumptions & Dependencies:

- **Class Dependencies:** By structuring our code to ensure low coupling, we have decided to separate our functionalities into multiple classes, to avoid interdependence. Although the majority of our classes are independent from one another, they all are highly dependent on the parsing of the string entered in English which is handled by our Dictionary Class.
- **Programming Dependencies [5]**
    - Tkinter library for GUI applications.
    - MessageBox: It is a module that is supported by Tkinter and provides template classes and inbuilt functions for alerting users when they write a wrong input.
    - FileDialog: It is a module that is supported by Tkinter and provides the classes and inbuilt functions for creating file or directory selection windows.

Constraints

- **User Input & Vocabulary:** The software's word interpreter will be limited to specific words and sentences that kids have to know in advance. Our software is therefore constrained and must have a very strict set of keywords/word mapping, as well as predefined invalid scenarios, due to the high dependency on a child's input.
- **Admin Powers:** Users will not have authorization to the software's resources and state.

- **Language:** The software is written in Python 3+ and the program interpreter will be analyzing sentences in the English language.
- **Framework:** Mathematical libraries like Numpy will be used for internal calculation procedures and Tkinter for implementing the UI.
- **Database:** There is no internal storage or memorization of instructions outside of the program context. Only a specific class will have the necessary variables saved only for the program's current run.
- **Testing:** Unit testing with expected and predicted values will be used to test all possible cases of the program's input.

Goals & Guidelines

- To create a relevant and well designed software which features both *high cohesion* and *low coupling*.
- To make the UI very *appealing* and *child-friendly*, including fun colours, easy navigation, and simplified input and returned result.
- To offer a fun opportunity for children to learn the basics of coding through arithmetics and basic operations

System Environment

- This software will be available on any machine that has a Python Virtual Machine.

3. Architectural Strategies

To achieve modularity, the architecture must be built easily upon existing components. In Object-Oriented Programming, Composition and Inheritance are vital in creating clean modules that interact with one another in a cohesive manner. In addition, reuse of existing components will be frequent.

An important but necessary class is the Interpreter. Like explained in the Component Level Architecture, the Interpreter will be responsible for breaking down the sentence provided by the user and match it with the correct operation. Having a class responsible for this process ensures modularity and facilitates the addition of new features for sentence processing. In the same manner, all other algorithms are built this way, so if a new calculation feature is required, it can be added without having to restructure the current code base.

The reason why Python is the programming language of choice is because of its dynamic flow and high-level data types. This will allow for faster development and simplified code. The development team is also more comfortable with this language as well as the easy-to-use frameworks, like Tkinter, where it requires little to no experience in GUI programming to create a graphical text editor.

4. System Architecture

Overview

The overall architecture of the text editor is the Model - View - Controller:

I. Model: to encapsulate the data that **View** displays to users and **Controller** shows as result for user input.

II. View: to present data to users. It uses user input to manipulate **Model** that then updates **View** and shows results to users.

III. Controller: to take the requests from users, and then execute those. After executing, those datas are stored in **Model.**

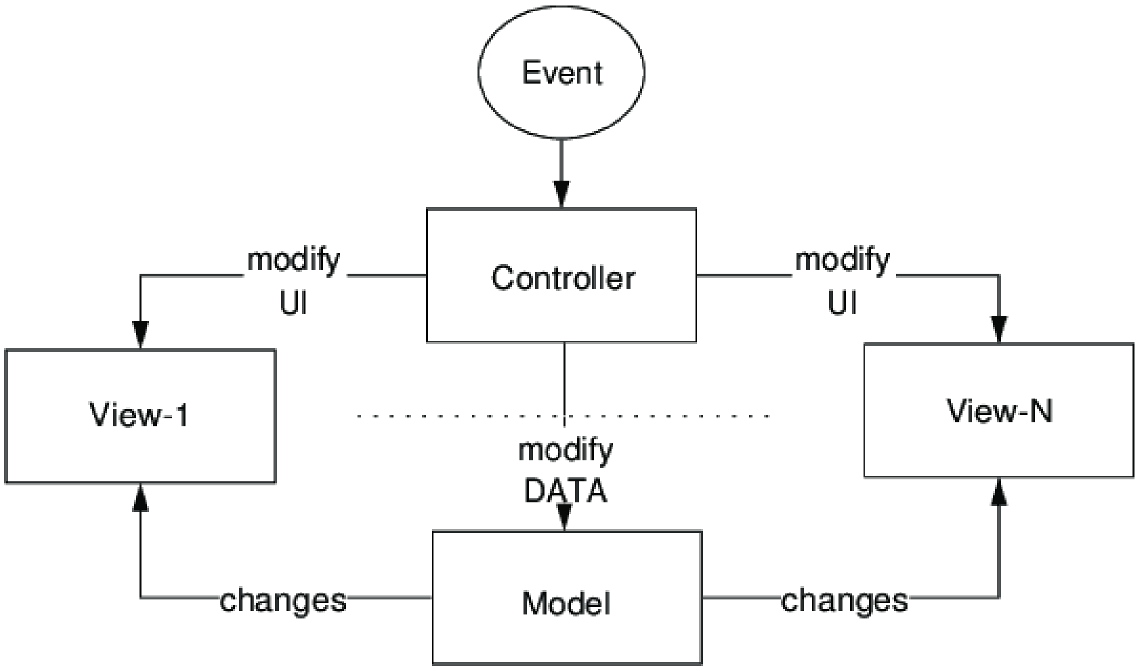


Figure 1: How MVC works.

Source: Adapted from [4]

The overall architecture of the calculation functionality can be represented by the following 6 classes:

I. Interpreter Class

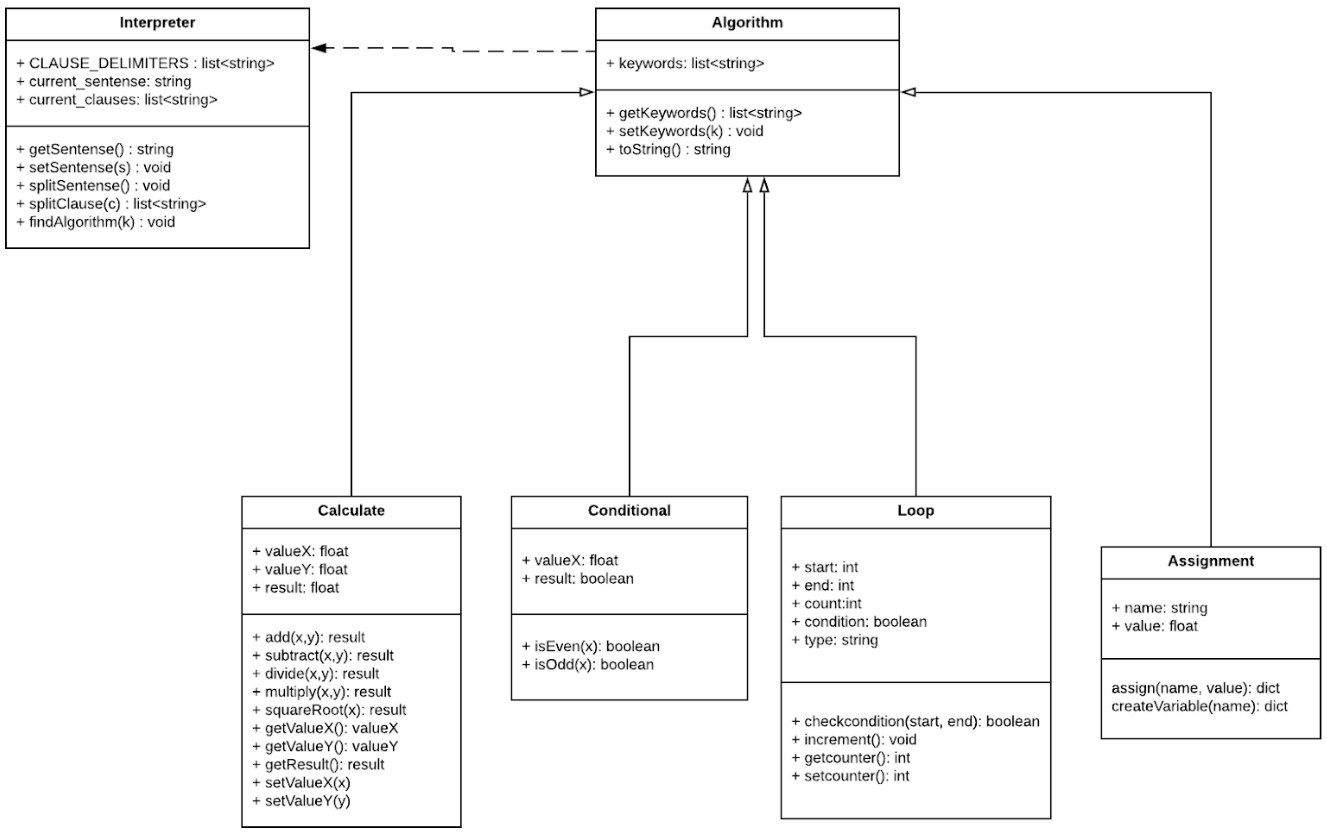
II. Algorithm Class

III. Calculate Class

IV. Loop Class

V. Conditional Class

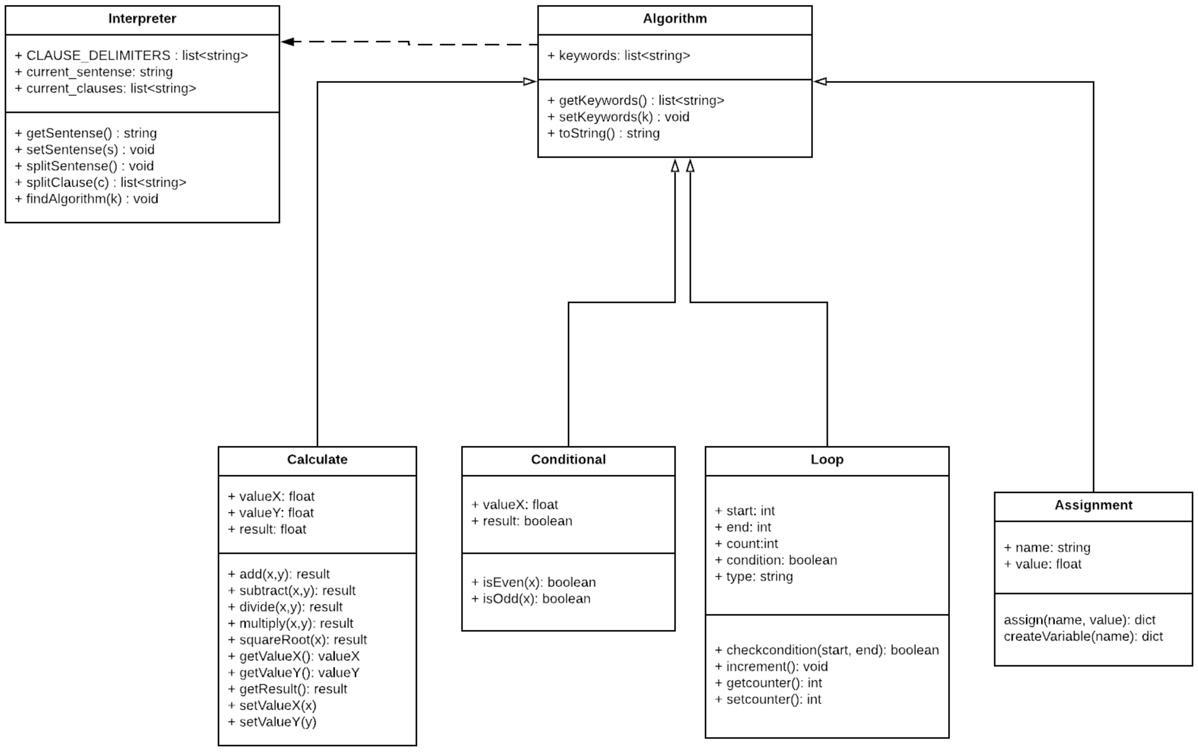
VI. Assignment Class



The definition, responsibilities, and uses/interactions will be described for each of our components below. As for the composition and interface, they don't apply to all 6 individual components, rather are more general aspects to our software.

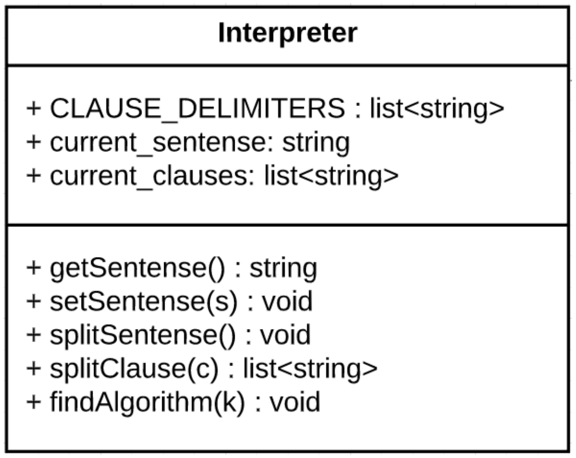
Interface: We will have a single UI which can be described in detail in section 5. UI Design. Apart from that, each class does not have an individual interface, they are the behind-the-scenes components of our software.

Composition: Our classes are structured to be independent from one another to implement our principle of low coupling. By having each step separated as much as possible, we ensure unrelated steps are independent, so the risk of an error will minimize the system's failure. High cohesion is achieved by structuring the code in such a way that attributes and functions are connected in a unified and organized manner. Each step of the code is clearly defined from beginning to end and ensures the purpose of the class that is part of.



Component Level Architecture

I. Interpreter Class


Interpreter
+ CLAUSE_DELIMITERS : list<string>
+ current_sentense: string
+ current_clauses: list<string>
+ getSentense() : string
+ setSentense(s) : void
+ splitSentense() : void
+ splitClause(c) : list<string>
+ findAlgorithm(k) : void

Definition: The Interpreter class deconstructs sentences and analyzes their keywords to match it with the correct algorithm and operation. It will also check for the validity of the sentence and its clauses.

Responsibilities:

getSentense(): return the current input sentence in the form of a string.

setSentense(): sets the current sentence that will be used for deconstruction.

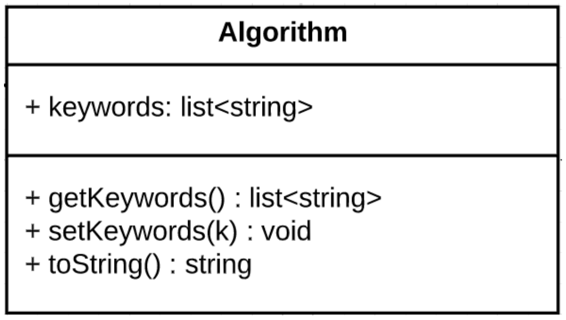
splitSentense(): splits sentence string into clauses, using the clause delimiters and return list.

splitClause(c): splits clause c into words using the space delimiter and return list of words.

findAlgorithm(k): find appropriate algorithm using list of keywords k and perform necessary operation.

Uses/Interaction: When a new sentence is entered in the message box, the interpreter will perform the necessary steps to break down the sentence and match it with the correct algorithm to display the proper results.

II. Algorithm Class



Definition: The Algorithm class is the base class for all operations that will be performed once the interpreter analyzes the sentence and matches it with an algorithm. It contains the list of keywords that different algorithms will have labelled so the operation can be performed.

Responsibilities:

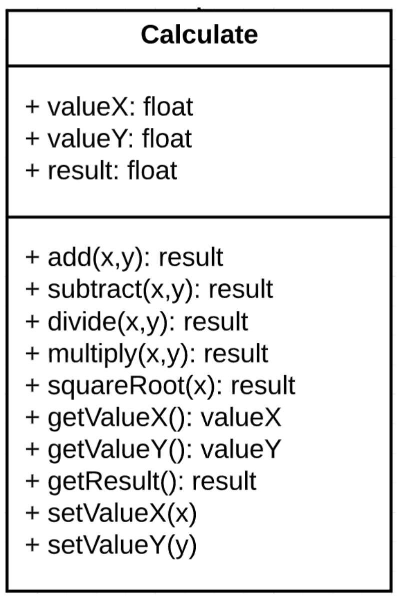
getKeywords(): returns the list of the algorithms specific keywords.

setKeywords(k): sets the list of keywords for the specific algorithm class.

toString(): returns the output/result of the performed algorithm in the form of a sentence.

Uses/Interaction: Called upon when a user wants to compute an algorithm, when the interpreter detects an algorithm.

III. Calculate Class



Definition: A class to handle all mathematical calculations, based on a user input of two values, this class will be utilized to perform a said operation and return the correct result to the user.

Responsibilities:

add(x,y): method that adds value x and value y and returns the result.

subtract(x, y): method that subtracts value y from value x and returns the result.

divide(x, y): method that divides value y from value x and returns the result.

multiply(x, y): method that multiplies value x and value y and returns the result.

squareRoot(x): method that takes the square root of value x and returns the result.

getValueX(): method that returns value x.

getValueY(): method that returns value y.

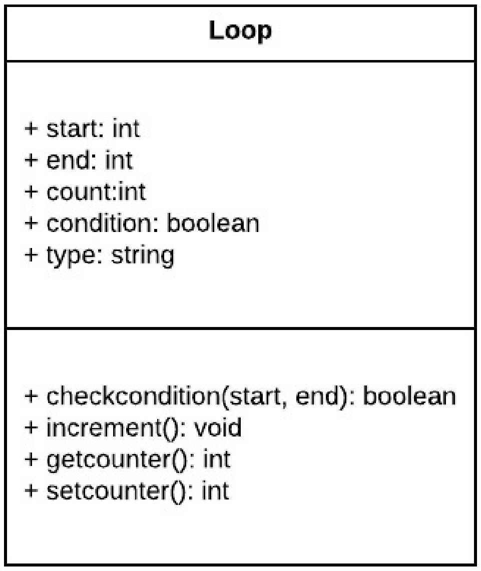
getResult(): method that returns the calculation result.

setValueX(): method that sets a value to x.

setValueY(): method that sets a value to y.

Uses/Interaction: Called upon when a user wants to compute a calculation. This class offers 5 calculations: addition, subtraction, multiplication, division, and square root.

IV. Loop Class



Definition: This Loop class is responsible for all loop-based operations. The loop class can operate one mathematical operation on multiple data on a list. This class is built with two types of loops: for loop and while loop, and the user can specify a stop condition the loop runs. The user will have to input the start number and end number which will be used as conditions for both loops. For example:

For 1 to 10, add 2 to list_num

The following instruction means that a value of 2 will be added to the first ten elements in the list which is list_num.

Responsibilities:

checkcondition(start, end): return true when condition is correct.

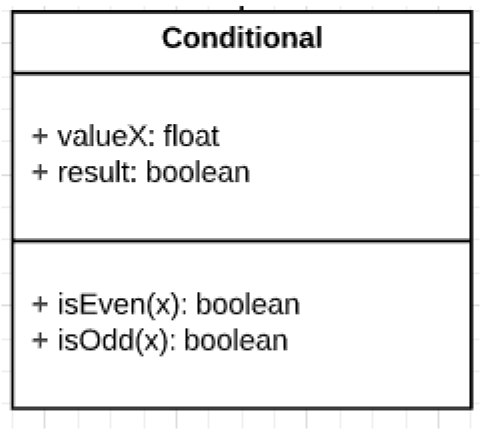
increment(): method to increment counter.

getcounter(); method to return count value.

setcounter(): method to set count value.

Uses/Interaction: This class is used when the user wants to process the same calculation on multiple values in a list. It offers two types of loops, and the user can specify its own condition.

V. Conditional Class



Definition: The conditional class is used to perform conditions catered towards determining whether or not a number is even or odd. Once the correct syntax has been entered by the user, a check, from either the isEven or isOdd function, would be performed to assess the validity of the condition. If the result is true, the code following the *'then'* keyword is executed. If false, it will skip the code following the *'then'* keyword, executing only what comes after the entire condition-based construct.

Responsibilities:

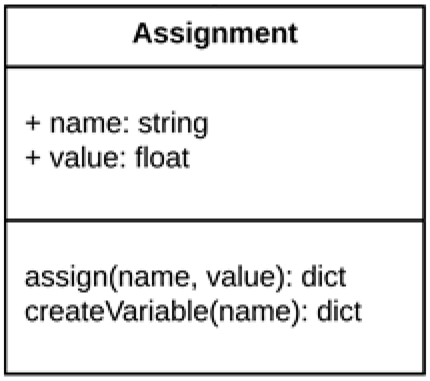
isEven(x): a method that returns True only when x mod 2 = 0

isOdd(x): a method that returns True only when x mod 2 = 1

Uses/Interaction:

The purpose of this class is to execute a specific code block only if the condition to determine whether or not a number is even or odd has been satisfied. For proper usage, the user should have their condition, written as *[integer] is even* or *[integer] is odd*, placed between *'if'* and *'then'.* Such a construct can be used within loops to achieve even or odd patterns that would have been more difficult or impossible, given the limitations of the programming language, to implement otherwise.

VI. Assignment Class



Definition:

This assignment class is small but essential to the general algorithm. It will assign values to variables and will also create uninitialized variables. The dictionary object that holds the variables (keys) and values is not part of the class as it will be created during the execution of the software in the driver. This class is simply to perform the action of variable assignment and creation.

Responsibilities:

assign(name, value): returns a key value pair where the key is the variable name.

createValue(name): returns a key value pair where the value is null, to be defined later.

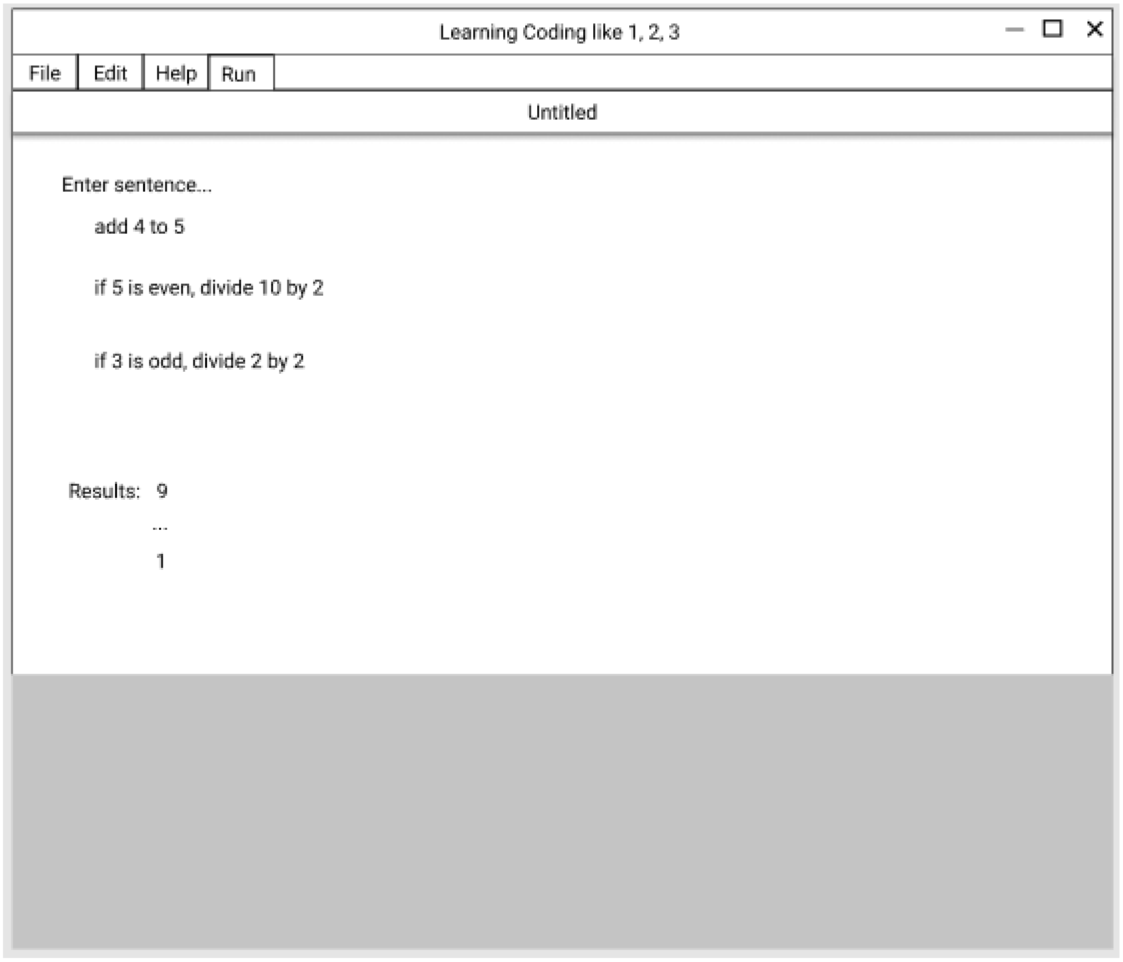
Uses/Interaction:

This class is used when the user wants to store a value in a variable. They will have to input keywords like “is”, or “equals”, or “assign”. E.g. “assign 2 to X”. This class will also be called after the *Calculate* and *Loop* operations are over to store the results in a variable.

5. UI Design

Since the software is for children, we want to prioritize the simplest and most child-friendly design.

Although the text editor has not gone through many evolutions of design due to its simplistic nature, all of its core features are there and designed to be used in an efficient manner. Because children are expected to use this software, a design that does not feel clustered is set with high priority to ensure excellent user experience. We are able to achieve this effect by using white space in a way to draw the eyes of the user to what's most important: the sentences. Furthermore, a way to highlight certain text, various sizes, opacities and boldness of text is used in order to subtly stylize the experience without using too many distracting shapes. Speaking of distracting shapes, despite the fact that the newer design has more shapes, they are laid out in such a way to guide the user's eyes in a way that feels more natural. The figure below shows the initial design, the phase before refining it to what it currently looks like.



Enter sentence...

add 4 to 5

if 5 is even, divide 10 by 2

if 3 is odd, divide 2 by 2

Results: 9
...
1

Figure 2: Beginning Stages of Design Evolution

Final stage of design evolution is the following. About the **Information Design**, the text editor's name, and control buttons (minimize, maximize/restore down, close) will be on top. The second bar is for **File**, **Edit**, and **Help** button. Every feature inside **File** (which are **New**, **Open**, **Save**, **Quit**), **Edit** (which are **Undo**, **Redo**, **Copy**, **Paste**, **Cut**), and **Help** (which are **Dictionary**, **Tips**) will be hidden to the children unless they click those buttons. The third bar is the name of the file. Under it, there will be space for children to enter the sentences whose keywords are defined in our dictionary. For example: "Add 4 to 5". After finishing the sentence, the **Run** button has to be clicked to show the results. The **Results** are presented on the right of the text editor.

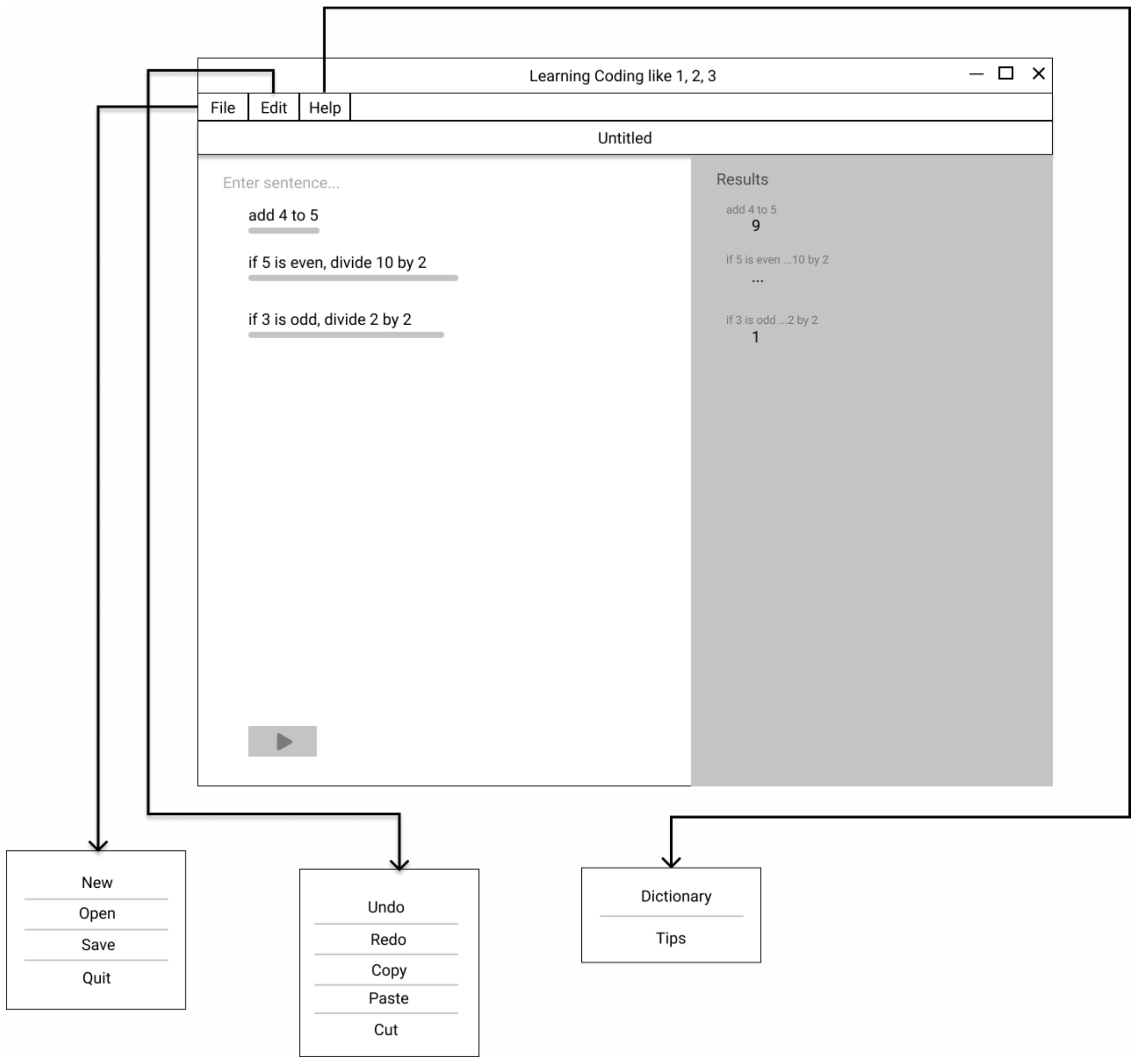


Figure 3: UI Design of the LCL123

The **User Flow Diagram** below illustrates the experience that users will undergo when using the text editor. The arrows connect different system states, forming a path of logic through which the user can make sense of. Moreover, this text editor has been designed in such a way to guarantee that each user will have the same experience, replicating the user flow diagram below.

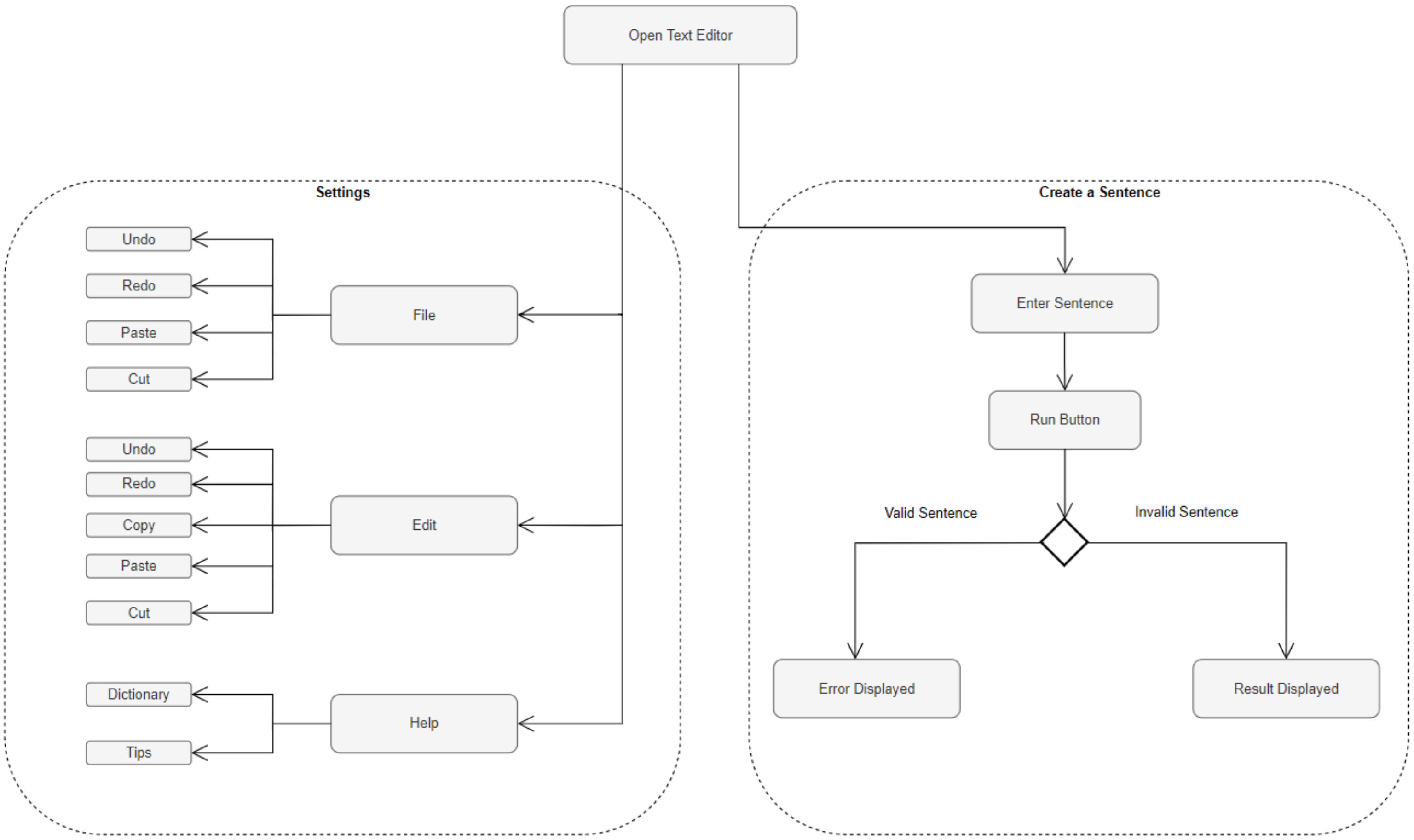


Figure 4: User Flow Diagram for using LCL123

Contribution Of Team Members

Student Name - Student ID	**Sections Contributed**
Phuong Anh Trinh - 40069870	➢ UI Design ➢ Figure 2 ➢ Figure 3 ➢ Figure 4 ➢ UML Diagram ➢ Design Considerations
George Mavroeidis - 40065356	➢ I. Interpreter Class ➢ II. Algorithm Class ➢ UML Diagram ➢ Architecture Strategies ➢ System Architecture Composition ➢ Design Considerations
Annika Timermanis - 40131128	➢ III. Calculate Class ➢ UML Diagram ➢ Design Considerations ➢ Architecture Strategies
Jordan Chan Kum Sang - 40125997	➢ IV. Loop Class ➢ UML Diagram
Jahrel Stewart - 40115728	➢ V. UI Design ➢ Figure 2 ➢ Figure 3 ➢ Figure 4 ➢ Conditional Class ➢ UML Diagram
Axel Solano - 40046154	➢ VI. Assignment Class ➢ UML Diagram

References

[1] R. Pressman and B. Maxim, Software Engineering: A Practitioner's Approach, 9th Edition, McGraw Hill, 2020, ISBN 13: 9781259872976.

[2] Diagrams created in Lucidchart, www.lucidchart.com

[3] UI Design created in Figma, https://www.figma.com/

[4] Figure 2.1 of J. Wielemaker, Logic Programming for Knowledge-Intensive Interactive Applications. [Online]. Available: https://www.researchgate.net/figure/Model-View-Controller-MVC-design-pattern-Controllers-modify-UI-aspects-of-a-view_fig3_254852917

[5] S. Saxena, Create a Text Editor in Python. [Online]. Available: https://www.codespeedy.com/create-a-text-editor-in-python/