COMP 354 Software Engineering

LCL123: **LEARNING CODING LIKE 1,2,3**

ASSIGNMENT 3: SOFTWARE REQUIREMENTS SPECIFICATIONS

FALL 2021

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

November 8th 2021

Dr. Rajagopalan Jayakumar
Farbod Farhour

Revision History

Version	Change	Author(s)	Date
v1.0	Creation of Document, Section Headings	Anh	2021-10-30
v1.1	Filled Introduction	Anh	2021-10-30
v1.2	Organized group meeting time, and prepared list to review for technical review	Anh	2021-10-30
v1.3	Technical Review	All members during group discussion	2021-10-2
v1.4	Component Level Tests	All members, each responsible a individual component	2021-10-2
v1.5	Technical Review Summary	Annika	2021-10-5
v1.6	Metrics	Annika, Axel	2021-10-5
v1.7	Added References	Annika	2021-10-5
v1.8	Integration Tests	Jahrel, Annika, Anh, Axel	2021-10-5

Contribution Of Team

Student Name - Student ID	**Pages Contributed**	**Component Level Test**
Phuong Anh Trinh - 40069870	1,2,3,4,5,13,17	Calculate (square root)
George Mavroeidis - 40065356	5,6,7,8,9,10,12,13	Interpreter & Algorithm Class
Axel Solano - 40046154	16,17,18,19,20,21	Assignment Class
Annika Timermanis - 40131128	4,5,6,7,8,9,17,18,19,20,21	Calculate & Algorithm Class
Jordan Chan Kum Sang - 40125997	9,14,15,16	Loop Class
Jahrel Stewart - 40115728	13,14,15,17,18,19,20	Conditional Class

Introduction

Purpose and Overview

The purpose of this document is to report on the outcome of the technical review and outline the component and integration tests we will be incorporating into the development process for the project.

Technical Review

I. Review Agenda

All the tasks are completed by the date set in the agenda.

Task / Responsibility	Responsible Members	Date
Read over and understand the tasks given in the assignment	Annika, Anh, Axel, George, Jahrel, Jordan	10/31/2021
Vote for the review leader, conducting informal review	Annika, Anh, Axel, George, Jahrel, Jordan	10/31/2021
Each member starts 6-step reviewing process, conducting formal review	Annika, Anh, Axel, George, Jahrel, Jordan	11/3/2021
Conduct review meeting	Annika, Anh, Axel, George, Jahrel, Jordan	11/4/2021

II. Review Recording and Record Keeping

Checklist for products that is likely to be reviewed:

- Design Consideration & Architecture Dependencies
- Design Constraints
- UI Design
- UML Designs

Review List

Product	Issue	Summary
Design consideration & Architecture dependencies	The Interpreter Class was responsible for interpreting the input entered by the user and parsing the clause(s), the Algorithm Class was responsible for calling the corresponding class and functions, but 2 issues arised: 1) No class is handling the verification of the numerical values 2) No class is handling the verification of a variable input	Splitting up the responsibility of validating the string input: After looking at our original design, we think it would be more reasonable to have the Algorithm class check the values (whether numerical or variable) in the input string.
Design constraints	1) If the values entered are not numerical, or if a variable entered does not exist, then a class needs to handle this.	We must consider in our modified design that the Algorithm Class must handle this: If the string value passed is NAN, then we should return an ERROR message to the user. The user must enter a valid numeric string: '5' would be accepted, 'five' would not be accepted. If the variable entered does not exist, we should return an ERROR message to the user.
UI design	No Issues Found	We decided to still use the Tkinter library to implement the text editor, and use the grid() method that is imported from Tkinter to have the grey features that are described in figure 3 of Assignment 2.
UML Design	1) There should not be inheritance. 2) After revising our design considerations, the UML diagram is missing a few	We must update our previous UML diagram with our newly added functions, and get rid of the inheritance between the Algorithm Class and Calculate,

	new functions that are needed as a result of issues found and stated above.	Loop, Conditional and Assignment classes.

III. Review Outcome

As a team, we have accepted and acknowledged both some minor and major errors in both the logic and implementation of our design so far. We have summarized our changes in design below.

IV. Formal Technical Review Summary Report

In summary, after revising our initial requirements, and our design thus far, we have noted a few discrepancies that we believe are important to take note of and fix. In our review, we found:

- As the **Interpreter Class** is responsible for parsing the input string and splitting it into clause(s) (which are comma separated), we needed a specific way to handle extracting the value input (if the user inputs ‘5’ or ‘five’) OR variable input (if the user inputs ‘X’):
  - **Value Input:** Checking to see if the string entered could be converted to float, in order to continue execution.
    - If valid, continue execution. (Example: ‘5’)
    - If invalid, terminate the program, prompt for re-entry. (Example: ‘five’)
  - **Variable Input:** Checking to see if the variable used in the input exists or not.
    - If the variable exists, continue execution.
    - If the variable does not exist, terminate the program, prompt for re-entry, UNLESS they are properly declaring a variable in the form: “**Let** X **be** __”.
- We have decided that this should all be handled by the **Algorithm Class**. We have added *five* functions, *three* new class variables and *six* new accessor and mutator methods to the **Algorithm Class** which are highlighted below in red.

Algorithm
+ current_words: list<string> + validValue1: float + validValue2: float + temp_key: string
+ getWords() : list<string> + setWords(k) : void + getValidValue1(): float + getValidValue2(): float + getValid_key(): string + setValidValue1(float): + setValidValue2(float): + setValid_key(string): + isVariable(string): bool + doesVariableExist(key): bool + getVariableValue(key): void + convertStringToFloat(string): void + toString() : string + findAlgorithm() : void

- These new functions will work together to verify if the string that has been previously parsed by the Interpreter Class holds a value or variable, and if input is valid, we will find (**findAlgorithm()**) the class (Calculate, Loop, Conditional) needed to perform and execute the operation.
- **isVariable(string)**: This function will look at the value in the input string and will return true if it is a variable (cannot be converted to float), and return false if it is not a variable (can be converted to a float).
- **doesVariableExist(key)**: This function will be called if the **isVariable()** function returns true, and will check if the variable exists by passing the key to our Dictionary of pre-existing variables (a global variable). Will return true if the variable exists, false otherwise.
- **getVariableValue(key)**: This function will be called if the **doesVariableExist(key)** returns true. Will access the value at the passed key, and the value will be set to class variable validValue1 or validValue2. The key will be set to class variable **valid_key**.

- **convertStringToFloat(string)**: This function will be called if **isVariable()** returns false, and will be used to convert the string input (“5”) into a float value (5), the new float value will be set to class variable **validValue1** or **validValue2**.
- Due to keywords list in the **Algorithm Class** that were passed by the **Interpreter Class**, if the **findAlgorithm()** is called, it will know exactly what class to call: Calculate, Loop, Conditional or Assignment, and will pass the preprocessed values (**validValue1** and/or **validValue2)** that have now been validated.
- Another point to mention that was noticed during our review was that the **Calculate Class** requires an extra helper function called **findOperation(operation,x,y=0)**. This helper function will be called directly by the **findAlgorithm()** method of the **Algorithm class**, and internally handle the decision of calling the particular method (add(), subtract(), ect…) based on the passed operation.

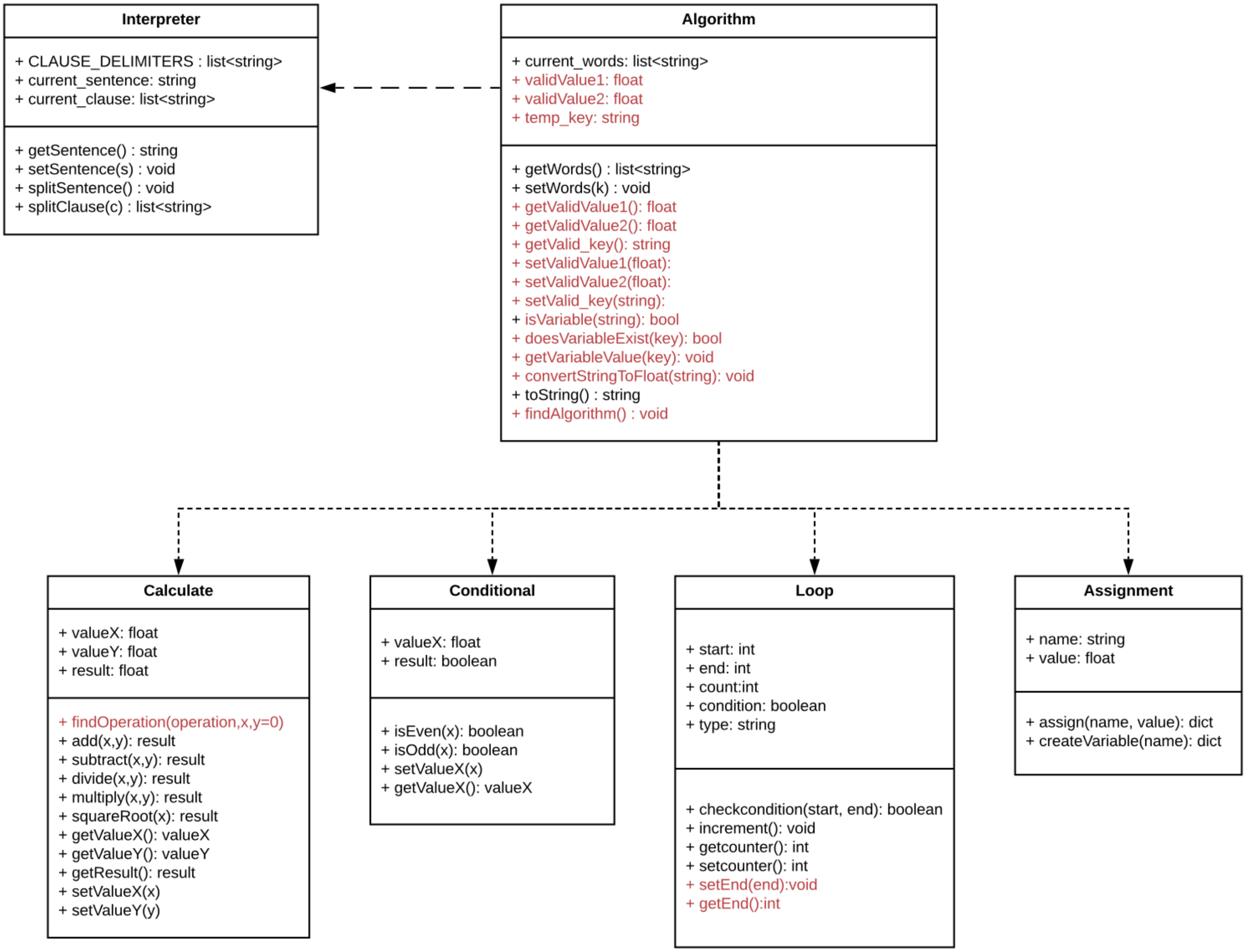
Calculate
+ valueX: float + valueY: float + result: float
+ findOperation(operation,x,y=0) + add(x,y): result + subtract(x,y): result + divide(x,y): result + multiply(x,y): result + squareRoot(x): result + getValueX(): valueX + getValueY(): valueY + getResult(): result + setValueX(x) + setValueY(y)

- It is also important to note that the user input could take on multiple individual clauses which are comma separated. These clauses are **independent** from one another. For example: “Add 5 and 2, multiply 5 by 2”, here we have 2 separate clauses that do not depend on one another, and therefore the user can enter multiple clauses in a single line (allowing our software to be extendable).

- Another modification was made to the Loop class after our review. In the Loop class, two additional methods were added i.e.: setEnd(end) and getEnd(). After additional consideration, we have realised that these two mentioned methods will facilitate the execution of a loop instruction. The setEnd(end) method will help to set the number of loop passes and it will also validate the end value if it is invalid for example: -1 (Negative value). The getEnd() method will help to access the end value in the checkcondition() method to compare stop condition after each loop pass.

Loop
+ start: int + end: int + count:int + condition: boolean + type: string
+ checkcondition(start, end): boolean + increment(): void + getcounter(): int + setcounter(): int + setEnd(end):void + getEnd():int

- Our final point after reviewing our design, is that in our UML diagram from Assignment 2, we had the Calculate Class, Loop class, Conditional Class and Assignment class all inheriting from our Algorithm Class. After our technical review, we have realized that there is no need for inheritance, and therefore have replaced the solid arrows with dotted arrows to express communication, rather than child-parent class relationship.



V. Review Metrics

- **Preparation effort:** The effort in hours that our team required to review our software design prior to our actual review meeting was ½ **hour**.
- **Assessment effort**: The effort in hours that we expended during the actual review was **2 hours.**
- **Rework effort:** The effort in hours that our team dedicated to correcting the errors uncovered during the review was **3 hours**.
- **Work product size**: the size of the work product that has been reviewed involves our original, **single UML diagram**, and revising around **20 pages** including Assignment 1's requirements and much of Assignment 2's design considerations.

- ❖ **Minor errors found**: The number of errors found that can be categorized as minor is **4** (adding findOperation(operation,x,y=0) of the Calculate Class, adding setEnd(end) and getEnd() to the Loop Class, and removing the inheritance scheme from our UML.)
- ❖ **Major errors found**: The number of errors found that can be categorized as major is **8**, which involved the reworking of the Algorithm Class including the **3** new class variables and **5** new methods.
- ❖ **Total errors found** = 4 minor errors + 8 major errors= **12 errors**
- ❖ **Error density** = 12 total errors / 20 pages (work product size)= **60%**

Component Level Tests

I. Interpreter Class

Test #1: Module Interface
Test Purpose: Tested to ensure that information properly flows into and out of the program unit
Test #1.1: To demonstrate how a sentence is stored:

- **The input to be applied:**
  dict = {‘X’: 10}
  setSentense(s): s = “If X is odd, add 5 to number”
  getSentense()
- **The correct output**: “If X is odd, add 5 to number”

Test #1.2: To demonstrate how a sentence is split into clauses and stored:

- **The input to be applied:**
  dict = {‘X’: 10}
  current_sentense = “Add 5 to X”
  splitSentense()
- **The correct output (void):**
  current_clauses = [‘If number is odd’, ‘add 5 to X’]

Test #1.3: To demonstrate how a clause is split into words:

- **The input to be applied:**
  c = “add 5 to number”
  splitClause(c)
- **The correct output:** [‘add,’ ‘5,’ ‘to’, ‘number’]

Test #2 Error Handling with sentence deconstruction
Test Purpose: To test when a sentence is made up of less than 2 words.
Test #2.1

- **The input to be applied:**
  setSentense(s): s = “Sentence”
  splitSentense()
- **The correct output (void):** Sentence cannot be split and is invalid. User is prompted to enter another valid sentence.

II. Algorithm Class

Test #1: Variable Input
Test Purpose: To demonstrate how Algorithm Class will handle variable input.
Test #1.1: To demonstrate how we will determine if it is a variable or numerical value.

- **The input to be applied**:
  dict = {‘X’: 10}
  current_words = [‘add’, ‘X’, ‘to’, ‘2’]
  isVariable(current_words[1])
- **The correct output:** true

Test #1.2: To demonstrate how we will determine if a variable exists.

- **The input to be applied**:
  dict = {‘X’: 10}
  current_words = [‘add’, ‘X’, ‘to’, ‘2’]
  doesVariableExist(current_words[1])
- **The correct output:** true

Test #1.3: To demonstrate how we will get a pre-existing variable’s value.

- **The input to be applied**:
  dict = {‘X’: 10}
  current_words = [‘add’, ‘X’, ‘to’, ‘2’]
  getVariableValue(current_words[1])
- **The correct output:** 10

Test #2: Value Input
Test Purpose: To demonstrate how Algorithm Class will handle value input.
Test #2.1: To demonstrate how we will determine if it is a variable or numerical value.

- **The input to be applied**:
  current_words = [‘add’, ‘5’, ‘and’, ‘2’]
  isVariable(current_words[1])
- **The correct output:** false

Test #2.2: To demonstrate how we will convert a numeric string value to a float.

- **The input to be applied**:
  current_words = [‘add’, ‘5’, ‘and’, ‘2’]
  convertStringToFloat(current_words[1])
- **The correct output:** 5

Test #3: Algorithm Input
Test Purpose: To demonstrate how Algorithm Class will match the sentence with the correct algorithm.
Test #3.1: To demonstrate how a sentence is matched with the Calculate Class

- **The input to be applied**:
  dict = {‘X’: 10}
  current_words = [‘add’, ‘5’, ‘to’, ‘X’]
  findAlgorithm()
- **The correct output (void):** “5 has been added to X. Now X equals 15.”

Test #3.2: To demonstrate how a sentence is matched with the Conditional + Calculate

- **The input to be applied**:
  dict = {‘X’: 10}
  current_words = [‘if’, ‘X’, ‘is’, ‘even’, ‘add’, ‘5’, ‘to’, ‘X’]
  findAlgorithm()
- **The correct output (void):** “X is even, so 5 has been added to X. Now X equals 15.”

Test #4: Error Handling with matching algorithms
Test Purpose: To test when an algorithm cannot be found:
Test #4.1

- **The input to be applied:**
  dict = {‘X’: 10}
  current_words = [‘this’, ‘algorithm’, is’, ‘wrong’]
  findAlgorithm()
- **The correct output (void):** Algorithm not found. User is prompted to enter another valid sentence

III. Calculate Class

Test #1: Independent Paths Through Control Structures
Test Purpose: Exercised to ensure all statements/methods are executed at least once.
Test #1.1 Add:

- **The input to be applied**:
  setValueX(5)
  setValueY(2)
  add(valueX, valueY)
- **The correct output:** 7

Test #1.2 Subtract:

- **The input to be applied:**
  setValueX(5)
  setValueY(2)
  subtract(valueX, valueY)
- **The correct output:** 3

Test #1.3 Multiply:

- **The input to be applied:**
  setValueX(5)
  setValueY(2)
  multiply(valueX, valueY)
- **The correct output:** 10

Test #1.4 Divide:

- **The input to be applied:**
  setValueX(5)
  setValueY(2)
  divide(valueX, valueY)

- **The correct output:** 2.5

Test #1.5 Square Root:

- **The input to be applied:**
  setValueX(5)
  squareRoot(valueX)
- **The correct output:** 2.236067977

Test #2: Error-Handling & Boundary Conditions
Test Purpose: To verify how each test path accommodates and handles errors.
Test #2.1 Division by 0: How the information properly flows into and out of the program unit.

- **The input to be applied**:
  setValueX(5)
  setValueY(0)
  divide(valueX, valueY)
- **The correct output:** “Error! Division by 0.”

Test #2.2 Square root of a negative number:

- **The input to be applied**:
  setValueX(-9)
  squareRoot(valueX)
- **The correct output:** “Square root of a negative number is not a number!”

IV. Conditional Class

Test #1 Error Handling with Non-Integers
Test Purpose: To test what entered numerical value within the syntax should be considered as valid when determining whether it is even or odd.
Test #1.1

- **The input to be applied:**
  setValueX(5.3)
  isEven(5.3)
- **The correct output:**
  “Incorrect value! Use Integers instead.”

Test #1.2

- **The input to be applied:**
  setValueX(5.0)
  isOdd(5.0)
- **The correct output:**
  proceeds to do arbitrary sentence

Test #1.3

- **The input to be applied:**
  setValueX(5.00000)
  isOdd(5.00000)
- **The correct output:**

proceeds to do arbitrary sentence

Test #1.4

- **The input to be applied:**

  setValueX(5.)

  isOdd(5.)

- **The correct output:**

  proceeds to do arbitrary sentence

Test #1.5

- **The input to be applied:**

  setValueX(.2)

  isEven(.2)

- **The correct output:**

  "Incorrect value! Use Integers instead."

Test #1.5

- **The input to be applied:**

  setValueX(2)

  isEven(2)

- **The correct output:**

  proceeds to do arbitrary sentence

Test #2 Error Handling with Negative Numbers

Test Purpose: To show in what forms negative numbers can be considered as valid/invalid.

Test #2.1:

- **The input to be applied:**

  setValueX(-3)

  isOdd(-3)

- **The correct output:**

  proceeds to do arbitrary sentence

Test #2.2:

- **The input to be applied:**

  setValueX(-5.000)

  isOdd(-5.000)

- **The correct output:**

  proceeds to do arbitrary sentence

Test #2.3:

- **The input to be applied:**

  setValueX(-5.11)

  isOdd(-5.11)

- **The correct output:**

  "Incorrect value! Use Integers instead."

Test #2.4:

- **The input to be applied:**

  setValueX(-0.0)

  isEven(-0.0)

- **The correct output:**

  proceeds to do arbitrary sentence

Test #2.5:

- **The input to be applied:**

  setValueX(-.00000)
  isEven(-.00000)
- **The correct output:**

  proceeds to do arbitrary sentence

Test #2.6:

- **The input to be applied:**

  setValueX(-.4)
  isEven(-.4)
- **The correct output:**

  "Incorrect value! Use Integers instead."

V. Loop Class

Test #1: Loop Testing for loop passes
Test Purpose: To test if the number of loop passes is lower, higher or correct
Test #1.1: Skip entire loop

- **The input to be applied:**

  setEnd(0)
  checkcondition(start,end)
- **The correct output:** "Error! Inappropriate number of loop runs to end"

Test #1.2: Only one loop pass

- **The input to be applied:**

  setEnd(1)
  checkcondition(start, end)
- **The correct output: "**Result of Calculation:**"**

Test #1.3: Two loop passes

- **The input to be applied:**

  setEnd(2)
  checkcondition(start, end)
- **The correct output: "**Result of Calculation:"

Test #1.4: m passes through loop where m < n

- **The input to be applied:**

  setEnd(n)
  checkcondition(start, end)
- **The correct output: "**Result of Calculation:"

Test #1.5: n+1 passes through loop where m < n

- **The input to be applied:**

  setEnd(n+1)
  checkcondition(start, end)
- **The correct output: "**Result of Calculation:"

Test #2: Incorrect counter value
Test Purpose: To validate if specified counter value by user is valid
Test #2.1: Negative counter value

- **The input to be applied:**
  setEnd(-2)
  checkcondition(start,end)
- **The correct output:** "Error! Inappropriate number of loop runs to end"

Test #2.2: Positive counter value

- **The input to be applied:**
  setEnd(2)
  checkcondition(start, end)
- **The correct output: "**Result of Calculation:**"**

VI. Assignment Class

Test #1: Independent Paths Through Control Structures
Test Purpose: Exercised to ensure all statements/methods are executed at least once.
Test #1.1 Creating unassigned variable:

- **The input to be applied**:
  createVariable("X")
- **The correct output:** dict = { "X": None }

Test #1.2 Creating assigned variable:

- **The input to be applied**:
  createVariable("X")
  assign("X", 50)
- **The correct output:** dict = { "X": 50}

Test #1.3 Replacing variable value:

- **The input to be applied**:
  createVariable("X")
  assign("X", 50)
  assign("X", 100)
- **The correct output:** dict = { "X": 100}

Test #2: Error Handling
Test Purpose: To show in what types of variables and values are accepted
Test #2.1

- **The input to be applied:**
  createVariable(10)
- **The correct output:** "Error, variable name must be a string."

Test #2.2

- **The input to be applied:**
  createVariable("X")
  assign("X", "Y")

- **The correct output:** “Error, value to be assigned must be of type float.”

Test #2.3

- **The input to be applied:**
  createVariable(“X”)
  assign(“X”, “Y”)
- **The correct output:** “Error, value to be assigned must be of type float.”

Integration Tests

Strategy

The integration strategy chosen will be top-down integration testing. This strategy works best with the architecture chosen for LCL123 since it allows us to test the dependencies of the components by simply following the sequence of events. The following integration tests all follow the same hierarchical pattern: from the interpreter class to the algorithm class, and then followed by the combination of corresponding classes (Assignment, Loop, Conditional, Calculate). Moreover, the manner in which these classes are accessed follows after the depth first approach.

Integration Test 1

Selected integration strategy: Top Down, Scenario-based
Its purpose: To demonstrate how although independent clauses, variables are stored globally and can be changed throughout the program. This test can also be used to demonstrate how the **Interpreter**, **Algorithm**, **Assignment** and **Calculate** Class function together.
How this test should be done:

Input: “Let Y be 100, let X be 50, divide Y by X”
Interpreter Class Output:
Clause 1: [“Let Y be 100”] → Sentence 1: [“Let”, “Y”, “be”, “100”]
Clause 2: [“Let X be 50”] → Sentence 2: [“Let”, “Y”, “be”, “100”]
Clause 3: [“divide Y by X”] → Sentence 3: [“divide”, “Y”, “by”, “X”]
Algorithm Class Output:
Clause 1:
… validated
findAlgorithm() →
Assignment Class Output:
createVariable(“Y”), assign(“Y”, 100)
Clause2:

… validated
findAlgorithm() →
Assignment Class Output:
createVariable(“Y”), assign(“Y”, 100) createVariable(“X”), assign(“X”, 50)
Clause 3:
findAlgorithm() →
Calculate Class Output:
divide(“Y”, “X”) → 2
Expected Final Output: “X==50”, “Y==2”

Integration Test 2

Selected integration strategy: Top Down, Scenario-based
Its purpose: To demonstrate the execution of a specific sentence upon the satisfaction of a condition. The following test further emphasizes the chaining of multiple sentences whereby the value stored within variables are updated as sentence execution occurs sequentially. This test can also be used to demonstrate how the **Interpreter**, **Algorithm**, **Assignment**, **Conditional,** and **Calculate** Class function together.
How this test should be done:
Input: “Let X be 20, if 3 is odd then add 10 to X, subtract 3 from X.”
Interpreter Class Output:
Clause 1: [“Let X be 20”] → Sentence 1: [“Let”, “X”, “be”, “20”]
Clause 2: [“if 3 is odd then add 10 to X”] →
Sentence 2: [“if”, “3”, “is”, “odd”, “then”, “add”, “10”, “to” ,“X”]
Clause 3: [“subtract 3 from X”] → Sentence 3: [“subtract”, “3”, “from”, “X”]
Algorithm Class Output:
Clause 1:
… validated
findAlgorithm() →
Assignment Class Output:
createVariable(“X”), assign(“X”, 20)
Clause 2:
… validated
findAlgorithm() →
Conditional Class Output:
isOdd(3) → true
Calculate Class Output:
add(10, “X”) → 30
Clause 3:
… validated
findAlgorithm() →
Calculate Class Output:
subtract(“X”, 3) → 27
Expected Output: “X==27”

Integration Test 3

Selected integration strategy: Top Down, Scenario-based
Its purpose: To demonstrate the nonexecution of a specific sentence/s upon the dissatisfaction of a condition. This particular block of sentence/s is placed between the *then* keyword and a period(.). This test can also be used to demonstrate how the Interpreter, Algorithm, Assignment, Conditional and Calculate Class function together.
How this test should be done:

Input: "Let X be 95, if 4 is odd then add 10 to X, subtract 5 from X, divide X by 2"
Interpreter Class Output:
Clause 1: ["Let X be 95"] → Sentence 1: ["Let", "X", "be", "95"]
Clause 2: ["if 4 is odd then add 10 to X"] →
Sentence 2: ["if", "4", "is", "odd", "then", "add", "10", "to" ,"X"]
Clause 3: ["subtract 5 from X"] → Sentence 3: ["subtract", "5", "from", "X"]
Clause 4: ["divide X by 2"] → Sentence 4: ["divide", "X", "by", "2"]
Algorithm Class Output:
Clause 1:
… validated
findAlgorithm() →
Assignment Class Output:
createVariable("X"), assign("X", 95)
Clause 2:
… validated
findAlgorithm() →
Conditional Class Output:
isOdd(4) → false
Clause 3:
… validated
findAlgorithm() →
Calculate Class Output:
subtract("X", 5) → 90
Clause 4:
… validated
findAlgorithm() →
Calculate Class Output:
divide("X", 2) → 45
Expected Output: "X==45"

Integration Test 4

Selected integration strategy: Top Down, Scenario-based
Its purpose: To demonstrate that at any point, reassigning variables to other values or variables is possible regardless of what calculations were performed on them. The following test keeps track of variable X and assigns its most recent change in value to variable Y. This test can also be used to demonstrate how the **Interpreter**, **Algorithm**, **Assignment**, **Loop** and **Calculate** Class function together.
How this test should be done:

Input: "Let X be 200, let Y be 50, add 5 to X 2 times, let Y be X, add 2 to Y"
Interpreter Class Output:
Clause 1: ["Let X be 200"] → Sentence 1: ["Let", "X", "be", "200"]
Clause 2: ["Let Y be 50"] → Sentence 2: ["Let", "Y", "be", "50"]
Clause 3: ["add 5 to X 2 times"] → Sentence 3: ["add", "5", "to", "X", "2", "times"]
Clause 4: ["let Y be X"] → Sentence 4: ["let", "Y", "be", "X"]
Clause 5: ["add 2 to Y"] → Sentence 4: ["add", "2", "to", "Y"]
Algorithm Class Output:
Clause 1:
… validated
findAlgorithm() →
Assignment Class Output:
createVariable("X"), assign("X", 200)
Clause 2:
… validated
findAlgorithm() →
Assignment Class Output:
createVariable("Y"), assign("Y", 50)
Clause 3:
… validated
findAlgorithm() →
Loop Class Output:
Calculate Class Output:
add("X", 5) → 205
Assignment Class Output:
assign("X", 205)
setEnd(2)
increment()
checkCondition(1, 2) → false
Calculate Class Output:
add("X", 5) → 210
Assignment Class Output:
assign("X", 210)
increment()
checkCondition(2, 2) → true
Clause 4:

… validated
findAlgorithm() →
    Assignment Class Output:
    assign(“Y”, “X”)
Clause 5:
… validated
findAlgorithm() →
    Calculate Class Output:
    add(“Y”, 2) → 212
Expected Output: “X==210”, “Y==212”

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