Project Report

Group 23

COMP2021 Object-Oriented Programming (Fall 2022)

Member 1: Liu Yuzhou

Member 2: Niyitegeka Berwa Aime Noel

Member3: Lee Jer Tao

Member 4: Li Tong

1. Introduction

This document describes the design and implementation of a command-line based interpreter for the **Simple programming language** by Group 23. The project is one of the assessments assigned in the course COMP2021 – Object-Oriented Programming at The Hong Kong Polytechnic University (PolyU) .

2. Command-Line-Based Interpreter for Simple Programs

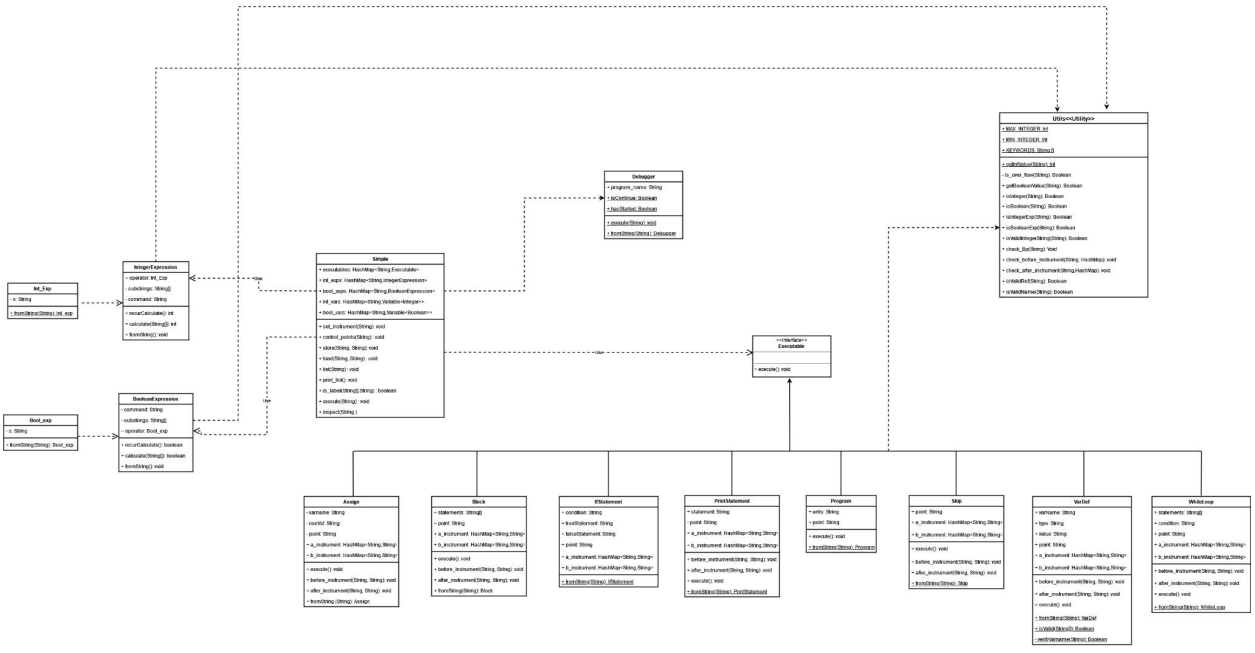
In this section, we will first describe the overall design and the implementation details of the interpreter.

For each requirement we divide them into three parts

- Fields and methods(in the form)
- Logic of the functions for the requirement

2.1 Design

UML CLASS DIAGRAM



DESIGN PATTERNS

1. **FACTORY DESIGN PATTERN**

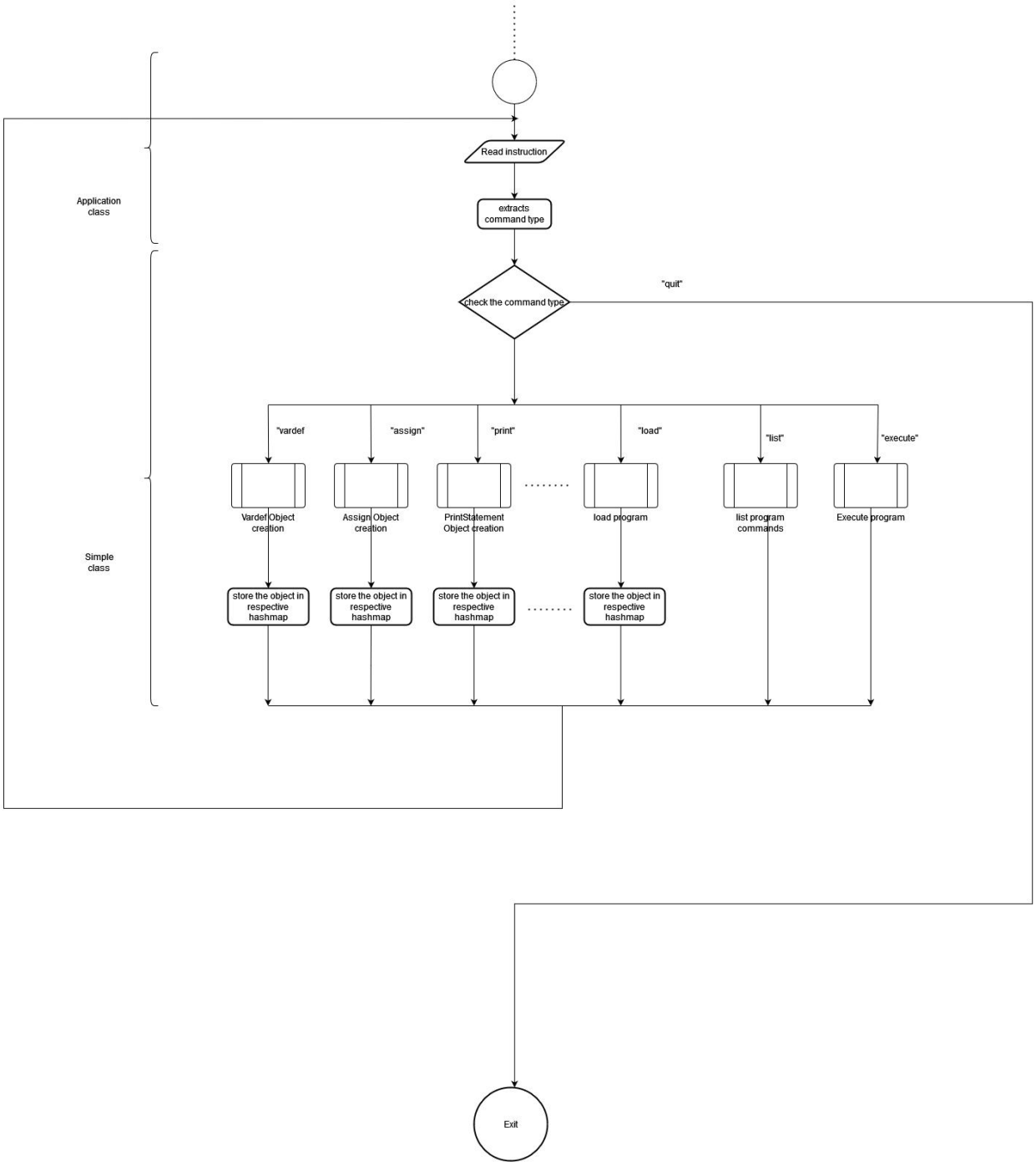
   The interpreter was designed based on the factory design pattern. With this pattern objects of statements and expressions' entities are created based on the input command that the user keys in.

   Working principle:

   - Users will interact with the interpreter in the main function of the Application class.
   - Through command-line, java Scanner will collect the user's command string.
   - The instruction string will be passed to ***addInstruction()*** method of SImple class.
   - The method will extract the command type and the reference label of instruction.
   - Based on the command type, the ***addInstruction()*** method will call a static function ***fromString()*** from the appropriate class.
   - ***fromString()*** function will tokenize the string and build an object of the class and return the build object.
   - In the ***addInstruction()*** function, the built object will be received and then added to the appropriate hashmap of statements/expressions.

   This will be repeated for every command that user enters until they enter "quit".

The working principle is shown on the diagram below:



2. SINGLETON DESIGN PATTERN

The Singleton design pattern was applied to the class Simple. Class Simple hosts globally shared resources including variable hashmaps, and statements/expressions hashmaps. Therefore, to enforce the integrity and universal sharing of the resources, we decided to use the Singleton design pattern.

Why singleton design pattern?

- To ensure that all statements being executed are using the same variable pool and expressions hashmaps, we initially made the resources static.
- However, there was a potential that a developer in the team mistakenly invokes the constructor for the simple class. This would cause the resetting of all static data structures of the programming.
- To prevent this catastrophe, we employed the Singleton design pattern, we chose to hide the constructor from the public and make it private, and have a method that returns an instance of Simple class and creates one when it does not exist yet.

NOTE:Testing the system raised a special case of having a reset function because the test needed to run independently. This resulted in creation of a *reset()* function which generates new data structures for Simple and it is executed after each test.

The implementation of the design is shown below:

```
private static Simple instance;
3 usages  new *
private Simple() {
    Simple.bool_exps = new HashMap<String, BooleanExpression>();
    Simple.int_vars = new HashMap<String, Variable<Integer>>();
    Simple.bool_vars = new HashMap<String, Variable<Boolean>>();
    Simple.executables = new HashMap<>();
    Simple.programs = new HashMap<String, Program>();
    Simple.int_exps = new HashMap<String, IntegerExpression>();
    Simple.store_file = new ArrayList<String>();
    Simple.all_ref = new ArrayList<String>();
    Simple.result = new HashSet<String>();
    Simple.break_point=new HashMap<>();
    Simple.before_instruments=new HashMap<>();
    Simple.after_instruments=new HashMap<>();
}
new *
public static Simple getSimpleInstance(){
    if( instance == null){
        return new Simple();
    }
    return instance;
}
4 usages  Noel2002 *
public static void reset() { instance = new Simple(); }
```

Simple

Fields

Field	Type	Description
int_exps	HashMap<String, IntegerExpression>	Store the pair of the reference and the object of integer expression.
bool_exps	HashMap<String, BooleanExpression>	Store the pair of the reference and the object of boolean expression.
bool_vars	HashMap<String,Variable<Boolean>>	Store the pair of the reference and the object of boolean variable.
int_vars	HashMap<String,Variable<Integer>>	Store the pair of the reference and the object of integer variable.
executables	HashMap<String,Executable>	Store the pair of the reference and the object of executable.
programs	HashMap<String,Program>	Store the pair of the reference and the program class.
store_file	ArrayList<String>	Store the command lines in ArrayList and save in the given filename..
all_ref	ArrayList<String>	Store the references to the functions in ArrayList
result	HashSet<String>	Store the command lines in ArrayList.
break_point	HashMap<String,HashSet<String>>	Store the string that function as a reference of the statement which has the breakpoints.
entry	String	Function as the entry point to run the program.
before_instruments	HashMap<String,HashSet<String>>	
after_instruments	HashMap<String,HashSet<String>>	

Class Methods

Return type	Description
void	**addInstruction(String commandType, String instruction)** Add the command type and the command string to the HashMap executables based on the command type.

void	**set_instrument(String instruction)** Place curly braces before and after the statement.
void	**control_points(String instruction)** Act as a toggle breakpoint when entering debugging mode.
void	**store (String ref, String path)** Store the command lines as a .SIMPLE file in the specific path.
void	**load (String path, String new_name)** Load the .SIMPLE file using the keyword
void	**list(String ref)** Tracks every expression used in the program and prints them out.
void	**print_list()** Print the list of commands that are stored in HashSet.
boolean	**Is_label(String [] labels, String lab)** Check if the command contains a label.
void	**execute(String instruction)** Evaluate the condition and execute.
void	**inspect(String instruction)** Implementation of the inspect method.

In this project, we divided all the requirements into two parts: ‘Objects’ and ‘Methods’. For requirements that can be implemented as an object, we write it as a separate object class (e.g., Blocks, Program, Expressions...), and store it in multiple HashMaps. By using the advantage of HashMap's unordered storage and ability to store data dynamically, it could make our classes more flexible and efficient. The independent encapsulation of objects also allows our code to have more extension interfaces, While ensuring the security of internal modules, it also increases the scalability of the project.

On the other hand, for the requirements that can be implemented as a method, we define it in a main class (Simple) (e.g., Execute, Load, List…). It serves as an advantage as we can call multiple HashMaps stored in the main class to store, call, update or upload data directly. This also improves the readability of the code, making our operations on objects clearer.

It involves the usage of the class Enum to store the variable. When it comes to the classification of the Expression class, we divide it into two types according to the type of return value of the class: IntegerExpression and BooleanExpression . It is not defined by the number of operands as both binary and unary expressions can call and contain each other. We also define the operators in the expressions as Enum classes, because some of them are only suitable for integer data calculations ("+", "-", "*"...) or boolean value calculations ("== ", "!").

Use of interface:

```
public interface Executable {
    8 implementations  Noel2002
    public void execute();
    8 implementations  new *
    public void before_instrument(String program_name,String instrument);
    8 implementations  new *
    public void after_instrument(String program_name,String instrument);
}
```

During the implementation of the program, an interface called ‘Executable’ will be defined. By using this interface, it not only simplifies the recognition of the input sentence, but also greatly simplifies the sentence processing when executing the program. Therefore, when the execute function is called, it does not execute the whole sentence, but executes the words in the sentence.This reduces the judgment of the statement and shortens the time for us to locate the specific error in the program.

Use of fromString functions:

```
public static Assign fromString(String expString){
    String[] tokens = expString.split( regex: " ");
    String varName = tokens[2];
    String rawVal = tokens[3];
    return new Assign(varName, rawVal,tokens[1]);
}
```

```
public static VarDef fromString(String expString){
    String[] tokens = expString.split( regex: " ");
    String type = tokens[2];
    String varName = tokens[3];
    String value = tokens[4];
    return new VarDef(varName, type, value,tokens[1]);
}
```

There are multiple fromString methods in classes which are used to receive the inputted sentence and store the splitted words in specific fields. It will call the constructor of its class and return a well-defined class object.

For instance, the fromString() method in the Assign and VarDef class, it will first split the sentence into tokens. The tokens will be assigned to different String fields such as ‘type’, ‘varName’, ‘value’ and more. We will then return and call the specific function recursively to proceed to the remaining functions.

[REQ1]

1) The requirement is implemented.
2) Implementation details.

[Implementation in VarDef Class]

2.2 VarDef

Vardef Syntax:

vardef statementRef datatype variableName value

- statementRef - unique identifier of the statement
- Datatype - datatype of the declared variable
- variiableName - inique identifier of the variable being declared
- Value - value to to initialize to the variable upon creation

Class fields

Field	Type	Description
varName	String	Stores the variable name of the variable being declared
value	String	Stores the value to be initialized to the variable upon creation. This can be a constant or a reference to an expression or a variable name. ● Constant: “123”, “true”, “false” ● Expression name: “exp1”, “exp2” ● Variable name: “x”, “y”, “var”
point	String	Stores the statement label

Class methods

Return type	Description
	VarDef(String varName, String type, String value,String point) Constructs an object of class Vardef
Vardef	**fromString(String expString)** Forms an object of the class from an instruction string passed. **Parameters**: ● expString: the instruction string E.g: “vardef vardef1 int x 10” **Return**: ● Returns an object of class Vardef **Throws**: ● SyntaxErrorException - When the instruction’s string does not conform with the predefined syntax of the Vardef command.
boolean	**isValid(String[] tokens)** Validates the instruction’s string against the defined syntax of the Vardef command. **Parameters**: ● Tokens - array of string tokens which represents a tokenized version of the instruction string. **Return**: ● True when the format of the instruction’s string matches with Vardef command’s syntax **Throws**: ● InvalidFormatException - When the instruction is malformed; i.e less tokens or more tokens than expected. There should be 5 tokens. ● InvalidIdentifierException: When the instruction’s reference is not valid(for more details on validity of reference, see the naming rules.
void	**execute()** Creates the variable in the variable’s hashmap and assigns the value to the created variable.

All the defined variables are stored in the two HashMaps based on their data type: Boolean and Integer.

```
case "vardef":
    VarDef varDef = VarDef.fromString(instruction);
    executables.put(ref, varDef);
    break;
```

```
public void execute(){
    Utils.check_before_instrument(this.point,b_instrument);
    Utils.check_Bp(this.point);
    if(type.equals("int")){
        Integer actualValue = Utils.getIntValue(value);
        Simple.int_vars.put(varName, new Variable<Integer>(actualValue));
    }
    else if (type.equals("bool")){
        Boolean actualValue = Utils.getBooleanValue(value);
        Simple.bool_vars.put(varName, new Variable<Boolean>(actualValue));
    }
    else {
        throw new IllegalArgumentException("Invalid command of variable declaration");
    }
    Utils.check_after_instrument(this.point,a_instrument);
}
```

When the keyword “execute” is inputted, the program will identify and store the data into HashMap according to their data types. For example, if the data type is ‘int’, it will store into the int_vars HashMap where it stores variables which have the ‘int’ data type.

Error at executing time

At the beginning, we found out that the ‘execute’ method will run after detecting the keyword “vardef”. It will create and add the variable to the HashMap, which does not meet the requirements of the project and leads to the occurrence of unknown problems. Fortunately, we solved this problem by first storing the "Vardef" class, calculating and adding variables when the program received "execute" (which means It will be executed only if it is stored in a block defined in SIMPLE language).

[REQ2]

1) The requirement is implemented.
2) Implementation details.

Binexpr

We define the binary expression into two data types: “Int” and “Boolean”,which are linked to the two enums: “Int_exp” and “Bool_exp”.

IntegerExpression

Class fields

Field	Type	Description
command	String	Holder for command
substrings	String[]	Holder for substrings from command
operator	Int_exp	Holder for Integer operator

Class methods

Return type	Description
int	**recurCalculate()**
int	**calculate(String[] substrings)**
void	**fromString()**

Integer Expression - Binary

```
public static HashMap<String, IntegerExpression> int_exps;
```

HashMap is used to store the IntegerExpression classes

```
case "binexpr":
    if (Utils.isIntegerExp(instruction)) {
        IntegerExpression intExp = new IntegerExpression(instruction.substring( beginIndex: 9 + ref.length()));
        int_exps.put(ref, intExp);
```

When the word “binexpr” is detected, it will use ’isIntegerExp’ from the class Utils to detect the data type. If it is an integer expression, it will split the instruction, add the command label and the remaining command into the int_exps HashMap.

Integer Expression - Enums

```
public enum Int_exp {
    1 usage
    PLUS( s: "+"),SUBTRACT( s: "-"),MULTIPLY( s: "*"),DIVISION( s: "/"),REMAINDER( s: "%"),U_PLUS( s: "#"),U_MINUS( s: "~");
    3 usages
    private String s;
```

This enum class defines and enumerates all the arithmetic operators related to this project, which makes the calling of symbols clearer. (some symbols are used in Unary Expression.)

Integer Expression - Calculation

```java
public int calculate(String[] substrings)
{
    if (substrings.length == 3) {
        Integer left = Utils.getIntValue(substrings[0]);
        Integer right = Utils.getIntValue(substrings[2]);
        switch (this.operator) {
            case PLUS:
                return left + right;
            case SUBTRACT:
                return left - right;
            case DIVISION:
                return left / right;
            case MULTIPLY:
                return left * right;
            case REMAINDER:
                return left % right;
            default:
                throw new ArithmeticException();
        }
    }
```

This function determines the expression type by the length of the substring array.After it is determined that it is a binary expression, since the expression type of the left and right operands are uncertain (may be another expression, variable, or specific value), getIntValue function will be called and the integer value of the operands will be returned. After determining the arithmetic operator, it will then return the value after the arithmetic operation.

```java
public static int getIntValue(String value){
    Integer actualValue;
    if( Simple.int_vars.get(value) != null){
        actualValue = Simple.int_vars.get(value).getvalue
    }
    else if( Simple.int_exps.get(value) != null){
        actualValue = Simple.int_exps.get(value).recurCal
    }
    else{
        if(!isValidIntegerString(value)){
            throw new IllegalArgumentException("the integ
        }
        if(is_over_flow(value)){
            if(value.charAt(0)=='-'){
                return MIN_INTEGER;
            }
            else {
                return MAX_INTEGER;
            }
        }
        actualValue = Integer.valueOf(value);
    }
    return actualValue;
}
```

getIntValue function in Utils class

Boolean Expression - Binary

```java
public static HashMap<String, BooleanExpression> bool_exps;
```

HashMap is used to store the BooleanExpression classes

```
else if (Utils.isBooleanExp(instruction)) {
    BooleanExpression booleanExp = new BooleanExpression(instruction.substring( beginIndex: 9 + ref.length()));
    bool_exps.put(ref, booleanExp);
}
```

When the word “binexpr” is detected, it will use ’isBooleanExp’ from the class Utils to detect the data type. If it is a boolean expression, it will split the instruction, add the command label and the remaining command into the int_exps HashMap.

Boolean Expression - Enum

```
public enum Bool_exp {
    1 usage
    GreaterThan( s: ">"), GreaterOrEqual( s: ">="), SmallerThan( s: "<"), SmallerOrEqual( s: "<="), Equal( s: "=="),
    2 usages
    Unequal( s: "!="), And( s: "&&"), Or( s: "||"),NOT( s: "!");
    3 usages
    private String s;
```

This enum class for BooleanExpression defines and enumerates all the boolean computing symbols related to this project, which makes the calling of symbols clearer. (some symbols are used for Unary Expression.)

With the similar use of int_exp enum, the fromstring() function enables enum class definitions to be made externally.

Boolean Expression - Calculation

Unlike the binary expression of integer type, the left and right operands of the binary expression of boolean type contain more types than the binary expression of integer type. Therefore, after determining that the length is a binary expression, it is also necessary to judge the operator to further decide the operand type. The judgment of the symbol is still divided into three categories.

(1) Operator only for integer: “>”, “>=”, “<”, “<=”
(2) Operator only for boolean: “&&”, “||”
(3) Operator for two types: “==”, “!=”

(1)Operator only for integer:

```
public boolean calculate(String[] substrings)
{
    if (substrings.length == 3) {
        if(get_substring()[get_substring().length - 2].equals(">") ||
                get_substring()[get_substring().length - 2].equals(">=") ||
                get_substring()[get_substring().length - 2].equals("<")  ||
                get_substring()[get_substring().length - 2].equals("<=")) {

            Integer left = Utils.getIntValue(substrings[0]);
            Integer right = Utils.getIntValue(substrings[2]);
            switch (this.operator){
                case GreaterThan:
                    return left > right;
                case GreaterOrEqual:
                    return left >= right;
                case SmallerThan:
                    return left < right;
                case SmallerOrEqual:
                    return left <= right;
                default:
                    throw new ArithmeticException();
```

According to the characteristics of symbols, this type of symbols can only be used to calculate integer data. Similar to the calculation of binary expressions, the value of the operand is obtained recursively through the call of the Utils class function, and then the calculation symbol is confirmed through the switch structure, and finally a Boolean value is returned.

(2)Operator only for boolean:

```
else if(get_substring()[get_substring().length - 2].equals("&&") ||
        get_substring()[get_substring().length - 2].equals("||")){

    Boolean left = Utils.getBooleanValue(substrings[0]);
    Boolean right = Utils.getBooleanValue(substrings[2]);
    switch (this.operator){
        case And:
            return left && right;
        case Or:
            return left || right;
        default:
            throw new ArithmeticException();
    }
```

As these symbols can only be used to calculate boolean values, it is different from the calculation function. This function will call the function to calculate the boolean type in the Utils class and also obtain the value of the operand through recursion. and then obtain the corresponding calculation through Switch operator, and finally return a boolean value.

```
else{// when the operator is "==" and "!="
    if(Utils.isInteger(substrings[0]) && Utils.isInteger(substrings[2])){
        Integer left = Utils.getIntValue(substrings[0]);
        Integer right = Utils.getIntValue(substrings[2]);
        switch (this.operator){
            case Equal:
                return left.equals(right);
            case Unequal:
                return !(left.equals(right));
            default:
                throw new ArithmeticException();
        }
    }
    else if( Utils.isBoolean((substrings[0])) && Utils.isBoolean(substrings[2])){
        Boolean left = Utils.getBooleanValue(substrings[0]);
        Boolean right = Utils.getBooleanValue(substrings[2]);
        switch (this.operator){
            case Equal:
                return left.equals(right);
            case Unequal:

                return !(left.equals(right));
            default:
                throw new ArithmeticException();
        }

    }
    else{
        throw new ArithmeticException("Operations not defined on the give arguments");
    }

}
```

(3)Operator for two types: The two symbols "==" and "!=" can be used to calculate integers and Boolean values, so in this type of calculation, it is necessary to further determine the type of the operand, and then call the same method to calculate and return a boolean value.

```
public static boolean getBooleanValue(Str
    Boolean actualValue;
    if( Simple.bool_vars.get(value) != nu
        actualValue = Simple.bool_vars.ge
    }
    else if( Simple.bool_exps.get(value)
        actualValue = Simple.bool_exps.ge
    }
    else{
        if(!(value.equals("true") || valu
            throw new IllegalArgumentExce
        }
        actualValue = Boolean.parseBoolea
    }
    return actualValue;
}
```

getBooleanValue function in Utils class

```
public static boolean isInteger(String value) throws NumberFormatException{
    if(Simple.int_exps.get(value) != null || Simple.int_vars.get(value) != null){
        return true;
    }
    try {
        Integer.valueOf(value);
        return true;
    }
    catch (NumberFormatException e){
        return false;
    }
}
8 usages  Noel2002
public static boolean isBoolean(String value) throws NumberFormatException{
    if(Simple.bool_exps.get(value) != null || Simple.bool_vars.get(value) != null){
        return true;
    }
    return value.equals("true") || value.equals("false") ? true : false;

}
```

Two functions in Utils class to detect the expression type. When determining the expression type, the two functions will first search in their corresponding HashMap: to find whether it is an expression or whether it is a defined variable. If null is returned, it will try to convert the type for further judgment.

3.**Error conditions and how they implemented**

In the early stage of project development, the code for deciding the type was written inside the function to calculate the return value, which led to the complexity of the code – 600 lines at one time. Later, because the judgment condition was encapsulated in the Utils class, the calculation function became clear and easy to understand, and the problem of redundant code was solved.

Error condition:(can refer to the form above)

Only happens when executing the error will be generated.it is runtime error.

- Integer: IllegalArgumentException("InvalidFormat: The command is malformed! The statement has unexpected number of tokens.");

[REQ3]

1)The requirement is implemented.

2)Implementation details.

Unexpr

Similar to the development method of Binexpr, we define the unary expression into two data types: “Int” and “Boolean”,which are linked to the two enums: “Int_exp” and “Bool_exp”.

Integer expression - Unary

```
public static HashMap<String, IntegerExpression> int_exps;
```

HashMap use to store the IntegerExpression classes

```
case "unexpr":
    if (Utils.isIntegerExp(instruction)) {
        IntegerExpression intExp = new IntegerExpression(instruction.substring( beginIndex: 8 + ref.length()));
        int_exps.put(ref, intExp);
```

When the word “unexpr” is detected, it will use ’isIntegerExp’ from the class Utils to detect the data type. If it is an integer expression, it will split the instruction, add the command label and the remaining command into the int_exps HashMap.

Integer Expression - Enum

```
public enum Int_exp {
    1 usage
    PLUS( s: "+"),SUBTRACT( s: "-"),MULTIPLY( s: "*"),DIVISION( s: "/"),REMAINDER( s: "%"),U_PLUS( s: "#"),U_MINUS( s: "~");
    3 usages
    private String s;
```

Binary and unary calculations in terms of integer share the same enumeration class, where the "#" and "~" symbols are only used for unary calculations.

Integer Expression - Calculation

```
else {//for unary integer expression
    Integer operand = Utils.getIntValue(substrings[1]);
    switch (this.operator) {
        case U_PLUS:
            return +operand; //'+' MAYBE CAN REMOVE
        case U_MINUS:
            return -operand;
        default:
            throw new ArithmeticException();
    }

}
```

Unary calculations only involve one operand. Therefore, compared to binary integer expressions, the calculation of unary integer expressions is simpler and clearer, and the principle is the same.
Use the getIntValue function in the Utils class to obtain the value of the operand through recursion, use the switch structure to find the corresponding symbol, calculate it, and return the corresponding integer value.

Boolean Expression - Unary

```
public static HashMap<String, BooleanExpression> bool_exps;
```

HashMap use to store the BooleanExpression classes

```
else if (Utils.isBooleanExp(instruction)) {
    BooleanExpression booleanExp = new BooleanExpression(instruction.substring( beginIndex: 8 + ref.length()));
    bool_exps.put(ref, booleanExp);
}
```

When the word “unexpr” is detected, it will use ’isBooleanExp’ from the class Utils to detect the data type. If it is a boolean expression, it will split the instruction, add the command label and the remaining command into the int_exps HashMap.

Boolean Expression - Calculation

```
else{
    Boolean operand=Utils.getBooleanValue(substrings[1]);
    switch (this.operator){
        case NOT:
            return !operand;
        default:
            throw new ArithmeticException();
    }
}
```

There is only one symbol involved in calculating unary boolean values("!"). Therefore, the calculation method is simple. We can use the getBooleanValue function in the Utils class to obtain the value of the operand through recursion, use the switch structure to find the corresponding symbol, perform calculations, and return the corresponding Boolean value.

3) Error conditions and how they implemented:

Similar to binary expression, the code for deciding the type was written inside the function to calculate the return value, which led to the complexity of the code. This problem has been solved after implementing the encapsulation in the Utils class.

Error condition:(can refer to the form above)

- Integer: IllegalArgumentException("InvalidFormat: The command is malformed! The statement has unexpected number of tokens.");

[REQ4]

1)The requirement is implemented.

2) Implementation details.

Assign

Assign Syntax:

assign statementRef varname value

- *statementRef* - unique identifier of the statement
- *Varname* - name of the variable
- *Value* - value to be assigned

Class fields

Field	Type	Description
varName	String	Stores the variable name of the variable to assign the value.
rawVal	String	Stores the value or the source of the value assigned to the variable. This can be a constant or a reference to an expression or a variable name. ● Constant: “123”, “true”, “false” ● Expression name: “exp1”, “exp2” ● Variable name: “x”, “y”, “var”
point	String	Stores the statement label

Class methods

Return type	Description
	Assign(String varName, String rawVal, String point) Constructs an object of class Assign
Assign	**Assign fromString(String expString)** Forms an object of the class from an instruction string passed. **Parameters**: ● expString: the instruction string

	E.g: “assign assign1 x 10” **Return**: ● Returns an object of class Assign **Throws**: ● SyntaxErrorException - When the instruction’s string does not conform with the predefined syntax of the Assign command.
boolean	**isValid(String[] tokens)** Validates the instruction’s string against the defined syntax of the Assign command. **Parameters**: ● Tokens - array of string tokens which represents a tokenized version of the instruction string. **Return**: ● True when the format of the instruction’s string matches with Assign command’s syntax
void	**execute()** Updates the value stored in the specified variable(*varName*) to the new specified value(*rawVa*l).

[implementation on the class Assign]

```
case "assign":
    Assign assign = Assign.fromString(instruction);
    executables.put(ref, assign);
    break;
```

We will read the user’s input. After calling fromString function to create an object of it,it will be added into the executables HashMap.Similar to VarDef, assignments of the correct form will be made after sentences containing "execute" are inputted.

3) Error conditions and how they implemented:

Error condition:(can refer to the form above)

- InvalidFormatException - When the instruction is malformed; i.e less tokens or more tokens than expected. There should be 4 tokens
- InvalidIdentifierException: When the instruction’s reference is not valid(for more details on validity of reference, see the naming rules.

[REQ5]

1)The requirement is implemented.

2) Implementation details.

PrintStatement

PrintStatement Syntax:

print statementRef expression

- *statementRef* - unique identifier of the statement
- *expression*: value/expression to be printed

Class fields

Field	Type	Description
statement	String	Boolean or Integer value/ expression to be printed on the screen This can be: - Constant - “123”, “true” - Variable - “bool1”, “x” - Integer/Boolean expression - “exp1”, “exp4”
point	String	Stores the statement label

Class methods

Return type	Description
	PrintStatement(String statement,String point) Constructs an object of class PrintStatement
PrintStatement	**fromString(String instruction)** Forms an object of the class from an instruction string passed. **Parameters**: ● instruction: the instruction string E.g: - “print print1 false ”, “print print2 123” **Return**: ● Returns an object of class Block **Throws**: ● SyntaxErrorException - When the instruction’s string does not conform with the predefined syntax of the Block command.
boolean	**isValid(String[] tokens)** Validates the instruction’s string against the defined syntax of the PrintStatement command. **Parameters**: ● Tokens - array of string tokens which represents a tokenized version of the instruction string. **Return**: ● True when the format of the instruction’s string matches with PrintStatement command’s syntax
void	**execute()**

	Evaluates the value of *expression* and display the value on the screen

```
5 usages  Noel2002 *
public class PrintStatement implements Executable{
    6 usages
    String statement;
```

[Implementation of printStatement class]

Member of the class:
Statement: label of the value that will be printed (eg. exp1).

The above is the fields we have for printStatment class, the statement is the Expression of value needed to print here.

```
    @Override
    public void execute(){
        Utils.check_Bp(this.point);
        Utils.check_before_instrument(this.point,b_instrument);
        if(Utils.isBoolean(this.statement)){
            System.out.print("["+ Utils.getBooleanValue(statement) +"]");
        }
        else if (Utils.isInteger(statement)) {
            System.out.print("[" + Utils.getIntValue(statement) + "]");
        }
        else{
            throw new IllegalArgumentException(statement+ " is undefined in the scope");
        }
        Utils.check_after_instrument(this.point,a_instrument);
    }
```

In this execute function we will check the data type of the statement, according to it,we can check through different HashMaps to find it, then print it with brackets. (gerBooleanValue() and getIntValue() are function from Utils class which have already display before)

[Implementation in print class]

```
case "print":
    PrintStatement print = PrintStatement.fromString(instruction);
    executables.put(ref, print);
    break;
```

We will first read the user's instruction,then fromString function will be called to create an object of it, and finally put it into the executables HashMap.

3) Error conditions and how they implemented:

Error condition:(can refer to the form above)

- InvalidFormatException - When the instruction is malformed; i.e less tokens or more tokens than expected. There should be 3 tokens.

- InvalidIdentifierException - When the instruction's reference is not valid(for more details on validity of reference, see the naming rules.

[REQ6]

1) The requirement is implemented.
2) Implementation details.

Skip

Skip Syntax:

Skip statemIentRef

- *statementRef* - unique identifier of the statement
- *sk*: value/expression to be printed

Class fields

Field	Type	Description
point	String	Stores the statement label

Class methods

Return type	Description
	Skip(String point) Constructs an object of class Skip
Skip	**fromString(String instruction)** Forms an object of the class from an instruction string passed. **Parameters**: ● instruction: the instruction string E.g: - "skip skip1" **Return**: ● Returns an object of class Skip **Throws**: ● SyntaxErrorException - When the instruction's string does not conform with the predefined syntax of the Skipcommand.
boolean	**isValid(String[] tokens)** Validates the instruction's string against the defined syntax of the Skin command.

	Parameters: ● Tokens - array of string tokens which represents a tokenized version of the instruction string. **Return**: ● True when the format of the instruction's string matches with Skin command's syntax **Throws**: ● InvalidFormatException - When the instruction is malformed; i.e less tokens or more tokens than expected. There should be 2 tokens. ● InvalidIdentifierException - When the instruction's reference is not valid(for more details on validity of reference, see the naming rules.
void	**execute()** Evaluates the value of *expression* and display the value on the screen

[Implementation in skip class]

```
4 usages   Noel2002 *
public class Skip implements Executable
{
```

```
    Noel2002 *
    public static Skip fromString(String instruction){
        String[] tokens=instruction.split( regex: " ");
        return new Skip(tokens[1]);
    }
```

[implementation in simple class]

```
case "skip":
    Skip skip = Skip.fromString(instruction);
    executables.put(ref, skip);
    break;
```

We will read the user's instruction,then the fromString function will be called to create an object of it,then put it into the executable's hashmap.

3) Error conditions and how they implemented:

Error condition:(can refer to the form above)

- InvalidFormatException - When the instruction is malformed; i.e less tokens or more tokens than expected. There should be 2 tokens.
- InvalidIdentifierException - When the instruction's reference is not valid(for more details on validity of reference, see the naming rules.

[REQ7]

1) The requirement is implemented.
2) Implementation details.

Block

Block Syntax:

block statementRef statement1 statement2 . . . statement_n

- *statementRef* - unique identifier of the statement
- *Statement1 … n* - statements to be executed in the block

Class fields

Field	Type	Description
statements	String[]	List of statement references of the executable statements that will be executed in the block. These statements have to be executable; i.e they should be the objects of classes that implement the executable interface.
point	String	Stores the statement label

Class methods

Return type	Description
	Block(String[] statements, String point) Constructs an object of class Block
Block	**fromString(String expString)** Forms an object of the class from an instruction string passed. **Parameters**: ● expString: the instruction string E.g: “block block1 vardef1 assign2 ” **Return**: ● Returns an object of class Block **Throws**: ● SyntaxErrorException - When the instruction’s string does not conform with the predefined syntax of the Block command.
boolean	**isValid(String[] tokens)** Validates the instruction’s string against the defined syntax of the Block command. **Parameters**: ● Tokens - array of string tokens which represents a tokenized version of the instruction string. **Return**: ● True when the format of the instruction’s string matches with Block command’s syntax

	Throw: ● InvalidFormatException - When the instruction is malformed; i.e less tokens or more tokens than expected. There should be at least 3 tokens. ● InvalidIdentifierException: - When the instruction's reference is not valid(for more details on validity of reference, see the naming rules. - When the identifier for one of the *statements* has not yet been declared an executable statement.
void	**execute()** Executes the list of executable *statements* one by one in the order they were listed in the instruction's string.

[implementation of block class]

```
public class Block implements Executable{
    2 usages
    String[] statements;
```

```
Noel2002 +1 *
public void execute(){
      System.out.println(Simple.break_points);
      System.out.println(this.point);
    Utils.check_before_instrument(this.point,b_instrument);
    Utils.check_Bp(this.point);
    for(String expref: statements){
        Executable exp = Simple.executables.get(expref);
        exp.execute();
    }
    Utils.check_after_instrument(this.point,a_instrument);
}
```

[The usage in simple class]

```
case "block":
    Block block = Block.fromString(instruction);
    executables.put(ref, block);
    break;
```

When the main class Simple receives the statement input with the "while" field, a block class will be created through its fromString and added to the corresponding HaspMap.

```
public static HashMap<String, Executable> executables;
```

HashMap use to store Block class

The labels stored in the block must be reference executable

3) Error conditions and how they implemented:

Error condition:(can be check in the form above)

- InvalidFormatException - When the instruction is malformed; i.e less tokens or more tokens than expected. There should be at least 3 tokens.

- InvalidIdentifierException:
  i) When the instruction's reference is not valid(for more details on validity of reference, see the naming rules.
  ii) When the identifier for one of the *statements* has not yet been declared an executable statement.

[REQ8]

1) The requirement is implemented.
2) Implementation details.

[implementation in IfStaement class]

IfStatement Syntax:

If statementRef condition statement1 statement2

- *statementRef* - unique identifier of the statement
- *condition* - an boolean expression
- *Statement1* - statement to be executed when the condition evaluates to true
- *Statement2* - statement to be executed when the condition evaluates to false

Class fields

Field	Type	Description
condition	String	Stores the boolean value or source of the value for the if statement's condition This can be a constant or a reference to an expression or a variable name. ● Constant: "123", "true", "false" ● Expression name: "exp1", "exp2" ● Variable name: "x", "y", "var"
truestatement	String	Statement to be executed when the condition evaluates to true Note: This has to be an executable statement; i.e instance of the executable interface.
falsestatement	String	Statement to be executed when the condition evaluates to false Note: This has to be an executable statement; i.e instance of the executable interface.
point	String	Stores the statement label

Class methods

Return type	Description
	IfStatement(String condition, String trueStatement, String falseStatement, String point) Constructs an object of class IfStatement
IfStatement	**fromString(String instruction)** Forms an object of the class IfStatement from an instruction string passed. **Parameters**: ● expString: the instruction string E.g: “if if false print1 assign1” **Return**: ● Returns an object of class IfStatement **Throws**: ● SyntaxErrorException - When the instruction’s string does not conform with the predefined syntax of the IfStatement command.
boolean	**isValid(String[] tokens)** Validates the instruction’s string against the defined syntax of the Vardef command. **Parameters**: ● Tokens - array of string tokens which represents a tokenized version of the instruction string. **Return**: ● True when the format of the instruction’s string matches with IfStatement command’s syntax **Throws**: ● InvalidFormatException - When the instruction is malformed; i.e less tokens or more tokens than expected. There should be 5 tokens. ● InvalidIdentifierException - ○ When the instruction’s reference is not valid(for more details on validity of reference, see the naming rules. ○ When the identifier for *trueStatement* or *falseStatements* has not yet been declared as executable statements.
void	**execute()** Evaluate the condition and execute: ● *trueStatement* when the *condition* evaluates to true ● Otherwise, execute *falseStatement* when the *condition* evaluates to false

[IfStatement class]

```
public class IfStatement implements Executable{
    3 usages
    private String condition;
    3 usages
    private String trueStatement;
    3 usages
    private String falseStatement;
    5 usages
    private String point;
```

```
public void execute(){
    Executable exp;
    Utils.check_before_instrument(this.point,b_instrument);
    Utils.check_Bp(this.point);
    if(Utils.getBooleanValue(condition)){
        exp = Simple.executables.get(trueStatement);
    }
    else{
        exp = Simple.executables.get(falseStatement);
    }
    exp.execute();
    Utils.check_after_instrument(this.point,a_instrument);
}
```

[Implementation in Simple class]

```
case "if":
    IfStatement ifStatement = IfStatement.fromString(instruction);
    executables.put(ref, ifStatement);
    break;
```

When the main class Simple receives a statement input with an "if" field, an IfStatement class will be created through its fromString and added to the corresponding HaspMap.

```
public static HashMap<String, Executable> executables;
```

HashMap use to store IfStatement class

3) Error conditions and how they implemented

In the beginning stage of this class setting, we have calculated the return values in both cases before calculating the judgment conditions, which is unnecessary. Later, we first calculated the judgment condition and then performed the calculation to solve this problem.

Error condition:(can refer to the form above)

- InvalidFormatException - When the instruction is malformed; i.e less tokens or more tokens than expected. There should be 5 tokens.

- InvalidIdentifierException -
    - When the instruction's reference is not valid(for more details on validity of reference, see the naming rules.
    - When the identifier for *trueStatement* or *falseStatements* has not yet been declared as executable statements.

[REQ9]

1) The requirement is implemented.
2) Implementation details.

WhileLoop

WhileLoop Syntax:

while statementRef condition statement1 statement2 . . . statement_n

- *Condition*: boolean expression
- *statementRef* - unique identifier of the statement
- *Statement1 … n* - statements to be executed in the block

Class fields

Field	**Type**	**Description**
condition	String	Boolean value/ expression for determining the stopping of the loop. The loop terminates once the condition evaluates to false. This can be: - Constant - "false", "true" - Variable - "bool1", "b1" - Boolean expression - "exp1", "exp4"
statements	String[]	List of statement references of the executable statements that will be executed when the condition of the loop evaluates to true. . These statements have to be executable; i.e they should be the objects of classes that implement the executable interface.
point	String	Stores the statement label

Class methods

Return type	Description
	WhileLoop(String condition, String[] statements,String point) Constructs an object of class WhilleLoop
WhileLoop	**fromString(String expString)** Forms an object of the class from an instruction string passed. **Parameters**: ● expString: the instruction string E.g: “block block1 vardef1 assign2 ” **Return**: ● Returns an object of class Block **Throws**: ● SyntaxErrorException - When the instruction’s string does not conform with the predefined syntax of the Block command.
boolean	**isValid(String[] tokens)** Validates the instruction’s string against the defined syntax of the Block command. **Parameters**: ● Tokens - array of string tokens which represents a tokenized version of the instruction string. **Return**: ● True when the format of the instruction’s string matches with Block command’s syntax **Throws**: ● InvalidFormatException - When the instruction is malformed; i.e less tokens or more tokens than expected. There should be at least 3 tokens. ● InvalidIdentifierException: ○ When the instruction’s reference is not valid(for more details on validity of reference, see the naming rules. ○ When the identifier for one of the *statements* has not yet been declared an executable statement.
void	**execute()** Executes the list of executable *statements* one by one in the order they were listed in the instruction’s string.

[implementation in WhileLoop class]

```
public class WhileLoop implements Executable{
    2 usages
    private String[] statements;
    2 usages
    private String condition;
```

```
public void execute(){
    Utils.check_before_instrument(this.point,b_instrument);
    Utils.check_Bp(this.point);
    while(Utils.getBooleanValue(condition)){
        for(String expref: statements){
            Executable exp = Simple.executables.get(expref);
            exp.execute();
        }
    }
    Utils.check_after_instrument(this.point,a_instrument);
}
```

[The implementation in simple class]

```
case "while":
    WhileLoop loop = WhileLoop.fromString(instruction);
    executables.put(ref, loop);
    break;
```

When the main class Simple receives a statement input with a "while" field, a WhlipLoop class is created through its fromString and added to the corresponding HaspMap.

```
public static HashMap<String, Executable> executables;
```

HashMap use to store IfStatement class

3) Error conditions and how they implemented

Error condition:(can refer to the form above)

- InvalidFormatException - When the instruction is malformed; i.e less tokens or more tokens than expected. There should be at least 3 tokens.

- InvalidIdentifierException:
    - When the instruction's reference is not valid(for more details on validity of reference, see the naming rules.
    - When the identifier for one of the *statements* has not yet been declared an executable statement.

- SyntaxErrorException - When the instruction's string does not conform with the predefined syntax of the Block command.

[REQ10]

1) The requirement is implemented.
2) Implementation details.

Program

Program Syntax:

program programName

- *programName*- name of the program

Class fields

Field	Type	Description
entry	String	Function as the entry point to run the program.
point	String	Stores the statement label

Class methods

Return type	Description
void	**execute()** Since the string "entry" stored in this class must have an "executable" interface, so when this class is run, it is only necessary to simply find the storage object in the corresponding HashMap and call its "execute" function, when we will get the feedback.
Program	**fromString(String instruction)** Forms an object of the class from an instruction string passed. **Parameters**: ● instruction: the instruction string **Return**: ● Returns an object of class program **Throws**: ● SyntaxErrorException - When the instruction's string does not conform with the predefined syntax of the program command.

```
public class Program implements Executable {
    3 usages
    private String entry;
```

This class has one main field, which is used to store the sentence which is represented as the program. (entry eg. “while1”, “block1”)

```
case "program":
    Program program = Program.fromString(instruction);
    executables.put(ref, program);
    break;
```

When the main class Simple receives a statement input with a "program" field, a Program class will be created through the fromString function and added to the corresponding HashMap.(ref will be the program name eg. “printeven”)

```
public static HashMap<String, Executable> executables;
```

HashMap is used to store program class

```
public void execute(){
    if(Simple.executables.containsKey(entry)){
        Simple.executables.get(entry).execute();
        Debugger.hasStarted=false;
    }
}
```

3) Error conditions and how they implemented

Error condition:(can refer to the table above)

- SyntaxErrorException - When the instruction’s string does not conform with the predefined syntax of the program command.

[REQ11]

1) The requirement is implemented.
2) Implementation details.

Field	Type	Description
entry	String	Function as the entry point to run the program.
executables	HashMap<String,Executable>	Store the pair of the reference and the object of executable.

Class methods

Return type	Description
void	**execute(String instruction)** The "execute" function of the main class is the first to run among all the execute functions of the class. This function will get the name of

	the project that needs to be run and look it up in the HashMap. When the program name exists and is not null, please execute it. The program's Execution ability, which also indicates that a program starts its operation. **Parameters:** ● *Instruction* - the command from user input **Throws** ● IllegalArgumentExpectation When we cannot get valid entry for the program

[implementation in execute]

```
case "execute":
    entry=instruction.split( regex: " ")[1];
    execute(instruction);
    entry=instruction.split( regex: " ")[1];
    System.out.println();
    break;
```

Set the static entry stores the name of the program which is currently executing.
Since this statement of running the program does not need to be stored, when the main class Simple receives the statement input with the "execute" field, this code will directly obtain the program name in the input statement, and then it will call the function "execute" in the main class.

[Execute function in Simple class]

```
public static void execute(String instruction){
    String expRef = instruction.split( regex: " ")[1];
    Executable exp = executables.get(expRef);
    if(exp != null){
        exp.execute();
    }
    else{
        throw new IllegalArgumentException("Invalid expression reference");
    }
}
```

3) Error conditions and how they implemented

Throws

- IllegalArgumentExpectation

When we cannot get valid entry for the program

[REQ12]

1) The requirement is implemented.
2) Implementation details.

Field	Type	Description
store_file	ArrayList<String>	Store the command lines in ArrayList and save in the given filename.
result	HashSet<String>	Store the command lines in ArrayList.

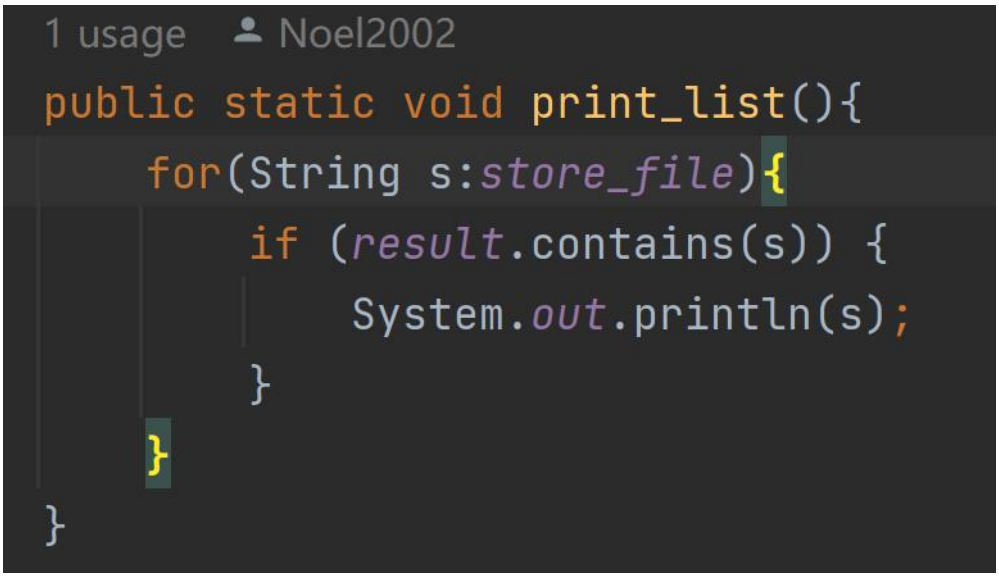
Class methods

Return type	Description
void	**list(String ref)** ● We will first store all the commands into an array,by the way find the entry of the program and store it. ● we choose to use a queue to traverse the input of the users, We will enqueue the label of each commands related to this program, and use it to find other programs, then dequeue after checking. In case there might be a lot of duplicates,we choose to use a hashset to store those commands. **Parameters:** ● *ref* - the name of the program that we want to list.
void	**print_list()** ● We call this function to print the commands stored in the HashSet result . To sort it, we use the ArrayList store_file . If it contains the commands used, then we will print the command. As store_file is ordered, our list should be ordered too.

[The implementation of list function in Simple]

```
2 usages   Noel2002 +1
public static void list(String ref){
    String[] array=new String[store_file.size()];
    Queue<String> queue= new LinkedList<>();
    int index=0;
    HashSet<String> contain=new HashSet<>();
    String label="";
    for(String command:store_file){
        String[] temp=command.split( regex: " ");
        array[index++]=temp[1];
        if(temp[1].equals(ref)&&temp[0].equals("program")){
            label=command;}
    }
    queue.add(label);
```

```
    while(!queue.isEmpty()){
        String[] test=queue.remove().split( regex: " ");
        for(int k=store_file.size()-1;k>=0;k--){
            String[] temp=store_file.get(k).split( regex: " ");
            for(int i=0;i<temp.length;i++){
                if(is_label(array,temp[i])){
                    for(int j=2;j<test.length;j++){
                        if(temp[i].equals(test[j])){
                            result.add(store_file.get(k));
                            if(!contain.contains(store_file.get(k))){
                                queue.add(store_file.get(k));
                                contain.add(store_file.get(k));
                            }
                        }
                    }
                }
            }
        }
    }
}
```



Error conditions and how they implemented

We used the idea in the Data Structure course which we have studied this semester. By using this structure, we can start traversing the data from the bottom to the top of the commands. However, the disadvantage is the time complexity, which is (O(n^3)).

[REQ13]

1) The requirement is implemented.
2) Implementation details.

Field	Type	Description
store_file	ArrayList<String>	Store the command lines in ArrayList and save in the given filename.
result	HashSet<String>	Store the command lines in ArrayList.

Class methods

Return type	Description
void	**list(String ref)** ● We will first store all the commands into an array,by the way find the entry of the program and store it. ● we choose to use a queue to traverse the input of the users, We will enqueue the label of each commands related to this program, and use it to find other programs, then dequeue after checking. In case there might be a lot of duplicates,we choose to use a hashset to store those commands. **Parameters:**

	• *ref* - the name of the program that we want to list.
void	**print_list()** • We call this function to print the commands stored in the HashSet result . To sort it, we use the ArrayList store_file . If it contains the commands used, then we will print the command. As store_file is ordered, our list should be ordered too.

[implementation in store]

In the development of this project, we defined the storage method as calling in the main class, rather than defining a class for the storage function.

```
case "store":
    String path = instruction.substring( beginIndex: 7 + ref.length());
    store(ref, path);
    break;
```

When the main class Simple receives a statement input with a "store" field, this method will intercept the path in it. It will pass the program name and path to be stored into the "store" function. (This function is defined in the main class Simple)

```
public static ArrayList<String> store_file;
public static ArrayList<String> all_ref;
```

The subsequent "store" function will call the stored statement from the two ArrayList shown above.

[store function in Simple class]

```java
public static void store(String ref, String path){
    result.clear();
    list(ref);
    try {
        FileWriter fwrite = new FileWriter(path,  append: true);
        for(int i = 0; i < store_file.size(); i++){
            if(result.contains(store_file.get(i))) {
                fwrite.write( str: store_file.get(i) + "\n");
            }
        }
        fwrite.close();
    }
    catch (IOException e) {
        System.out.println("Unexpected error occurred");
        e.printStackTrace();
    }
}
```

When storing a program, we will call the "list" function mentioned earlier to ensure that the stored statements are free of redundant and irrelevant statements. Therefore, before calling the "list" function, we need to clear the HashSet result used to store the results returned by the "list" function to ensure the normal operation of the function.

```java
public static HashSet<String> result;
```

The HashSet use to store the consequence of List function

In this function, we use the "FileWriter" package (this package can change the suffix name of the stored file), to write the statements in the ArrayList "store_file" into the file we created in a circular manner (each correct statement will be added to the ArrayList after inputting).

Before writing into the file, we will first check whether the statement is related to the program that needed to be stored by comparing it with the ArrayList "result" created by the List function. This is to refrain from storing unrelated statements.

3) Error conditions and how they implemented

At the beginning, we did not consider that there would be unrelated statements from the program stored in the file. At that time, this function would directly store all the statements entered before the field "store", which made the generated file very messy. Using the "list" function, we only store the fields related to the program name in it. This also simplifies the development of the "load" function.

[REQ14]

1) The requirement is implemented.
2) Implementation details.

In the development of this project, we defined the method of "load" as calling in the main class, rather than defining a class specifically.

```
case "load":
    load(ref, instruction.split( regex: " ")[2]);
    break;
```

When the main class Simple receives the statement input with the "load" field, this method will intercept the path in it. It will then pass the program name and path to be stored into the "load" function. (This function is defined in the main class Simple)

[Clear all the things stored in Simple class]

```
public static void load(String path, String new_name){
    int_vars.clear();
    bool_vars.clear();
    bool_exps.clear();
    int_exps.clear();
    executables.clear();
    programs.clear();
    store_file.clear();
    result.clear();
    break_point.clear();
    Simple simple=new Simple();
```

Since the statements contained in the program loaded into this class that may conflicts with the statements entered by the user before calling this function, we need to clear all stored data before loading the program.

[read the file]

```
try {
    // Create f1 object of the file to read data
    File f1 = new File(path);
    Scanner dataReader = new Scanner(f1);
    while (dataReader.hasNextLine()) {
        String instruction = dataReader.nextLine();
        store_file.add(instruction);
    }
    dataReader.close();
}
catch (FileNotFoundException exception) {
    System.out.println("Unexpected error occurred!");
    exception.printStackTrace();
}
```

After the file is obtained from the path, this function reads the statements in the file into the ArrayList "store_file" and stores them in a loop for subsequent operation.

[Re-add data to the storage

structure]

```
String[] temp = store_file.get(store_file.size() - 1).split( regex: " ");
temp[1] = new_name;
store_file.set(store_file.size() - 1, temp[0] + " " + temp[1] + " " + temp[2]);
for(String command : store_file){
    String commandType=command.split( regex: " ")[0];
    addInstruction(commandType,command);
}
```

After storing the data in the static ArrayList "store_file ", we need to re-store the data into the corresponding storage structure, which may involve every function introduced previously. Therefore, we need to change the name of the stored program to be the entered program name to work properly. Afterwards, we recall the "addInstruction" function in the main class "Simple", and add statements to it in order to restore all the program information into the storage structure so that it can run accurately.

3) Error conditions and how they implemented

When designing this function at the beginning, we did not consider the need to clear the data in the storage structure, which caused the program to conflict with the previous data. However, we are able to solve the problem by first clearing the data of the storage structure.

[REQ15]

1) The requirement is implemented.
2) Implementation details.

[The implementation of quit]

```java
public static void main(String[] args){
    Scanner sc = new Scanner(System.in);
    String instruction;
    boolean isStop = false;
    while(!isStop){
        instruction = sc.nextLine();
        String commandType = instruction.split( regex: " ")[0];
        if(commandType.equals("quit")){
            isStop = true;
        }
        else {
            Simple.addInstruction(commandType, instruction);
            if(commandType != "load"){
                Simple.store_file.add(instruction);
            }
        }
    }
}
```

In the design of our program, the detection of the "quit" instruction is before all other statement detection. In the "main" function, we set the loop condition. When the user enters "quit", the loop will be terminated and the program will exit.

[BON1]

1) The requirement is implemented.

2) Implementation details.

[Debugger]

Debugger

Class fields

Field	Type	Description
program_name	String	name of the program to execute in debug mode
isContinue	Boolean	The flag to signal whether the debugger should resume(in program class)execution of the program after stopping at the break point.
hasStarted	Boolean	The flag to signal whether the debugger has already started execution of the program or not. ● When false, command 'debug' starts the execution of the program from the beginning. ● When true, command 'debug' resumes the execution of the program from where it stopped at the breaking point. ● When it is false, the inspect function cannot be used

Class methods

Return type	Description
void	**execute(String expRef)** It is different from other classes implementing the executable interface, instead it is similar to the one in Simple class.When exp.execute() invoked, the program will keep running until it meets a break point,then it will stop there. **Parameters:** ● *expRef* **-** the command from user

[The debugger class]

```
7 usages  AlexLiuyz *
public class Debugger{
    3 usages
    public static boolean isContinue = true;
    4 usages
    public static boolean hasStarted = false;
    AlexLiuyz *
    public static void execute(String expRef){
        Executable exp= Simple.executables.get(expRef);
        if(exp!=null&&!Simple.break_point.get(Simple.entry).contains(expRef)){
            exp.execute();
        }
        else{
            throw new IllegalArgumentException();
        }
        Simple.break_point.get(Simple.entry).add(expRef);
    }
}
```

The usage of Simple.entry(static String variable,which denotes the program name) is to notify which program we are executing currently.Therefore, if two or more programs share the same commands, it will break the running of the program with breakpoints, but other programs will remain in running process.

[The implementation in program class](To resume the debugger)

```
Noel2002 *
@Override
public void execute(){
    if(Simple.executables.containsKey(entry)){
        Simple.executables.get(entry).execute();
        Debugger.hasStarted=false;
    }
}
```

This is to resume the debug function,once it finishes we can change the state of hasStarted to false, and get ready for the debug next time.

Method in Simple class

Field	Type	Description
break_point	HashMap<String,HashSet<String>>	Store the string that function as a reference of the statement which has the breakpoints.
entry	String	Function as the entry point to run the program.

Return type	Description
void	**control_points(String instruction)** Sets the breakpoints to the instruction. By checking the commands we can find the program name, and put

	the label name of the breakpoint into the static hashset which is called break_point(If the hashset contains the breakpoints, we will just remove it). **Parameters** ● *Instruction* - command from the user
	inspect(String instruction) ● When not in debug mode we cannot use this method. ● For checking, it is easy, just use the varname as key,and find the value of it in hashmap. Then print it. **Parameters** ● *Instruction* - command from the user

[The implementation of debugger in Simple class]

```
case "debug":
    String programName = instruction.split( regex: " ")[1];
    entry=programName;
    if(!Debugger.hasStarted){
        if( Simple.executables.get(programName) != null){
            Executable pgrm = Simple.executables.get(programName);
            if( pgrm instanceof Program){
                Debugger.hasStarted=true;
                pgrm.execute();
            }
            else {
                System.out.println("The statement can be executed as program");
            }
        }
        else{
            System.out.println("Program cannot be found!");
        }
    }
    Debugger.isContinue = true;
    break;
```

The image shows how we read the command in add_instruction function.

[The implementation of togglebreakpoint in simple class]

```
    case "togglebreakpoint":
        control_points(instruction);
        break;
```

```
1 usage  AlexLiuyz *
public static void control_points(String instruction){
    String[] input=instruction.split( regex: " ");
    for(String command:store_file){
        String[] tokens=command.split( regex: " ");
        if(tokens[0].equals("program")&&tokens[1].equals(input[1])){
            if(break_point.containsKey(tokens[1])){
                if(break_point.get(tokens[1]).contains(input[2])){
                    break_point.get(tokens[1]).remove(input[2]);
                    return;
                }
                break_point.get(tokens[1]).add(input[2]);
            }
            else{
                HashSet<String> hashSet=new HashSet<>();
                hashSet.add(input[2]);
                break_point.put(tokens[1],hashSet);
            }
        }
    }
    System.out.println(break_point);
}
```

[The implementation of inspect function in simple class]

```
case "inspect":
    inspect(instruction);
    break;
```

```
1 usage  new *
public static void inspect(String instruction){
    if(!Debugger.hasStarted){
        System.out.println("Not in debug mode");
        return;
    }
    String varname = instruction.split( regex: " ")[2];
    if(Simple.int_vars.get(varname) != null){
        System.out.println("<" +Simple.int_vars.get(varname).getvalue()+">");
    }
    else if(Simple.bool_vars.get(varname)!=null){
        System.out.println("<"+Simple.bool_vars.get(varname).getvalue()+">");
    }
    else {
        System.out.println("Variable is not defined");
    }
}
```

The first image shows how we read the command in add_instruction function.
The second image shows the implementation of the inspect method.

Method in util class

Return type	Description
void	**check _Bp(String point)** This method is used to check whether the current executable class is a breakpoint or not.

	When you invoke this method, it will first check breakpoint is setted for the current program(Without checking it might encounter NullPointerError for the HashMap breakpoint). If it is a breakpoint, we can find it and go to the second if statement. We will now set the isContinue flag(mentioned in Debugger class) to false in order to break the running process of the current program. However, users are still allowed to continue to add instructions. **Parameters** ● *point* - the label of current executable class.

[check_Bp method](in Utils class)

```
7 usages  new *
public static void check_Bp(String point){
    Scanner sc = new Scanner(System.in);
    if(Simple.break_point.get(Simple.entry)==null){
        return;
    }
    if( Simple.break_point.get(Simple.entry).contains(point)) {
        Debugger.isContinue = false;
        while (!Debugger.isContinue) {
            String instruction = sc.nextLine();
            String commandType = instruction.split( regex: " ")[0];
            Simple.addInstruction(commandType, instruction);
        }
    }
}
```

Other examples

[The example shows how we break the execution of a executable class]

The following image is an example of the check_Bp method to be used in assigning class.(All executable class should have the ability to be set as a breakpoint)

```
9 usages  Noel2002 +1 *
public class Assign implements Executable{
    6 usages
    private String varName;
    4 usages
    private String rawVal;
    4 usages
    private String point;
    3 usages
    public HashMap<String,String> b_instrument;
    3 usages
    public HashMap<String,String>  a_instrument;
    1 usage  Noel2002 *
    public Assign(String varName, String rawVal,String point) {
        this.varName = varName;
        this.rawVal = rawVal;
        this.point=point;
        this.b_instrument=new HashMap<>();
        this.a_instrument=new HashMap<>();
    }
```

Unique name will be used for each class which will be stored in ‘point’, its function is to check if it is a breakpoint in check_Bp function.

```
Noel2002 *
@Override
public void execute(){
    Utils.check_before_instrument(this.point,b_instrument);
    Utils.check_Bp(this.point);
    if(Simple.int_vars.get(varName) != null){
        Integer actualValue = Utils.getIntValue(rawVal);
        Simple.int_vars.get(varName).setvalue(actualValue);
    }
    else if (Simple.bool_vars.get(varName) != null){
        Boolean actualValue = Utils.getBooleanValue(rawVal);
        Simple.bool_vars.get(varName).setvalue(actualValue);
    }
    Utils.check_after_instrument(this.point,a_instrument);
}
```

3) Error conditions and how they implemented

After implementing the debugger, we find that there is a bug that when we set a breakpoint on a command, then it will also break the execution of other programs. That is because for the first time we just use a hashset to store all the breakpoints, so later on we come up with the hashmap to store both program name and its breakpoints. And in the check_Bp function,we will first check the program name, then check if it is a breakpoint belonging to the current program.

[BON2]

1) The requirement is implemented.
2) Implementation details.

[The implementation in Executable interface]

```
Noel2002 *
public interface Executable {
    8 implementations  Noel2002
    public void execute();
    8 implementations  new *
    public void before_instrument(String program_name,String instrument);
    8 implementations  new *
    public void after_instrument(String program_name,String instrument);
}
```

We will ensure that each executable command will overwrite the method ‘before_instrument’, which is to make sure the instrument will be printed out before the execution of that label occurs, and ‘after_instrument’ which is to make sure the instrument will be printed out after the label occurs.

We will use the Assign class which implements executable as an example. Here, we have two HashMaps, one is for storing before the instrument of the current class, we design because it may have different instrument values in different programs. Therefore,

we will store the program name as the key of the HashMap, and the second place is the instrument value. The usage of a_instrument is somehow similar.

```
3 usages
public HashMap<String,String> b_instrument;
3 usages
public HashMap<String,String>  a_instrument;
```

```
Noel2002 *
@Override
public void before_instrument(String program_name,String instrument){
    this.b_instrument.put(program_name,instrument);
}
new *
@Override
public void after_instrument(String program_name,String instrument){
    this.a_instrument.put(program_name,instrument);
}
```

We put the check of before instrument before the check of break points, this can achieve the print of instrument value before the execution of current label,the check of after_instrument is somehow similar.

```
Noel2002 *
@Override
public void execute(){
    Utils.check_before_instrument(this.point,b_instrument);
    Utils.check_Bp(this.point);
    if(Simple.int_vars.get(varName) != null){
        Integer actualValue = Utils.getIntValue(rawVal);
        Simple.int_vars.get(varName).setvalue(actualValue);
    }
    else if (Simple.bool_vars.get(varName) != null){
        Boolean actualValue = Utils.getBooleanValue(rawVal);
        Simple.bool_vars.get(varName).setvalue(actualValue);
    }
    Utils.check_after_instrument(this.point,a_instrument);
}
```

[The implementation in simple class]

```
case "instrument":
    set_instrument(instruction);
    break;
```

```
6 usages
public static HashMap<String,HashSet<String>> before_instruments;
6 usages
public static HashMap<String,HashSet<String>> after_instruments;
```

```
1 usage   AlexLiuyz *
public static void set_instrument(String instruction){
    String[] input=instruction.split( regex: " ");
    for(String command:store_file){
        String[] tokens=command.split( regex: " ");
        if(tokens[0].equals("program")&&tokens[1].equals(input[1])){
            if(input[3].equals("before")){
                if(before_instruments.containsKey(tokens[1])){
                    before_instruments.get(tokens[1]).add(input[2]);
                    executables.get(input[2]).before_instrument(tokens[1],input[4]);
                } else{
                    HashSet<String> hashSet=new HashSet<>();
                    hashSet.add(input[2]);
                    before_instruments.put(tokens[1],hashSet);
                    executables.get(input[2]).before_instrument(tokens[1],input[4]);
                }
            }
            else if(input[3].equals("after")){
                if(after_instruments.containsKey(tokens[1])){
                    after_instruments.get(tokens[1]).add(input[2]);
                    executables.get(input[2]).after_instrument(tokens[1],input[4]);
                } else{
                    HashSet<String> hashSet=new HashSet<>();
                    hashSet.add(input[2]);
                    after_instruments.put(tokens[1],hashSet);
                    executables.get(input[2]).after_instrument(tokens[1],input[4]);
                }
            } else{
                System.out.println("Wrong command");
            }
        }
    }
```

The first image shows how we read the command in add_instruction function.

To set instrument, we have made a method set_instrument to achieve ut, here we have two different cases, one is the instrument will be set before the occur of the label, anther is that it may occur after the label, so we have two different if conditions to deal with that, the main difference between it and the check_Bp here is that after add the label's class into hashmap,we also need to input value into its hashmap.

[The implementation in Utils class]

```
7 usages   new *
public static void check_before_instrument(String point, HashMap before){
    if(Simple.before_instruments.get(Simple.entry)==null){
        return;
    }
    if(Simple.before_instruments.get(Simple.entry).contains(point)){
        System.out.print("{"+before.get(Simple.entry)+"}");
    }
}
7 usages   new *
public static void check_after_instrument(String point, HashMap after){
    if(!Simple.after_instruments.containsKey(Simple.entry)){
        return;
    }
    if(Simple.after_instruments.get(Simple.entry).contains(point)){
        System.out.print("{"+after.get(Simple.entry)+"}");
    }
}
```

These two method works somehow similar, the design of them shares the advantage by using the point field which was add when we made the debugger, here we just check if the point appeared in the before_instrument hashmap, and after_instrument hashmap, if so we will print its value once it appear before or after the execution of the hashmap.

3) Error conditions and how they implemented

[Set_instrument method]
The problem we have met in designing this method is that, in the beginning, we didn't know how to access the hashmap in label classes as there are different kinds of labels, because it is an attribute of each label class, using object didn’t solve the problem. Finally, the idea we came up with is to add two methods (after_instrument(String program_name,String instrument), before_instrument(String program_name,String instrument)) in the executable interface,because that is the common attribute all the executable classes share with. By overriding those methods, we successfully set the instrument value to those executable classes.

[Implementation in simple class]
Here we just share the same idea as the design of the debugger function,because we need to both store the program name and class which has been set instrument value in that program together. The only difference between set instrument method and set breakpoint method is that further on, the instrument setting will continue to set value to the point.