Computer Science Department

San Francisco State University

CSC 413 Spring 2018

Assignment 1 - Expression Evaluator and Calculator GUI

GitHub repository link:
https://github.com/sfsu-csc-413-spring-2018/assignment-1-expression-evaluator-and-gui-Gayatri27

Compiling and Run instructions

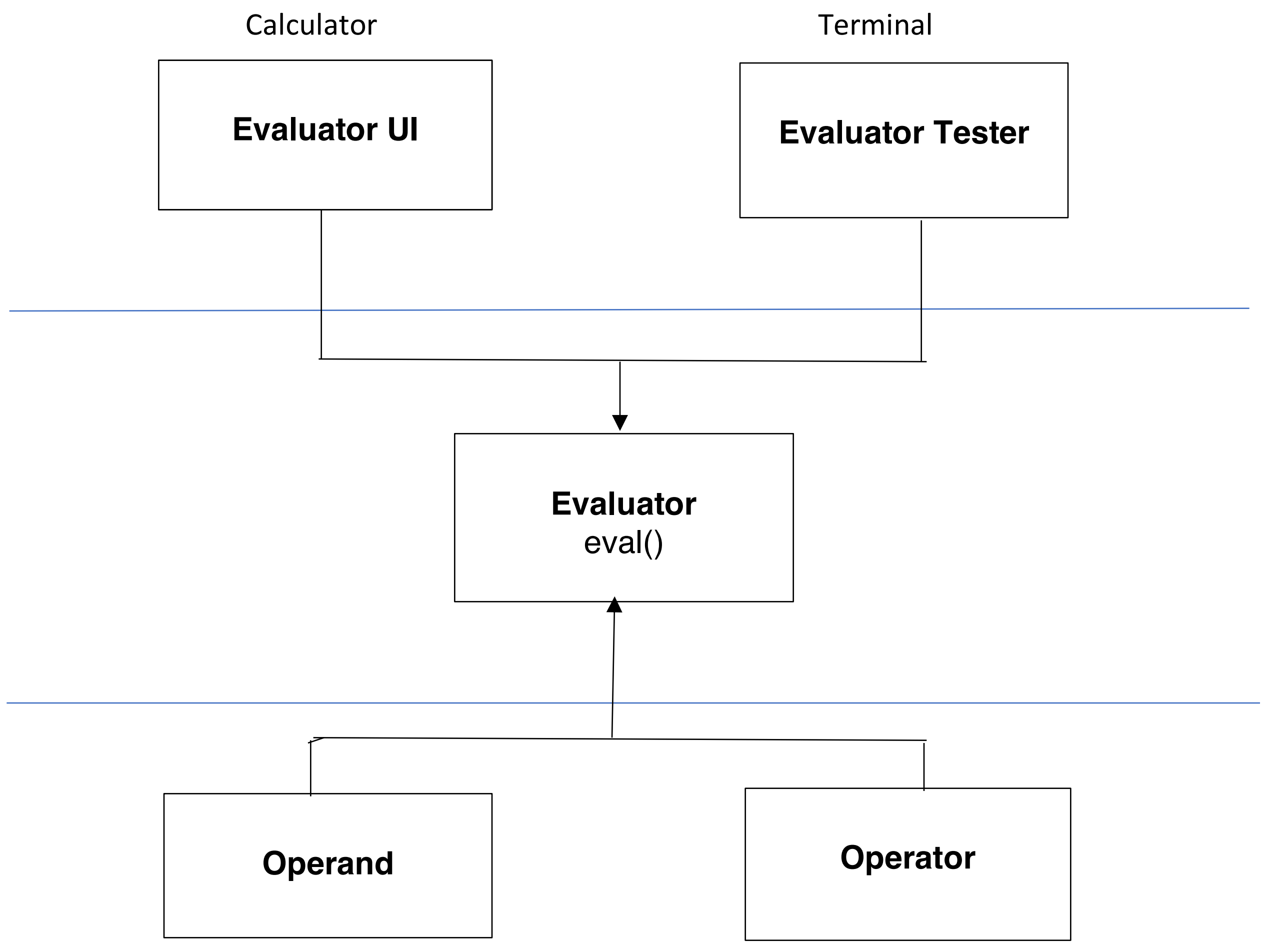
```
javac *.java
java EvaluatorUI   OR   java EvaluatorTester
```

BY

Gayatri Pise	917922296	gpise@mail.sfsu.edu

1.**Project Introduction**

The assignment is to complete the implementation of the Evaluator class which is responsible to calculate output of a given mathematical expression. The method eval() in this class takes String as input parameter, processes the mathematical expression and outputs an integer value. The operators in the mathematical expression follow the PEMDAS priority rule for calculations. The operator and operands of the expression are processed and saved in the Operator and Operand objects respectively. There are two stacks designed to hold the Operator's and Operand's. The priority of operators is saved in a static hashmap and cannot be altered. The super-class Operator is further sub classed by several operator sub-classes such as AdditionOperator, SubtractionOperator and so on. During processing, we pop the operator and operands, calculate their result and the result is pushed back to the operand stack. Two operands are popped for every operator. The process of popping from stack is performed until the stack is empty and the last result is returned from the eval() method.



2. **Summary of Technical Work**

The *EvaluatorUI* and *EvaluatorTester* classes take the mathematical expression input from the user. *EvaluatorUI* is a calculator GUI which extends the *javax.swing.JFrame* class. In this *JFrame,* we draw a *java.awt.TextField* to display the expression and a *java.awt.Panel* to show calculator buttons. We use the *java.awt.Button* for calculator buttons and each button is attached to an *ActionListener* with the *EvaluatorUI* class context. Due to the *ActionListener*, we receive the button clicks in *actionPerformed(ActionEvent arg0)* method and the user input is processed in this method.

The *EvaluatorTester* class is the terminal version of accepting user input. It runs a *for* loop to keep accepting expressions from the terminal, evaluate the expression and display the result. From the terminal, expression must be entered in double quotation marks in order to avoid ambiguity. Such as *"(3+4) * 2".*

Evaluator class is the heart of this application and is responsible to process the mathematical expression and output its result. The public method *eval(String input)* in this class is called from *EvaluatorUI* and *EvaluatorTester*. This method takes the mathematical expression as the *String* datatype and returns solution as an *integer* datatype. The constructor of *Evaluator* class initializes the *OperatorStack, OperandStack* and a *HashMap* mapping the mathematical operators to its respective classes. First the *eval()* method separates operators and operands from the expression using *StringTokenizer*. The following delimiters are provided to StringTokenizer: "+-*^/#!() ". The separated operators and operands are inserted into their respective stacks and the stack-based expression evaluation algorithm is used to compute the result. The stack-based expression evaluation algorithm is described in the following paragraphs.

The *Operand* class is used as an object for each operand identified in the mathematical expression. The operand class has two constructors to save the operand value in the object. First constructor takes the input as *String* datatype, converts to *integer* and saves as *integer* datatype. Second constructor takes input as *integer* datatype and saves as an *integer*. This class exposes two public methods to other classes. First is *getValue()* to expose the operand integer value from *Operand* object and second is *check(String token)* to check if the *String* parameter is a valid operand.

The Operator class is an abstract class and holds the static *HashMap* to map different operator sub-classes to the mathematical operators. This class exposes one public method *check(String token)* to see if the token is a valid operator for our implementation. Other abstract methods such as *priority()* and *execute(Operand op1, Operand op2)* are implemented by respective operator sub-classes.

There are 8 operator subclasses namely: *AdditionOperator, SubtractionOperator, MultiplicationOperator, DivisionOperator, PowerOperator, LeftParenthesesOperator,* *RightParenthesesOperator* and *InitOperator*. All these classes extend the *Operator* class and *@Override* the two abstract methods from *Operator*. The *InitOperator* is used as a dummy operator in our implementation and is not a valid mathematical operator.

The operators are assigned priorities based on the PEMDAS rule (*https://en.wikipedia.org/wiki/Order_of_operations*) as follows:

Operator	**Priority**
AdditionOperator, SubtractionOperator	2
MultiplicationOperator, DivisionOperator	3
PowerOperator	4
LeftParenthesisOperator, RightParenthesisOperator	1
InitOperator	-1

We use the following algorithm to evaluate the mathematical expression:

- If an operand token is scanned, an *Operand* object is created from the token, and pushed to the operand *Stack*
- If an operator token is scanned, and the operator *Stack* is empty, then an *Operator* object is created from the token, and pushed to the operator *Stack*
- If an operator token is scanned, and the operator *Stack* is not empty, and the operator's precedence is greater than the precedence of the *Operator* at the top of the *Stack*, then and *Operator* object is created from the token, and pushed to the operator *Stack*
- If the token is (, and *Operator* object is created from the token, and pushed to the operator *Stack*
- If the token is), the process *Operators* until the corresponding (is encountered. Pop the (*Operator*.
- If none of the above cases apply, process an *Operator*.

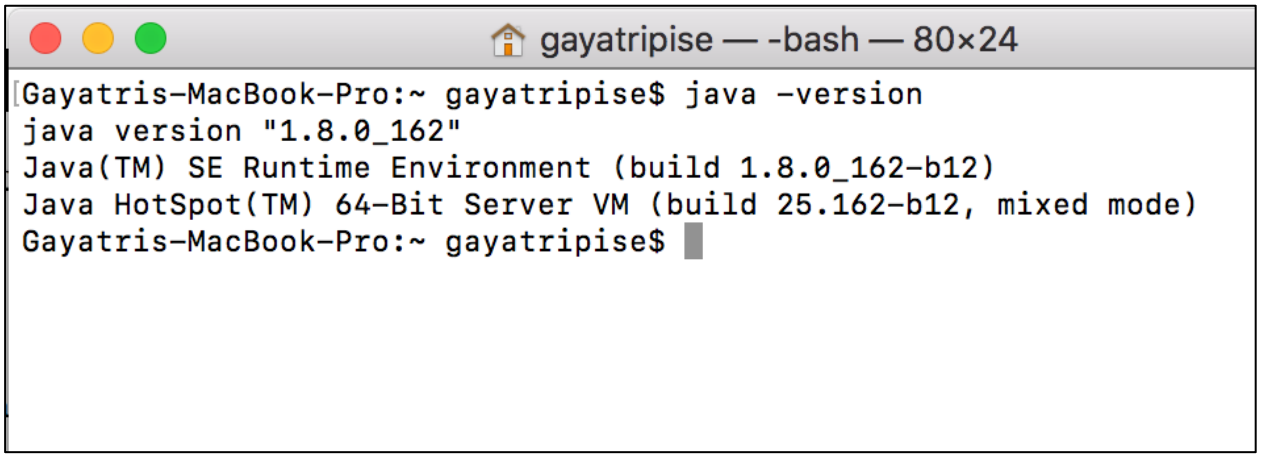
  Processing an *Operator* means to:
- Pop the operand *Stack* twice (for each operand - note the order!!)
- Pop the operator *Stack*
- Execute the *Operator* with the two *Operands*
- Push the result onto the operand *Stack*

- When all tokens are read, process *Operators* until the operator *Stack* is empty.

3. **Execution and Development environment**

The expression evaluator described above is coded in java language. Java is a programming language that is concurrent, class-based, object-oriented, and specifically designed to have as few implementation dependencies as possible. JVM is a computing machine which enables the computer to run a Java program. For compiling and execution, Java source code is compiled into bytecode when we use the javac compiler. The bytecode gets saved on the disk with the file extension .class. When the program is to be run, the bytecode is converted using the compiler. The result is machine code which is then fed to the memory and is executed.

We use the following java development environment for our application.



```
Gayatris-MacBook-Pro:~ gayatripise$ java -version
java version "1.8.0_162"
Java(TM) SE Runtime Environment (build 1.8.0_162-b12)
Java HotSpot(TM) 64-Bit Server VM (build 25.162-b12, mixed mode)
Gayatris-MacBook-Pro:~ gayatripise$
```

4. **Scope of Work and Implementation**

- The calculator is able to evaluate simple mathematical expressions such as addition, subtraction, multiplication and division of numbers.
- The calculator is able to evaluate mathematical expressions involving multiple operators such as: “3 + 4 * 2”.
- The calculator is able to evaluate mathematical expressions involving parenthesis such as: “2 * (6 + 3) – 1”.
- The calculator is able to evaluate mathematical expressions involving power expressions such as: “6 ^ 2”.
- The calculator is able to evaluate mathematical expressions following PEDMAS rule for priorities.
- The scope of work was fully completed.

5. **Command line instructions to compile and execute**

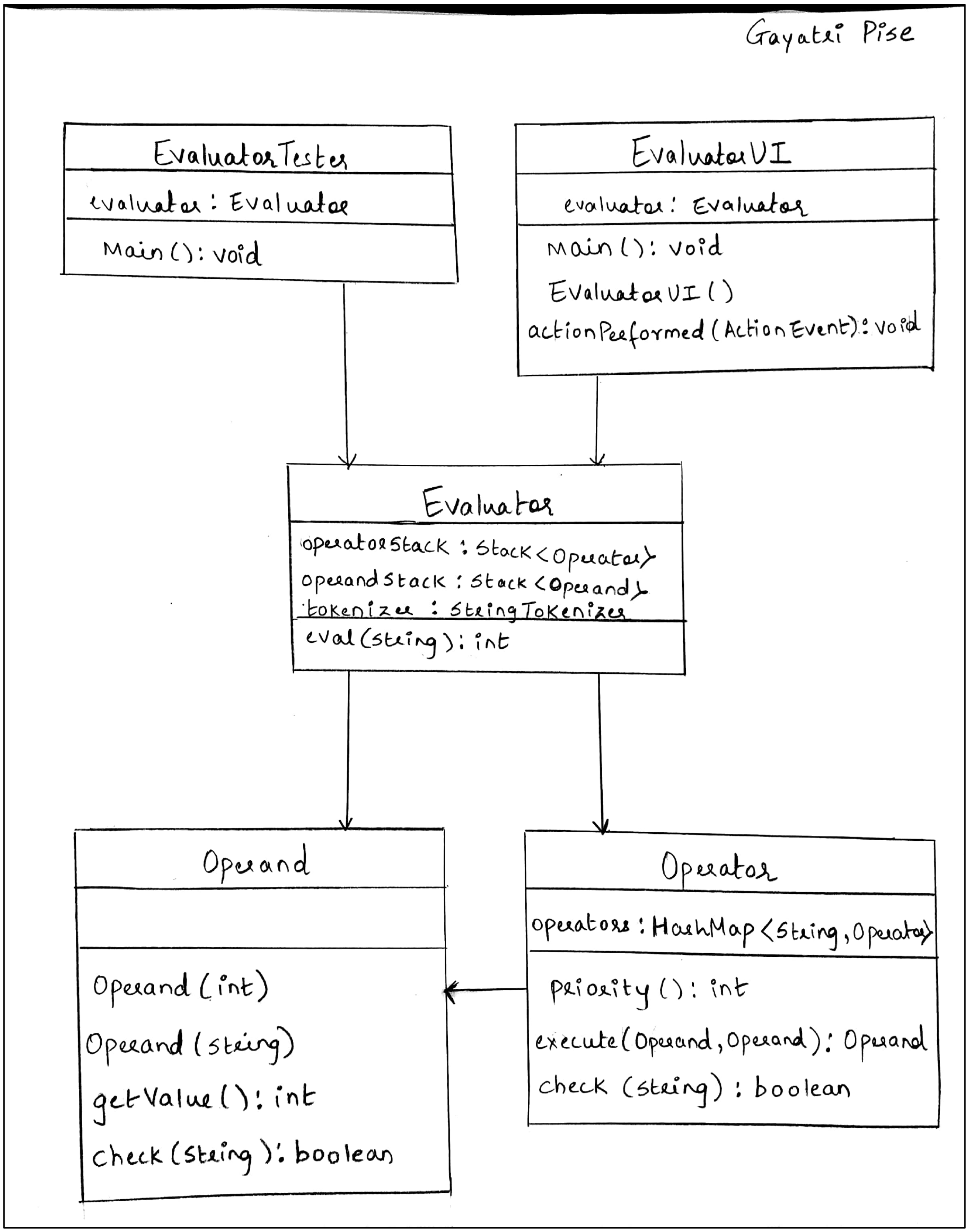
Compile	javac *.java
Execute	java EvaluatorUI **OR** java EvaluatorTester

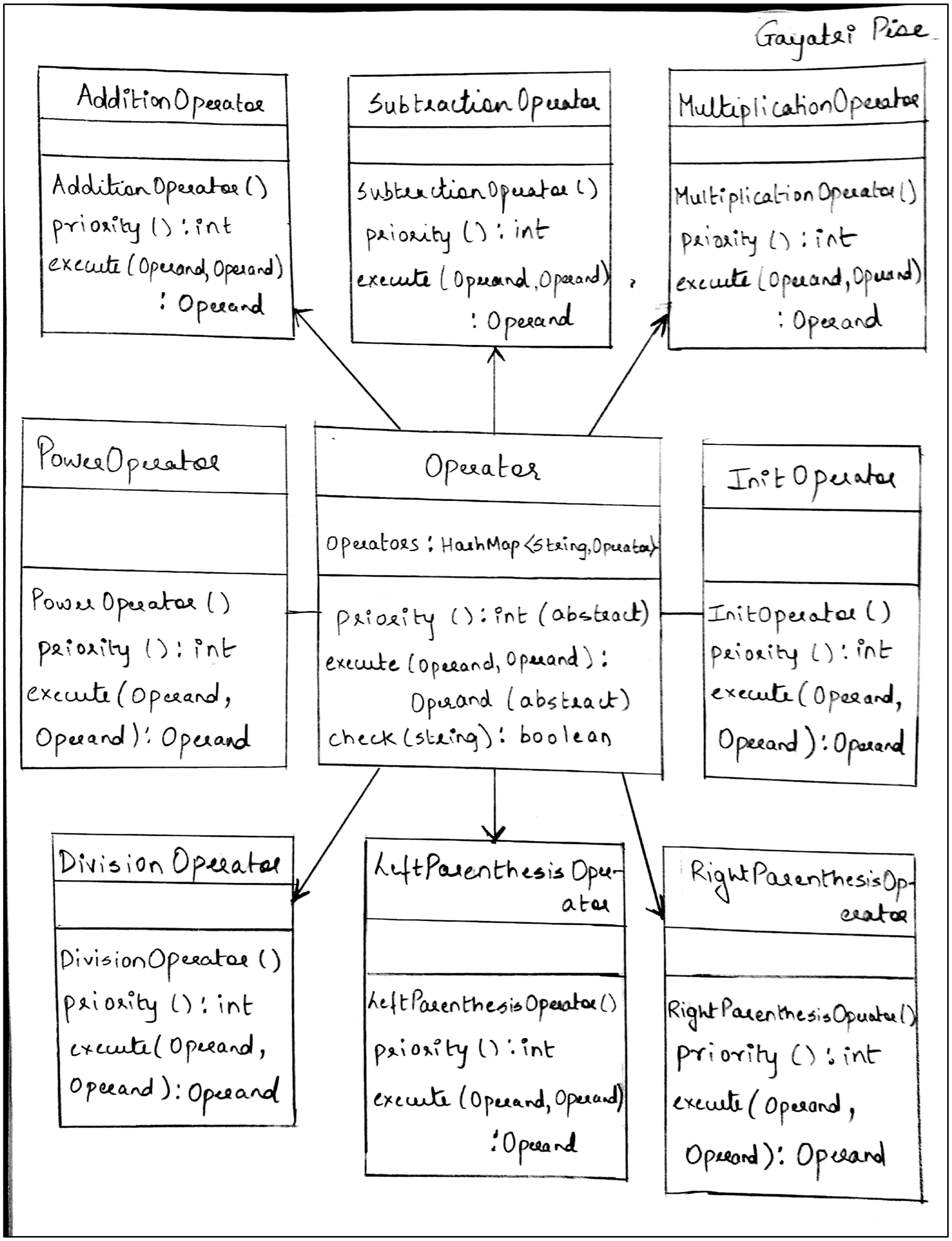
6. **Assumptions**

- User always enters valid positive input.
- User will not give input values in decimal number.
- User does not expect result in decimal number.
- Exceptions are not handled.
- While executing the EvaluatorTester in Terminal, mathematical expression is entered in double quotation marks in order to avoid ambiguity.
  Such as *"(3+4) * 2".*
- C (global clear) clears the entire calculation.
- CE (Clear Entry) – clears the current display.
- Mathematical expression does not start or end with an operator such as: "-5-2" or "5-2-".

7. **Class diagram with hierarchy**

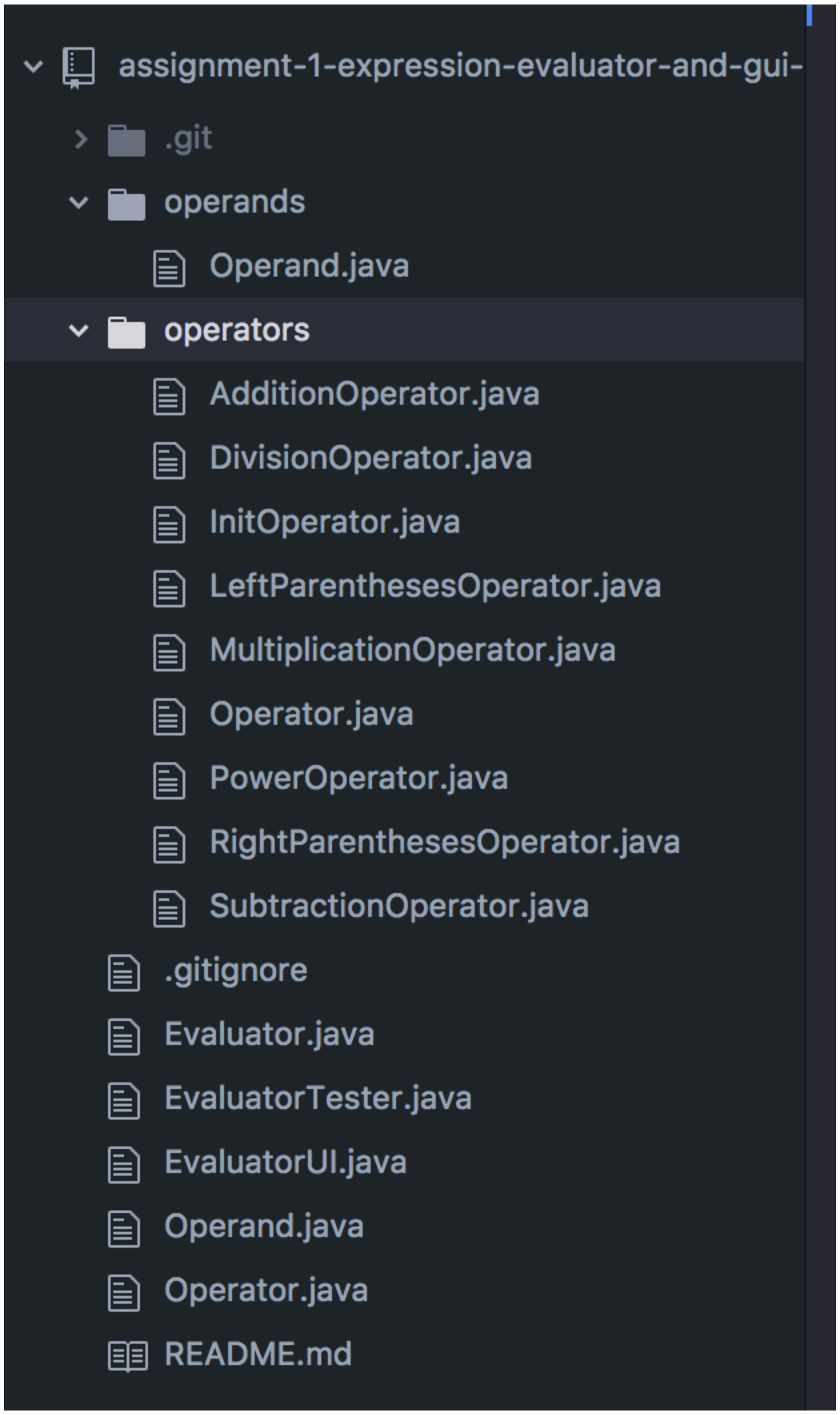
The class diagrams for this project is as below:


Gayatri Pise
EvaluatorTester
evaluator : Evaluator
Main(): void
EvaluatorUI
evaluator: Evaluator
Main(): void
EvaluatorUI()
actionPerformed(ActionEvent): void
Evaluator
operatorStack : Stack<Operator>
operandStack : Stack<Operand>
tokenizer : StringTokenizer
eval(string): int
Operand
Operand(int)
Operand(string)
getValue(): int
Check(String): boolean
Operator
operators: HashMap<String, Operator>
priority(): int
execute(Operand, Operand): Operand
check (string) : boolean


Gayatri Pise
AdditionOperator
AdditionOperator()
priority():int
execute(Operand, Operand) : Operand
SubtractionOperator
SubtractionOperator()
priority():int
execute(Operand, Operand) : Operand
MultiplicationOperator
MultiplicationOperator()
priority():int
execute(Operand, Operand) : Operand
PowerOperator
PowerOperator()
priority():int
execute(Operand, Operand): Operand
Operator
Operators : HashMap<String,Operator>
priority():int (abstract)
execute(Operand, Operand): Operand (abstract)
check(string): boolean
InitOperator
InitOperator()
priority():int
execute(Operand, Operand):Operand
DivisionOperator
DivisionOperator()
priority():int
execute(Operand, Operand): Operand
LeftParenthesisOperator
LeftParenthesisOperator()
priority():int
execute(Operand, Operand) :Operand
RightParenthesisOperator
RightParenthesisOperator()
priority():int
execute(Operand, Operand): Operand

8. Code Organization

The entire project is organized in the below file structure.



```
assignment-1-expression-evaluator-and-gui-
    .git
    operands
        Operand.java
    operators
        AdditionOperator.java
        DivisionOperator.java
        InitOperator.java
        LeftParenthesesOperator.java
        MultiplicationOperator.java
        Operator.java
        PowerOperator.java
        RightParenthesesOperator.java
        SubtractionOperator.java
    .gitignore
    Evaluator.java
    EvaluatorTester.java
    EvaluatorUI.java
    Operand.java
    Operator.java
    README.md
```

9. Conclusion

The above implementation evaluates the mathematical expression within the scope of the requirements. Implementation has been tested on terminal as well as the java based graphical user interface. The assumptions and limitations of this implementation have been described in this document.

The code is written following the concepts of object-oriented programming system (OOPS) such as inheritance, polymorphism, abstraction, etc. The project also involves interaction between Java Swing and Core Java.