SOEN 6011 - Software Engineering Processes

Summer 2019

Scientific Calculator- ETERNITY: FUNCTIONS

Project Report

Deliverable 2

Presented to

Instructor: PANKAJ KAMTHAN

By
Prashanthi Ramesh

July 30, 2019

Contents

Chapter 1

Changes

1.1 From Deliverable 1 to Deliverable 2

1. In deliverable 1, the requirement with ID= EF_REQ_6 states that $0^0 = Error$ which is incorrect. The requirement is corrected as $0^0 = 1$ and the unit test cases are created accordingly.

2. In deliverable 1, the requirement with ID= EF_REQ_8 states that zero raised to a real number is 1 which is incorrect. The requirement is corrected as $0^y = 0$ where y is a real number and the unit test cases are created accordingly.

3. In deliverable 2 there is an addition of a new algorithm to calculate the power of non-fractional number by the method of squaring. The fractional part uses the pseudo-code mentioned in deliverable 1 which is bit manipulation approximation method.

Chapter 2

Problem 4

This section presents an overview of the source code of the Eternity Function application and the practices followed during the development.

2.1 Programming Style of Source Code

We collaboratively as a team brainstormed and decided to follow the Google Java Style[2] programming style.

2.1.1 Google Java Style- Benefits

The benefits are as follows:

- Compatible with popular IDE (Integrated Development Environment) like Eclipse and IntelliJ IDEA.
- The style checker will automatically highlight any deviations from the code standards as we write code.

2.2 User Interface Design Pattern

2.2.1 Memento Design Pattern

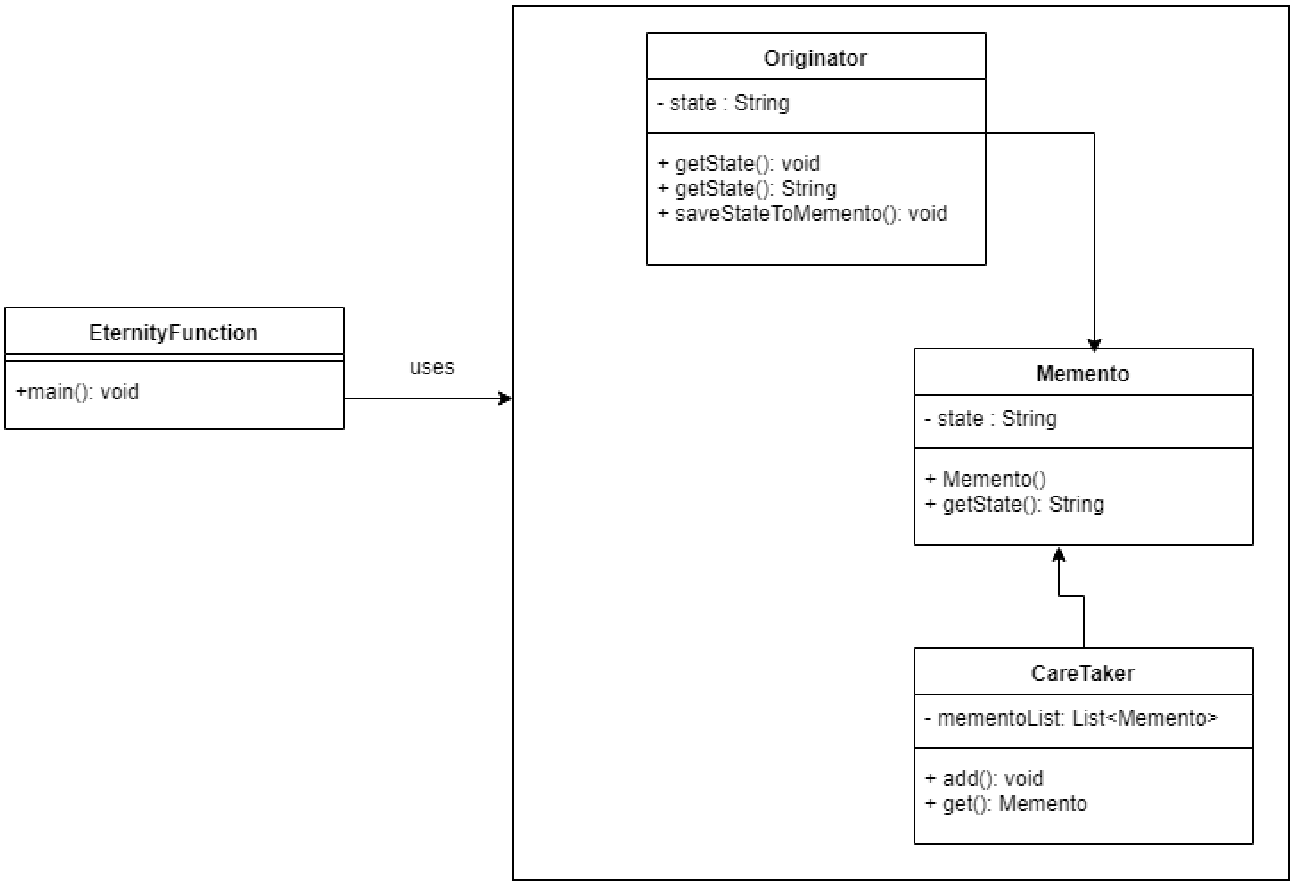


Figure 2.1: The Memento Design pattern UML diagram representation

The memento design pattern[4] is implemented in the Eternity Function to maintain a list of all previous operation that it has performed.

Hence, the user of the Eternity Function would be able to restore a previous calculation performed.

The use of memento design pattern prevents the EternityFunction object from becoming large and complex.

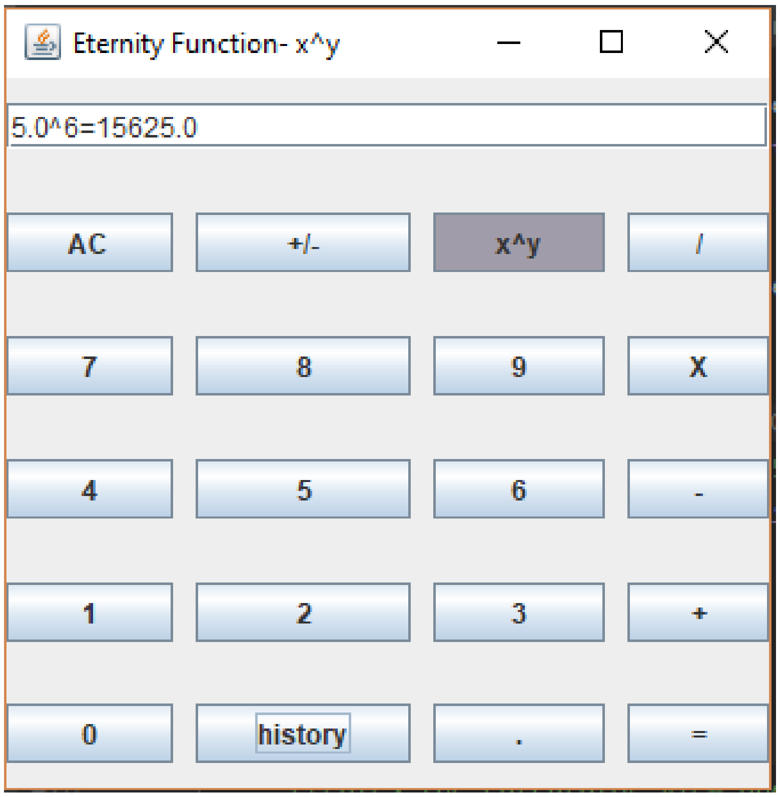


Figure 2.2: The history of operations of Eternity Function implemented using memento design pattern

2.3 Error Handling

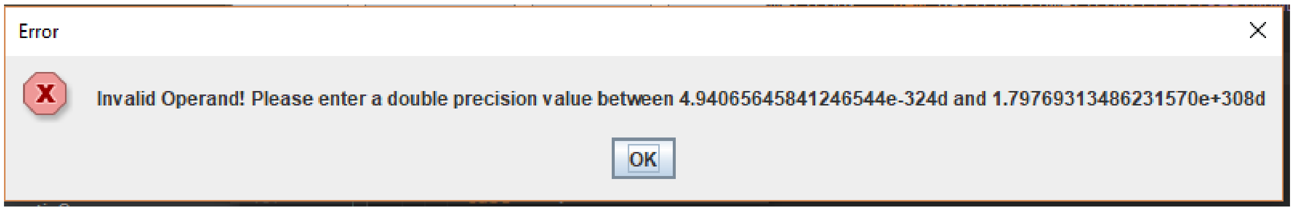
When an exception occurs in Eternity Function, we say that the exception is "thrown."

Double.parseDouble(textFieldValue) throws an exception of type NumberFormatException when the value entered in the text field is any character other than real numbers.

When an exception is thrown, it is possible to "catch" the exception and prevent it from crashing the program. This is done with a try..catch statement in Eternity Function.

In simplified form, the syntax for a try..catch statement can be:

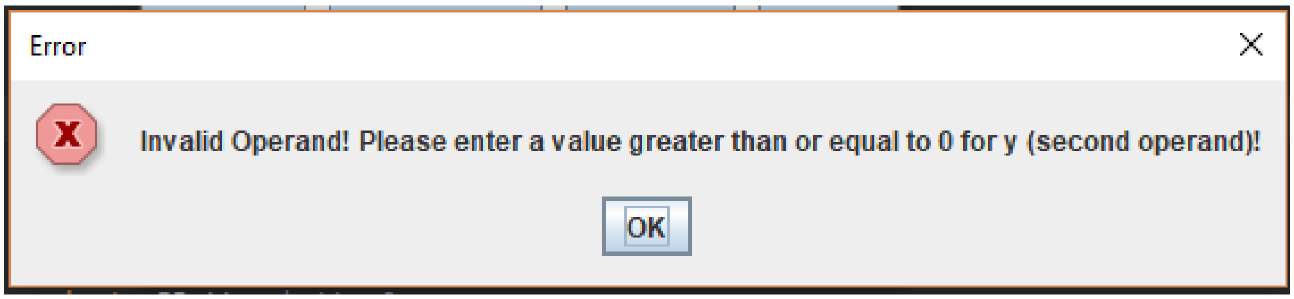
```
try {
  statements−1
   ...
  Double.parseDouble(textFieldValue)
   ...
}
catch ( NumberFormatException exception) {
    statements−2
}
```



Error

Invalid Operand! Please enter a double precision value between 4.94065645841246544e-324d and 1.79769313486231570e+308d

Figure 2.3: An Error message in the EternityFunction that aims to be helpful



Error

Invalid Operand! Please enter a value greater than or equal to 0 for y (second operand)!

Figure 2.4: Another instance of an error message in the EternityFunction that aims to be helpful

2.4 Debugger

IntelliJ IDEA build-in debugger is used for debugging the Eternity Function application.

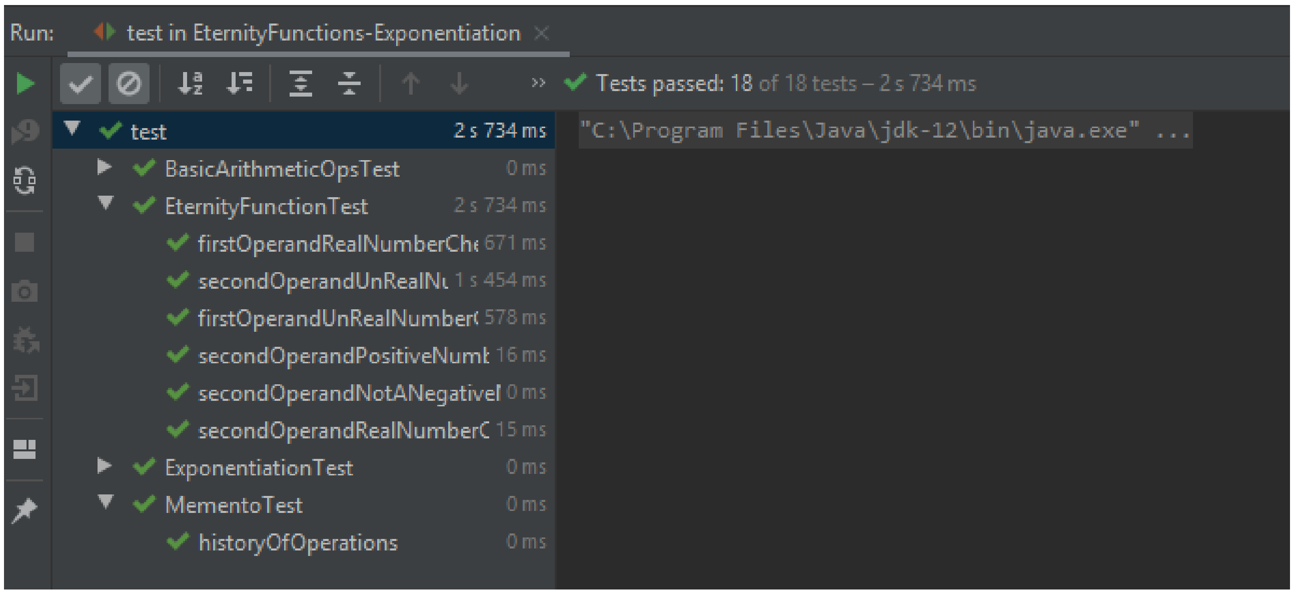
- The IntelliJ IDEA debugger offers a rich experience that helps us to easily debug anything from the simplest code to complex multi-threaded applications.
- Smart Step Into action allows to skip all intermediate invocations and proceed to the target method directly.
- IntelliJ allows creating breakpoints that pause the execution only if a user-defined condition is satisfied.
- IntelliJ allows to attach custom labels to JVM objects.
- An object can be marked when an application is stopped on a breakpoint and the target is reachable from stack frames.

Advantages	**Disadvantages**
If we don't have the source to specific code, IntelliJ IDEA will still decompile the class and show our steps in the decompiled source.	Code that was compiled without the debug flag cannot be debugged.
we can define in IntelliJ IDEA a breakpoint to stop before entry or exit from a specific method, even if the method itself was compiled without the debug flag	Line breakpoints are also not possible to define and hit.
The Remove Once Hit option is a very useful feature and its use is as a filter to triggering a breakpoint in a scenario where we're interested in a visit to a method, or a specific state in the code only after another state was reached.	Method Breakpoint and Field Watchpoint slow down code execution considerably.
If we stepped too far and want to go back up the stack to then re-execute the code, we can use the Drop Frame feature.	Drop Frame feature is also potentially dangerous: we must be aware that re-executing the code will execute the same instructions twice, and if those instructions modify state we might end up in a corrupted state.

2.5 Quality Attributes

2.5.1 Correct

- The Eternity Function application code is in agreement with specifications
- The correctness of the application is verified by unit testing using JUnit.
- Unit testing is used to ensure that the program quality standards are met and that the program is correct.



```
Run:   test in EternityFunctions-Exponentiation
Tests passed: 18 of 18 tests – 2 s 734 ms
✓ test                                  2 s 734 ms    "C:\Program Files\Java\jdk-12\bin\java.exe" ...
  ✓ BasicArithmeticOpsTest                    0 ms
  ✓ EternityFunctionTest               2 s 734 ms
    ✓ firstOperandRealNumberChe            671 ms
    ✓ secondOperandUnRealNu              1 s 454 ms
    ✓ firstOperandUnRealNumber             578 ms
    ✓ secondOperandPositiveNumb             16 ms
    ✓ secondOperandNotANegativel             0 ms
    ✓ secondOperandRealNumberC              15 ms
  ✓ ExponentiationTest                        0 ms
  ✓ MementoTest                               0 ms
    ✓ historyOfOperations                     0 ms
```

Figure 2.5: A success run of the JUnit test cases

2.5.2 Efficient

- The use of approximation algorithms[1] which are efficient algorithms makes Eternity Function efficient.
- Since the exponentiation routines of typical math libraries are rather slow, their replacement with a fast approximation can greatly reduce the overall computation time.
- An approximation is perfectly adequate for most neural computation purposes and can save much time.
- This approximation method is about 20 times as fast as Math.pow().

2.5.3 Maintainable

- The use of memento design provide easier maintainability and reusability, more understandable implementation and more flexible design.
- Writing comments and Javadoc increases understand-ability of code and thus makes it easier to be maintained.

2.5.4 Robust

- Eternity Function can cope with errors during execution.
- Robustness is achieved by using exception handlers.

2.5.5 Usable

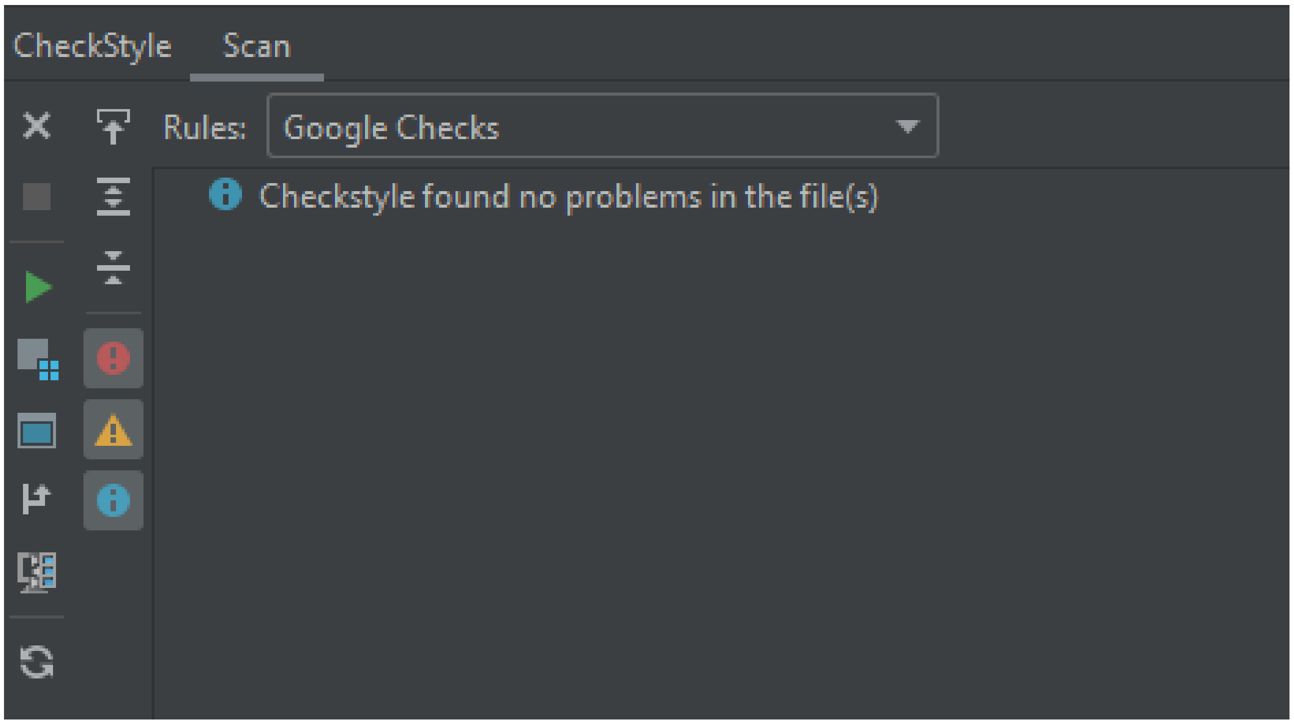
- The Eternity Function shall be used by members of public without training.
- The Eternity Function user interface design is clear and easy to understand

2.6 Quality Checker

To check the quality of the Eternity Functions's source code, Checkstyle[3] tool is used. This section presents an overview of Checkstyle tool.

2.6.1 Checkstyle

- Checkstyle is a development tool to help programmers write Java code that adheres to a coding standard.
- It automates the process of checking Java code to spare humans of this boring (but important) task.
- This makes it ideal for projects that want to enforce a coding standard.
- Checkstyle is highly configurable and can be made to support almost any coding standard.
- It can find class design problems, method design problems.
- It also has the ability to check code layout and formatting issues.



Google Checks

Checkstyle found no problems in the file(s)

Figure 2.6: The google checkstyle validates that the quality of the source code is good

Advantages	**Disadvantages**
Portable between IDEs. If you decide to use IntelliJ later, or you have a team using a variety of IDEs, you still have a way to enforce consistency.	It is not in-build with IDE's and needs to be configured manually.
better external tooling. It's much easier to integrate checkstyle with your external tools since it was really designed as a standalone framework.	With a bad configuration, you may check things twice or two opposite things i.e "Remove useless constructors" and "Always one constructor".
ability of creating your own rules. IDE defines a large set of styles, but checkstyle has more, and you can add your own custom rules.	Is not used for checking more complicate rules like during the design of your classes, or for more special problems like implementing correctly the clone function.
If we stepped too far and want to go back up the stack to then re-execute the code, we can use the Drop Frame feature.	Drop Frame feature is also potentially dangerous: we must be aware that re-executing the code will execute the same instructions twice, and if those instructions modify state we might end up in a corrupted state.

Chapter 3

Problem 6

3.1 Unit Test Cases

This section presents the unit test cases implemented using JUnit for Eternity Function which are traceable to requirements.

Test Case ID	EF_TESTCASE_1
Requirement ID	EF_REQ_4, EF_REQ_5
Action	The user enters input in the text field and clicks a button
Input(s)	first operand is 563248.2656E and operator is $\wedge$
Expected Output	isFirstNotRealNumber() returns true
Actual Output	isFirstNotRealNumber() returns true
Test Result	Success

Test Case ID	EF_TESTCASE_2
Requirement ID	EF_REQ_3
Action	The user enters input in the text field and clicks a button
Input(s)	first operand is 563248.2656 and operator is $\wedge$
Expected Output	isFirstNotRealNumber() returns false
Actual Output	isFirstNotRealNumber() returns false
Test Result	Success

Test Case ID	EF_TESTCASE_3
Requirement ID	EF_REQ_4
Action	The user enters input in the text field and clicks a button and enters another value in the text field
Input(s)	first operand is 563248.2656 and operator is $\wedge$ and second operand is 6526.25654
Expected Output	isSecondNotRealNumber() returns false
Actual Output	isSecondNotRealNumber() returns false
Test Result	Success

Test Case ID	EF_TESTCASE_4
Requirement ID	EF_REQ_3, EF_REQ_5
Action	The user enters input in the text field and clicks a button and enters another value in the text field
Input(s)	first operand is 563248.2656 and operator is $\wedge$ and second operand is 6526.25654E
Expected Output	isSecondNotRealNumber() returns true
Actual Output	isSecondNotRealNumber() returns true
Test Result	Success

Test Case ID	EF_TESTCASE_5
Requirement ID	EF_REQ_7
Action	The user enters input in the text field and clicks a button and enters another value in the text field
Inputs	first operand is 563248.2656 and operator is $\wedge$ and second operand is -543543.43432
Expected Output	isaNegativeValue() returns true
Actual Output	isaNegativeValue() returns true
Test Result	Success

Test Case ID	EF_TESTCASE_6
Requirement ID	EF_REQ_7
Action	The user enters input in the text field and clicks a button and enters another value in the text field
Input(s)	first operand is 563248.2656 and operator is $\wedge$ and second operand is -543543.43432
Expected Output	isaNegativeValue() returns false
Actual Output	isaNegativeValue() returns false
Test Result	Success

Test Case ID	EF_TESTCASE_7
Requirement ID	EF_REQ_1
Action	The user enters input in the text field and clicks a button and enters another value in the text field
Input(s)	first operand is 5.698468 and operator is $\wedge$ and second operand is 0
Expected Output	exponentiation.power() returns 1
Actual Output	exponentiation.power() returns 1
Test Result	Success

Test Case ID	EF_TESTCASE_8
Requirement ID	EF_REQ_2
Action	The user enters input in the text field and clicks a button and enters another value in the text field
Input(s)	first operand is 72645.3625892 and operator is $\wedge$ and second operand is 1
Expected Output	exponentiation.power() returns 72645.3625892
Actual Output	exponentiation.power() returns 72645.3625892
Test Result	Success

Test Case ID	EF_TESTCASE_9
Requirement ID	EF_REQ_6
Action	The user enters input in the text field and clicks a button and enters another value in the text field
Input(s)	first operand is 0 and operator is $\wedge$ and second operand is 0
Expected Output	exponentiation.power() returns 1
Actual Output	exponentiation.power() returns 1
Test Result	Success

Test Case ID	EF_TESTCASE_10
Requirement ID	EF_REQ_8
Action	The user enters input in the text field and clicks a button and enters another value in the text field
Input(s)	first operand is 0 and operator is $\wedge$ and second operand is 856954745.2545523
Expected Output	exponentiation.power() returns 0
Actual Output	exponentiation.power() returns 0
Test Result	Success

Test Case ID	EF_TESTCASE_11
Requirement ID	EF_REQ_9
Action	The user enters input in the text field and clicks a button and enters another value in the text field
Input(s)	first operand is -4 and operator is $\wedge$ and second operand is 7
Expected Output	exponentiation.power() returns -16384
Actual Output	exponentiation.power() returns -16384
Test Result	Success

Test Case ID	EF_TESTCASE_12
Requirement ID	EF_REQ_10
Action	The user enters input in the text field and clicks a button and enters another value in the text field
Input(s)	first operand is -4 and operator is $\wedge$ and second operand is 8
Expected Output	exponentiation.power() returns 65536
Actual Output	exponentiation.power() returns 65536
Test Result	Success

Test Case ID	EF_TESTCASE_13
Requirement ID	EF_REQ_15
Action	The user clicks the history button
Input(s)	History of operations
Expected Output	originator.getState() returns previous saved operation
Actual Output	originator.getState() returns previous saved operation
Test Result	Success

3.2 JUnit Standard Guidelines

The guidelines to create unit test cases of high-quality are:

- Did not use the test-case constructor to set up a test case
- Use of setUp() and tearDown() methods to initialize and release object resources
- Test case method names are meaningful
- Test cases are time-independent, small and fast
- Documented tests in javadoc

Appendix A

GitHub

A.1 Individual GitHub Link

https://github.com/PrashanthiRamesh/SOEN-6011-Project-Calculator/

A.2 Team GitHub Link

https://github.com/niravjdn/SOEN-6011-Project/

Bibliography

[1] Martin Ankerl. *Optimized pow() approximation for Java*. 2017. URL: https://martin.ankerl.com/2007/10/04/optimized-pow-approximation-for-java-and-c-c/ (visited on 10/01/2017).

[2] Joana Be. *Installing the google styleguide settings in intellij and eclipse*. 2019. URL: https://github.com/HPI-Information-Systems/Metanome/wiki/Installing-the-google-styleguide-settings-in-intellij-and-eclipse (visited on 05/12/2019).

[3] Jayanga Kaushalya. *How to configure CheckStyle and Findbugs plugins to IntelliJ IDEA for WSO2 products*. 2016. URL: https://medium.com/@jayanga/how-to-configure-checkstyle-and-findbugs-plugins-to-intellij-idea-for-wso2-products-c5f4bbe9673a (visited on 11/01/2016).

[4] TutorialsPoint. *Design Patterns - Memento Pattern*. 2019. URL: https://www.tutorialspoint.com/design_pattern/memento_pattern (visited on 07/06/2019).