

Postfix Calculator

Testing Document

EECS 2311 Group 2 v2.0

Team 2

Chauhan, Yash

Jin, Hui

Khan, Akbar

Martintsov, Vladimir

Nguyen, Andrew

Podimov, Michael

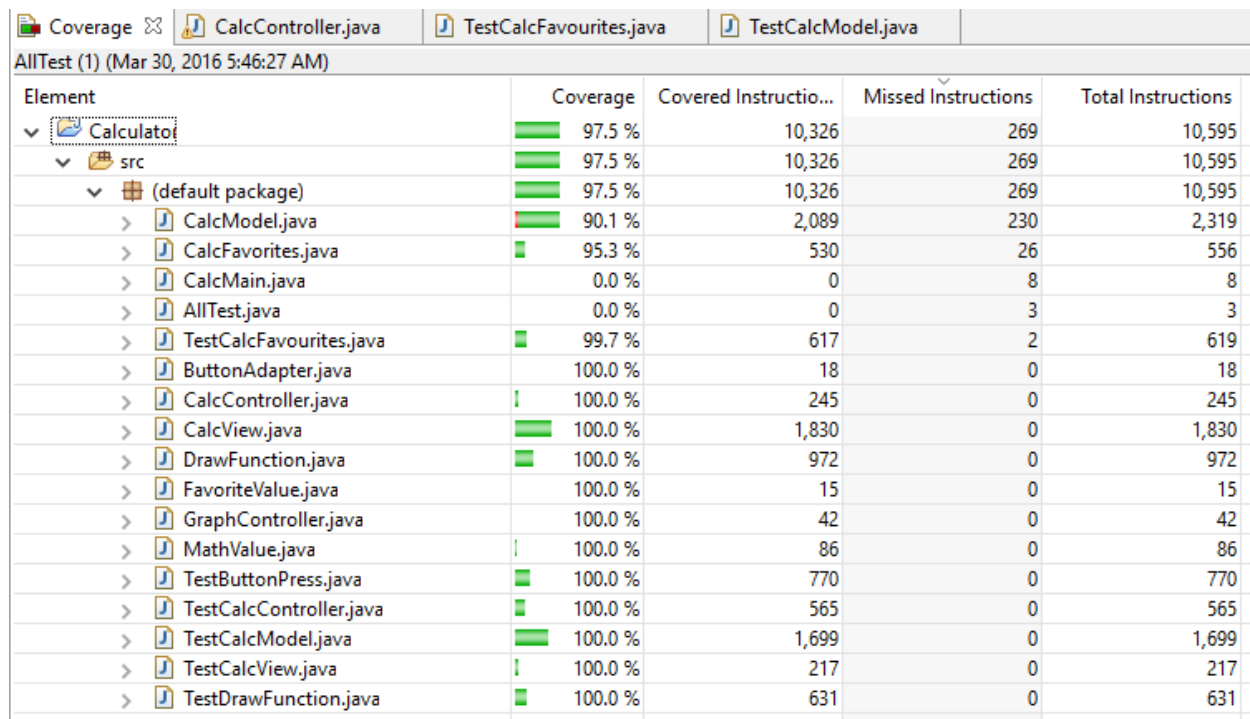
TABLE OF CONTENTS

1. Introduction	3
2. Testing the Model	3
2.1 testNumericButton() and testEnter()	3
2.2 testSum() and testSubtract()	3
2.3 testMultiply() and testDivide()	4
2.4 testPi()	4
2.5 testSine() and testCosine()	5
2.6 testFactorial()	5
2.7 testChangeSign()	6
2.8 testClear()	6
2.9 testUndo()	6
3. Testing the Controller	7
3.1 testUpdateView()	7
4. Testing the View	7
4.1 Testing Methods Within View and Design	7
4.2 Simulating Button Presses	9

1. Introduction

Testing was a critical part in the process of improving the code for the project as well as fixing a number of bugs that were apparent within the code. The test cases were split up into six classes -

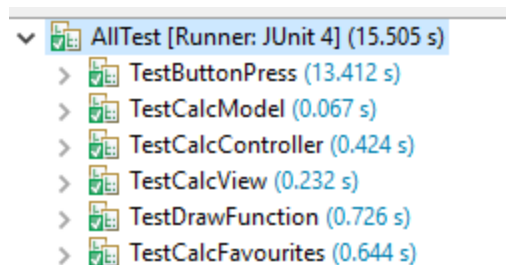
TestCalcModel, TestCalcController, TestCalcView, TestButtonPress, TestDrawFunction, and TestCalcFavorites. Each class contains a number of test cases with inputs that cover a wide variety of test cases sufficient enough to represent all inputs. The total coverage for all of the code was 97.5%. Out of 10,329 instructions only 269 were missed. The test coverage for all classes can be seen in fig. 1.



Element	Coverage	Covered Instructio...	Missed Instructions	Total Instructions
Calculator	97.5 %	10,326	269	10,595
src	97.5 %	10,326	269	10,595
(default package)	97.5 %	10,326	269	10,595
CalcModel.java	90.1 %	2,089	230	2,319
CalcFavorites.java	95.3 %	530	26	556
CalcMain.java	0.0 %	0	8	8
AllTest.java	0.0 %	0	3	3
TestCalcFavourites.java	99.7 %	617	2	619
ButtonAdapter.java	100.0 %	18	0	18
CalcController.java	100.0 %	245	0	245
CalcView.java	100.0 %	1,830	0	1,830
DrawFunction.java	100.0 %	972	0	972
FavoriteValue.java	100.0 %	15	0	15
GraphController.java	100.0 %	42	0	42
MathValue.java	100.0 %	86	0	86
TestButtonPress.java	100.0 %	770	0	770
TestCalcController.java	100.0 %	565	0	565
TestCalcModel.java	100.0 %	1,699	0	1,699
TestCalcView.java	100.0 %	217	0	217
TestDrawFunction.java	100.0 %	631	0	631

Figure 1: Total test coverage for all classes.

All of the test cases passed. This can be seen in fig. 2.



AllTest [Runner: JUnit 4] (15.505 s)
TestButtonPress (13.412 s)
TestCalcModel (0.067 s)
TestCalcController (0.424 s)
TestCalcView (0.232 s)
TestDrawFunction (0.726 s)
TestCalcFavourites (0.644 s)

Figure 2: Test classes as seen in Junit.

2. Testing the Model

The Model was tested using the following test cases. Each test case assesses the corresponding method in the Model class and includes all sufficient test coverages. The methods need to yield an appropriate output in order for them to pass the tests. In this case, all the tests passed. The total coverage for the

class **CalcModel()** was 90.1%. This class had the lowest coverage out of all the other classes. This class also had the largest amount of instruction – 1559 more than the second largest. This made it difficult to traverse every line of code, however, the following test cases are definitely sufficient.

2.1 testNumericButton() and testEnter()

The first method to be tested was **numericButton()** (used to enter single numbers or decimal points to the value). One key aspect to testing this method was to make sure every parameter was covered. This was done by calling the method through numbers 0 to 9 and decimal point, and using **getInputValue()** to test if the resulting value is correct. Another aspect was to make sure any extra zeros or decimals were removed in the input. Removing extra zeroes and decimals prevents the number from appearing in an improper format.

testNumericButton()

Input: numericButton() used with 0123456.789.

Expected output: Value equals 123456.789

For the method **enter()**, its basic function needed to be tested as well as using it on different types of inputs. The basic function was tested by using **enter()** on an input value and checking if the resulting number in history matches by using **getHistoryValue()**. Then, **enter()** was tested on multiple input values in order to see if the history would display each value separated by a comma. Within these multiple input values, a negative number and a decimal number was used to make sure the history would show both these types of numbers. This test also covers cases where the first or last character in the input value is a decimal. When displayed onto the history, zeros should be added to display the number in proper format.

testEnter()

Input: 1 [enter] 2 [change sign] [enter] 3 . 5 [enter] . 5 [enter] 6 . [enter]

Expected output: History displays 1, -2, 3.5, 0.5, 6.0

2.2 testSum() and testSubtract()

Testing the methods **sum()** and **subtract()** required that the resulting sum or difference equaled the expected value and that the history displayed the equation in infix notation. This was done by adding the numbers 1 and -2 and subtracting the numbers 5.5 and 2.5. These tests are important because getting a correct value is crucial in calculators and the infix notation is part of the requirements. Testing two numbers for each has sufficient coverage for all cases because it signifies that all other operands will result in a correct calculation.

testSum()

Input: 1 -2 +

Expected output: History displays 1 + -2 =, Value is -1

testSubtract()

Input: 5.5 2.5 -

Expected output: History displays 5.5 – 2.5 =, Value is 3,

In addition to the test cases above, there were also tests to check a feature of the calculator. When an operator is pressed without any input, the program uses two zeros as the operands. Also, when there is only one input for a binary operation, the program uses the single input and a zero as its operands. The following test case displays this property for sum, however, it is performed on all operators

testSum()

Input: +

Expected Output: History displays 0 + 0, Value is 0

Input: 1 +

Expected Output: History displays 1 + 0, Values is 1

2.3 testMultiply() and testDivide()

Testing **multiply()** consisted of getting a correct output value and displaying the history in infix notation with correct precedence. Two numbers are multiplied together and the product is compared to the expected value. If this passes, it confirms that **multiply()** works properly for all other operands. Thus, the test coverage is sufficient. Then, an equation involving addition, subtraction, and multiplication is input and is tested for infix notation as well as if multiplication has precedence over addition and subtraction.

testMultiply()

Input: 5 -5 ×

Expected output: History displays 5 × -5 =, Value equals -25

Input: 2 3 + 2 1 - ×

Expected output: History displays (2 + 3) × (2 - 1) =, Value equals 5

The test for **divide()** involved three cases. The first was the basic test of getting the correct value. As stated before, the purpose is to make sure **divide()** gives the correct value for all inputs and testing any two inputs is sufficient for this. The second case tests for precedence over **multiply()** (this precedence is situational i.e a number divided by a product). An equation involving **multiply()** and **divide()** is input in proper infix notation and precedence is checked by looking at the brackets. This is sufficient enough for precedence as it follows from the previous **testMultiply()** that if **divide()** has precedence over **multiply()**, it must have precedence over **sum()** and **subtract()**. The final case involves dividing a number by zero, which the expected output is a **MATH ERROR**. These three cases cover all major situations for divide.

testDivide()

Input: 7 2 ÷

Expected output: History displays 7 ÷ 2 =, Value equals 3.5

Input: 1 2 × 1 2 × ÷

Expected output: History displays (1 × 2) ÷ (1 × 2) =, Value equals 1

Input: 7 0 ÷

Expected output: Value equals MATH ERROR

2.4 testPi()

The method **pi()** was tested using two cases. The first compared the value the input to the expected value. After **pi()** is called, **getInputValue()** and **getHistoryValue()** are verified. This ensures that **pi()** gives the correct value for calculations and displays the correct symbol in the history. The second case tests a feature that lies within **pi()**. A number is input in the value and then **pi()** is used resulting in a multiplication of the two values. This feature creates easier input of equations involving **pi()**.

testPi()

Input: π

Expected output: History displays π , Value equals 3.1415926536

Input: 2π

Expected output: History displays $2 \times \pi =$, Value equals 6.2831853072

2.5 testSine() and testCosine()

For both of these test cases, **sine()** or **cosine()** were tested to see if they gave the correct values. This was done by first inputting a value into **inputValue**, then calling **sine()** or **cosine()**, then verifying if the resulting value is correct using **getInputValue()**. Just one **inputValue** is sufficient as one pass proves that all other values will give the correct output. This test also validates that the operator is shown in prefix notation with its operand. The history is checked and the test passes if sin or cos appears before its operand i.e. sin goes before the (x) in sin(x) where x is any operand.

testSine()

Input: $\pi \div 2 \div \sin$

Expected output: History displays $\sin(\pi \div 2) =$, Value displays 1

testCosine()

Input: $\pi \cos$

Expected output: History displays $\cos(\pi) =$, Value displays -1

2.6 testFactorial()

The test case **factorial()** consists of four components. The first part makes certain that the method returns the correct value. After **factorial()** is used, the value in the field **inputValue** is verified to be correct. The next two parts deal with cases of a non-integer input and a negative input. For both of these cases, the **inputValue** displays **MATH ERROR**. Although it may be possible to implement an algorithm to find these values, it is not required so a **MATH ERROR** will suffice. The last test uses factorial on a number over 170. The resulting number exceeds the max value so the result is infinity. These four cases are sufficient in all cases because it takes into consideration every situation factorial can be put in.

testFactorial()

Input: 5!

Expected output: History displays 5! =, Value equals 120

Input: 5.8!

Expected output: Value equals MATH ERROR

Input: -2!

Expected output: Value equals MATH ERROR

Input: 171!

Expected output: Input value is ∞

2.7 testChangeSign()

Testing of **changeSign()** resulted in four major situations. The first part verified that the operation changed the sign of the value. This was done by inspecting the value in **getInputValue()** after **changeSign()** was used on a value. The second part tested the capability of continuing an input after **changeSign()** was used. When the user presses **changeSign()**, the user should not be forced to enter the current value. The third part tested to see if an equation involving **changeSign()** gives the correct value. This ensures that the negative value is treated as such in calculations. The final situation tests to see if the negative sign is displayed properly outside of brackets. These four tests are all that is needed for coverage because they test all possible situations that change sign can experience.

testChangeSign()

Input: 6 [change sign] [enter]

Output: Value equals -6

Input: 6 [change sign] . 5

Output: Value equals -6.5

Input: -9 2 + -3 4 - ×

Output: History displays $(-9 + 2) \times (-3 - 4) =$, Value equals 49

Input: 6 [change sign] [enter] 7 + [change sign]

Output: History displays $- (-6 + 7) =$, Value equals -1

2.8 testClear()

This test is pretty straightforward. An input value is entered and **clear()** is called. The test checks to see if the **inputValue** and **historyValue** revert back to the initial conditions.

testClear()

Input: 8 [enter] [clear]

Expected output: History displays Start New Calculation, Value equals 0

2.9 testUndo()

When thinking about the testing of **Undo()**, there are four situations that are crucial to test. The first part tests to verify that using **undo()** while in the middle of inputs deletes the most recent input until the **inputValue** is empty while allowing the user to continue his inputs. The number 5 will be entered followed by the input of 6.3. When **undo()** is called three times, the **inputValue** should be empty and the **historyValue** untouched. The second part validates that using **undo()** after a number is entered

deletes the last value entered in the **historyValue**. The third segment tests that **undo()** works with all binary operators and the fourth part tests that **undo()** works with all unary operators.

`testUndo()`

Input: 5 [enter] 6 . 3 [undo] [undo] [undo] 2

Expected output: History displays 5, 2, Value displays nothing

Input: 5 [enter] 6 [enter] 7 [enter] 8 [enter] [undo] [undo]

Expected output: History displays 5, 6

Input: 1 2 3 4 5 + - × ÷ [undo] [undo] [undo] [undo]

Expected output: History displays 1, 2, 3, 4, 5

Input: 5! sin cos [undo] [undo] [undo]

Expected output: Value equals 5

2.10 `testX()`, `testGetFunction()`, `testGetEquation()`

The variable `x` is used for graphing, so the most important thing to test is to make sure the variable acts correctly when it is used within an equation. It is crucial to check that the function outputs the correct values for all of `x`. First, the variable `x` was tested to make sure that it displayed properly on the display.

`testX()`

Input: `x`

Expected Output: History displays `x`

When testing functions, it was concluded that they lie within four major categories –linear, sinusoidal, exponential, and discontinuous. All functions are one of these categories or a mixture of some so these test cases are sufficient. Each test case checks the `y` output by asserting on values from the output array created from `getFunction()`. `GetEquation()` is also used to display the string version of the function. Let `array` be the output array for each case. `Array` contains 1000 elements with 500 being the zero value of `x` and every 10 elements equals 1 value of `x`.

`testGetFunction()`

Input: `x [getFunction()]`

Expected Output: `array[500] = 0.0, array[0] = -50.0, array[750] = 25.0`

Equation: `y = x`

Note: `array[562]` is rounded because there is no exact point for 0.0

Input: `x sin [getFunction()]`

Expected Output: `array[500] = 0.0, (rounded)array[562] = 0.0`

Equation: `y = sin(x)`

Input: `x ! [getFunction()]`

Expected Output: `array[500] = 1.0, array[501] = NaN, array[510] = 1.0, array[530] = 6.0`

Equation: `y = x!`

Input: $x \times x$

Expected Output: `array[500] = 0.0, array[490] = 1.0, array [510] = 1.0`

Equation: $y = x \times x$

3. Testing the Controller

The tests for the Controller class are very similar to the test in the Model class since the methods basically call the methods within the Model. Because of this, all the test cases shown in part 2 can be applied to the controller. After running the tests, the Controller shows a coverage of 100%. All the test cases are not shown because they are identical to the model tests in part 2. The only method within Controller worth displaying is **updateView()**, which as its name suggests, updates the fields **display** and **historyDisplay** within the View with the matching **inputValue** and **historyValue**.

3.1 testUpdateView()

This test ensures that the fields **display** and **historyDisplay** from the view are set to its correct **inputValue** and **historyValue** from the model. This is done by using the controller to input a sequence of equations and comparing the values in **getDisplayText()** and **getHistoryText()** to its expected value. The test uses every operator within its equation to make sure they all update properly.

`testUpdateView()`

Input: `1 -2 + 3 4 - x π 2 \div sin π cos \times \div 0! \times`

Expected Output: History displays `((1 + -2) \times (3 - 4)) \div (sin($\pi \div 2$) \times cos(π)) \times 0! =,`
display shows `-1`

4. Testing the View

The view was testing by splitting it up into two categories. The first category is about the methods within the view and the design. The second part simulates button presses and tests to see if they perform the correct operation. Overall, the tests cases within this test class cover 100 % of the View class.

4.1 Testing Methods Within View and Design

Table 1: testSetDisplayText()

No	Action	Expected Result	Actual Result
1.1	Set displaytext as null	Display text field will be null	true
1.2	Set displaytext as "abc"	Display text field will show abc	true
1.3	Set displaytext as null	Display text field will be null	true

The function **setDisplayText()** changes the text to be shown on the value. This is tested by making the display null, then a string, then back to null. The purpose is to show the view can display any string as well as an empty one.

Table 2: testSetHistoryText()

No	Action	Expected Result	Actual Result
2.1	Set historytext as null	History text field will be null	true
2.2	Set historytext as "abc"	History text field will show abc	true
2.3	Set historytext as null	History text field will be null	true

This test, similar to **testSetDispalyText()**, makes the history display null, then a string, and then null. This shows how the history is able to display any string as well as an empty one.

Table 3: testBetterViewDesign()

No	Action	Expected Result	Actual Result
3.1	Check panel2's background color	Color will be black	true
3.2	Check historyDisplay's background color	Color will be black	true
3.3	Check historyDisplay's foreground color	Color will be white	true
3.4	Check display's background color	Color will be black	true
3.5	Check display's foreground color	Color will be white	true
3.6	Check display's border	Border will be null	true
3.7	Check historyDisplay's border	Border will be null	true
3.8	Check mainpanel's background color	Color will be (24,29,33)	true
3.9	Check panel's background color	Color will be (24,29,33)	true
3.10	Check numbers's background color	Color will be (24,29,33)	true
3.11	Check operators's background color	Color will be (24,29,33)	true
3.12	Check buttons's background color	Color will be (24,29,33)	true

These test cases check the color of the backgrounds as well as the foregrounds of the panel. Once the colors have been ensured to be correct, it is safe to say that the design of the panel is correct with the design.

Table 4: testButtonColor

No	Action	Expected Result	Actual Result
4.1	Check benter's background color	Color will be (218.100.2)	true
4.2	Check benter's foreground color	Color will be (250.251.255)	true
4.3	Check benter is focusable	Focusable will be false	true
4.4	Check benter's border	Benter's border will be null	true

These test cases ensure that the colors of the buttons as well as their borders are correct. This is to make sure that the buttons coincide with our design.

4.2 Simulating Button Presses

The view was also tested by simulating button presses on the panel. The purpose is to ensure that each button behaves as expected. This is done by using the built in method **doClick()** to simulate button presses within the view and **getDisplayText()** and **getHistoryText()** to verify the output. Each test case passed and they will be explained in the following:

The button enter was tested first, and a single press had the expected output of **getDisplayText()** to be 0 and **getHistoryText()** to be 0.

The buttons 0 through 9 had expected outputs of **getDisplayText()** and **getHistoryText()** corresponding to the respective value input.

The button dot was tested by expecting a decimal within its input.

The button clear was tested by expecting the fields to be reset to its initial values.

The buttons add, subtract, divide, and multiply were tested by using the method on a two values and expecting a certain sum, difference, quotient, or product.

The button pi was tested by making sure the button gave the correct value of 3.1415926536.

The buttons sin, cos, factorial, and change sign were testing by using the method on an input value and expecting the resultant output value.

The button undo was testing by simulating an undo press after a number has been pressed and expecting that number to be deleted from the input value.

The above button simulations are for the main view panel, however, the buttons in the graph view and the favorites view were simulated as well by using the method **doClick()**.

For the graph view, the button **Back** was tested to make sure that **view.frame.getContentPane()** agreed with the main view panel. The button **Open Favorites** was also tested with **getContentPane()** and it was checked to agree with the favorites panel. The button **Add to Favorites** was tested by ensuring that the current equation was added to the favorites list. More detail on how this is done will be explained in the following sections.

For the favorites view, the button **Back** was tested by making sure the current view was the graph view. The button **Graph** was tested to make sure the view was switched to the graph view as well as the correct graph being displayed. The button **Delete** was tested by making sure the correct equation was deleted from the favorites list.

4. Testing the Graph

In order to test the graph, the class that needed to be tested was **DrawFunction**. This class creates the graph view as well as its components such as buttons, the axis, and of course, the graph. The following test cases cover 100 % of **DrawFunction**. The way the tests check to see if the graph is correct is by checking the color of pixels on the image. This was done with **BufferedImage** since it has a **getRGB()** method. An example of how a **DrawFunction** is displayed with **BufferedImage** is shown in fig. 2.

```
bi = new BufferedImage(view.frame.getContentPane().getWidth(),
    view.frame.getContentPane().getHeight(), BufferedImage.TYPE_INT_RGB);
g = bi.createGraphics();
view.frame.getContentPane().paint(g);
```

Figure 2: Snippet of code how the panel is converted into BufferedImage

In order to maximize coverage, the four cases of functions – linear, sinusoidal, exponential, and discontinuous – were tested. Other methods that were created in the class **DrawFunction** were created as well.

4.1 Test Axis

The axis in the graph view consists of two straight lines running across the middle of the height and down the middle of the width. They were tested by check four pixels – the centre pixels along each border. The color expected was 97, 107, 116. The purpose of this test is to ensure that the axis is properly proportioned so that they are displayed correctly for every function.

Note: the size of the contentPane is 634 x 691

testAxis()

Input: Color of pixels at points (317, 0), (317, 690), (0, 345), (633, 345)

Expected Output: RGB color of pixels are 97, 107, 116

4.2 Test Linear Function

A linear function was tested by inputting the equation $y = x$ and checking the color of pixels where the graph is expected to appear. The color of the function is expected to be **Color.CYAN**. The that was tested appeared where $x = 1$ so the pixel should lie where $y = 1$. This point on the panel is (367, 296).

testLinearFunction()

Input: Color of pixel at point (367, 296).

Expected Output: Color.CYAN

4.3 Test Sinusoidal Function

The sinusoidal function to be tested was the function $\sin(x)$. The point where the pixel was checked lies on the point $(\pi/2, 1)$ on the graph, and on the point (332, 334) on the contentPane.

testSinusoidalFunction()

Input: Color of pixel at point (332, 334).

Expected Output: Color.CYAN

4.4 Test Quadratic Function