



# Testing

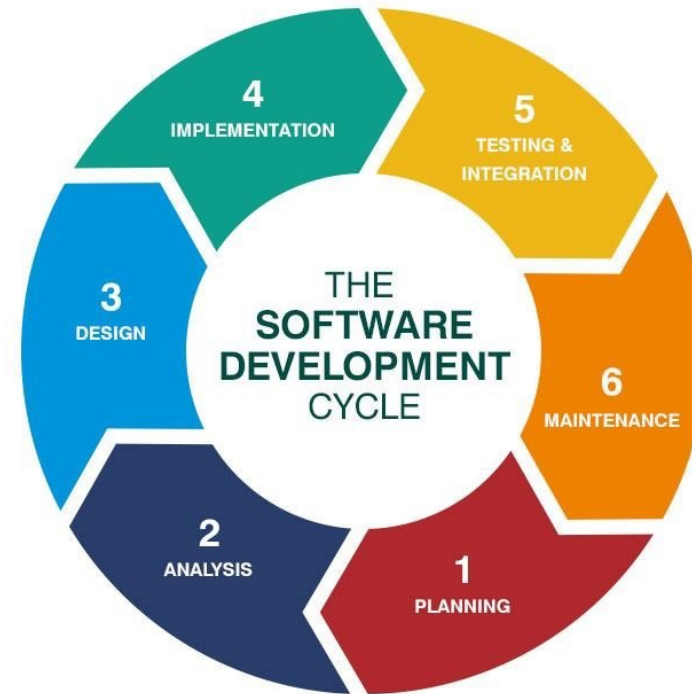
---

The coding of a program covers only a small part of the whole life of any software product.

One of the important phases of the software development cycle is **software testing**.

**The aim of testing is two-fold:**

- To investigate whether the system meets its given specifications.
- To expose defects before the system is delivered.

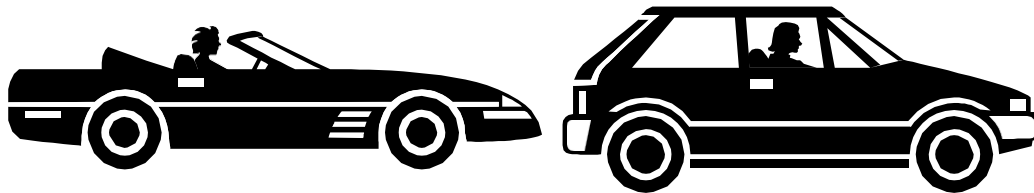


Source: datarob.com

# e.g. Before taking delivery of a car

---

- To investigate whether the car meets its given specifications.



Check it's the one I ordered

- To expose defects before the car is delivered.



Take it for a test drive



# Black-Box and White-Box Testing

---

For **black box testing**, we test the software from a user's point of view.

- In black box testing, we perform testing **without seeing the internal system code**.

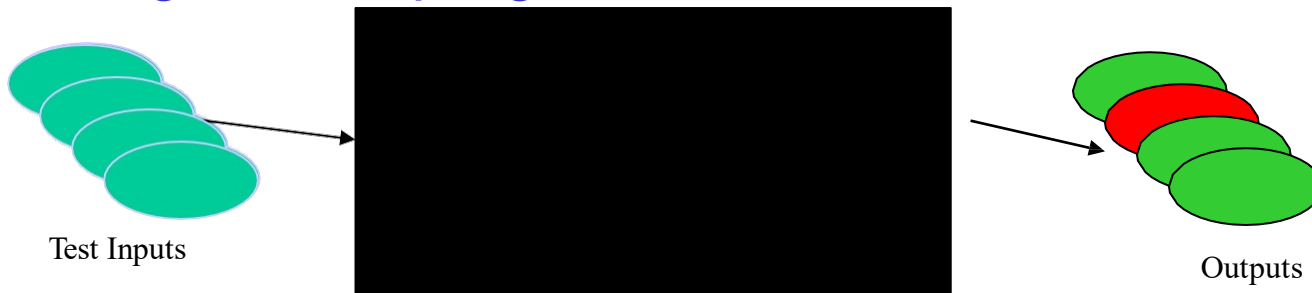
For **white box testing**, we see and test the system code

- It is used by both the developers as well as testers and helps them to understand which line of code is actually executed and which is not.

# Black box testing

---

- ***Functional* or *Black-box* testing.**
  - The main focus of Black Box Testing is on the **functionality of the system as a whole**. The term '**Behavioural Testing**' is also used for Black Box Testing.
  - Tests are derived from the program specification.
  - The tests are devised to explore various inputs and corresponding outputs, **without any knowledge of the workings of the program itself**



# Black Box Testing Example

---

**Specification:** A a program which  
**doubles** an input integer.

For black box testing, you can try  
some integer inputs without  
accessing to the code:

1, 3, 75, 100000, 9999999999

-1, -5, -68, -150000, -1000000

0.000001, 0.6666, 2.8, 100.9, 4.0e+6

-0.000001, -3.7, -10.002, -2.0e+4

0, 0.0

**input values are divided into different classes :**

Positive integers

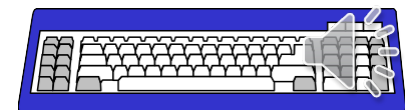
Negative integers

Positive real numbers

Negative real numbers

0

Multiply.java



# Black Box Testing

---

Test Case	Expected Output
1	2
5	10
50	100
100	200
500	1000

Unfortunately it would hard to find a fault in the program. (All it proves is that it works **only** for **these cases**)

We need to cover the majority of test cases so that most of the bugs will get discovered by the Black-Box method.



# White box testing

---

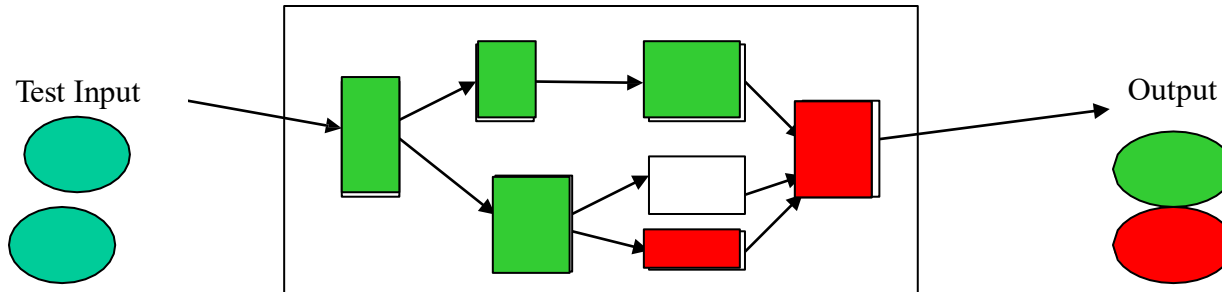
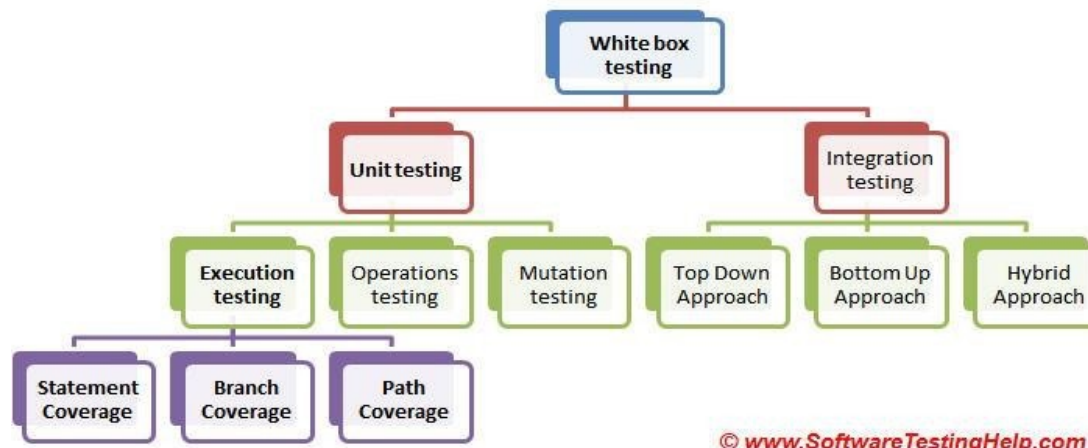
- **Structural** or **White-box** testing.
  - Tests are derived from the knowledge of the program code.
  - **To ensure:**
    - That all independent paths within a module have been exercised at least once.
    - All logical decisions verified on their true and false values.
    - All loops executed at their boundaries and within their operational bounds internal data structures validity.
  - **To discover the following types of bugs:**
    - Logical error (functions, conditions or controls that are out of the program)
    - The design errors due to difference between logical flow of the program and the actual implementation
    - Typographical errors and syntax checking





# White box testing types

Types of White Box Testing



# Main White Box Testing Techniques

---

**Statement Coverage:** which validates whether each line of the code is executed at least once.

**Branch Coverage:** In case of an “IF statement”, there will be two test conditions:

- One to validate the true branch and,
- Other to validate the false branch.

**Path Coverage:** which ensures that all the paths of the program are traversed at least once. This technique is useful for testing the complex programs.

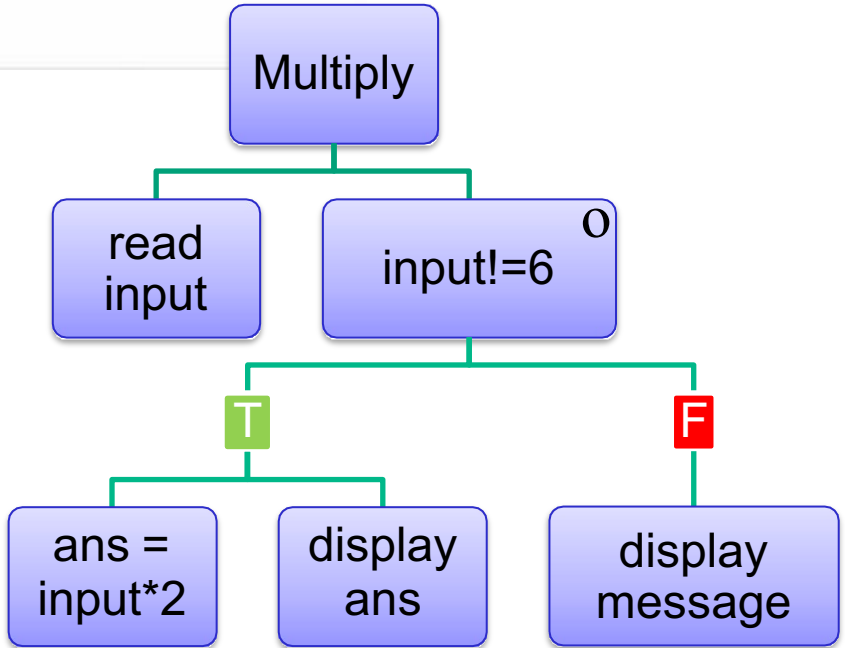
```

1  import javax.swing.*;
2
3
4  class Multiply {
5
6      public static void main(String[] args) {
7          String inputStr = JOptionPane.showInputDialog("Input?");
8          int input = Integer.parseInt(inputStr);
9          if (input != 6) {
10             int answer = input * 2;
11             System.out.println("the answer is " + answer);
12         } else {
13             System.out.println("Have a nice day");
14         }
15     }
16 }

```

Now we have access to the code we can see that there is a selection controlled by whether the input is 6 or not

White box testing will concentrate on making sure the **input set** is exercising every possible route through the structure i.e. **both** branches of the selection



# White – box testing 1

Let's see an example of path coverage:

```
public void testMethod(char a, char b) {  
    if (a < 'N') {  
        if( b < 'Y') {  
            System.out.println("path 1");  
        } else {  
            System.out.println("path 2");  
        }  
    } else {  
        if( b < 'D') {  
            System.out.println("path 3");  
        } else {  
            System.out.println("path 4");  
        }  
    }  
}
```



## White – box testing 2

---

The table below shows some possible values for  $a$  and  $b$  (with the smallest and largest upper case letters chosen) and the corresponding *path*:

$a$	$b$	Path
'A'	'A'	path 1
'A'	'Z'	path 2
'Z'	'A'	path 3
'Z'	'Z'	path 4

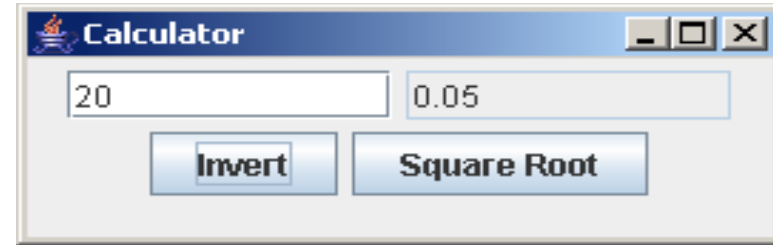
But the number of test cases required to test the method thoroughly can become quite large if values between A and Z were chosen, or lower case letters, or digit characters, etc.



# Example: Calculator

## Specification

Write a program to  
take in a number and either  
invert it or take its square root



(e.g. if the number is 16  
invert  $16 = 1/16 = 0.0625$  and square root  $16 = 4$ )

The display shows the answer when 20 is the  
input and the invert button is pressed giving  
 $1/20 = 0.05$



# Black box testing of Calculator

---

- Without knowing anything about the coding of the program we can see if it meets its specification
- We will try a number of inputs in different categories with a range of sizes of:

**Positive Integers**

**Negative Integers**

**Positive real numbers**

**Negative real numbers**

**0**

**Something that's not a number!**



# Black box testing of calculator

## typical testing values

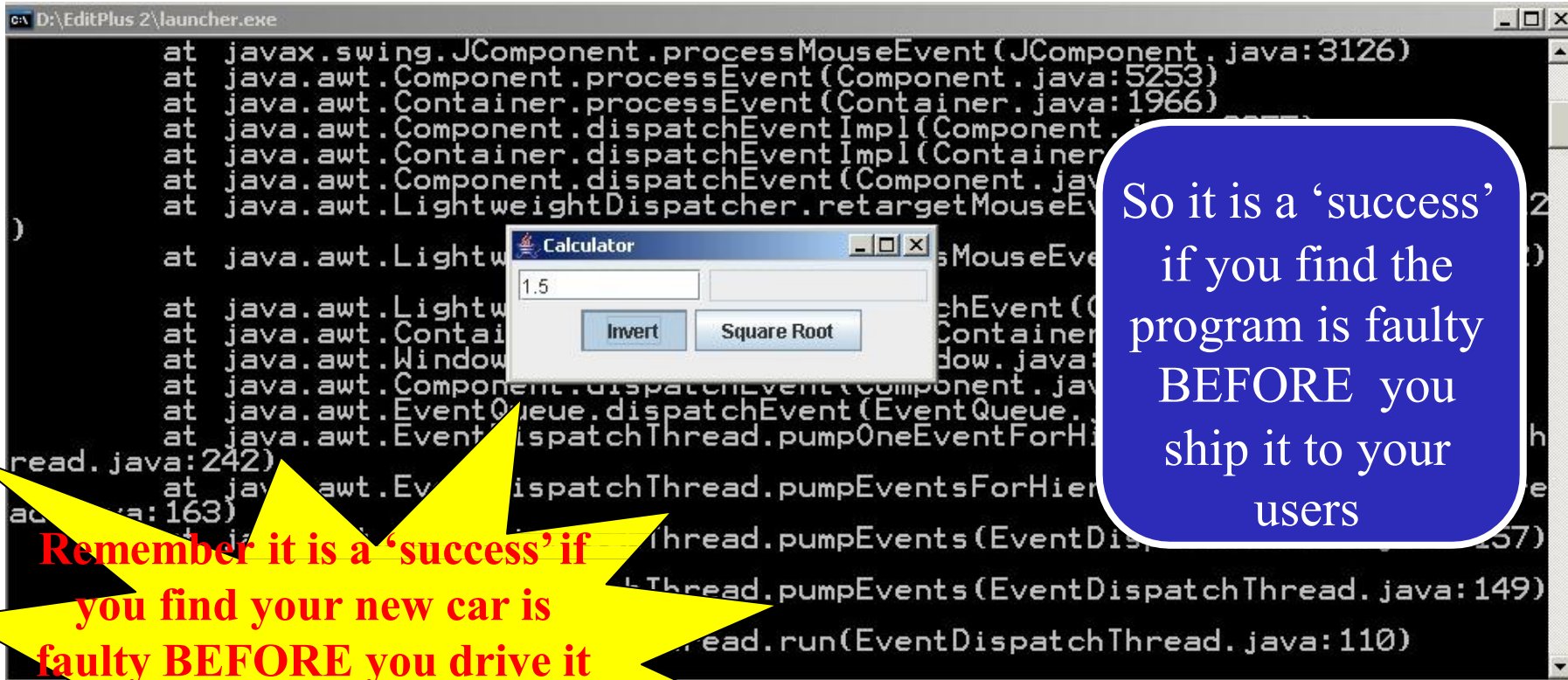
---

- 1, 2, 59, 10000, 99999999999
  - -1, -3, -34, -120000, -1000000
  - 0.00001, 0.3333, 1.5, 10.7, 5.0e+6
  - -2.5, -10.003, -0.00001(-1.0e-5)
  - 0, 0.0
  - fish
- 
- For each of these you should have calculated the expected output so you know whether your program is working correctly to the specification





# Some successful tests from Black box testing of Calculator

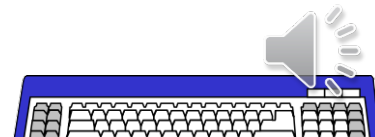


```
at javax.swing.JComponent.processMouseEvent(JComponent.java:3126)
at java.awt.Component.processEvent(Component.java:5253)
at java.awt.Container.processEvent(Container.java:1966)
at java.awt.Component.dispatchEventImpl(Component.java:3377)
at java.awt.Container.dispatchEventImpl(Container.java:2000)
at java.awt.Component.dispatchEvent(Component.java:3200)
at java.awt.LightweightDispatcher.retargetMouseEvent(Container.java:4534)
at java.awt.LightweightDispatcher.dispatchEvent(LightweightDispatcher.java:861)
at java.awt.LightweightDispatcher.dispatchEvent(LightweightDispatcher.java:854)
at java.awt.Window.dispatchEventImpl(Window.java:2704)
at java.awt.Component.dispatchEvent(Component.java:3200)
at java.awt.EventQueue.dispatchEvent(EventQueue.java:709)
at java.awt.EventDispatchThread.pumpOneEventForHierarchy(EventDispatchThread.java:242)
at java.awt.EventDispatchThread.pumpEventsForHierarchy(EventDispatchThread.java:163)
at java.awt.EventDispatchThread.pumpEvents(EventDispatchThread.java:157)
at java.awt.EventDispatchThread.pumpEvents(EventDispatchThread.java:149)
at java.awt.EventDispatchThread.run(EventDispatchThread.java:110)
```

So it is a 'success' if you find the program is faulty BEFORE you ship it to your users

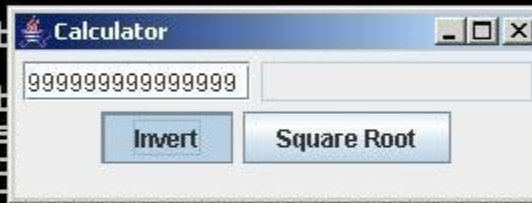
Remember it is a 'success' if you find your new car is faulty BEFORE you drive it home

Calculator.java

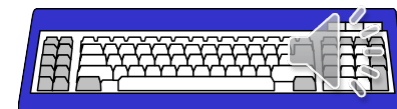


C:\D:\EditPlus 2\launcher.exe

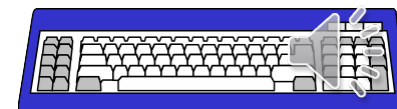
```
at javax.swing.JComponent.processMouseEvent(JComponent.java:3126)
at java.awt.Component.processEvent(Component.java:5253)
at java.awt.Container.processEvent(Container.java:1966)
at java.awt.Component.dispatchEventImpl(Component.java:3955)
at java.awt.Container.dispatchEventImpl(Container.java:2024)
at java.awt.Component.dispatchEvent(Component.java:3803)
at java.awt.LightweightDispatcher.retargetMouseEvent(Container.java:4212)
at java.awt.LightweightDispatcher.dispatchEvent(Container.java:3892)
at java.awt.LightweightDispatcher.dispatchEvent(Container.java:3822)
at java.awt.Container.dispatchEventImpl(Container.java:2010)
at java.awt.Window.dispatchEventImpl(Window.java:1774)
at java.awt.Component.dispatchEvent(Component.java:3803)
at java.awt.EventQueue.dispatchEvent(EventQueue.java:463)
at java.awt.EventQueue.dispatchEvent(EventQueue.java:463)
at java.awt.EventDispatchThread.pumpOneEventForHierarchy(EventDispatchThread.java:242)
at java.awt.EventDispatchThread.pumpEventsForHierarchy(EventDispatchThread.java:163)
at java.awt.EventDispatchThread.pumpEvents(EventDispatchThread.java:157)
at java.awt.EventDispatchThread.pumpEvents(EventDispatchThread.java:149)
at java.awt.EventDispatchThread.run(EventDispatchThread.java:110)
```



Calculator.java



```
D:\EditPlus 2\launcher.exe
at javax.swing.JComponent.processMouseEvent(JComponent.java:3126)
at java.awt.Component.processEvent(Component.java:5253)
at java.awt.Container.processEvent(Container.java:1966)
at java.awt.Component.dispatchEventImpl(Component.java:3955)
at java.awt.Container.dispatchEventImpl(Container.java:2024)
at java.awt.Component.dispatchEvent(Component.java:3803)
at java.awt.LightweightDispatcher.retargetMouseEvent(Container.java:4212)
at java.awt.LightweightDispatcher.dispatchEvent(Container.java:3892)
at java.awt.LightweightDispatcher.dispatchEvent(Container.java:3822)
at java.awt.Container.dispatchEventImpl(Container.java:2010)
at java.awt.Window.dispatchEventImpl(Window.java:1774)
at java.awt.Component.dispatchEvent(Component.java:3803)
at java.awt.EventQueue.dispatchEvent(EventQueue.java:463)
at java.awt.EventDispatchThread.pumpOneEventForHierarchy(EventDispatchThread.java:242)
at java.awt.EventDispatchThread.pumpEventsForHierarchy(EventDispatchThread.java:163)
at java.awt.EventDispatchThread.pumpEvents(EventDispatchThread.java:157)
at java.awt.EventDispatchThread.pumpEvents(EventDispatchThread.java:149)
at java.awt.EventDispatchThread.run(EventDispatchThread.java:110)
```



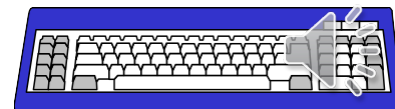
# Findings of Testing

---

Program cannot deal with large numbers e.g.  
9999999999

Program doesn't handle real numbers properly  
e.g. 1.5

Program has a run-time error for bad input e.g.  
fish rather than recognising it as non numerical



# Error handling in black box testing of Calculator

---



- The black box testing showed that it did recognise two standard errors
  - Trying to invert 0 (so you are trying to divide 1 by 0 and division by 0 is not allowed)
  - Trying to find the square root of a negative value





# Calculator program (moving to white box)

GUI set up

actions triggered  
when any button  
is clicked

actions if the  
source of click is  
the inv button  
(Invert)

actions if the  
source of click is  
the sq button  
(Square Root)

3 methods used by  
the if statements

```
import java.awt.*;
import javax.swing.*;
import java.awt.event.*;

public class Calculator extends JFrame implements ActionListener {
    JTextField input = new JTextField(10);
    JTextField result = new JTextField(12);
    JButton inv = new JButton("Invert");
    JButton sq = new JButton("Square Root");

    public static void main(String[] args) {
        new Calculator();
    }

    public Calculator() {
        setLayout(new FlowLayout());
        setSize(270, 100);
        setTitle("Calculator");
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        add(input);
        add(result); result.setEditable(false);
        add(inv); inv.addActionListener(this);
        add(sq); sq.addActionListener(this);
        setVisible(true);
    }

    public void actionPerformed(ActionEvent e) {
        int inp = Integer.parseInt(input.getText());
        if (e.getSource() == inv) {
            if (inp == 0) complain(); // can't invert 0
            else
                invertCalc(inp);
        }
        if (e.getSource() == sq) {
            if (inp < 0) complain(); // can't find sqrt of -ve number
            else
                sqrtCalc(inp);
        }
    }

    void invertCalc(int i) {
        result.setText("" + 1.0 / i); // "" + 1.0/i will convert 1.0/i to a string
    }

    void sqrtCalc(int i) {
        result.setText("" + Math.sqrt(i));
    }

    void complain() {
        result.setText("Error!");
    }
}
```



# Calculator program

```
public void actionPerformed(ActionEvent e)
{
    int inp = Integer.parseInt(input.getText());
    if (e.getSource() == inv)
    {
        if (inp == 0) complain(); // can't invert 0
        else
            invertCalc(inp);
    }
    if (e.getSource() == sq)
    {
        if (inp < 0) complain(); // can't find sqrt of -ve number
        else
            sqrtCalc(inp);
    }
}
```

if the input is not 0 then use the  
invertCalc method

if the input is not negative  
then use the sqrtCalc method

```
void invertCalc(int i)
{
    result.setText(" " + 1.0 / i); // "" + 1.0/i will convert 1.0/i to a string
}

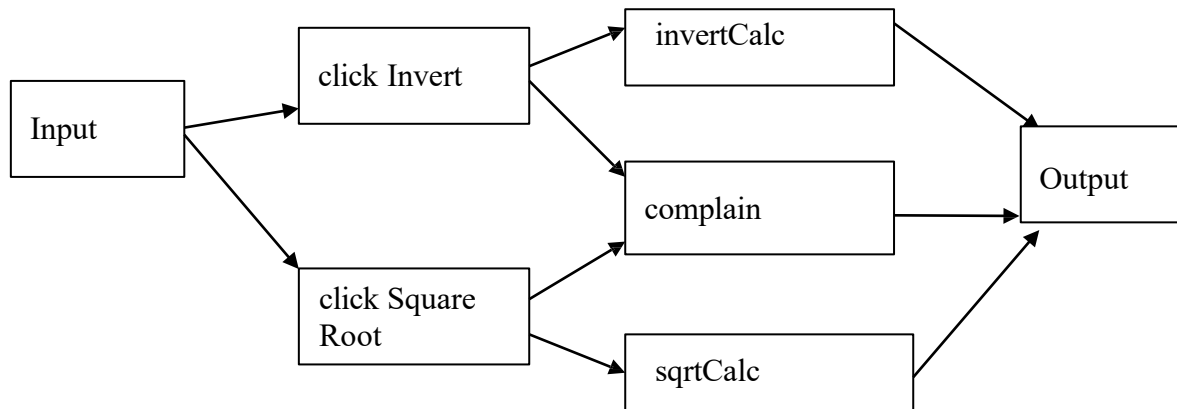
void sqrtCalc(int i)
{
    result.setText(" " + Math.sqrt(i));
}

void complain()
{
    result.setText("Error!");
}
```



# White box testing of Calculator

To white-box test the same program, we must consider the different paths that execution can follow (by analyzing the program's structure). These are:



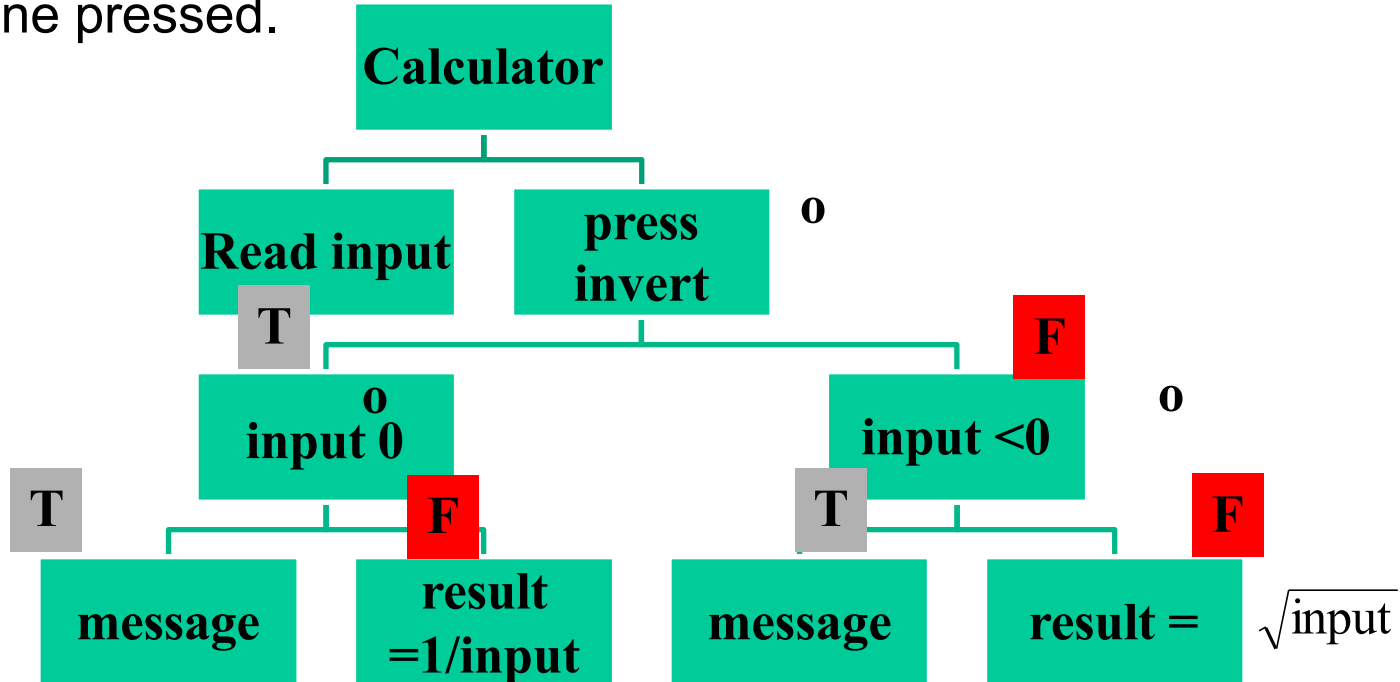
To help to analyze the structure of the program we need a systematic and organised way of writing this down in a diagram





# White box testing of calculator

An alternative way of displaying this structure is to use standard algorithm structure charts. Note that here the square root button is not mentioned explicitly but only as the other option if the invert is not the one pressed.



(You could have the 2 buttons as separate options with a dummy false = do nothing on each)



# Testing all paths

- Invert zero (should complain)

input 0

- Invert Non zero numbers

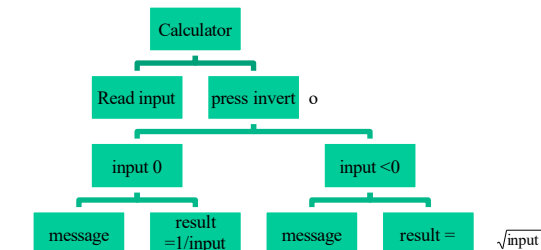
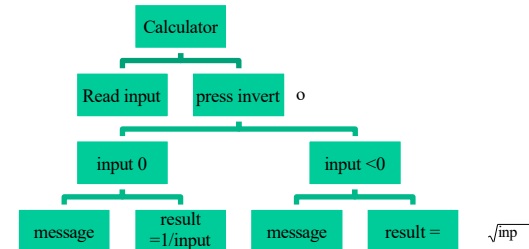
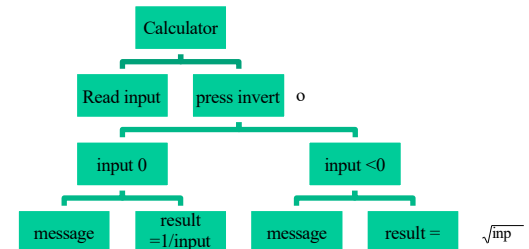
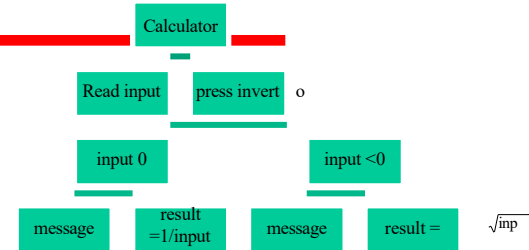
input 0.5

- Calculate the square root of negative numbers (should complain)

input -4.0

- Calculate the square root of non-negative numbers

input 4.0



# Test table

---

There are the 4 different paths through the program

At the very least we document the input values that will take us down each path and work out what the correct output should be

We can track the results in a table to make sure we have covered them all

Input	Click	Method	Expected Output	Actual Output
10	Invert	invertCalc()	0.1	
0	Invert	complain()	Error!	
-4	Square Root	complain()	Error!	
4	Square Root	sqrtCalc()	2	

# White box testing of calculator

---

This still doesn't check for bad input like 'fish' so we need to add validation checks on our input.

In general a good GUI should prevent the user from inputting values that are inadmissible and force the user to complete all required fields

It is sometimes difficult to think of all the wrong ways a user may treat your software so try and test it on your worst enemy! They will be trying to crash it and will help you find the faults

We should make it clear in the specification that it is to be designed to deal with integers only



# White box testing (continued)

---

In general, it is **impossible** to exhaustively test a program.

But you should at least:

- Make sure all statements are tested at least once!
- Make sure each **branch** (*if* or loop or try ... catch) is tested.



# White box testing (continued)

---

- For *try ... catch*, we need to check the branch for each catch, plus one (normal) path where no exception happens.

- The following loop should be tested at least twice:

```
while( ( line = inFile.readLine() ) != null) {  
    inputTextArea.append(line + "\n");  
}
```

- **Test cases:** a non-empty file containing and an empty file

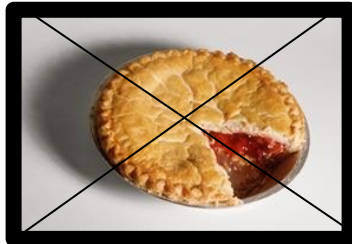




# Complex choices c&&p



- Test whether you are having coffee **AND** a piece of pie?



is there coffee?  $c = F$   
so  $c \& \& p = F$

is there pie?  $c = T$   $p = F$   
so  $c \& \& p = F$

is there both?  $c = p = T$   
so  $c \& \& p = T$



# Summary of decisions testing (continued)

---

- Bear in mind that a test such as *if (a && b) ...* has three branches –
  - *a = false*,
  - *a = true* but *b = false*,
  - *a = b = true*.
- Similarly *if (a || b) ...* has three branches –
  - *a = true*,
  - *a = false* but *b = true*,
  - *a = b = false*
- A *switch ...* has as many branches as it has *cases*, plus one if it doesn't have a *default*.





# CarPark Example

---

- For Black Box you will try a large number of combinations of money and number of hours chosen to see if the change is working correctly
- For White Box you will try to make sure every part of the code gets exercised
- We will expect to see this type of testing strategy used in **YOUR COURSEWORK**

# Test Plan

---

White box testing should always be carried out by the programmer as she/he knows the code!

You should draw up a *test plan* specifying:

- **Inputs**
- **Expected outputs**
- **Actual outputs**

The test plan must at a minimum carry out white box testing as described. s

**Any bugs should be highlighted!**



# Testing Harness

---

The term '**Harness**' in general refers to a 'Tool' for controlling something.

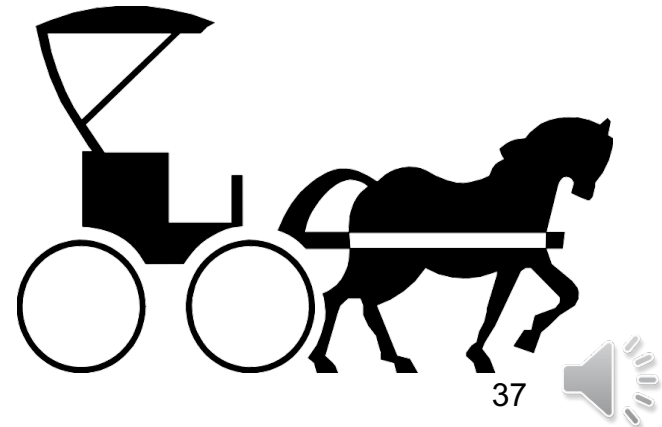
In the automate testing, **Test Harness** refers to the framework and the software systems that contain the test scripts, parameters to run these scripts, gather test results, compare them (if necessary) and monitor the results.

Test Harness helps automate the testing procedures and thus increase the productivity of the system through automation.

**Test Harness Tools:** HP ALM, JUnit, and NUnit

Having made a cart how do we test that it will run OK?

Put a horse in a harness and let it pull



# Summary

---

- We have looked at:
  - Black box testing
  - White box testing
  - Test Plan
  - Testing Harness

