UNIVERSITY of HOUSTON

DEPARTMENT OF COMPUTER SCIENCE

COSC 4351 Fall 2023

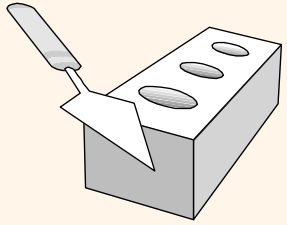
Software Engineering

M & W 4 to 5:30 PM

Prof. Victoria Hilford

PLEASE TURN your webcam ON

NO CHATTING during LECTURE



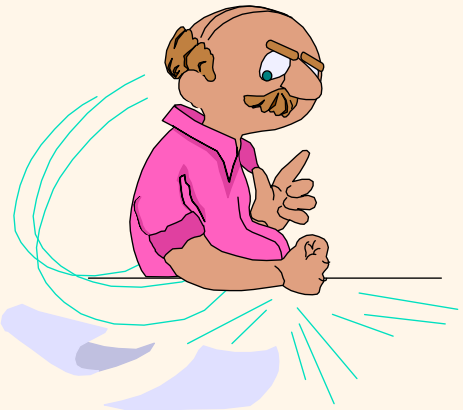
COSC 4351

4 to 5:30

PLEASE

LOG IN

CANVAS



Youyi [A-L]

Kevin [M-Z]

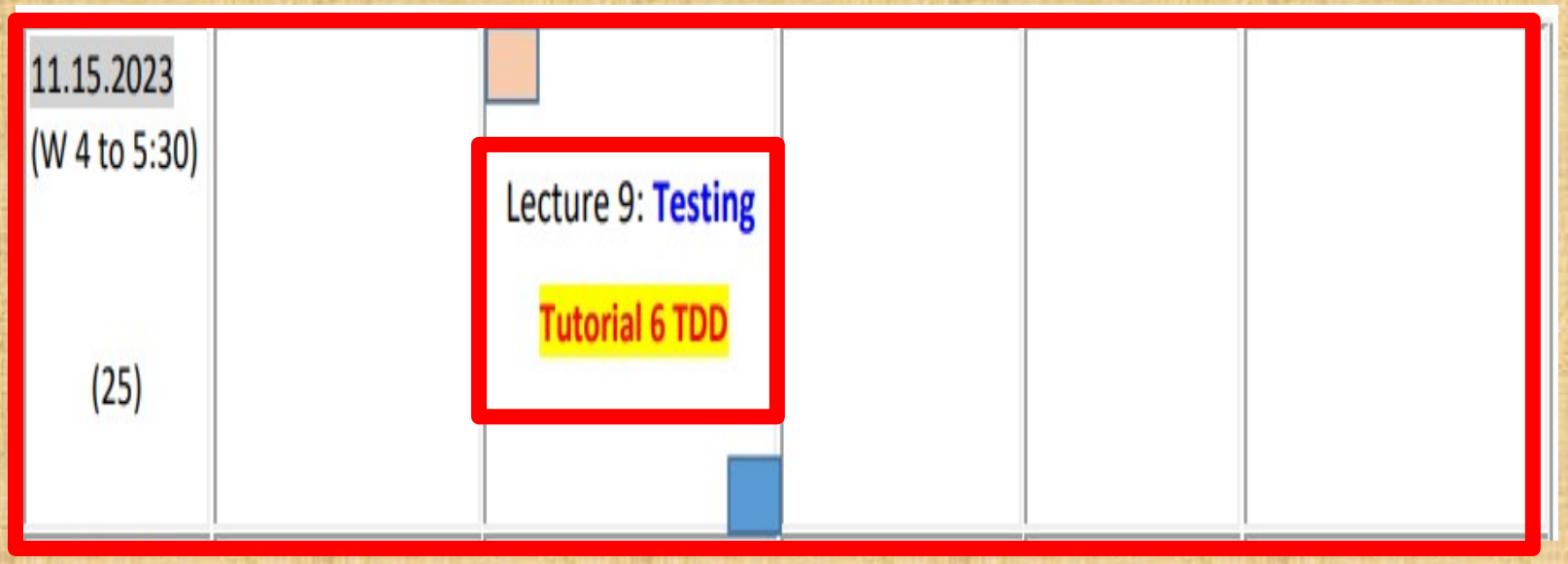
Please close all other windows.

11.15.2023 (W 4 to 5:30) (25)		Lecture 9: Testing Tutorial 6 TDD			
11.20.2023 (M 4 to 5:30) (26)		EXAM 4 REVIEW (CANVAS)	Download ZyBook: Sections 12-14		
11.27.2023 (M 4 to 5:30) Optional (27)					Q & A Set 4 topics.
11.29.2023 (W 4 to 5:30) (28) LAST CLASS					EXAM 4 (CANVAS)

Class 25

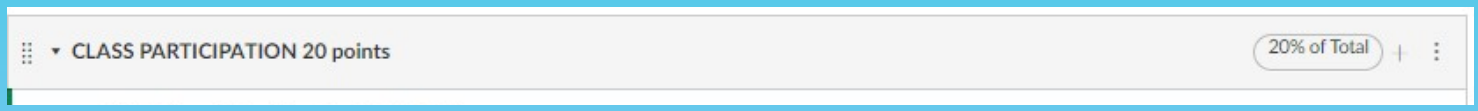
COSC 4351

Software engineering



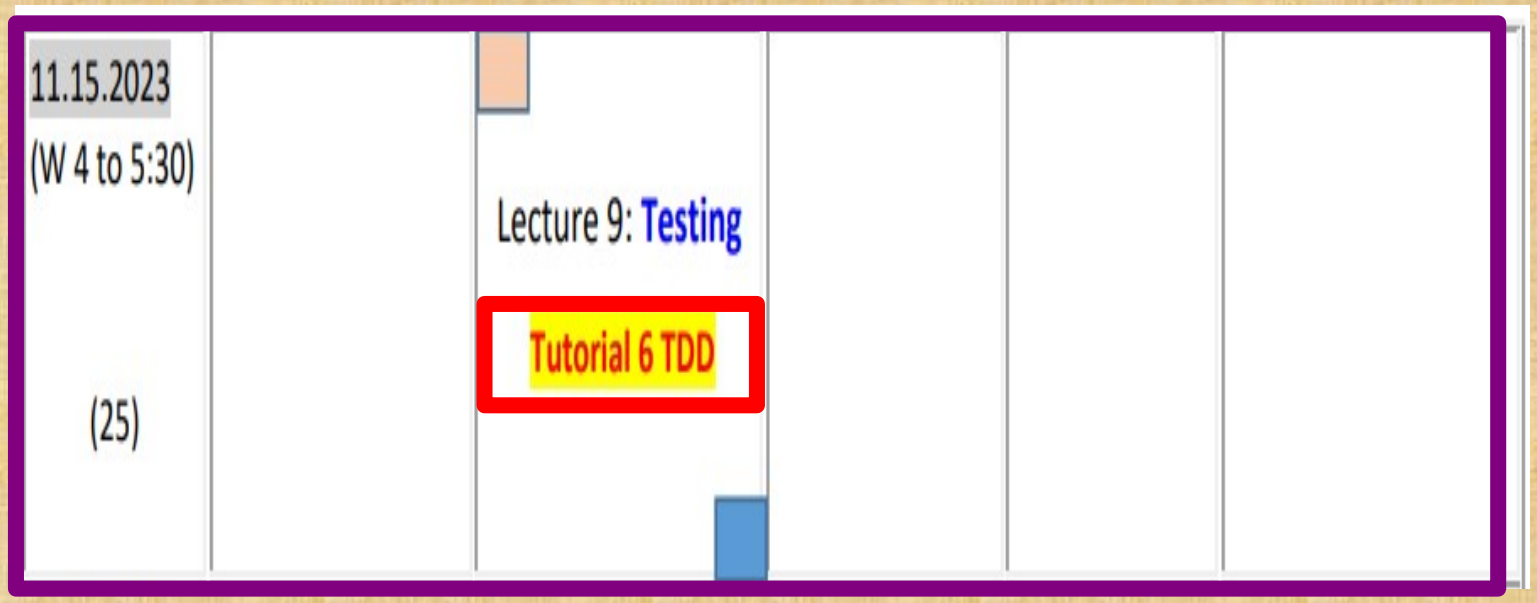
From 4:00 to 4:10 PM – 10 minutes.

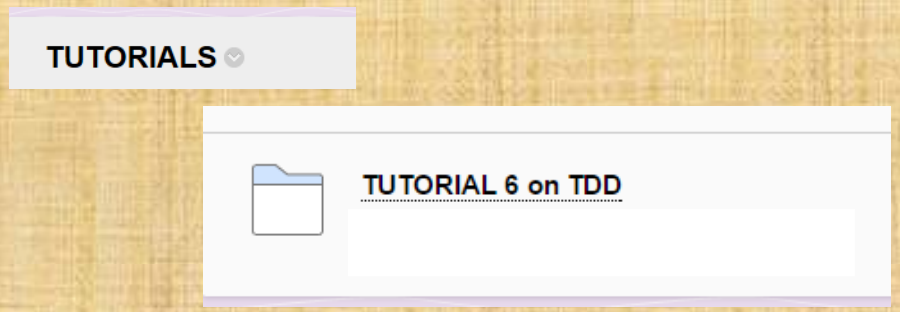
11.15.2023 (W 4 to 5:30) (25)		Lecture 9: Testing Tutorial 6 TDD			



PASSWORD: TESTING

From 4:10 to 4:20 – 10 minutes.





Describe some features of the Extreme Programming (XP) Life-Cycle Model.

Extreme programming [Beck, 2000] is a somewhat controversial new approach to software development based on the iterative-and-incremental model. The first step is that the software development team determines the various features (**stories**) the client would like the product to support. For each such feature, the team informs the client how long it will take to implement that feature and how much it will cost. This first step corresponds to the requirements and analysis workflows of the iterative-and-incremental model (Figure 2.4).

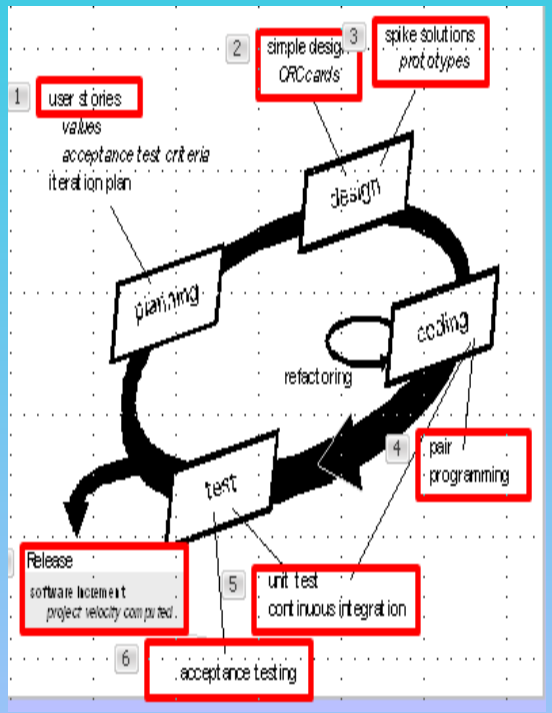
The client selects the features to be included in each successive build using cost–benefit analysis (Section 5.2), that is, on the basis of the duration and the cost estimates provided by the development team as well as the potential benefits of the feature to his or her business. The proposed build is broken down into smaller pieces termed **tasks**. A programmer first [illegible] test cases for a task; this is termed **test-driven development** (TDD). [illegible] mers work together on one computer (**pair programming**) [Williams, Kessler, Cunningham, and Jeffries, 2000], implementing the task and ensuring that all the test cases work correctly. The two programmers alternate typing every 15 or 20 minutes; the programmer who is not typing carefully checks the code while it is being entered by his or her partner. The task is then integrated into the current version of the product. Ideally, implementing and integrating a task should take no more than a few hours. In general, a num[illegible] implement tasks in parallel, so integration is essentially continuous [illegible] change coding partners daily, if possible; learning from the other team members increases everyone's skill level. The TDD test cases used for the task are retained and utilized in all further integration testing.

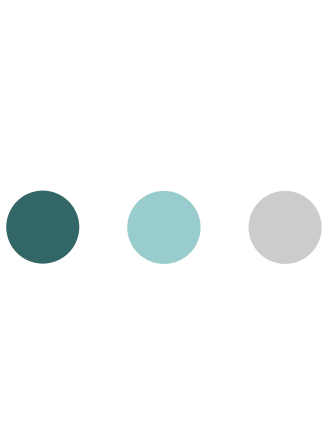
Some drawbacks to pair programming have been observed in practice [Drobka, Noftz, and Raghu, 2004]. For example, pair programming requires large blocks of uninterrupted time, and software professionals can have difficulty in finding 3- to 4-hour blocks of time. In addition, pair programming does not always work well with shy or overbearing individuals, or with two inexperienced programmers.

UML

TDD

ccnet





Tutorial 6 on TDD

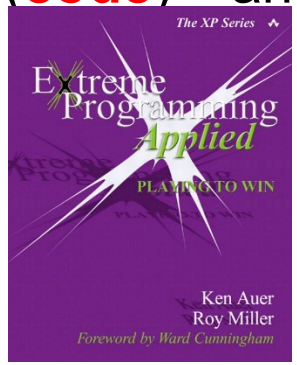
Test Driven Development

What is TDD?

- "Before you **write code**, **think** about what it will do.
  Write (**Code**) a **test** that will use the **methods** you **haven't even written yet**."

  Extreme Programming Applied: Playing To Win
  Ken Auer, Roy Miller
  "The Purple Book"

- A **test** is not something you "do", it is **something you "write** (**code**) **"** and **run** once, twice, three times, etc.
  - **Testing** can therefore be "**automated**"
  - **Repeatedly executed**, even after small changes



What is TDD?

TDD is a technique whereby you write your **test cases before** you write any **implementation code**

Tests drive or dictate **the code that is developed**

An indication of “intent”

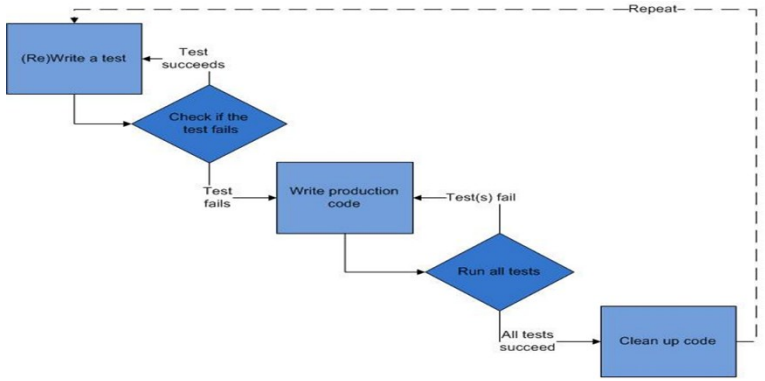
- **Tests** provide a **specification** of “**what**” **a piece of code** actually does

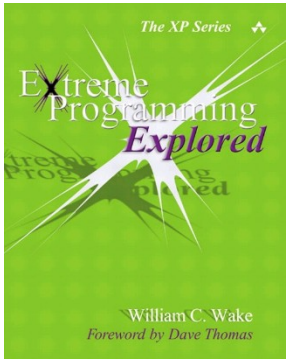
TDD Stages

- **In Extreme Programming Explored (The Green Book),**
  Bill Wake describes the **test / code cycle:**

1. Create a **single test**
2. Compile it. It **shouldn't compile** because you've not written the **testing code**
3. Write **just enough testing code** to get the **test** to Compile
4. Run the **test** and see it **fail**

5. Write **just enough method code** to get the **test** to **pass**
6. Run the **test** and see it **pass**
7. Repeat





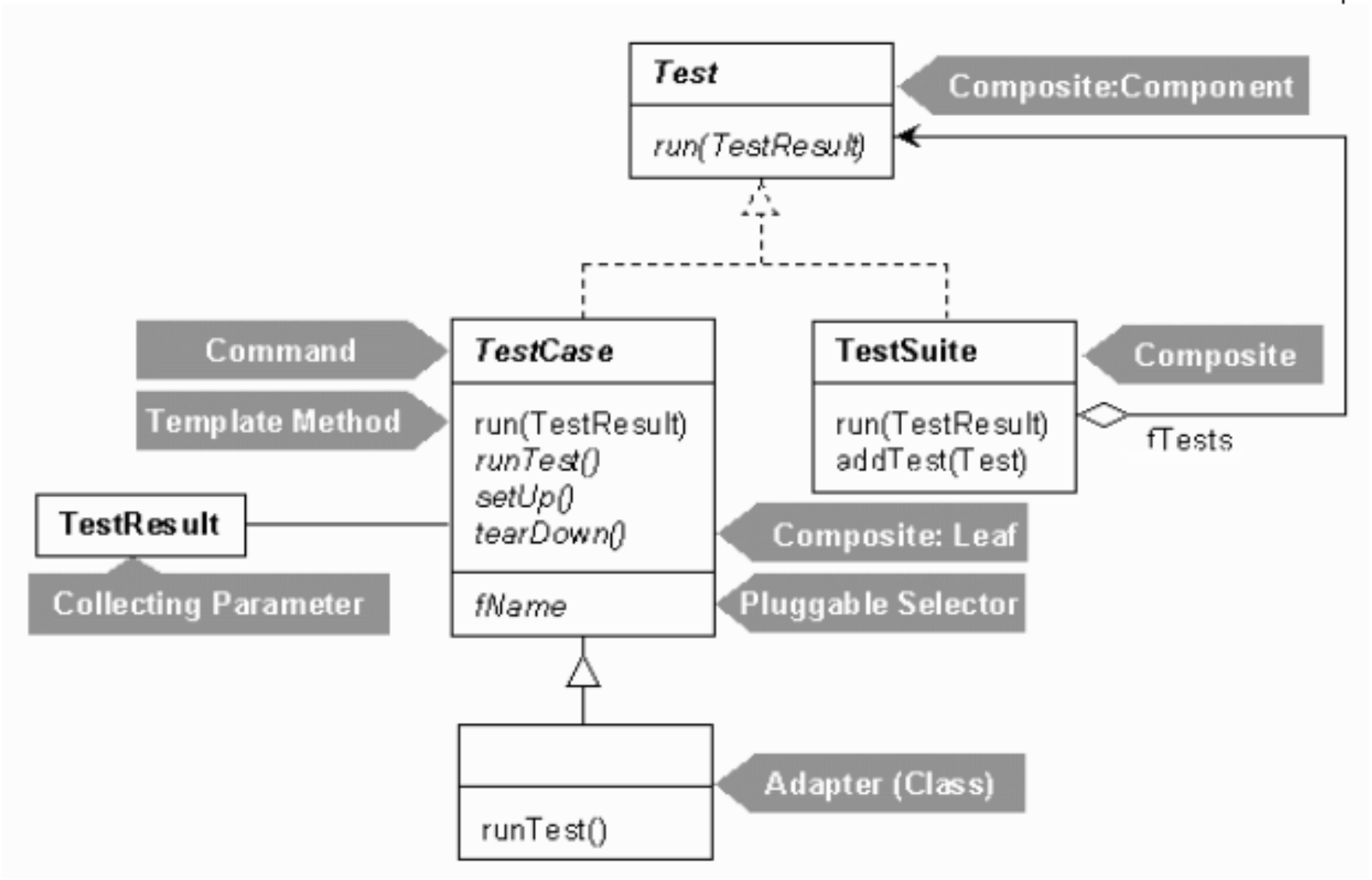
Junit - Java Unit Testing

- Java IDEs
  - Netbeans - http://www.netbeans.org/index.html
    - Includes JUnit

What is JUnit?

- JUnit is a **regression testing** framework written by Erich Gamma and Kent Beck
- It is found at www.junit.org
- It consists of **classes** that the developer can extend to write a **test**
- Notably: junit.framework.TestCase and related structure to Run and Report the **tests**

The Patterns Used in JUnit



Calculator - NetBeans IDE 6.5.1

File Edit View Navigate Source Refactor Run Debug Profile Versioning Tools Window Help

<default config>

Search (Ctrl+I)

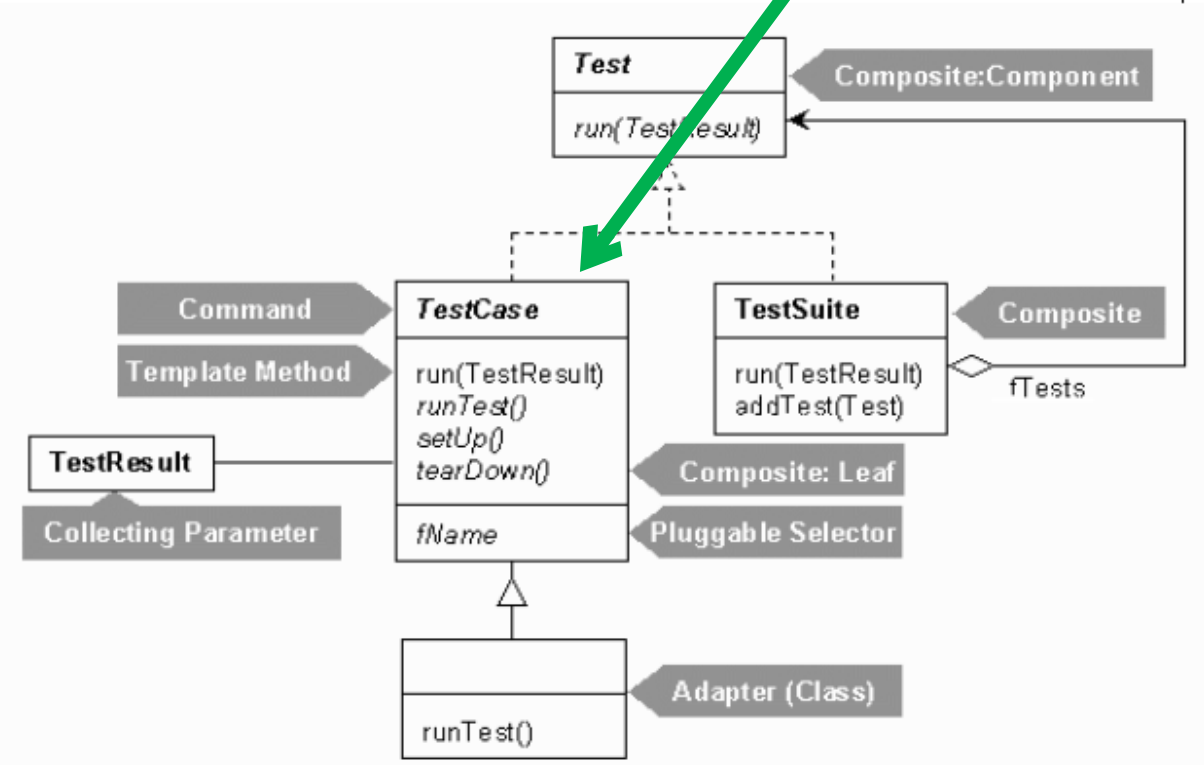
Projects

- Calculator
  - Source Packages
  - Test Packages
    - calculator
      - CalculatorTest.java
  - Libraries
  - Test Libraries

Calculator.java | CalculatorTest.java

```
3  import junit.framework.TestCase;
4
5  public class CalculatorTest extends TestCase {
6
```

The Patterns Used in JUnit




```
32
33  }
```

3:32 INS

Calculator - NetBeans IDE 6.5.1

File Edit View Navigate Source Refactor Run Debug Profile Versioning Tools Window Help

<default config>

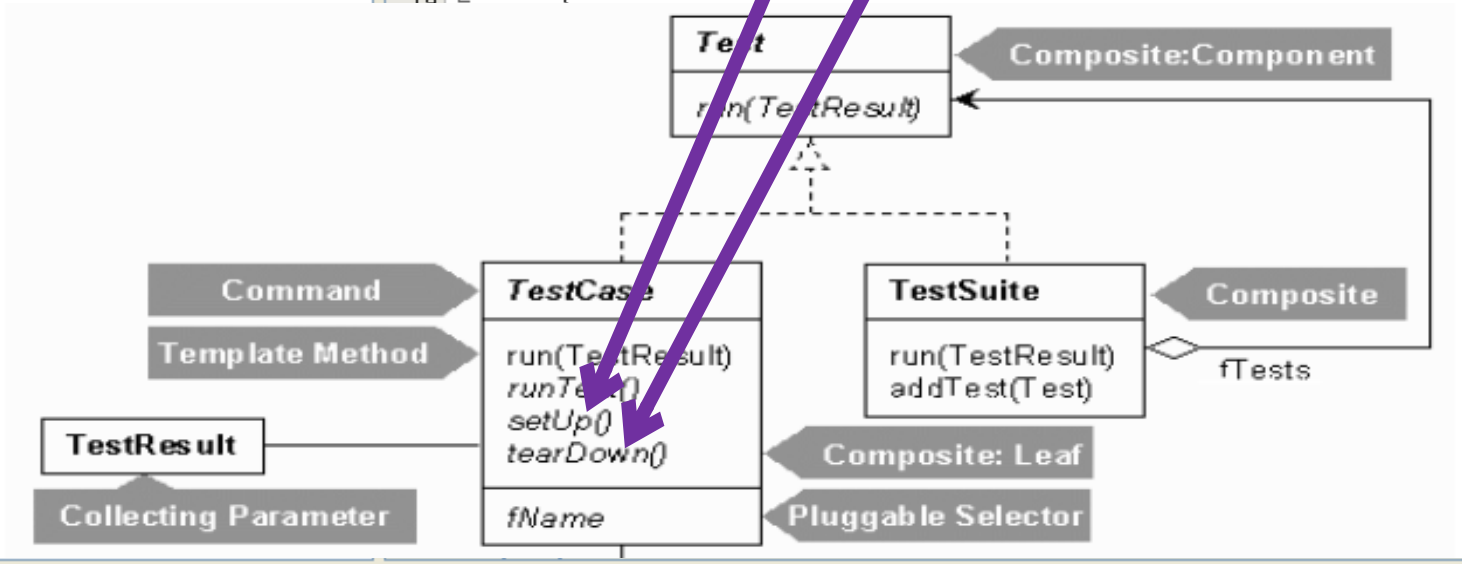
Search (Ctrl+I)

Projects

- Calculator
  - Source Packages
  - Test Packages
    - calculator
      - CalculatorTest.java
  - Libraries
  - Test Libraries

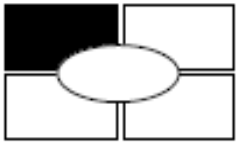
Calculator.java | CalculatorTest.java

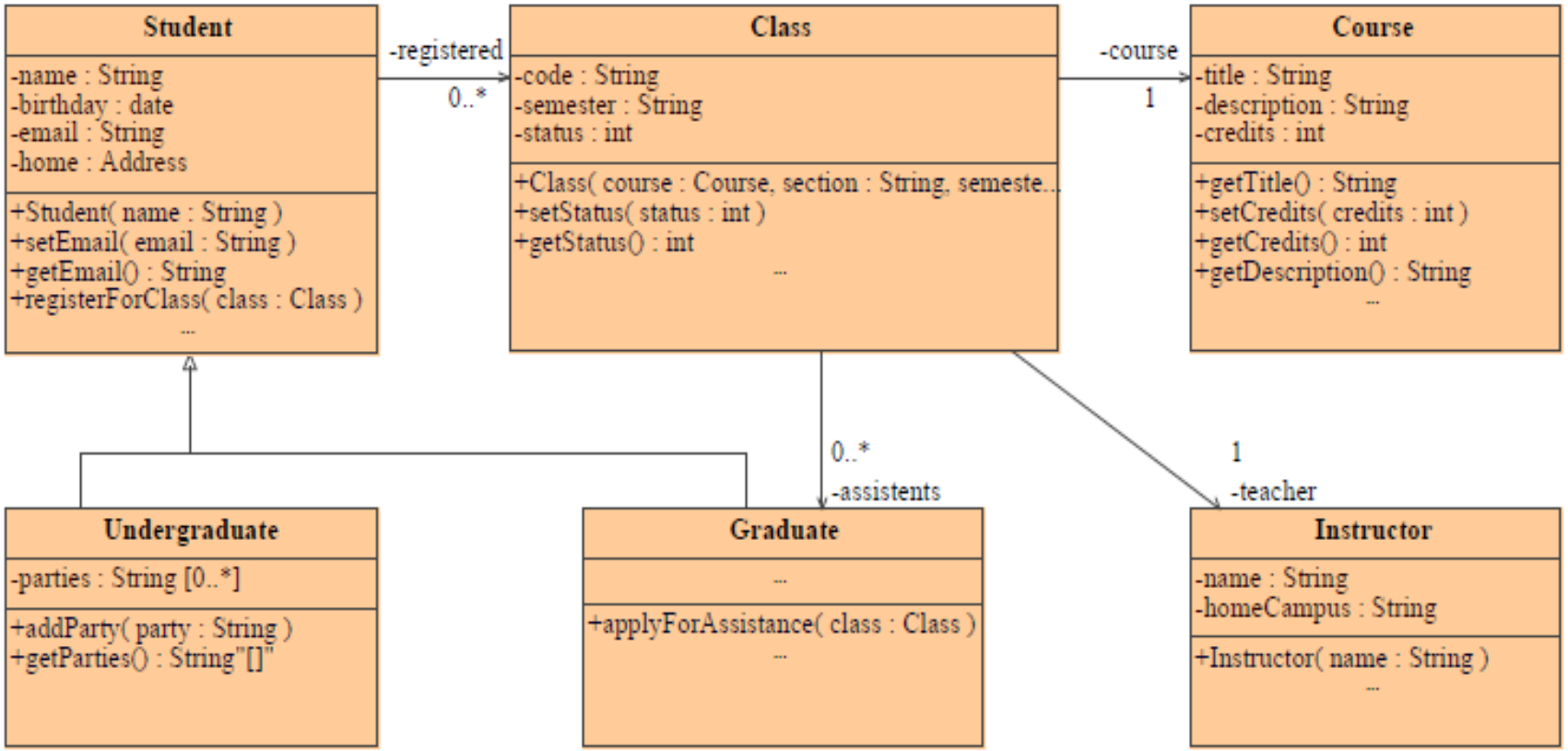
```
1  import junit.framework.TestCase;
4
5  public class CalculatorTest extends TestCase {
6
7      public CalculatorTest(String testName) {
8          super(testName);
9      }
10
       @Override
12     protected void setUp() throws Exception {
13         super.setUp();
14     }
15
       @Override
17     protected void tearDown() throws Exception {
18         super.tearDown();
```



UML OOD Class Diagram

- Describes static structure of the system
- Contains classes and relationships





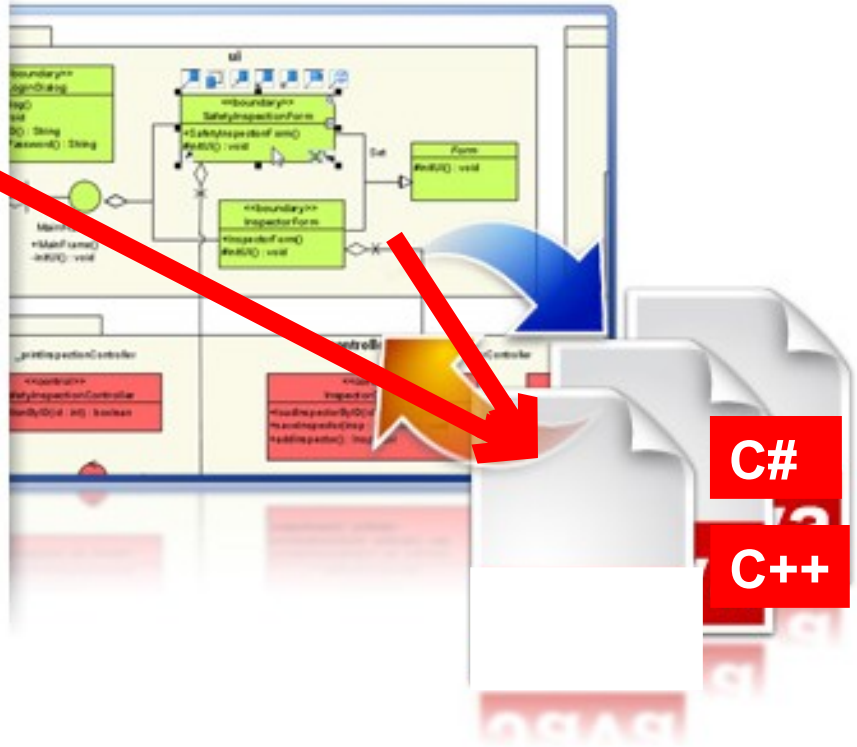
Model Driven Architecture (MDA)

http://www.visual-paradigm.com/product/vpuml/provides/codedbeng.jsp

Java round-trip code engineering

Generate/reverse Java source code from/to UML class model, and keeps the model synchronized with source code. You can synchronize the change make in source code to class model, or the other way round. This helps ensure that

The following movie demonstrates how to perform Java round-trip with Visual Paradigm for UML (VP-UML).



Tutorial

- Generate and synchronize Java code

User's Guide

- Generate/Update Java code
- Generate/Update UML classes from Java code

Online training

- Generate UML class diagram to Java (Paid)

Simple **Calculator** Application

Simple **Calculator** application that allows add, subtract, multiply, and divide operations on integers.

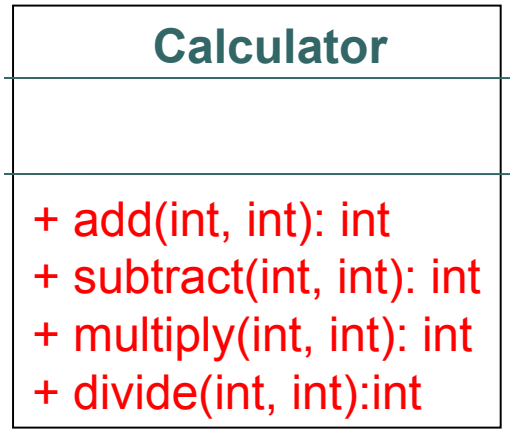
UML Class Diagram?

| Calculator

Requirements

OOA & **OOD** & **TDD** a Simple **Calculator** application that allows add, subtract, multiply, and divide operations on integers.

UML Class Diagram Model

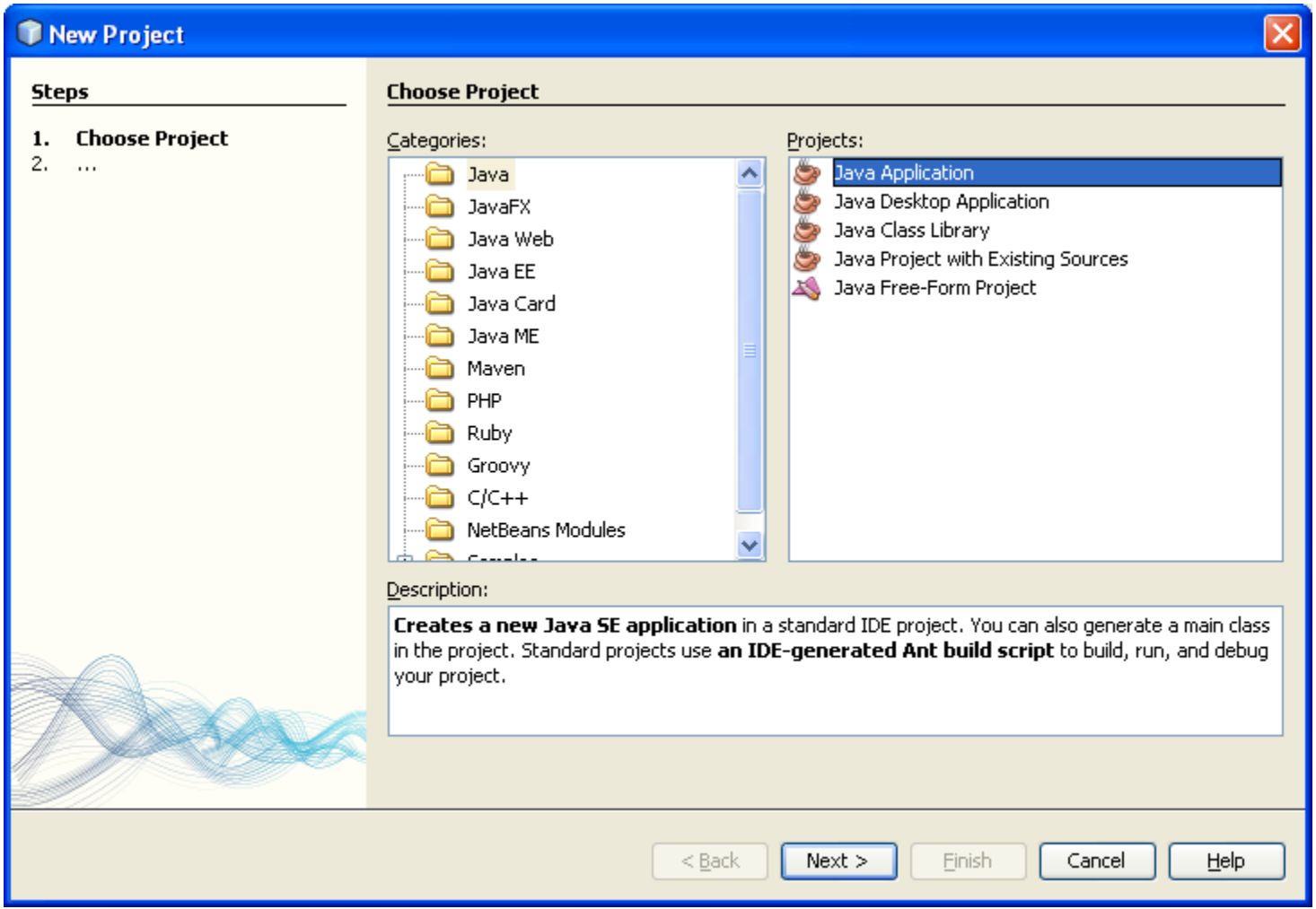


Class skeleton:

```
public class Calculator {
        int add(int number1, int number2) {
                return 0;
        }

        int subtract(int number1, int number2) {
                return 0;
        }

        int multiply(int number1, int number2) {
                return 0;
        }

        int divide(int number1, int number2) {
                return 0;
        }
};
```

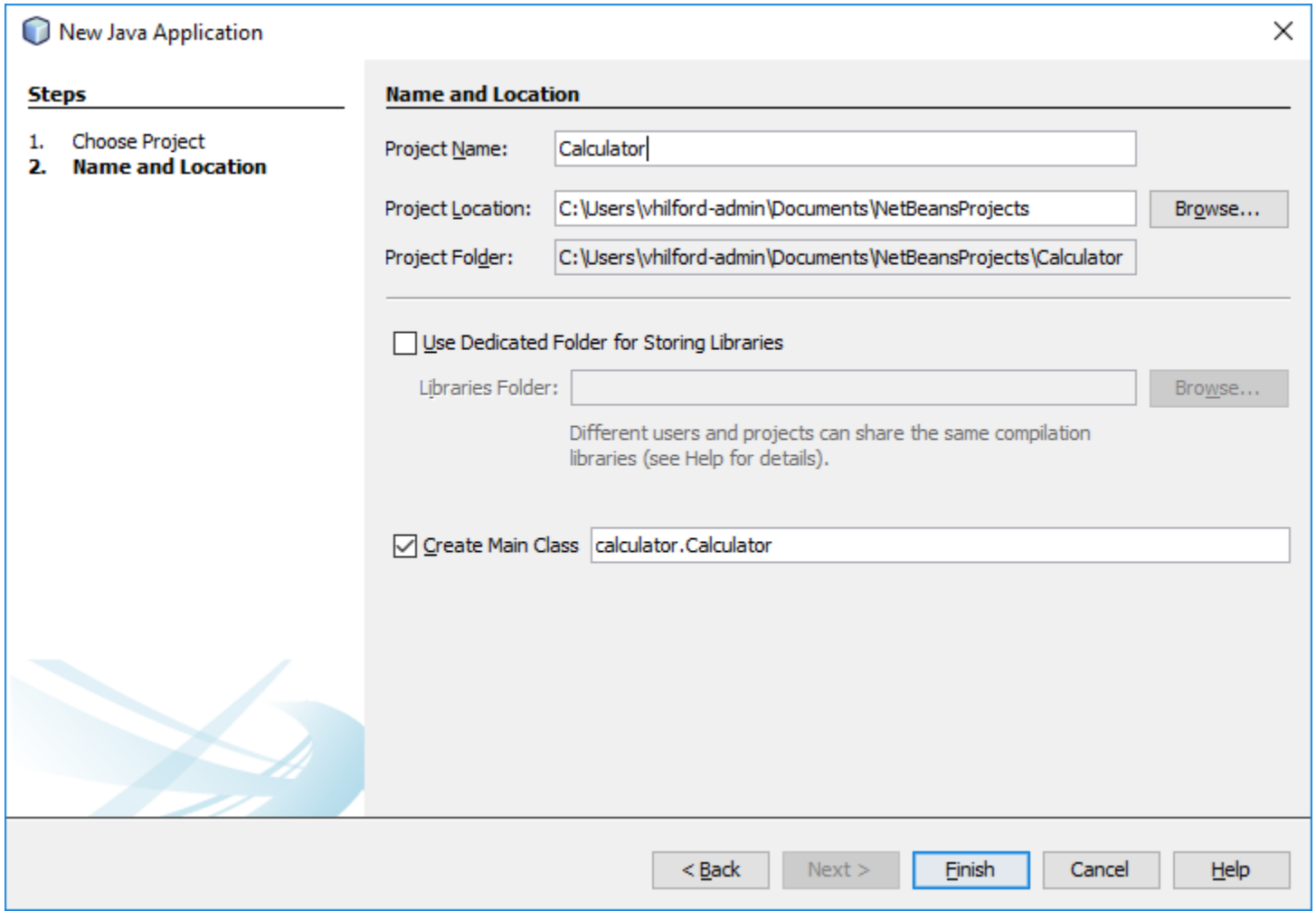
JUnit Tool Integration

Create a new Java Application **Calculator** in NetBeans 8.0.2

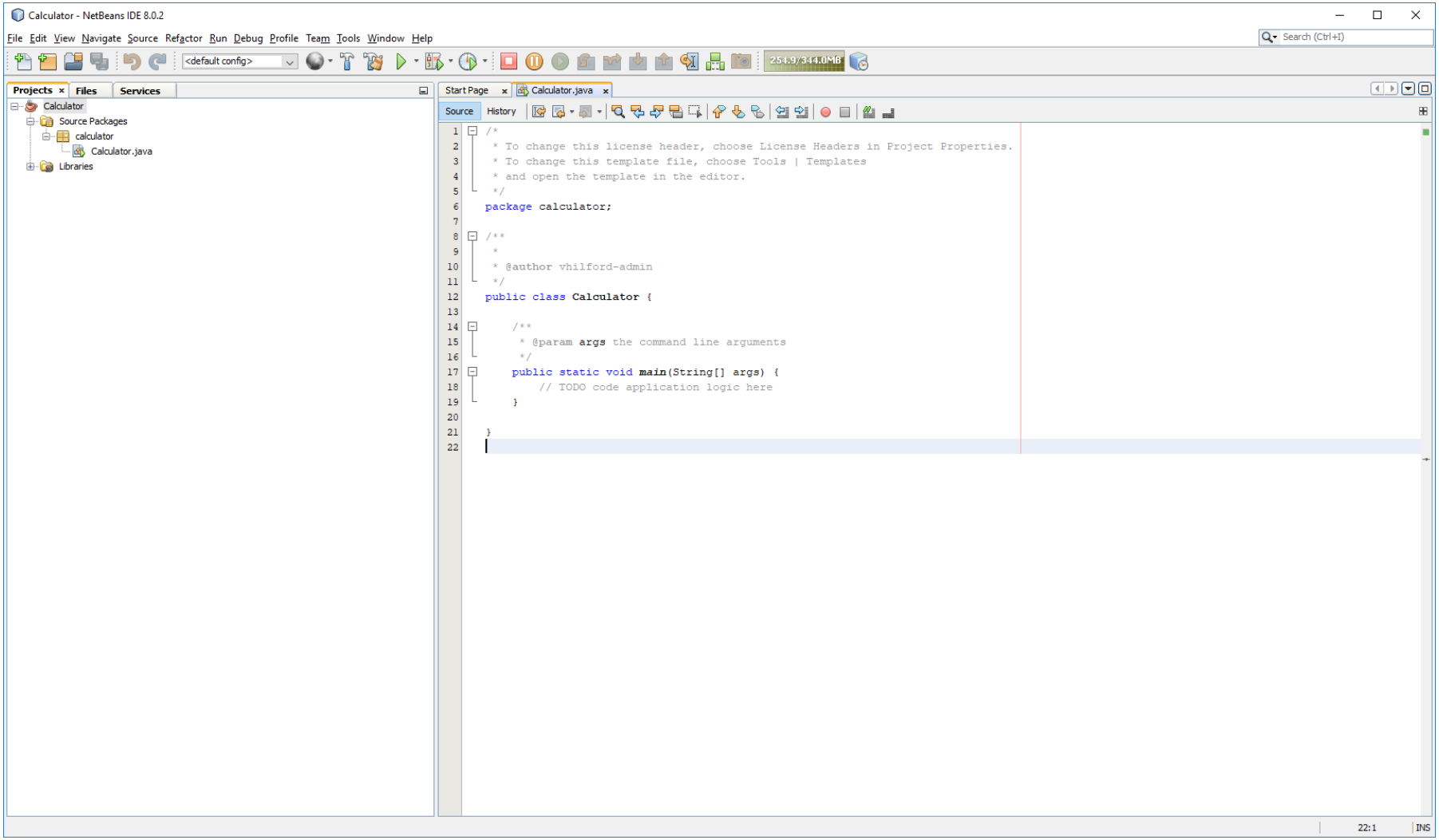


JUnit Tool integration

Calculator

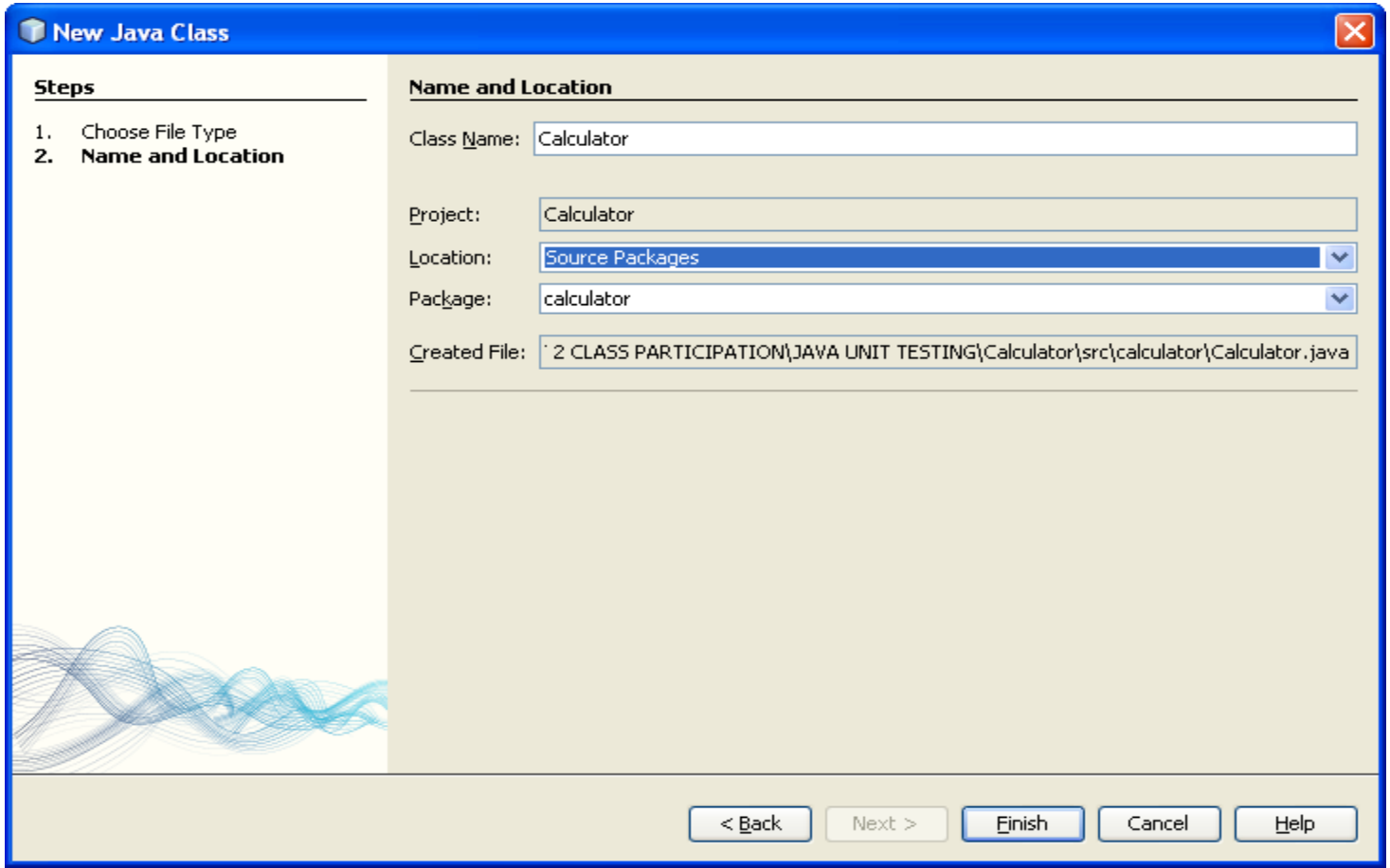


JUnit Tool integration



JUnit Tool integration

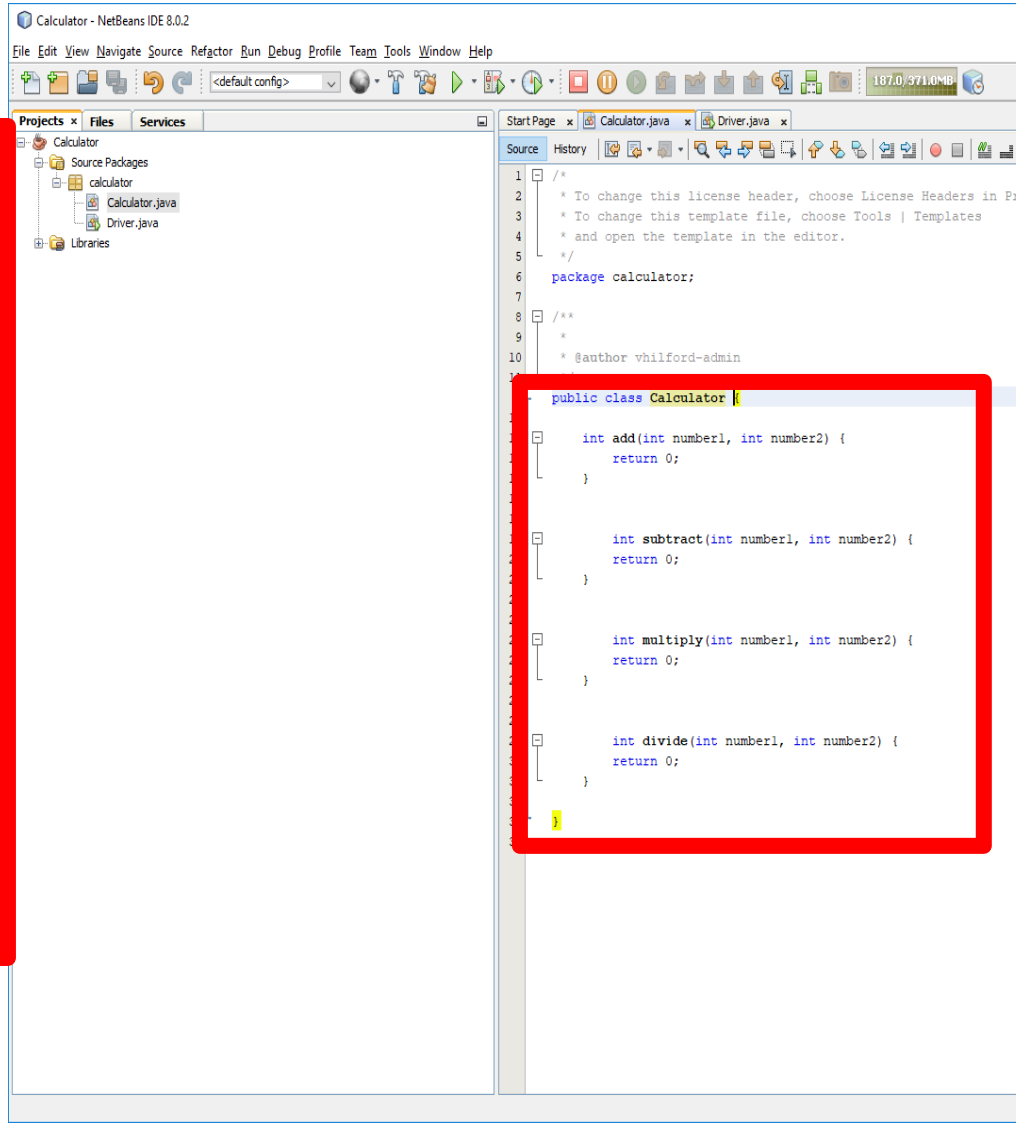
Java Class Calculator



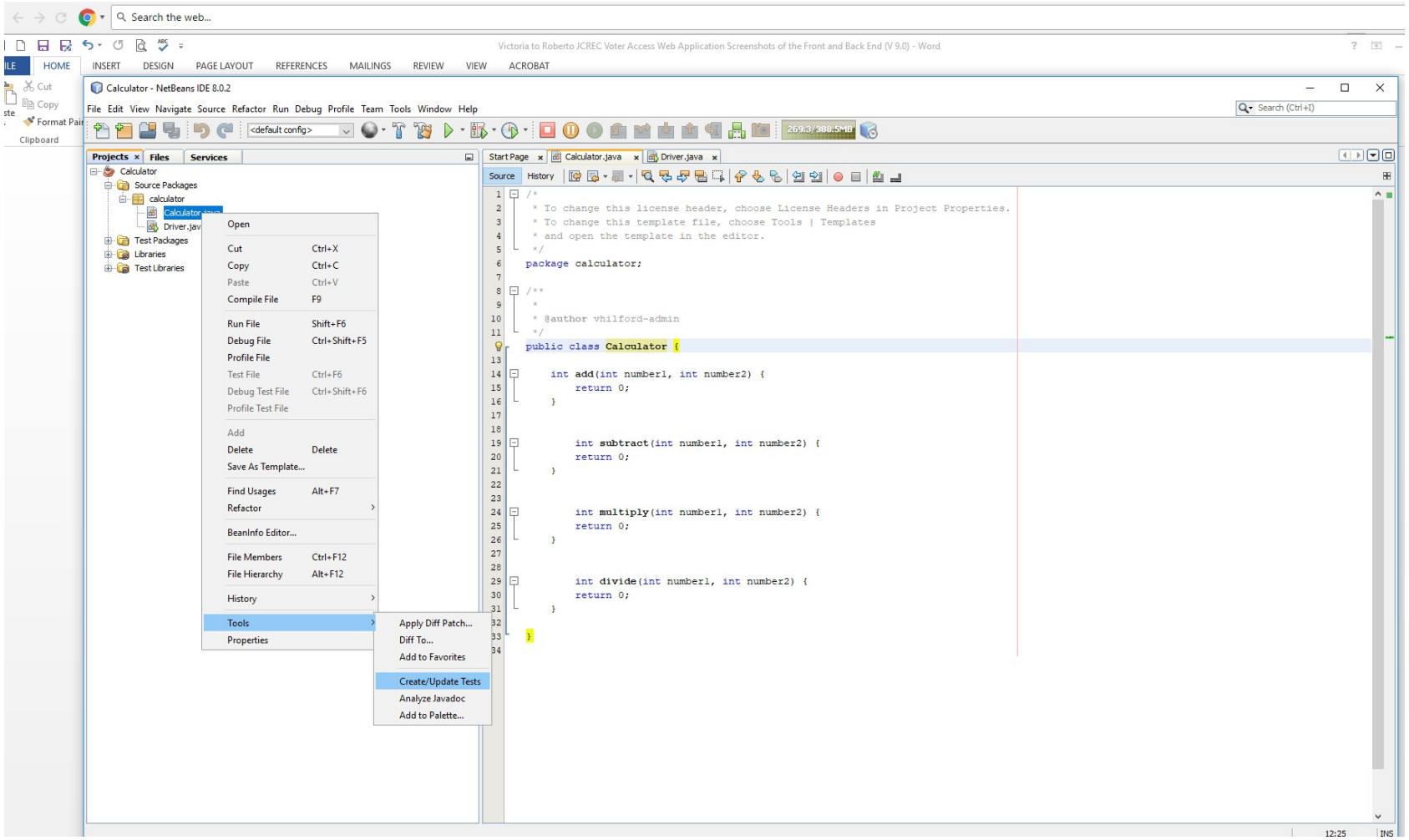
JUnit Tool integration

Write the following code in that file

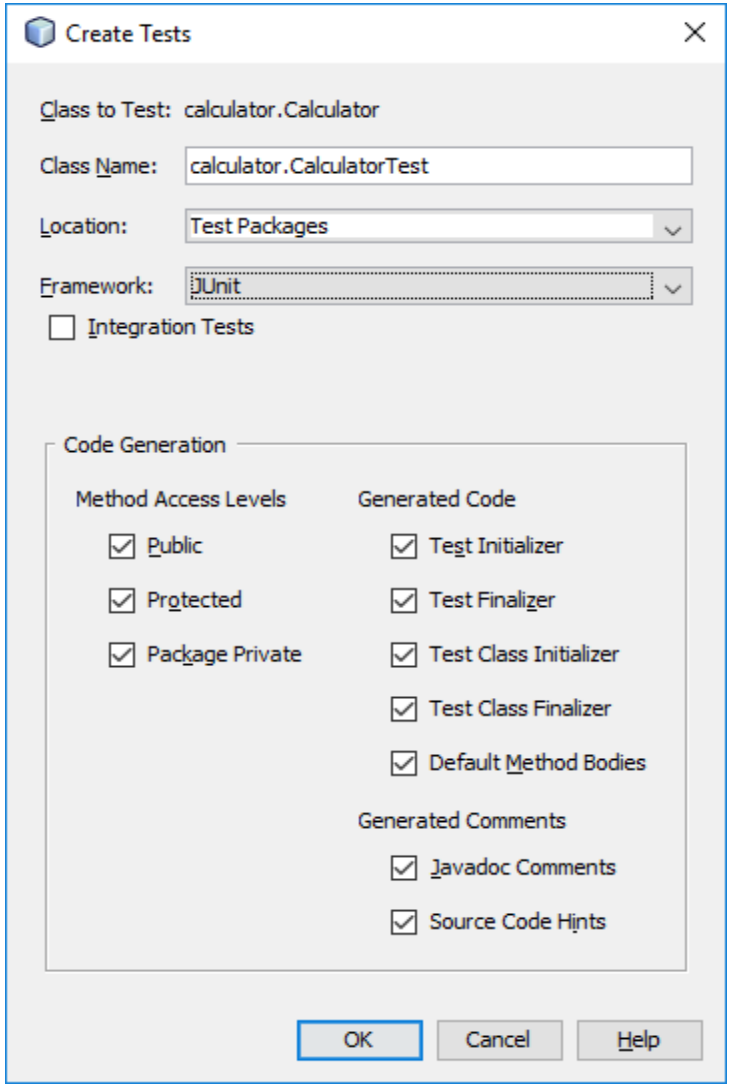
```
public class Calculator {

        int add(int number1, int number2) {
                return 0;
        }

        int subtract(int number1, int number2) {
                return 0;
        }

        int multiply(int number1, int number2) {
                return 0;
        }

        int divide(int number1, int number2) {
                return 0;
        }
};
```

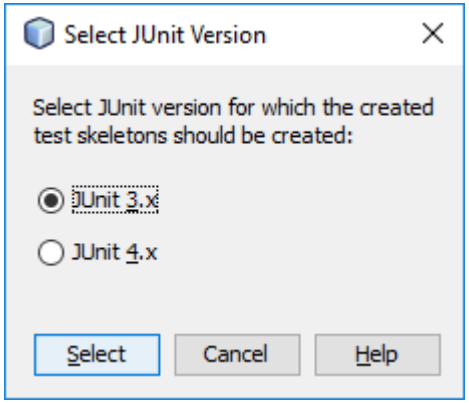


JUnit Tool integration

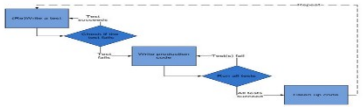


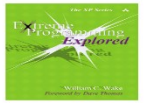
JUnit Tool integration





1. Create a **single test**
2. Compile it. It **shouldn't compile** because you've not written the **testing code**
3. Write **just enough testing code** to get the **test** to Compile
4. Run the **test** and see it **fail**
5. Write **just enough method code** to get the **test** to **pass**
6. Run the **test** and see it **pass**
7. Repeat





JUnit Tool integration

Calculator - NetBeans IDE 8.0.2

File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help

Search (Ctrl+I)

<default config>

Projects Files Services

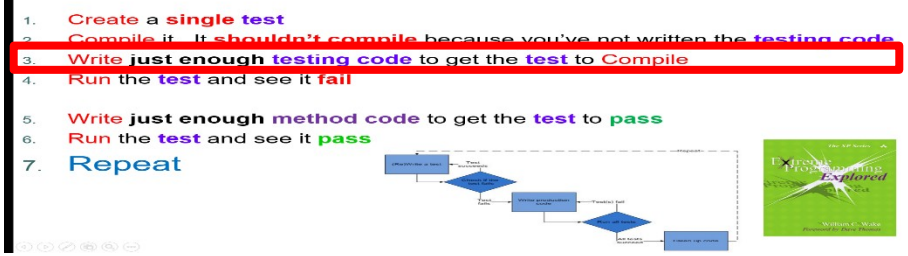
- Calculator
  - Source Packages
    - calculator
      - Calculator.java
      - Driver.java
  - Test Packages
    - CalculatorTest.java
  - Test Libraries

Calculator.java | Driver.java | CalculatorTest.java

Source History

```
1  /*
2   * To change this license header, choose License Headers in Project Properties.
3   * To change this template file, choose Tools | Templates
4   * and open the template in the editor.
5   */
6  package calculator;
7
8  import junit.framework.TestCase;
9
10 /**
11  *
12  * @author vhilford-admin
13  */
14 public class CalculatorTest extends TestCase {
15
16     public CalculatorTest(String testName) {
17         super(testName);
18     }
19
20     @Override
22     protected void setUp() throws Exception {
23         super.setUp();
24     }
25
26     @Override
27     protected void tearDown() throws Exception {
28         super.tearDown();
29     }
30
31     /**
32      * Test of add method, of class Calculator.
33      */
34     public void testAdd() {
35         System.out.println("add");
36         int number1 = 0;
37         int number2 = 0;
38         Calculator instance = new Calculator();
39         int expResult = 0;
40         int result = instance.add(number1, number2);
41         assertEquals(expResult, result);
42         // TODO review the generated test code and remove the default call to fail.
43         fail("The test case is a prototype.");
44     }
45
46     /**
      * Test of subtract method, of class Calculator.
```

14:28/1:14 INS



JUnit Tool integration

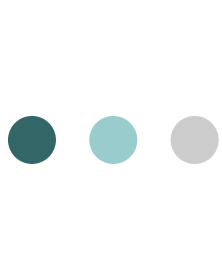
Fix the code for testAdd

```
public void testAdd() {
        System.out.println("add");
        int number1 = 3;
        int number2 = 2;
        Calculator instance = new Calculator();
        int expResult = 5;
        int result = instance.add(number1, number2);
        assertEquals(expResult, result);
    }
```

```
30      /**
31       * Test of add method, of class Calculator.
32       */
33      public void testAdd() {
34          System.out.println("add");
35          int number1 = 0;
36          int number2 = 0;
37          Calculator instance = new Calculator();
38          int expResult = 0;
39          int result = instance.add(number1, number2);
40          assertEquals(expResult, result);
41          // TODO review the generated test code and remove the default call to fail.
42          fail("The test case is a prototype.");
43      }
```

```
30      /**
31       * Test of add method, of class Calculator.
32       */
33      public void testAdd() {
34          System.out.println("add");
35          int number1 = 2;
36          int number2 = 3;
37          Calculator instance = new Calculator();
38          int expResult = 5;
39          int result = instance.add(number1, number2);
40          assertEquals(expResult, result);
41          // TODO review the generated test code and remove the default call to fail.
42          fail("The test case is a prototype.");
43      }
```

```
30      /**
31       * Test of add method, of class Calculator.
32       */
33      public void testAdd() {
34          System.out.println("add");
35          int number1 = 2;
36          int number2 = 3;
37          Calculator instance = new Calculator();
38          int expResult = 5;
39          int result = instance.add(number1, number2);
40          assertEquals(expResult, result);
41      }
```



assert*X* methods

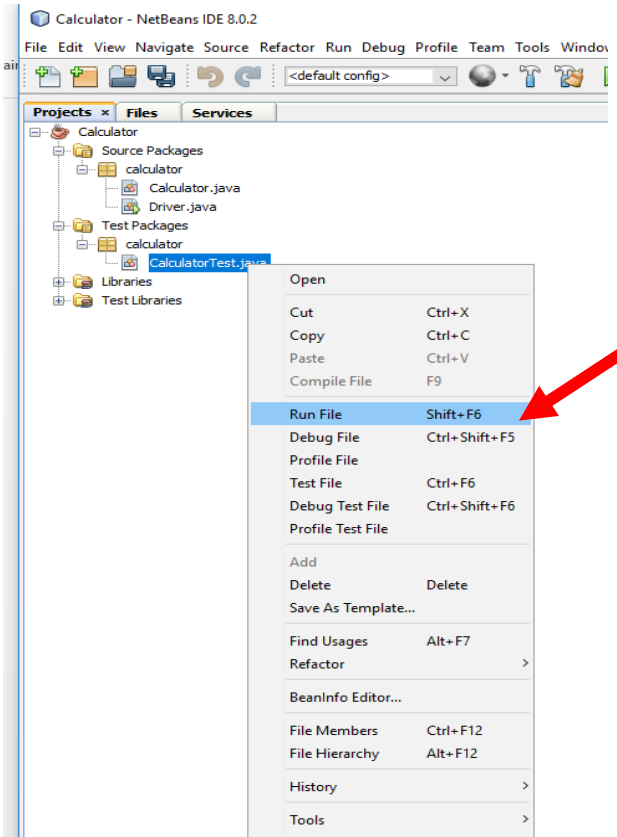
static void **assertTrue**(boolean ***test***)
static void **assertFalse**(boolean ***test***)
assertEquals(***expected***, ***actual***)
assertSame(Object ***expected***, Object ***actual***)
assertNotSame(Object ***expected***, Object ***actual***)
assertNull(Object ***object***)
assertNotNull(Object ***object***)
fail()

All the above may take an optional String message as the first argument, for example,

static void **assertTrue**(String ***message***, boolean ***test***)

1. Create a **single test**
2. Compile it. It **shouldn't compile** because you've not written the **testing code**
3. Write **just enough testing code** to get the **test** to Compile
4. Run the **test** and see it **fail**
5. Write **just enough method code** to get the **test** to **pass**
6. Run the **test** and see it **pass**

JUnit Tool integration

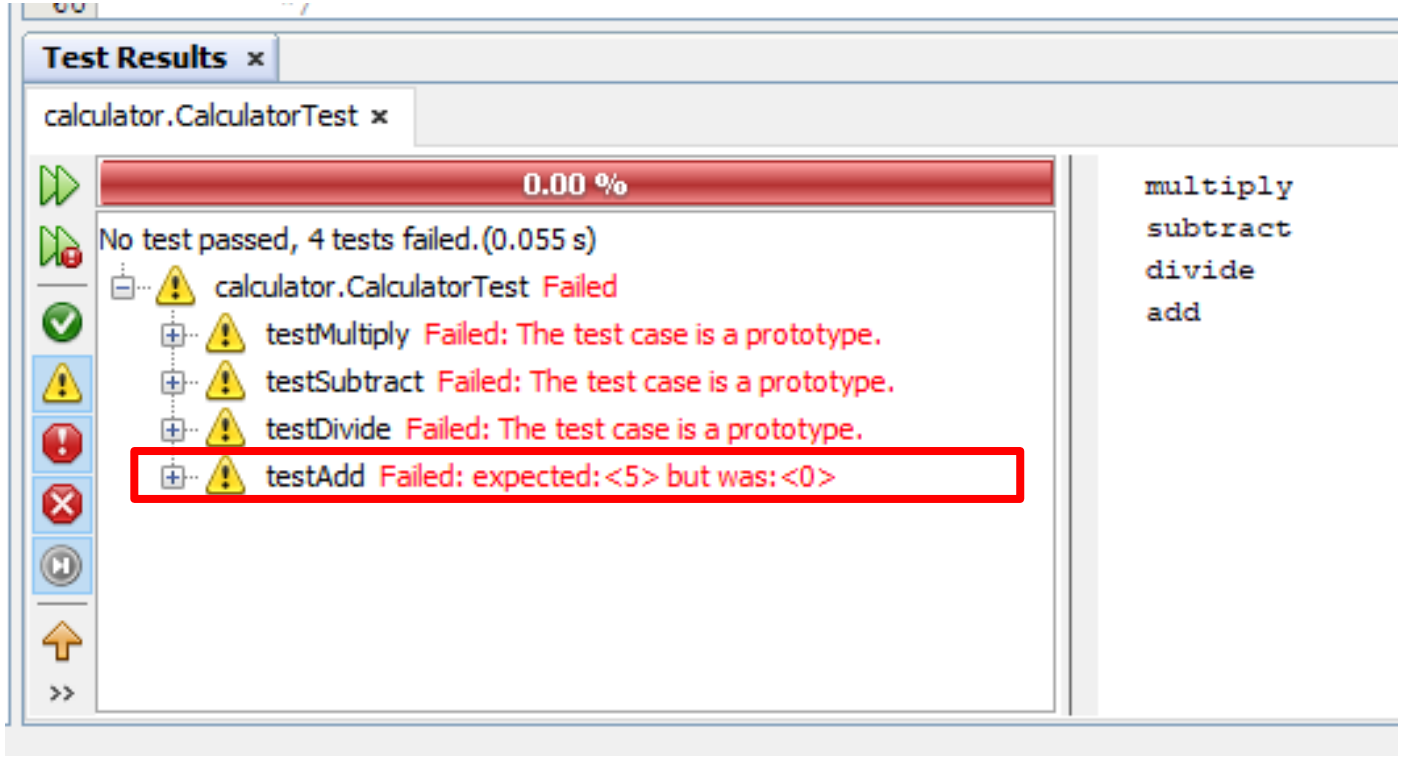


Run test File

1. Create a **single test**
2. Compile it. It **shouldn't compile** because you've not written the **testing code**
3. Write **just enough testing code** to get the **test** to Compile
4. Run the **test** and see it **fail**
5. Write **just enough method code** to get the **test** to **pass**
6. Run the **test** and see it **pass**

JUnit Tool integration

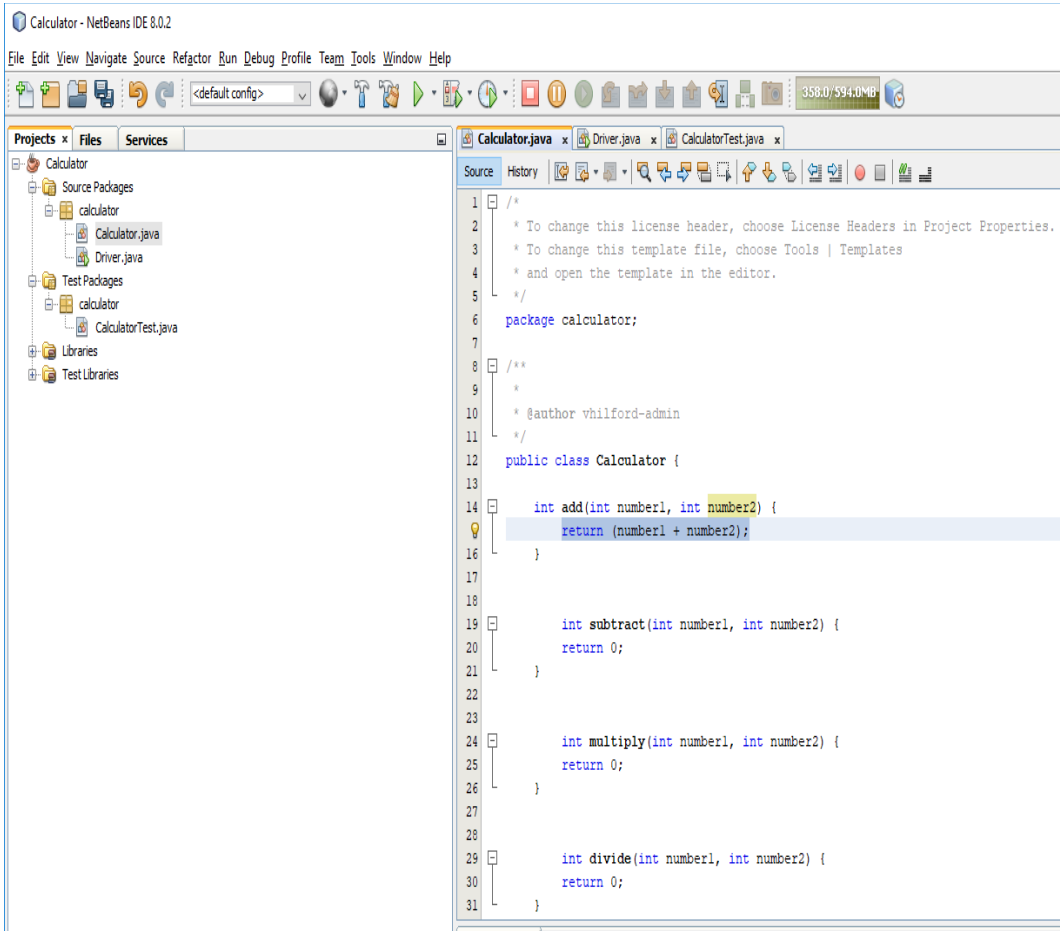
The test case **testAdd fails**!

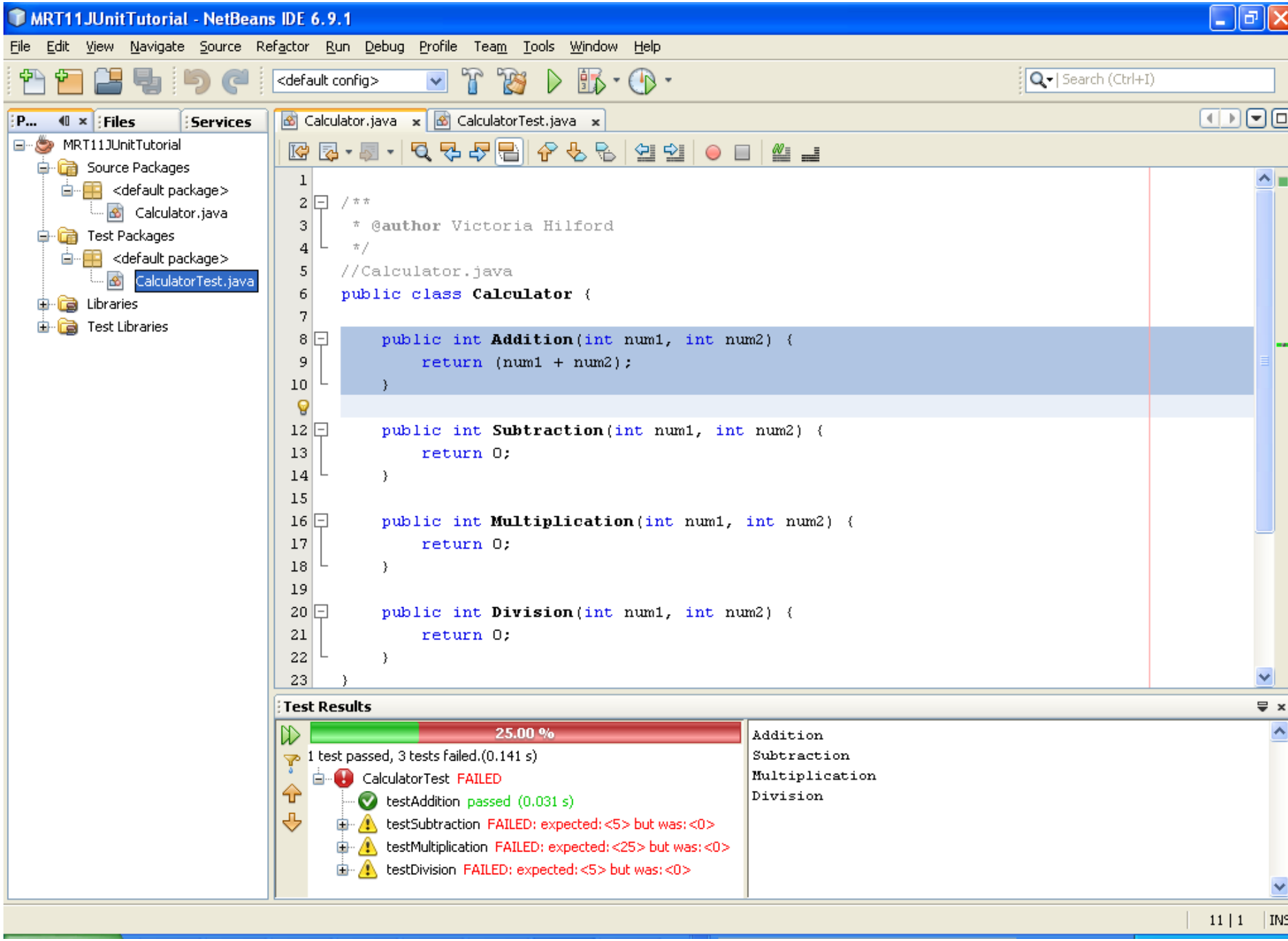


What do we need to do?

5. Write just enough method code to get the test to pass

```
public class Calculator
{

        int add (int number1, int number2)
        {
            return (number1 + number2);
        }

        int subtract (int number1, int number2)
        {
          return 0;
        }

        int multiply (int number1, int number2){

          return 0;
        }

        int divide (int number1, int number2)
        {
          return 0;
        }
};
```




MRT11JUnitTutorial - NetBeans IDE 6.9.1
File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help
<default config>
Search (Ctrl+I)
P... Files Services
MRT11JUnitTutorial
Source Packages
<default package>
Calculator.java
Test Packages
<default package>
CalculatorTest.java
Libraries
Test Libraries
Calculator.java
CalculatorTest.java
1 /**
2 * @author Victoria Hilford
3 */
4 //Calculator.java
5 public class Calculator {
6
7 public int Addition(int num1, int num2) {
8 return (num1 + num2);
9 }
10
12 public int Subtraction(int num1, int num2) {
13 return 0;
14 }
15
16 public int Multiplication(int num1, int num2) {
17 return 0;
18 }
19
20 public int Division(int num1, int num2) {
21 return 0;
22 }
23 }
Test Results
25.00 %
1 test passed, 3 tests failed.(0.141 s)
CalculatorTest FAILED
testAddition passed (0.031 s)
testSubtraction FAILED: expected:<5> but was:<0>
testMultiplication FAILED: expected:<25> but was:<0>
testDivision FAILED: expected:<5> but was:<0>
Addition
Subtraction
Multiplication
Division
11 | 1
INS

CppUnit - C++ Unit Testing

Download **Visual Assert** Build 3780

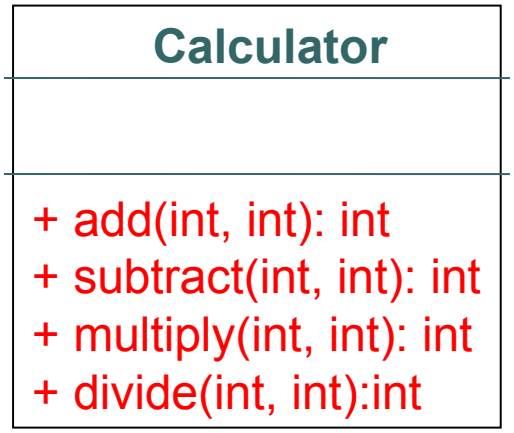
http://www.visualassert.com/unit-testing-framework/download.html

Calculator Application

Requirements

OOA & **OOD** & **TDD** a simple **Calculator** Application that allows add, subtract, multiply, and divide operations on integers.

OOA/OOD Modeling: UML Class Diagram Model

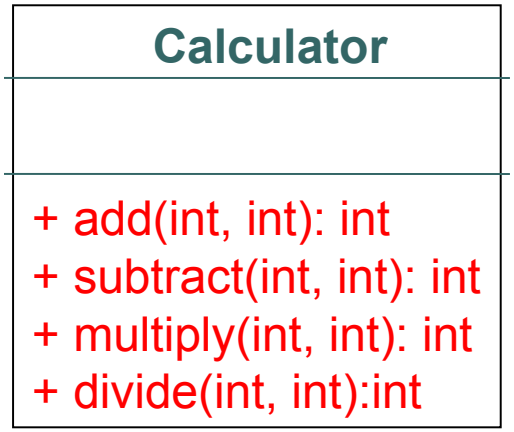


Calculator Application

OOA/OOD Modeling: UML Class Diagram Model

C++ Implementation class skeleton

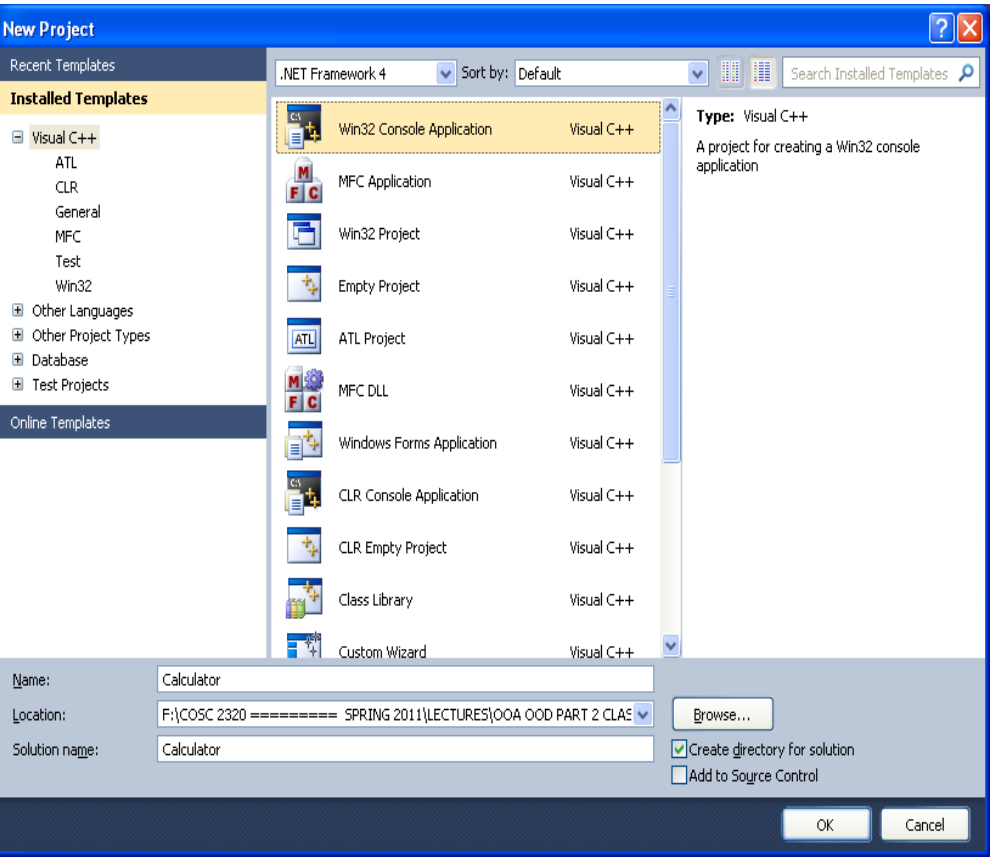
```
class Calculator {
        public:

        int add (int number1, int number2) {
            return 0;
        }

        int subtract (int number1, int number2) {
            return 0;
        }

        int multiply (int number1, int number2) {
            return 0;
        }

        int divide (int number1, int number2) {
            return 0;
        }
};
```



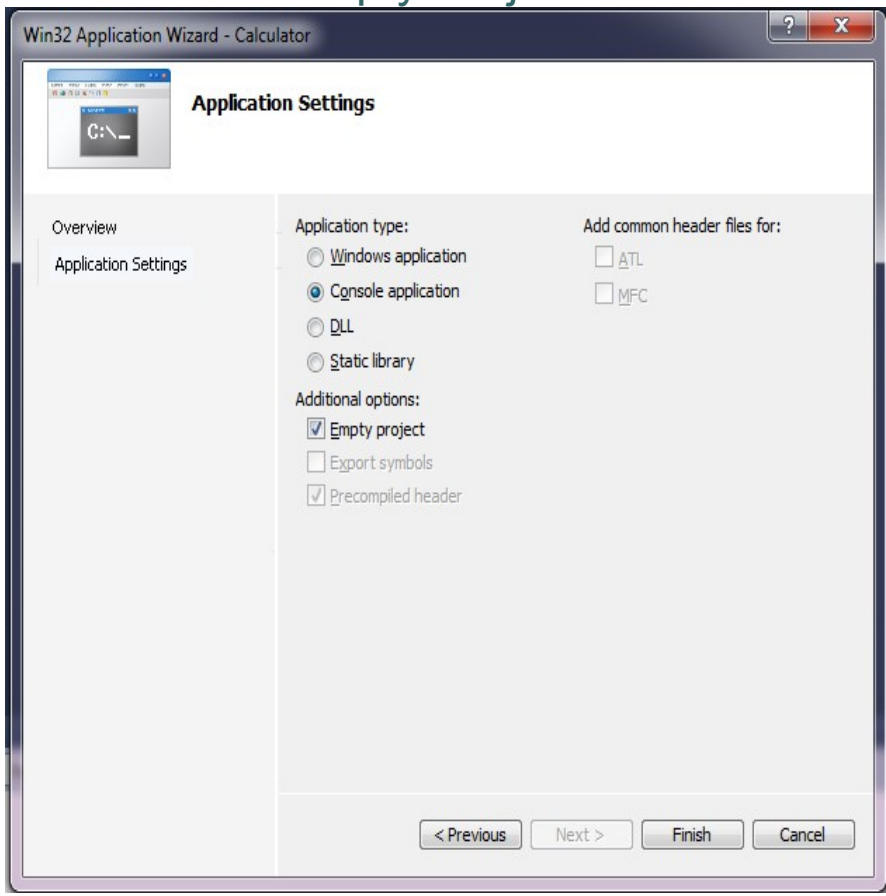
Create a Calculator.cpp file in the Calculator project:

Cpp Unit - Calculator

Create a new Win32 Console Application **Calculator**



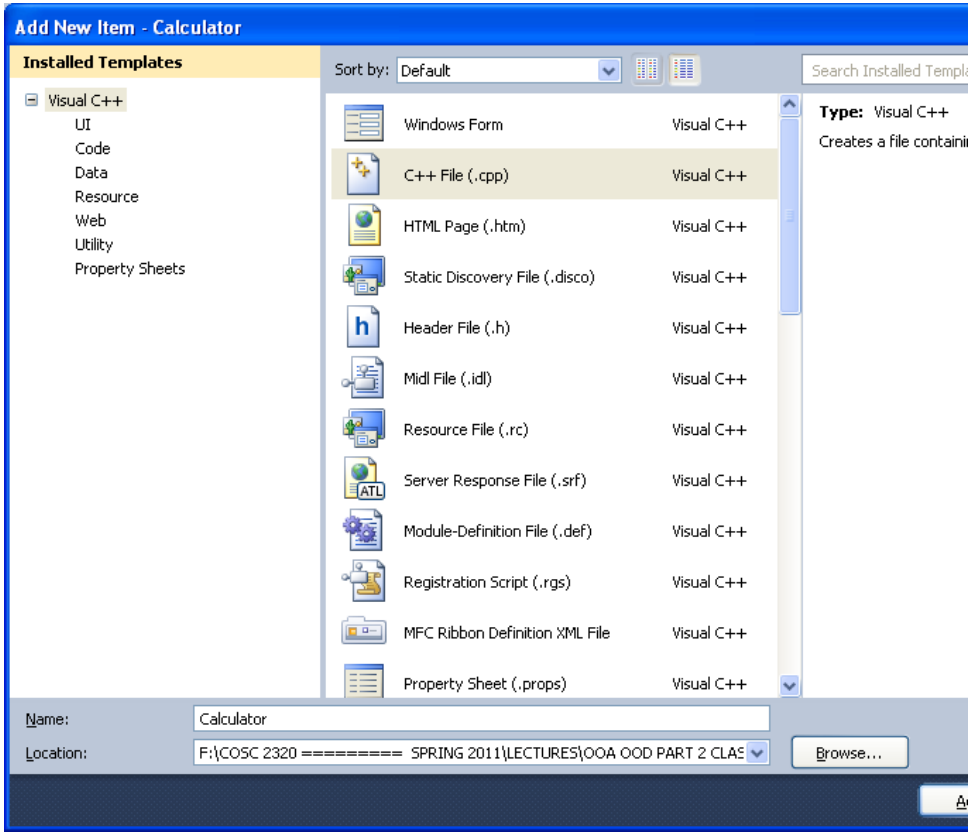
Empty Project



Finish

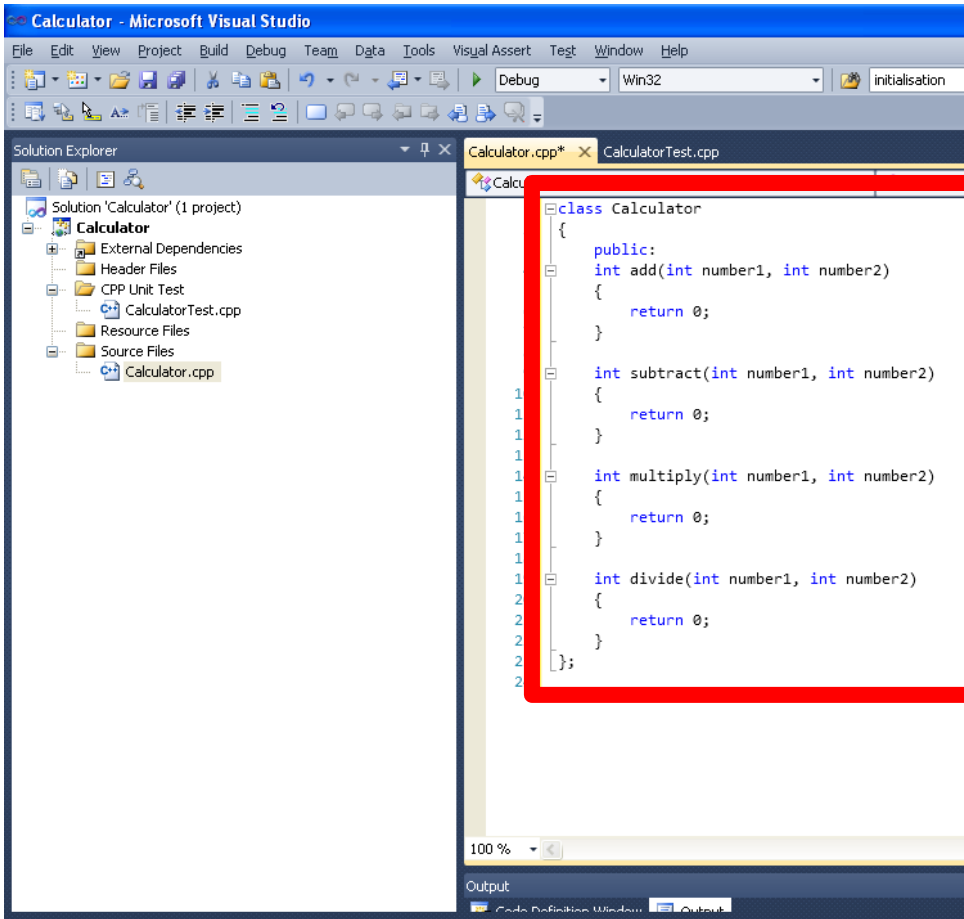
Create a Calculator.cpp file in the Calculator project:

```
class Calculator {

        public:
        int add (int number1, int number2) {

          return 0;
        }

        int subtract (int number1, int number2) {
          return 0;
        }

        int multiply (int number1, int number2) {
          return 0;
        }

        int divide (int number1, int number2) {
          return 0;
        }
};
```

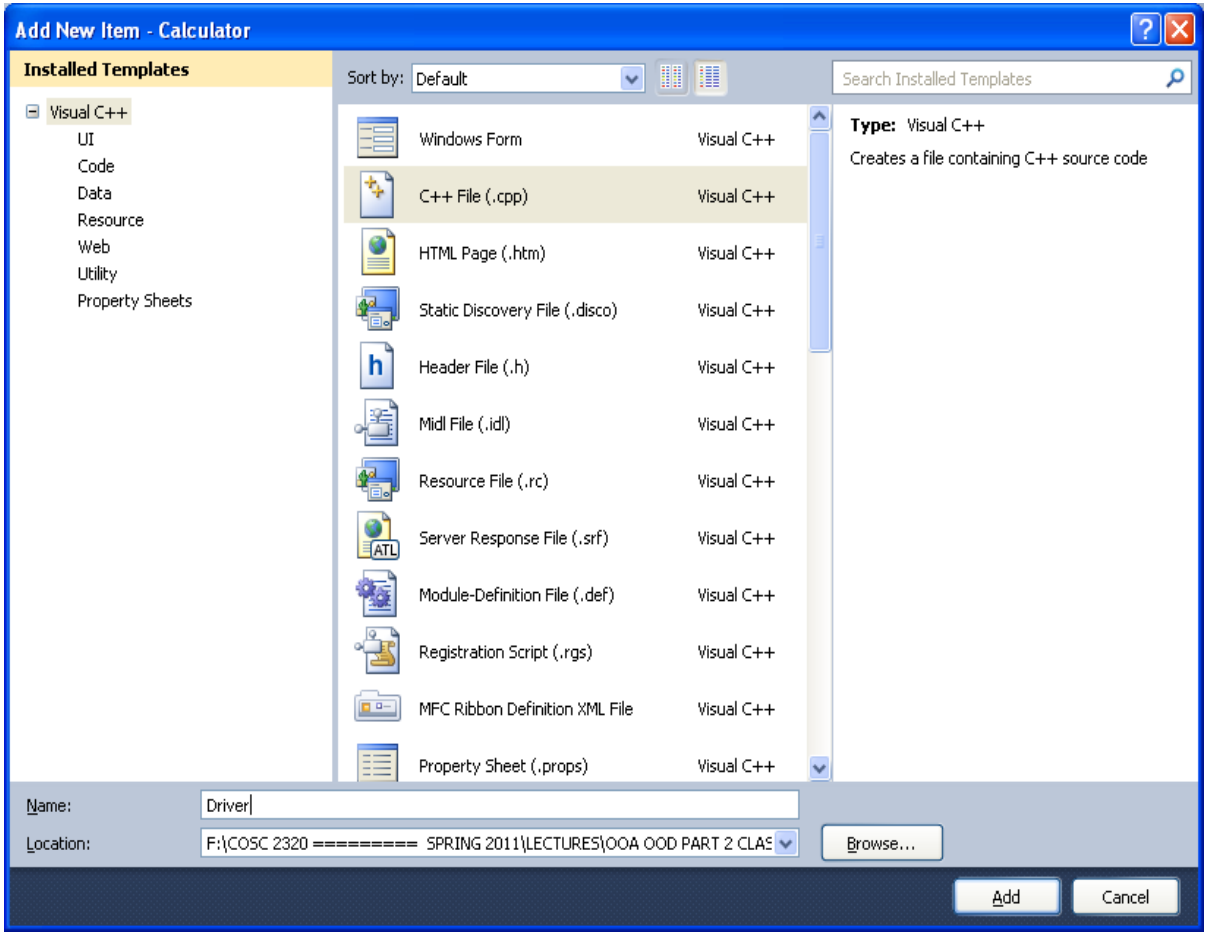


Type Class Calculator code:

```
class Calculator
{
        public:
        int add (int number1, int number2)
        {
          return 0;
        }

        int subtract (int number1, int number2)
        {
          return 0;
        }

        int multiply (int number1, int number2){

          return 0;
        }

        int divide (int number1, int number2)
        {
          return 0;
        }
};
```

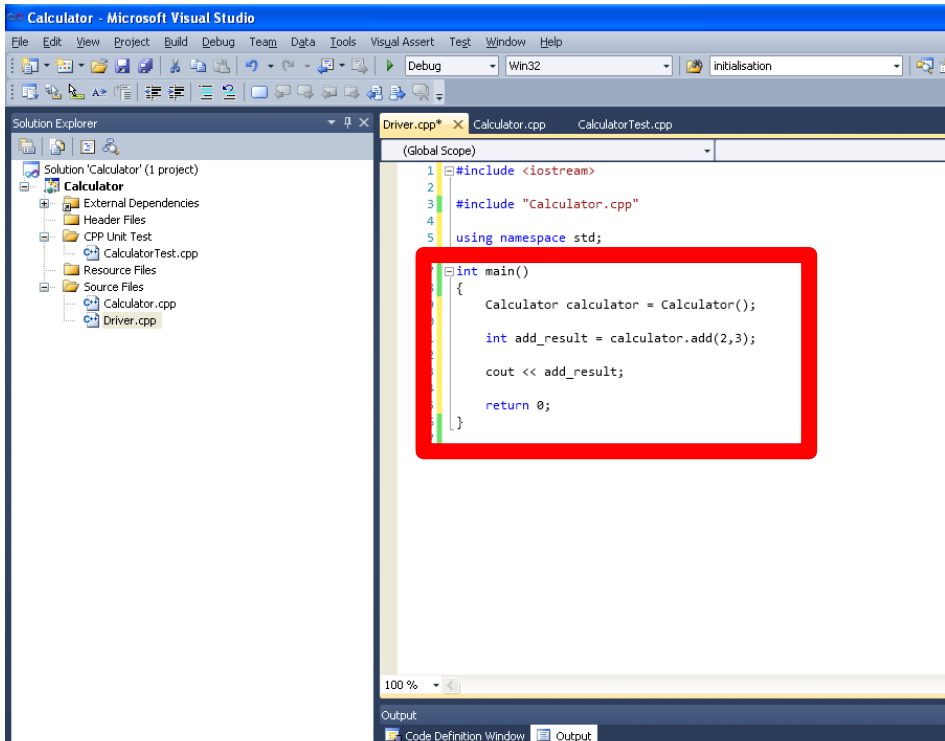


Create a Driver.cpp file in the Calculator project:



Type Driver code:

```
#include <iostream>

#include "Calculator.cpp"

using namespace std;

int main()
{
	Calculator calculator = Calculator();

	int add_result = calculator.add(2,3);

	cout << add_result;

	return 0;
	}
```

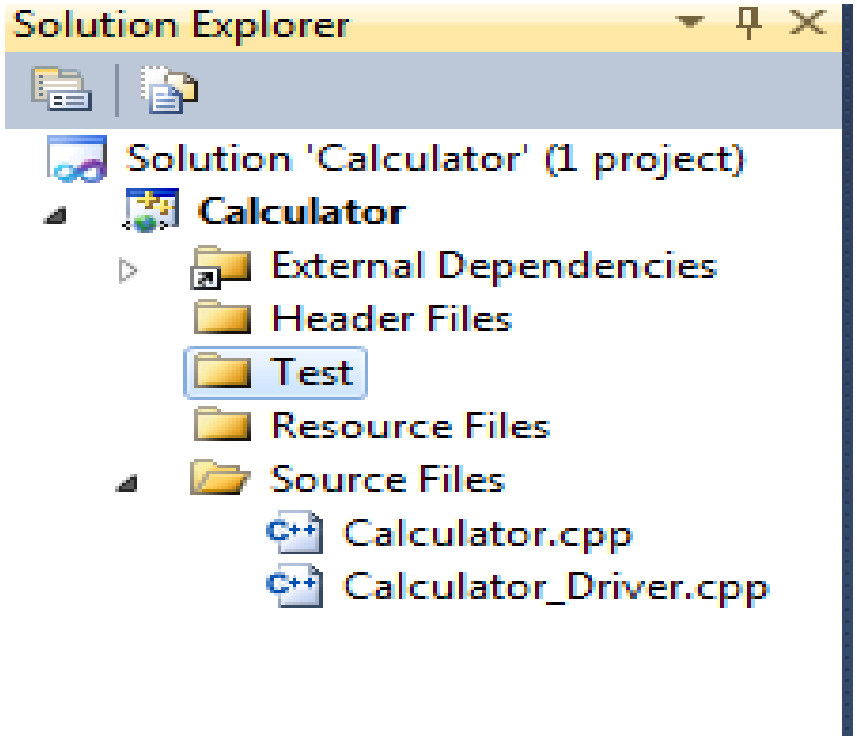


Now add a new filter to the project and name it “**Test**”. **Right click on Project**, click “New Filter” and Write Name “**Test**”.

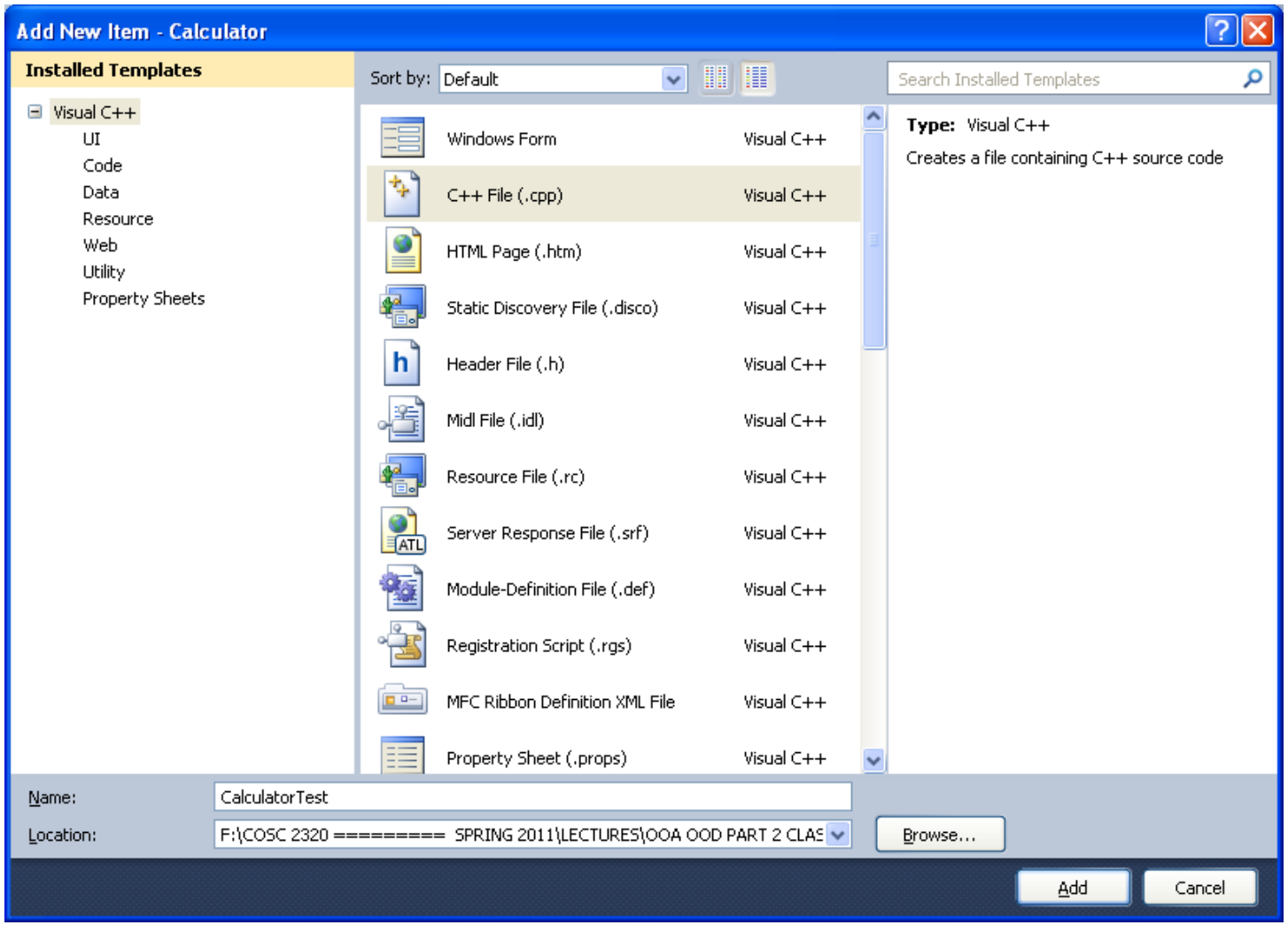
This filter will contain all the tests for the current project.

This filter has been added just to keep the testing code separate from the source code.

The test file can also be added into the source folder and it would still work correctly.



Now right click on this **Test** Fixture and add a new **CalculatorTest.cpp** file for unit testing.

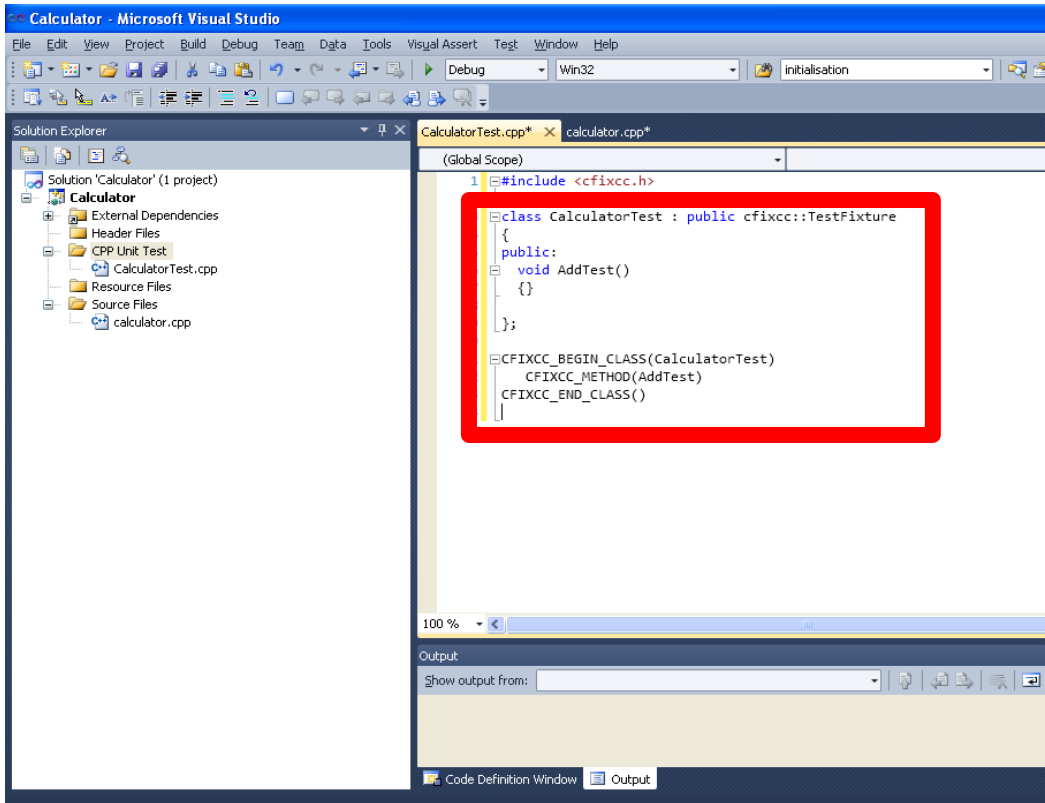


Add the following code in the file **CalculatorTest.cpp** to test the **Add** method:

```
#include <cfixcc.h>

class CalculatorTest : public cfixcc::TestFixture
{
  public:
  void AddTest()
  {}

};

CFIXCC_BEGIN_CLASS(CalculatorTest)
   CFIXCC_METHOD(AddTest)
CFIXCC_END_CLASS
```

Add the following code in the file CalculatorTest.cpp to test the Add method:

```
#include <cfixcc.h>

class CalculatorTest : public cfixcc::TestFixture
{
public:
  void AddTest()
  {}

};

CFIXCC_BEGIN_CLASS(CalculatorTest)
   CFIXCC_METHOD(AddTest)
CFIXCC_END_CLASS
```



Add the following code to **AddTest** method:

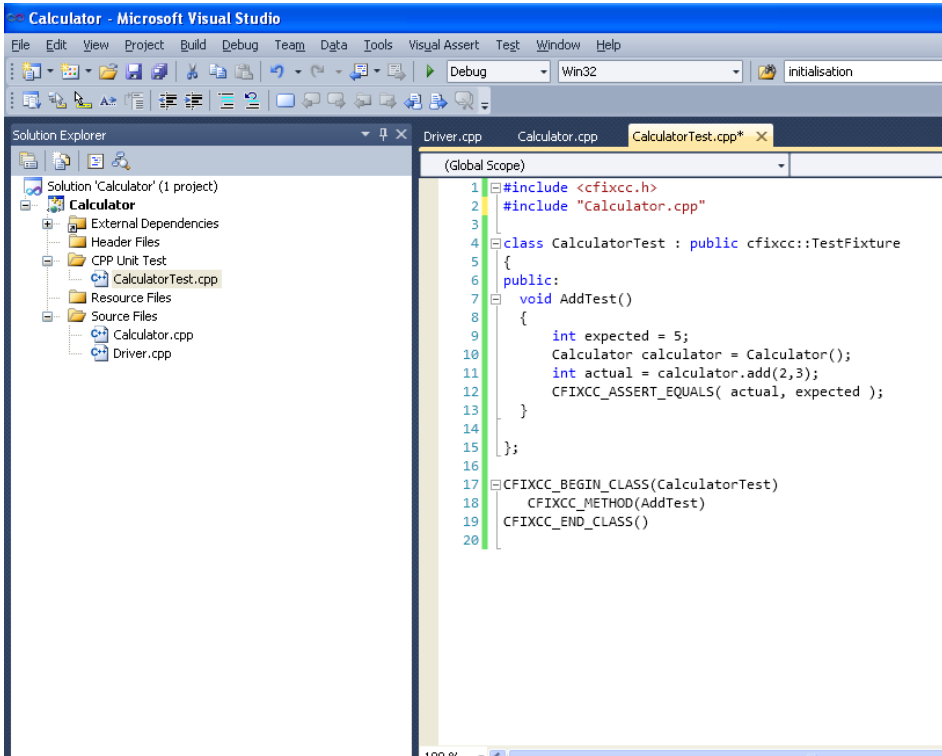
```
#include <cfixcc.h>
#include “Calculator.cpp “

class CalculatorTest : public cfixcc::TestFixture
{
  public:
  void AddTest()
  {
        int expected = 5;
        Calculator calculator = Calculator();
        int actual = calculator.add(2,3);
        CFIXCC_ASSERT_EQUALS( actual, expected );
   }

};

CFIXCC_BEGIN_CLASS(CalculatorTest)
  CFIXCC_METHOD(AddTest)
CFIXCC_END_CLASS
```

API for use in test cases (C++ only)

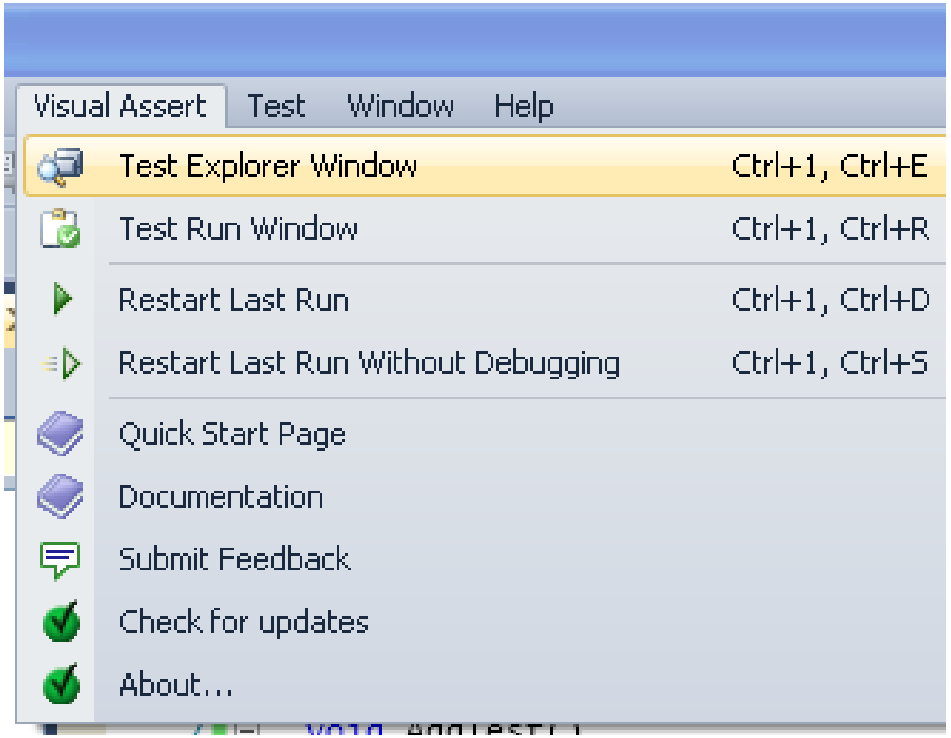
CFIXCC_ASSERT_EQUALS
CFIXCC_ASSERT_EQUALS_MESSAGE
CFIXCC_ASSERT_NOT_EQUALS
CFIXCC_ASSERT_NOT_EQUALS_MESSAGE
CFIXCC_ASSERT_LESS[_OR_EQUAL]
CFIXCC_ASSERT_LESS[_OR_EQUAL]_MESSAGE
CFIXCC_ASSERT_GREATER[_OR_EQUAL]
CFIXCC_ASSERT_GREATER[_OR_EQUAL]_MESSAGE

Add the following code to **AddTest** method:

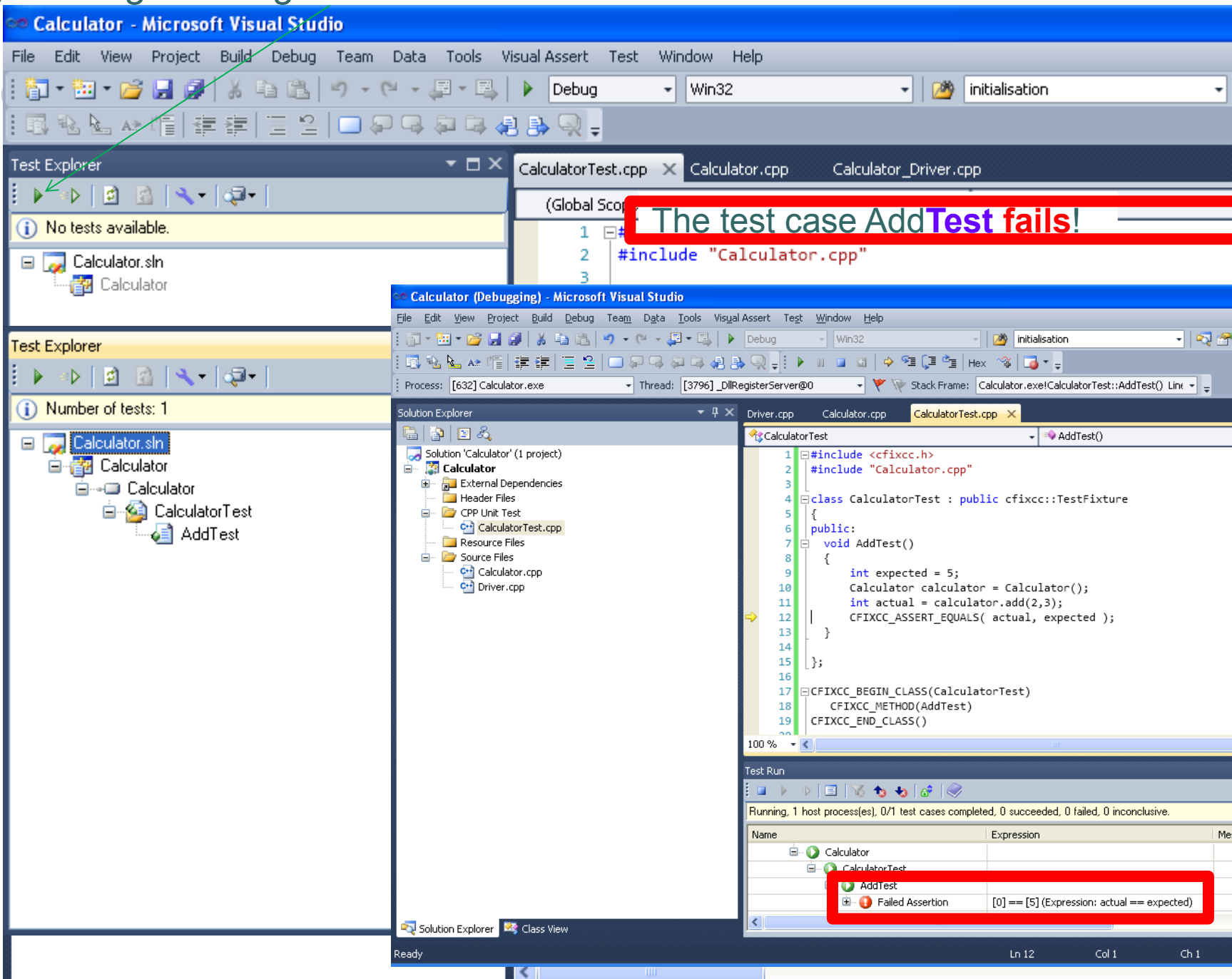
```
#include <cfixcc.h>
#include “Calculator.cpp “

class CalculatorTest : public cfixcc::TestFixture
{
public:
  void AddTest()
  {
        int expected = 5;
        Calculator calculator = Calculator();
        int actual = calculator.add(2,3);
        CFIXCC_ASSERT_EQUALS( actual, expected );
  }

};

CFIXCC_BEGIN_CLASS(CalculatorTest)
    CFIXCC_METHOD(AddTest)
CFIXCC_END_CLASS
```

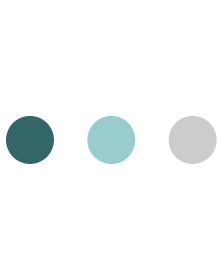


Now open VisualAssert -> Test Explorer Window



Run by clicking on the green button





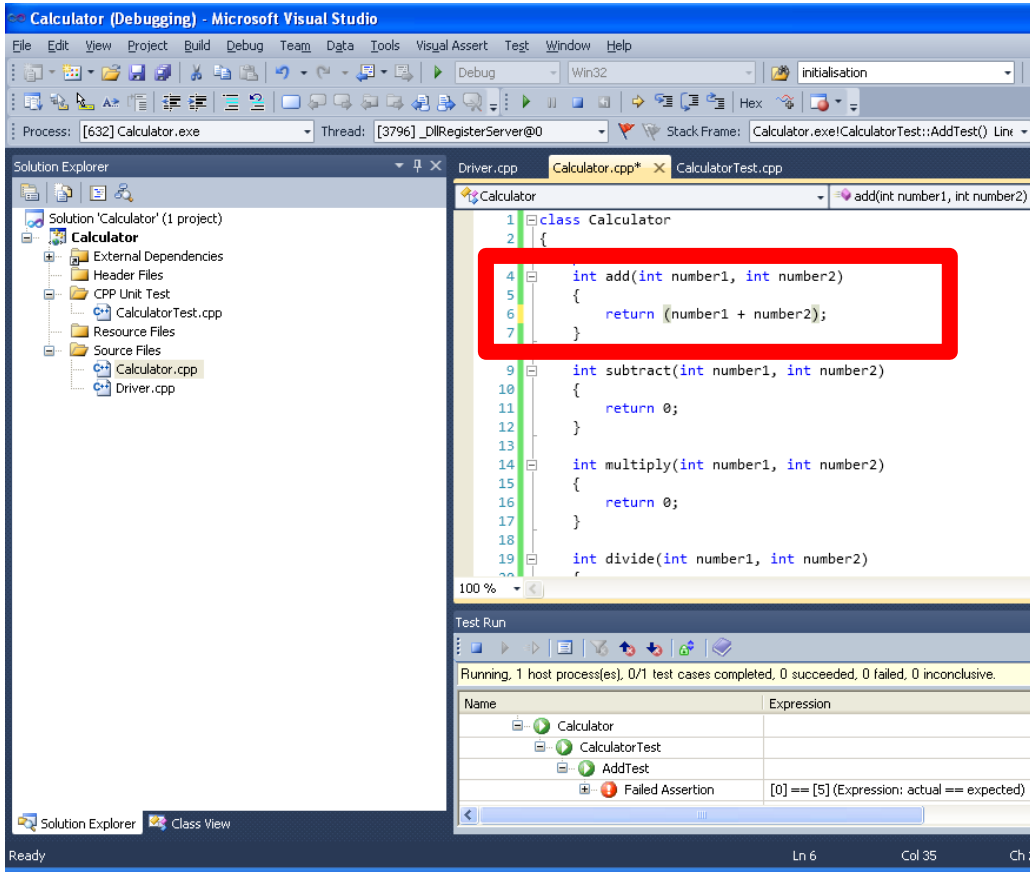
TDD Stages

5. Write **just enough** **method code** to get the **test** to **pass**

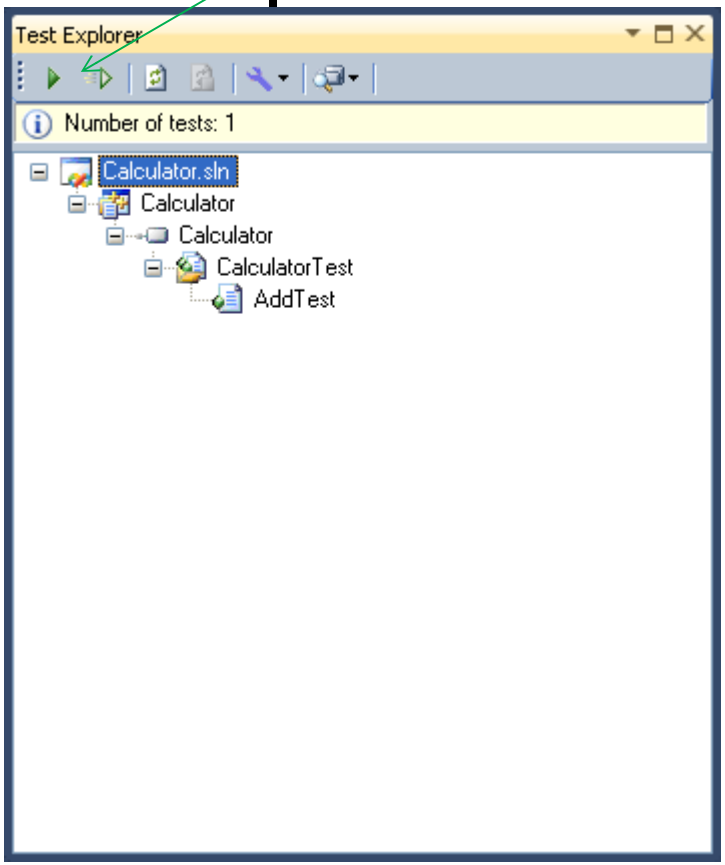
What do we need to do?

5. Write just enough method code to get the test to pass

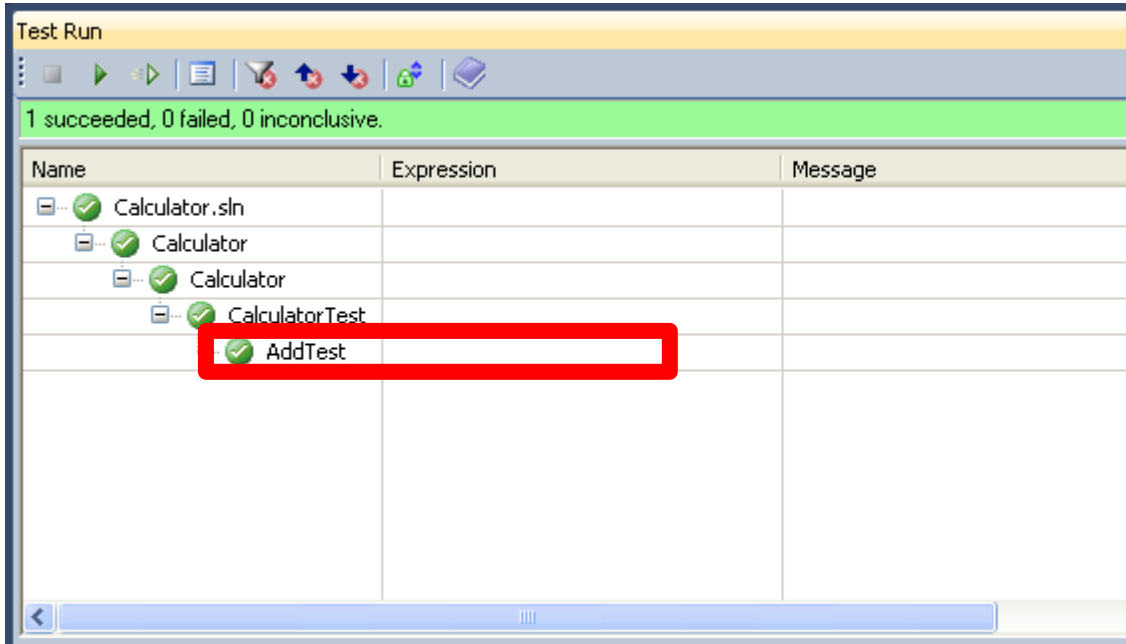
```
class Calculator {

        public:
        int add (int number1, int number2) {
            return (number1 + number2);
        }

        int subtract (int number1, int number2) {
          return 0;
        }

        int multiply (int number1, int number2) {

          return 0;
        }

        int divide (int number1, int number2) {
          return 0;
        }
};
```



Run by clicking on the green button



The test case **AddTest passes**!



NUnit - C# Unit Testing

Ruby on Rails Unit Testing

```
class Calc
    def add
    end
end
```

```
require 'test/unit'
require 'calc'
class TestAdd < Test::Unit::TestCase
    def test_add
        calc = Calc.new
        expected = calc.add 3,2
        assert_equal expected, 5
    end
end
```

5. Write **just enough** method code to get the test to pass

The test case **AddTest passes**!

```
$ ruby test_add.rb
Loaded suite test_add
Started
F
Finished in 0.01038 seconds.

    1) Failure:
test_add(TestAdd) [test_add.rb:10]:
<nil> expected but was
<5>.

1 tests, 1 assertions, 1 failures, 0
    errors
```

PyUnit - Python Unit Testing

From 4:20 to 5:00 – 40 minutes.

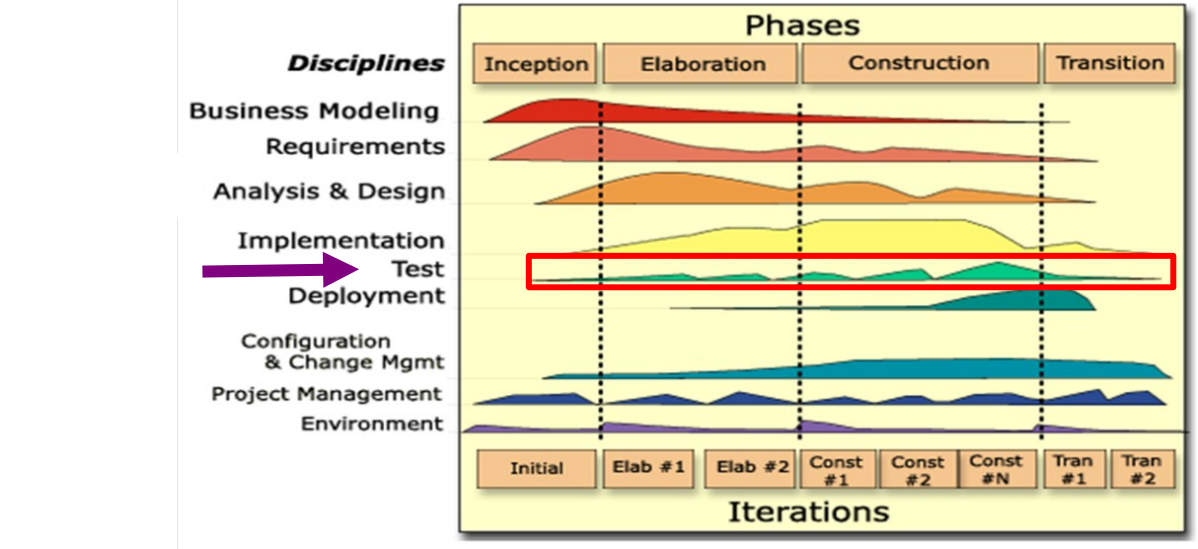
11.15.2023

(W 4 to 5:30)

Lecture 9: Testing

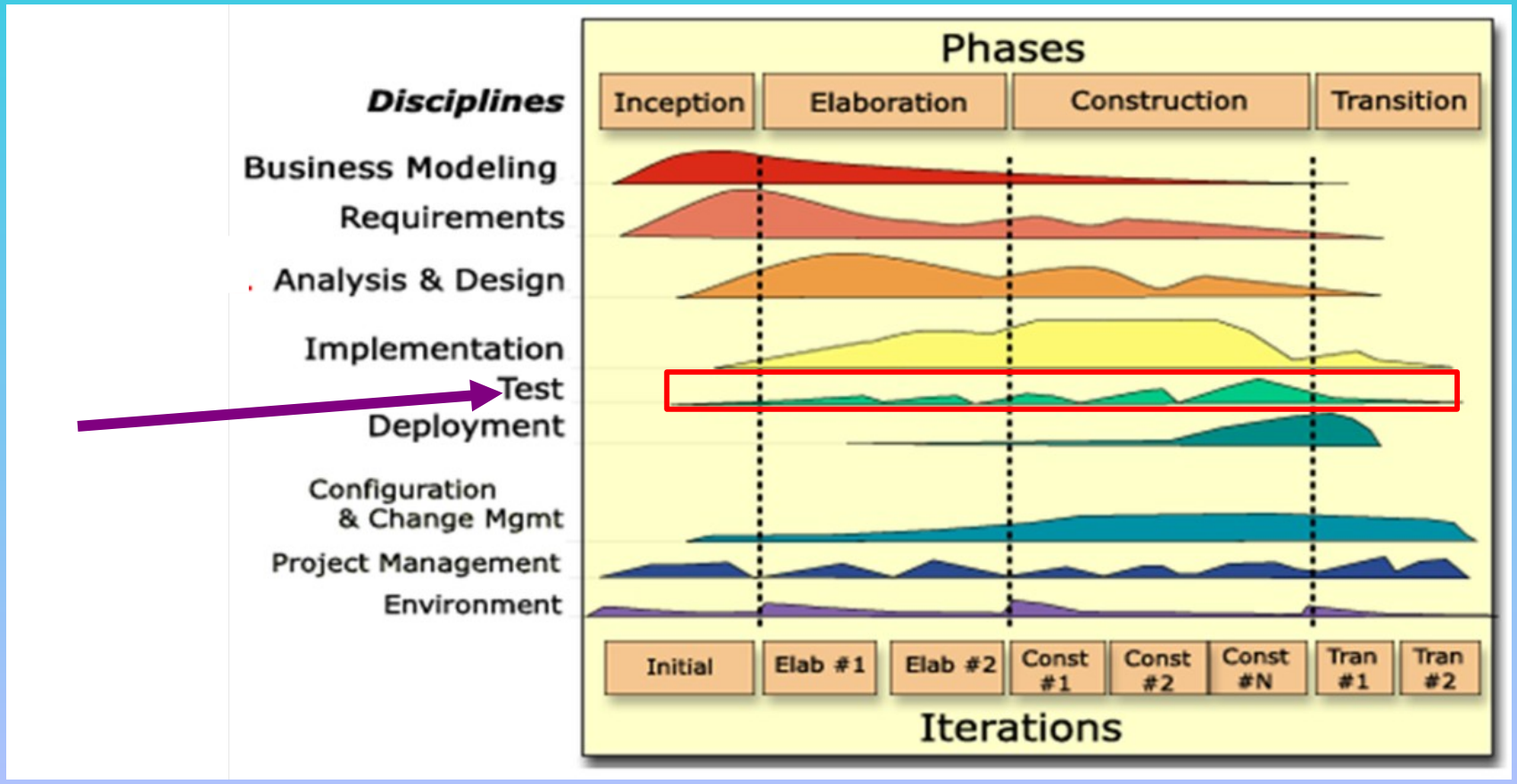
Tutorial 6 TDD

(25)



TESTING

Unified Process



Overview

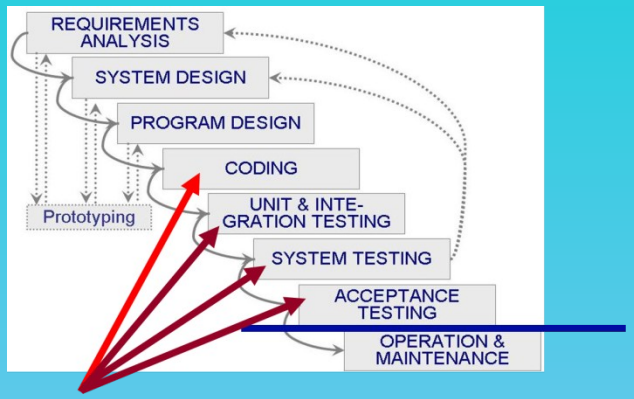
- The **Test Workflow**: Implementation
- **Test case selection**
- Black-Box unit-testing techniques
- Glass-Box unit-testing technique
- Code **Walkthroughs** and **Inspections**
- Comparison of **U**nit **T**esting Techniques

- **Potential problems when Testing objects**
- Management aspects of **U**nit **T**esting

EXAM 4 REVIEW

Overview

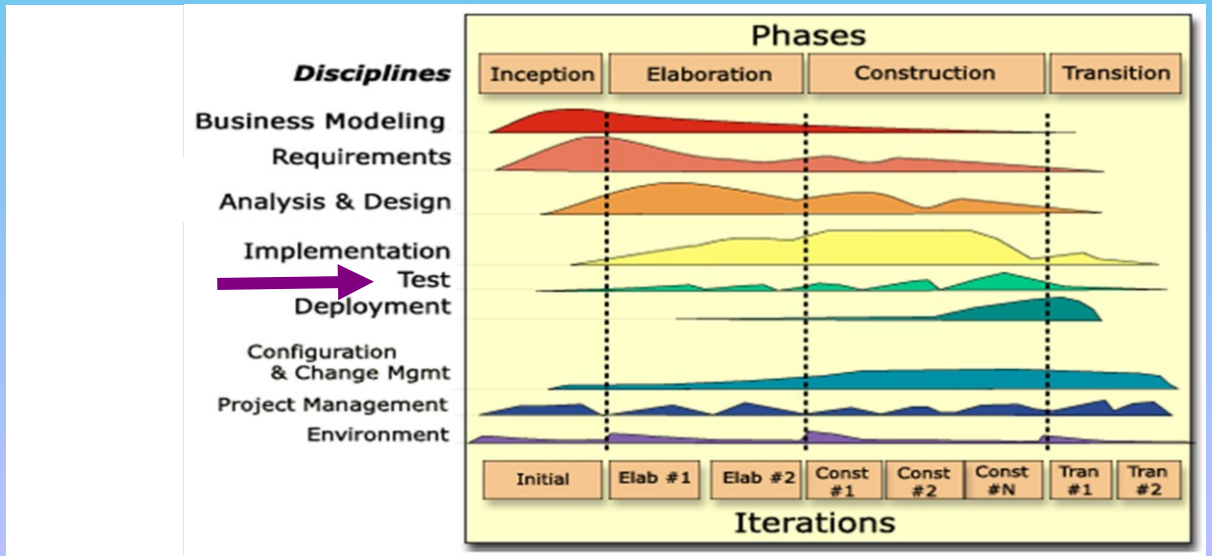
- When to rewrite rather than debug a **module**
- **Integration T**esting
- Product (**System**) **T**esting
- **A**cceptance **T**esting



The Implementation Workflow

- Once the **Programmer** has **Implemented** an artifact/ **module**, **he** or **she** ***Unit Tests*** it

- Then the **module** is passed on to the **SQA Group** for further **Testing**
  - This **Testing** is part of the **Test Workflow**



Test Case Selection

- Worst way — **random Testing**
  - There is **no time to test all** but the tiniest fraction of all possible **Test Cases**, totaling perhaps 10^{100} or more

- We need a **systematic** way **to construct Test Cases**

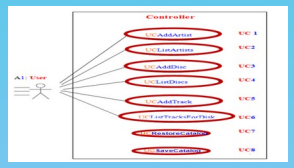
Testing to **Specifications** versus Testing to **Code**

There are two extremes to **Testing**

Test** to **Specifications (also called Black-Box, data-driven, functional, or input/output driven **Testing**)

- Ignore the **Code** — use the **Specifications** to select Test Cases

Black Box Test Cases



Test** to **Code (also called Glass-Box, logic-driven, structured, or path-oriented **Testing**)

- Ignore the ***Specifications*** — use the **Code** to select Test Cases

White Box Test Cases

EXAM 4 REVIEW

Feasibility of Testing to **Specifications**

- Example:
  - The **Specifications** for a **data processing product** include 5 types of commission and 7 types of discount
  - **35 Test Cases**

Feasibility of Testing to **Specifications**

Suppose the **Specifications** include **20 factors**, each taking on **4 values**

- There are **4^{20}** or **1.1×10^{12} Test Cases**
- If each takes **30 seconds to run**, running all **Test Cases** takes more than **1 million years**

The combinatorial explosion makes Testing to Specifications impossible

Feasibility of Testing to **Code**

Each **Path** through a **artifact** must be executed at least once

- Combinatorial explosion

Feasibility of Testing to Code

- **Code** example:

```
read (kmax)                          // kmax is an integer between 1 and 18
for (k = 0; k < kmax; k++) do
{
   read (myChar)                     // myChar is the character A, B, or C
   switch (myChar)
   {
      case 'A':
         blockA;
         if (cond1) blockC;
         break;
      case 'B':
         blockB;
         if (cond2) blockC;
         break;
      case 'C':
         blockC;
         break;
   }
   blockD;
}
```

Figure 14.9

Feasibility of Testing to Code

- The flowchart has over 10^{12} different paths

```
read (kmax)                          // kmax is an integer between 1 and 18
for (k = 0; k < kmax; k++) do
{
   read (myChar)                     // myChar is the character A, B, or C
   switch (myChar)
   {
      case 'A':
         blockA;
         if (cond1) blockC;
         break;
      case 'B':
         blockB;
         if (cond2) blockC;
         break;
      case 'C':
         blockC;
         break;
   }
   blockD;
}
```

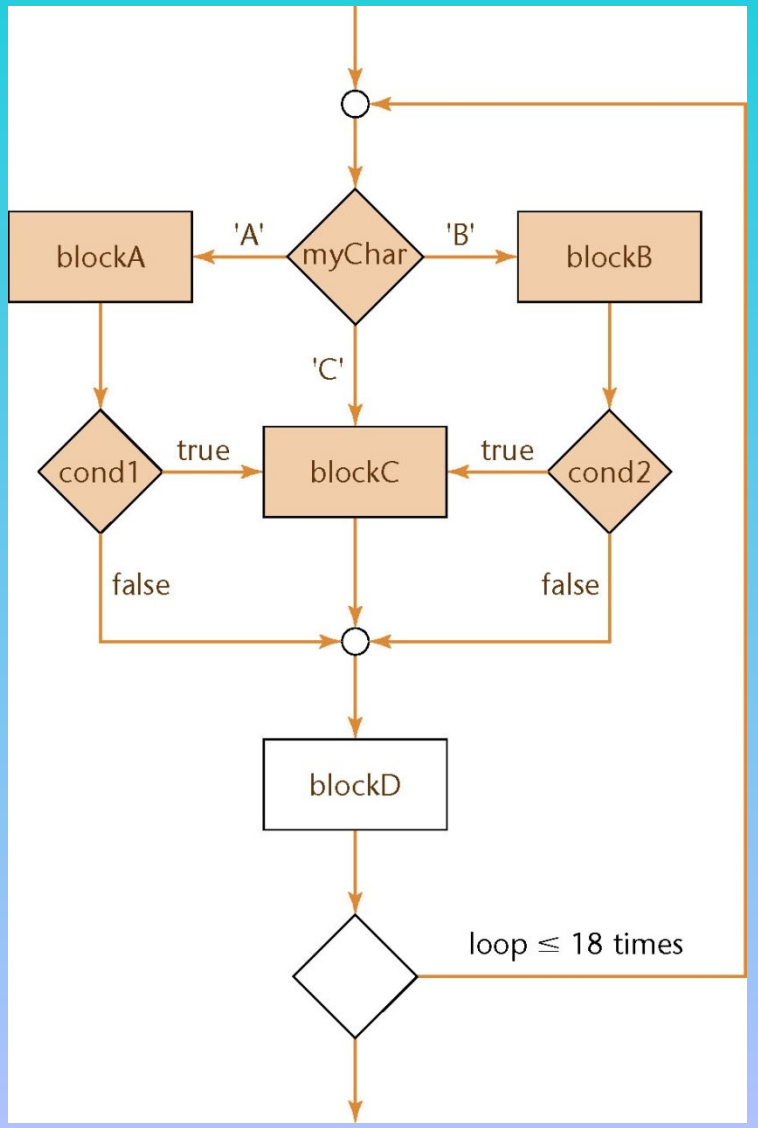


Figure 14.10

Feasibility of Testing to Code

- **Testing** to **Code** is **not Reliable**

```
if ((x + y + z)/3 == x)
   print "x, y, z are equal in value";
else
   print "x, y, z are unequal";
```

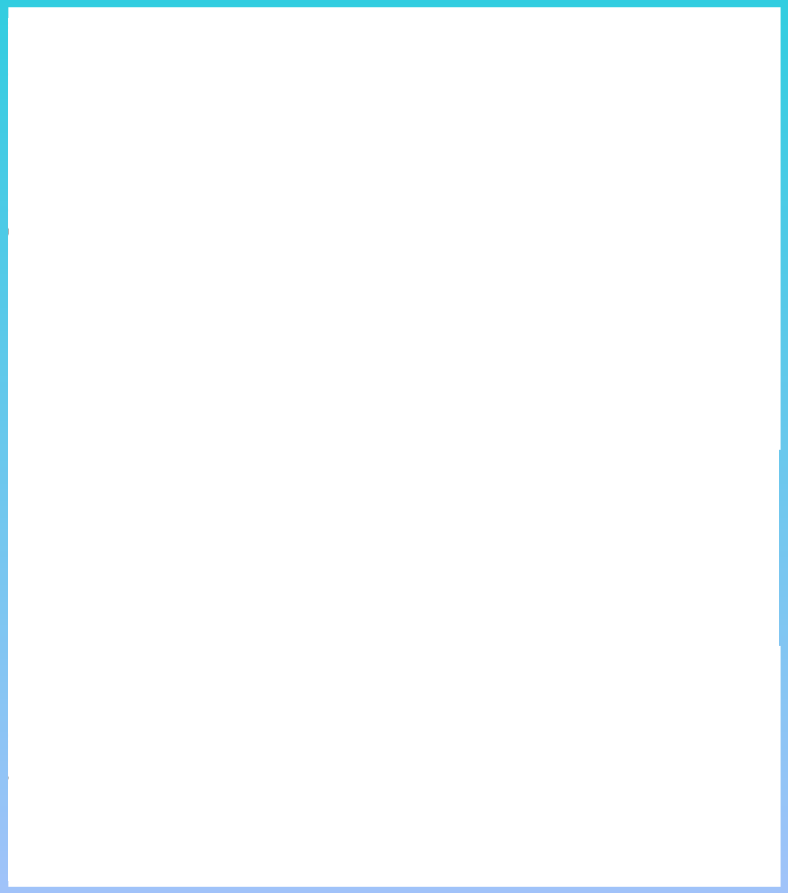
2 1

Anything wrong?

- **We can exercise every Path without detecting every fault**

Feasibility of Testing to Code

- A **Path** can be **Tested** only **if it is present**



- A **Programmer** who **omits the test for (d == 0)** in the **Code** probably is unaware of the possible danger

Feasibility of Testing to **Code**

- Criterion "exercise all **Paths**" is not ***Reliable***
  - **Products** exist for which some **data** exercising a given **Path** **detect** a **fault**, and other **data** exercising the same **Path** **do not**

- Black-Box Unit-Testing techniques
- Glass-Box Unit-Testing technique
- Code **Walkthroughs** and **Inspections**

Black Box Unit Testing Techniques

Neither **exhaustive Testing to Specifications** nor **exhaustive Testing to Code is feasible**

The art of **Testing**:

- Select a small, manageable set of **Test Cases** to
- **Maximize** the chances of detecting a **fault**, while
- **Minimizing** the chances of wasting a **Test Case**

Every **Test Case must detect a previously undetected fault**

EXAM 4 REVIEW

Equivalence Testing and Boundary Value Analysis

Example

- The **Input Specifications** for **a DBMS** state that **the produc** must handle any number of records between 1 and 16,383 $(2^{14} - 1)$

If the system can handle 34 records and 14,870 record **then it** probably will handle fine for 8,252 records

If the system works for any one **Test Case** in the range (**1..16,383**), **then** it will probably work for any other **Test Case** in the range

- Range (1..16,383) constitutes an ***Equivalence Class***

EXAM 4 REVIEW

Equivalence Testing

Any one member of an **Equivalence Class** is as good a **Test Case** as any other member of the **Equivalence Class**

Range (1..16,383) defines three different **Equivalence Classes**:

- **Equivalence Class 1**: Fewer than **1** record
- **Equivalence Class 2**: Between **1** and **16,383** records
- **Equivalence Class 3**: More than **16,383** records

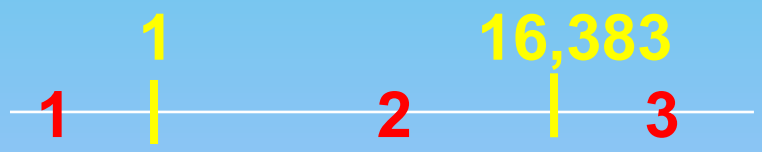
Boundary Value Analysis

Select **Test Cases** on or just to one side of the boundary of **Equivalence Classes**

- This greatly increases the probability of detecting a **fault**

Database Example

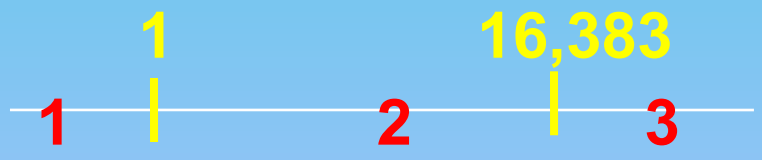
- Test case 1: 0 records — Member of **Equivalence Class 1** and adjacent to boundary value
- Test case 2: **1** record — Boundary value
- Test case 3: 2 records — Adjacent to boundary value
- Test case 4: 723 records — Member of **Equivalence Class 2**



Equivalence Class 1: Fewer than **1** record
Equivalence Class 2: Between **1** and **16,383** records
Equivalence Class 3: More than **16,383** records

Database Example

- Test case 5: 16,382 records Adjacent to boundary value
- Test case 6: **16,383** records Boundary value
- Test case 7: 16,384 records Member of **Equivalence Class 3** and adjacent to boundary value



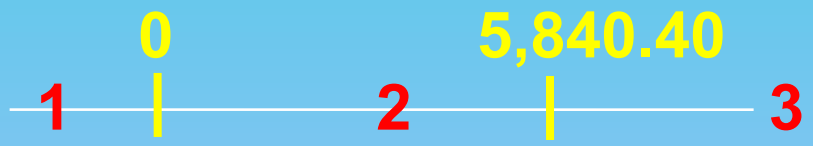
Equivalence Class 1: Fewer than **1** record
Equivalence Class 2: Between **1** and **16,383** records
Equivalence Class 3: More than **16,383** records

Equivalence Testing of Output Specifications

- We also need to perform **Equivalence Testing** of the **Output Specifications**

- Example:

  In 2006, the minimum Social Security (OASDI) deduction from any one paycheck was **$0.00**, and the maximum was **$5,840.40**



  - **Test Cases must include Input Data that should result in deductions of exactly $0.00** and **exactly $5,840.40**

  - Also, **Test Data that** might result in deductions of **less than $0.00** or **more than $5,840.40**

Overall Strategy

- **Equivalence Classes** together with **Boundary Value** analysis to **Test** both **Input Specifications** and **Output Specifications**
  - This approach generates a **small set of Test Data** with the potential of **uncovering a large number** of **faults**

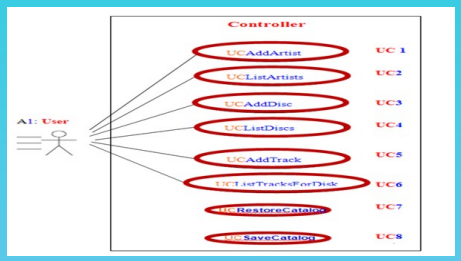
BBTC

Functional Testing

An **alternative form of** Black Box **Testing** for **software**

- We base the **Test Data** on the **functionality** of the **Code** artifacts

UML USE CASE MODEL



Each item of functionality or **function** is identified

Test Data are devised to **Test** each (lower-level) **function** separately

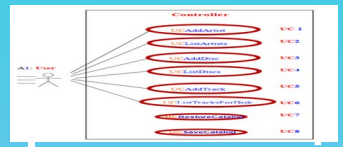
Then, higher-level **functions** composed of these lower-level **functions** are **Tested**

Functional Testing

In practice, **however**

- Higher-level functions are not always neatly constructed out of lower-level functions
- Instead, the **lower-level functions are often intertwined**

High COUPLING



Also, **functionality boundaries** do not always coincide with **Code artifact boundaries**

- The distinction between **Unit Testing** and **Integration Testing** becomes blurred
- This problem also can arise in the **Object Oriented Paradigm** when **messages** are passed between **objects**

Not MVC

Functional Testing

The resulting **random interrelationships between Code artifacts** can have negative consequences for **management**

- **Milestones** and **Deadlines** can become **ill-defined**
- The **Status** of the **Project** then becomes **hard to determine**

High COUPLING

- Black-Box Unit-Testing techniques
- Glass-Box Unit-Testing technique
- Code **Walkthroughs** and **Inspections**

Glass Box Unit Testing Techniques

- We will examine
  - Statement **Coverage**
  - Branch **Coverage**
  - Path **Coverage**
  - All-Definition-Use Path **Coverage**

- ***Statement Coverage***:
  - Running a set of **Test Cases** in which every **Statement** is executed at least once
  - **A CASE Tool needed to keep track**

```
if (s > 1 && t == 0)
  x = 9;
```

Test case: s = 2, t = 0.

Figure 14.15

Structural Testing: **Branch** Coverage

- Running a set of **Test Cases** in which every **Branch** is executed at least once (as well as all **Statements**)
  - Again, a **CASE Tool is needed**

```
if (s > 1 && t == 0)
   x = 9;
```

Test case: s = 2, t = 0.

Structural Testing: Path Coverage

Running a set of **Test Cases** in which every **Path** is executed at least once (as well as all statements)

Problem:

- The number of **Paths** may be very large

We want a **Weaker Condition** than all **Paths** but that shows up **more faults** than **Branch Coverage**

All-Definition-Use **Path Coverage**

Each occurrence of **variable**, `zz` say, is labeled either as

- The ***Definition*** of a **variable**

  `zz = 1` or `read (zz)`

- or the ***Use*** of **variable**

  `y = zz + 3` or `if (zz < 9) errorB ()`

Identify all **Paths** from the **Definition** of a **variable** to the **Use** of that **Definition**

This can be done by an automatic **CASE Tool**

A **Test Case** is set up for each such **Path**

All-Definition-Use- **Path** Coverage

Disadvantage:

- Upper bound on number of **Paths** is 2^d, where d is the number of **Branches**

In practice:

- The actual number of **Paths** is proportional to d

This is therefore a practical Test Case selection technique

Infeasible Code

- It may **not** be possible to Test a specific **Statement**
  - We may have an **infeasible Path** ("dead **Code**") in the artifact
- Frequently this is evidence of a **fault**

Figure 14.16

Complexity Metrics

- A **Quality** Assurance approach to Glass Box **Testing**
- **Artifact** `m1` is more "Complex" than **Artifact** `m2`
  - Intuitively, `m1` is more likely to have **faults** than artifact `m2`
- If the Complexity is unreasonably high, reDesign and then reImplement that **Code artifact**
  - This is **cheaper** and **faster** than trying to **debug** a **fault-prone Code Artifact**

Lines of Code

- The simplest measure of **Complexity**
  - Underlying assumption: There is a constant probability ***p*** that a Line of Code contains a **fault**
- Example
  - The **Tester** believes each Line of Code has a **2%** chance of containing a **fault**.
  - If the **Artifact** under Test is **100 Lines of Code long**, then it is expected to contain **2 faults**
- The number of **faults** is indeed related to the **size** of **the product** as a whole

Other Measures of **Complexity**

- **Cyclomatic Complexity *M*** (McCabe)
  - Essentially the number of decisions (**Branches**) in the **Artifact**
  - Easy to compute
  - A surprisingly **good measure** of **faults**

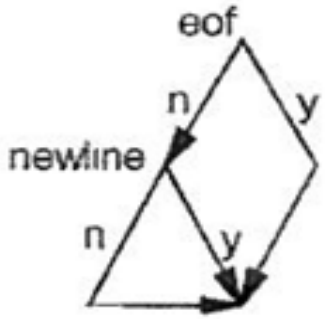
- In one experiment, Artifacts with ***M* > 10** were shown to have statistically **more errors**

EXAM 4 REVIEW

Cyclomatic Complexity *M* (McCabe)

The flowchart for one of the four procedures in Naur's text processing problem (see 6.5.2 on page 166) is shown below:

get_character / getCharacter

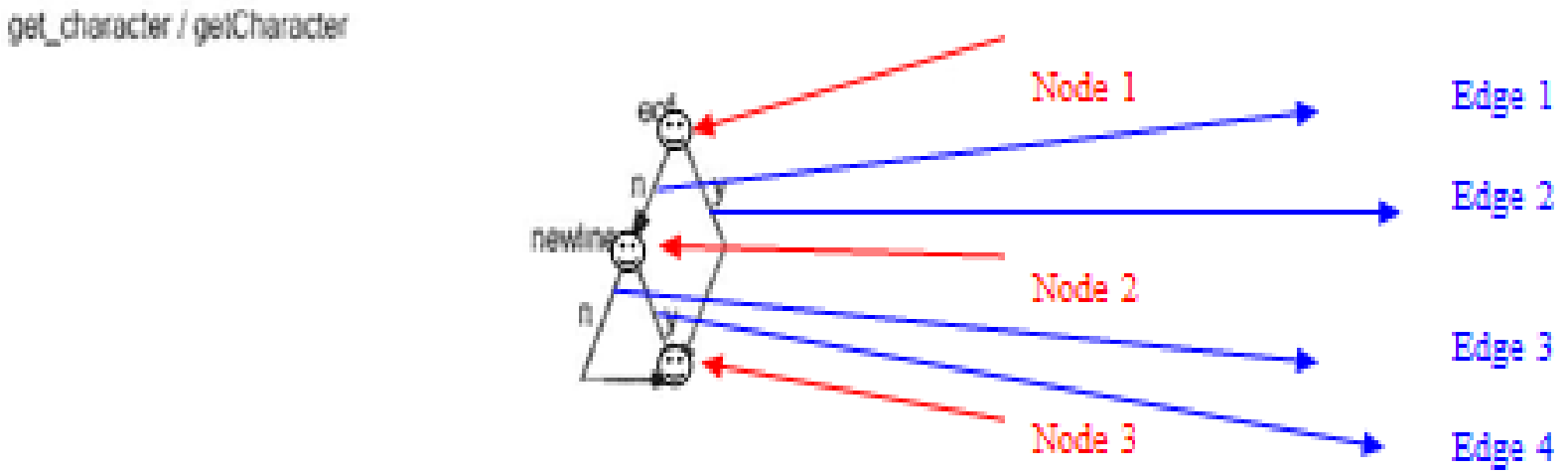


Determine the cyclomatic complexity. If you are unable to determine the number of branches, consider the flowchart as a directed graph. Determine the number of edges **e**, nodes **n** for the components **c** (**c = 1**). The cyclomatic complexity **M** is then given by the formula:

$$M = \square - n + c$$

Is this high compared to acceptable value of **M**?

Cyclomatic complexity *M* (McCabe)



Determine the cyclomatic complexity. If you are unable to determine the number of branches, consider the flowchart as a directed graph. Determine the number of edges **e**, nodes **n** for the components **c** (**c** = 1). The cyclomatic complexity **M** is then given by the formula:

$$M = e - n + c = 4 - 3 + 1 = 2$$

Is this high compared to acceptable value of **M**?

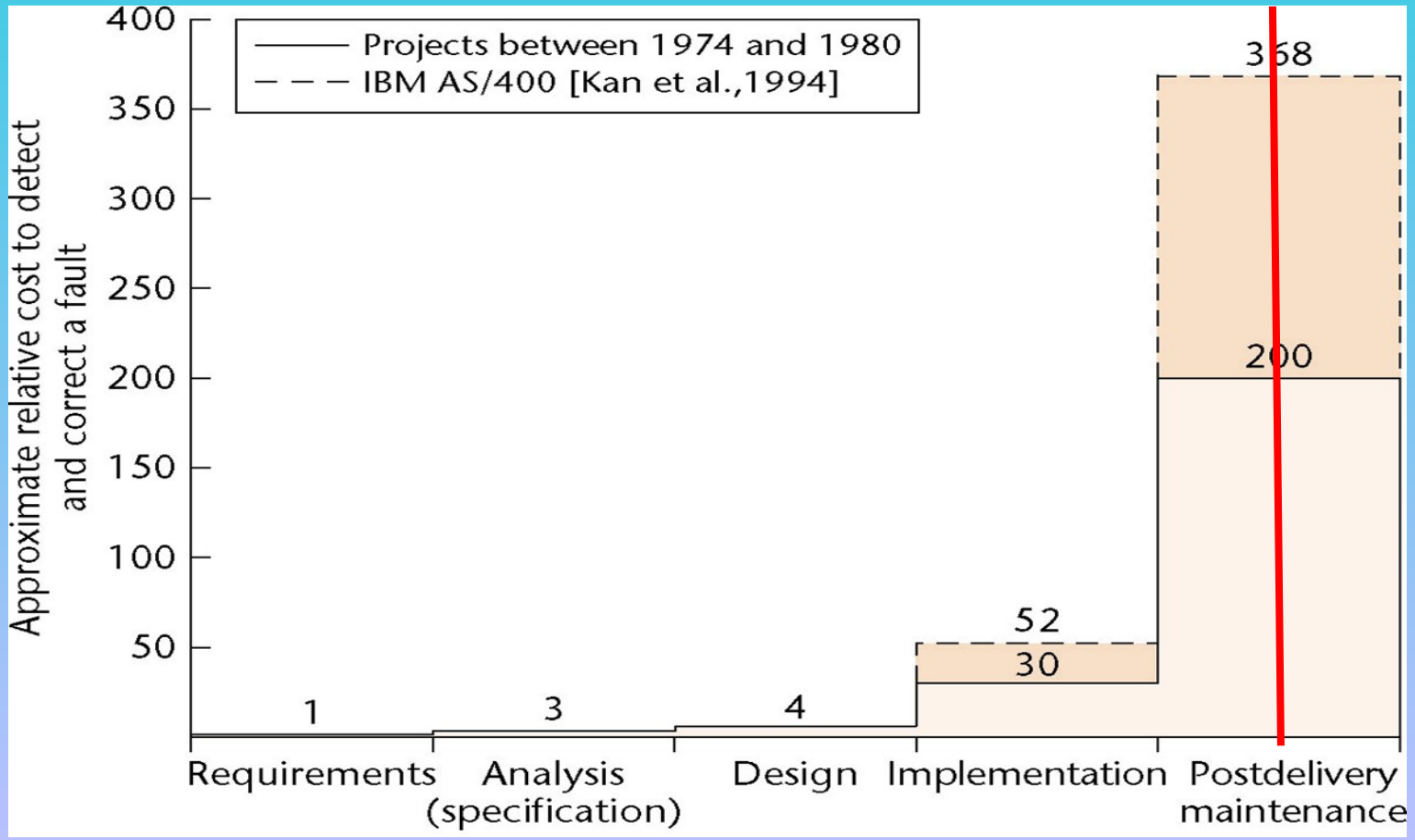
LOW

- Black-Box Unit-Testing techniques
- Glass-Box Unit-Testing technique
- Code **Walkthroughs** and **Inspections**

Code Walkthroughs and Inspections

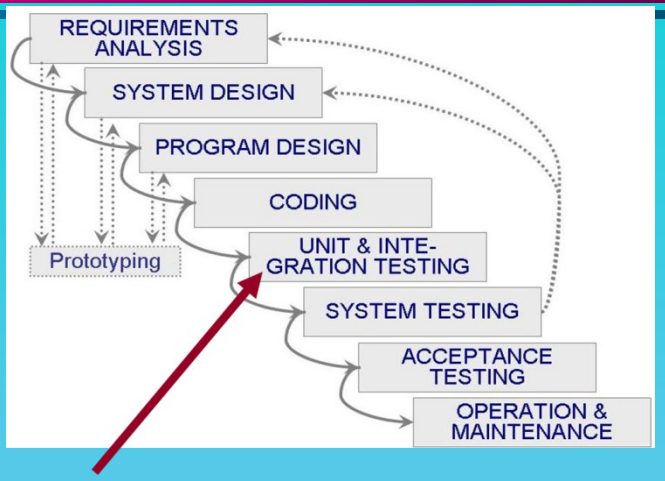
Code Reviews lead to rapid and thorough **fault** detection

- Up to 95% reduction in **Maintenance costs**



Comparison of Unit Testing Techniques

- Experiments comparing
  - **B**lack **B**ox **Testing**
  - **G**lass **B**ox **Testing**
  - **Code Inspections**



- [Myers, 1978] **59 highly experienced Programmers**
  - All three methods were **equally effective** in finding **faults**
  - **Code Inspections** were **less cost-effective**

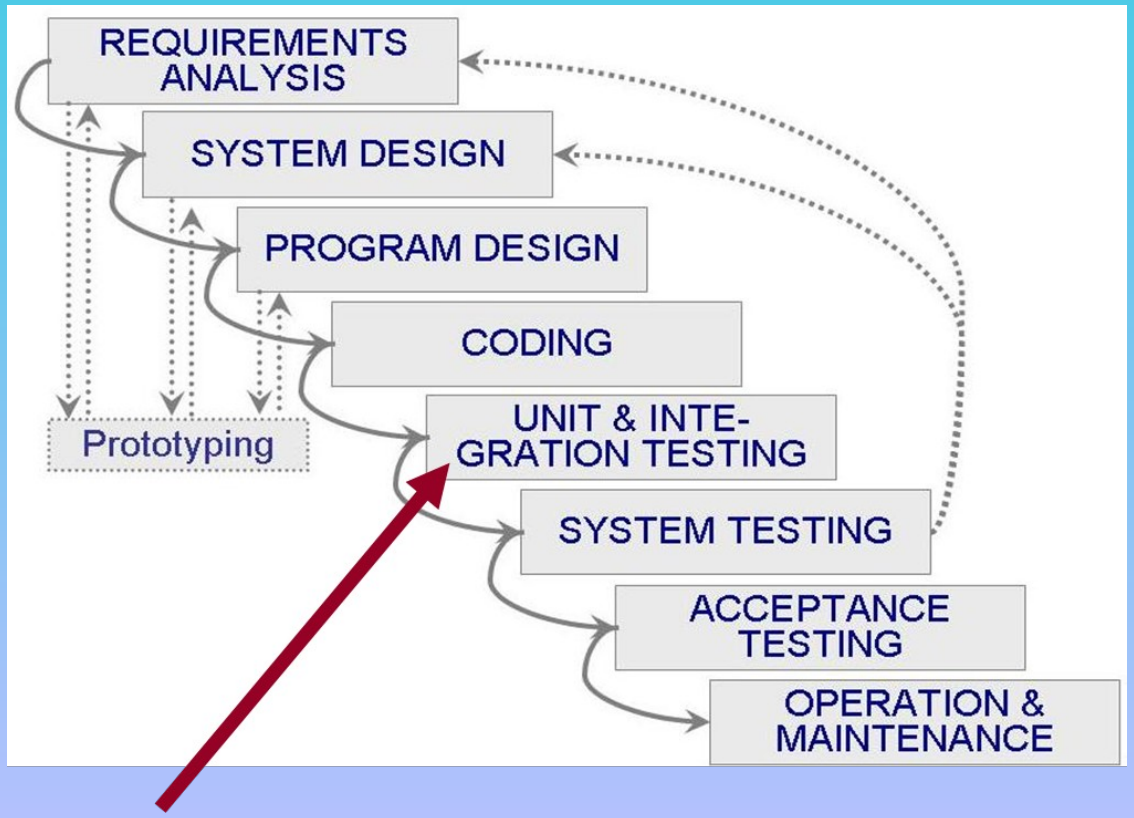
- [Hwang, 1981]
  - All three methods were **equally effective**

Comparison of Unit Testing Techniques

- [Basili and Selby, 1987] 42 advanced Students in two groups, 32 professional Programmers
- Advanced Students, group 1
  - No significant difference between the three methods
- Advanced Students, group 2
  - Code reading and Black Box Testing were equally good
  - Both outperformed Glass Box Testing
- Professional Programmers
  - Code reading detected more faults
  - Code reading had a faster fault detection rate

Comparison of Unit Testing Techniques

- Conclusion
  - ***Code Inspection** is at least **as successful** at **detecting faults** as Glass Box **and** Black Box **Testing***



22 software development trends for 2022

https://blog.codacy.com/software-development-trends-2022/

Believe it or not, the year 2022 is right around the corner! So what does next year have in store for the rapidly changing, ever-evolving software world? From code reviews to DevOps, software testing, and tech companies' culture, here are our 22 software development trends for 2022 🔮

- Black Box Testing
- Glass Box Testing
- Code Inspections

#1 – A rise of automated code reviews

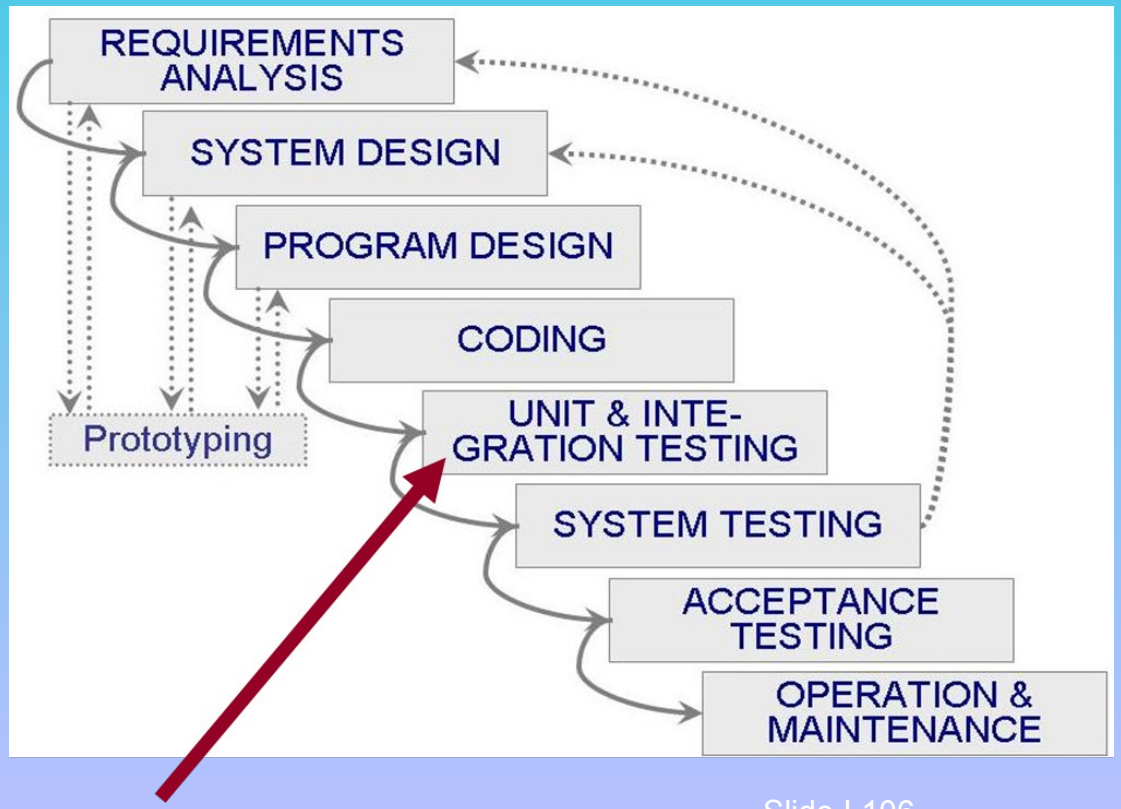
Tech companies increasingly see a well-defined code review process as a fundamental part of the software development process. Code reviews are among the best ways to improve code quality. Automated code review tools are increasing in popularity as more companies start to include them in their code review process, allowing developers to spend more time building new features instead of on code reviews. We can expect solutions like Codacy to see an increase in adoption in 2022.

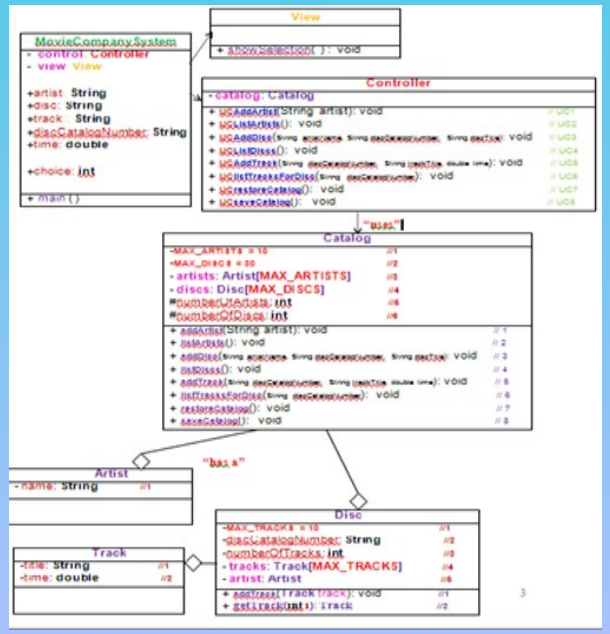
Conclusion

- ***Code Inspection** is at least as **successful** at* **detecting faults** as Glass Box **and** Black Box **Testing**

Potential Problems When Testing objects

- We must **Inspect Classes** and **objects**
- We can run **Test Cases** on **objects** (but not on **Classes**)





Potential Problems When Testing objects

A typical classical **module**:

- About 50 executable statements
- Give the input arguments, check the output arguments

A typical **object**:

- About 30 **methods**, some with only 2 or 3 statements
- A **method** often **does not return a value** to the **caller** — it changes **state** instead
- It may not be possible to check the **state** because of **information hiding (private instance variables)**
- Example: **method** `determineBalance` — we **need to know** `accountBalance` before, after

Potential Problems When Testing objects

- We need **additional methods** to return values of all **state variables**
  - They must be part of the **Test Plan**
  - **Conditional compilation** may have to be used

methods should not return void (untestable)

- An **Inherited method** may still have to be **Tested** (see next four slides)

Potential Problems When Testing Objects

- **Java Implementation** of a **Tree** hierarchy

????

Overwriting or Redefining

????

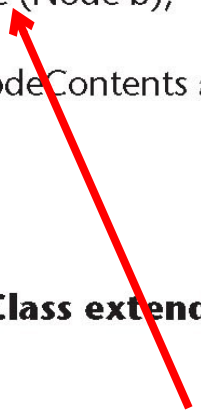
```
class RootedTreeClass
{
  ...
  void displayNodeContents (Node a);
  void printRoutine (Node b);
//
// method displayNodeContents uses method printRoutine
//
  ...
}

class BinaryTreeClass extends RootedTreeClass
{
  ...
  void displayNodeContents (Node a);
//
// method displayNodeContents defined in this class uses
// method printRoutine inherited from ClassRootedTree
//
  ...
}

class BalancedBinaryTreeClass extends BinaryTreeClass
{
  ...
  void printRoutine (Node b);
//
// method displayNodeContents (inherited from BinaryTreeClass) uses this
// local version of printRoutine within class BalancedBinaryTreeClass
//
  ...
}
```


Figure 14.17

Potential Problems When Testing objects

- Top half

```
class RootedTreeClass
{
   ...
   void displayNodeContents (Node a);
   void printRoutine (Node b);
//
// method displayNodeContents uses method printRoutine
//
   ...
}

class BinaryTreeClass extends RootedTreeClass
{
   ...
   void displayNodeContents (Node a);
//
// method displayNodeContents defined in this class uses
// method printRoutine inherited from ClassRootedTree
//
   ...
}
```



- When `displayNodeContents` is invoked in `BinaryTreeClass`, it uses `RootedTreeClass.printRoutine`

Potential Problems When Testing objects

Bottom half

```
class BinaryTreeClass extends RootedTreeClass
{
  ...
  void displayNodeContents (Node a);
//
// method displayNodeContents defined in this class uses
// method printRoutine inherited from ClassRootedTree
//
  ...
}

class BalancedBinaryTreeClass extends BinaryTreeClass
{
  ...
  void printRoutine (Node b);
//
// method displayNodeContents (inherited from BinaryTreeClass) uses this
// local version of printRoutine within class BalancedBinaryTreeClass
//
  ...
}
```

When `displayNodeContents` is invoked in `BalancedBinaryTreeClass` it uses `BalancedBinaryTreeClass.printRoutine`

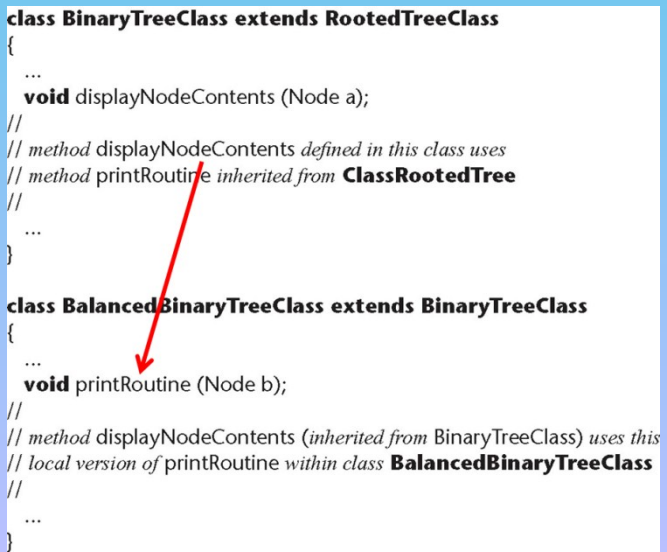
Potential Problems When Testing objects

Bad news

- `BinaryTreeClass.displayNodeContents` must be **ReTested** from scratch when **Reused** in `BalancedBinaryTreeClass`
- It invokes a different version of `printRoutine`

Worse news

- For theoretical reasons, we need to test using totally different **Test Cases**



Potential Problems When Testing objects

Making **state variables** visible (**private** – set and get)

- Minor issue

ReTesting before **Reuse**

- Arises only when **methods** interact
- We can determine when this **ReTesting** is needed

coupling

These are not reasons to abandon the **O**bject **O**riented **P**aradigm

Design by Contract

http://archive.eiffel.com/doc/manuals/technology/contract/ariane/page.html

The lesson for every software developer

The Inquiry Board makes a number of recommendations with respect to improving the software process of the European Space Agency. Many are justified; some may be overkill; some may be very expensive to put in place. There is a more simple lesson to be learned from this unfortunate event:

Reuse without a contract is sheer folly!

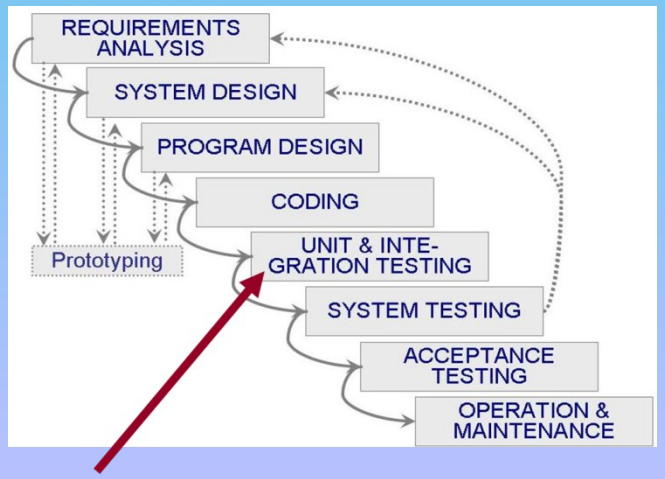
From CORBA to C++ to Visual Basic to ActiveX to Java, the hype is on software components. The Ariane 5 blunder shows clearly that naive hopes are doomed to produce results *far worse* than a traditional, reuse-less software process. To attempt to reuse software without Eiffel-like assertions is to invite failures of potentially disastrous consequences. The next time around, will it only be an empty payload, however expensive, or will it be human lives?

It is regrettable that this lesson has not been heeded by such recent designs as Java (which added insult to injury by *removing* the modest **assert** instruction of C!), IDL (the Interface Definition Language of CORBA, which is intended to foster large-scale reuse across networks, but fails to provide any semantic specification mechanism), Ada 95 and ActiveX.

For reuse to be effective, Design by Contract is a requirement. Without a precise specification attached to each reusable component -- precondition, postcondition, invariant -- no one can trust a supposedly reusable component.

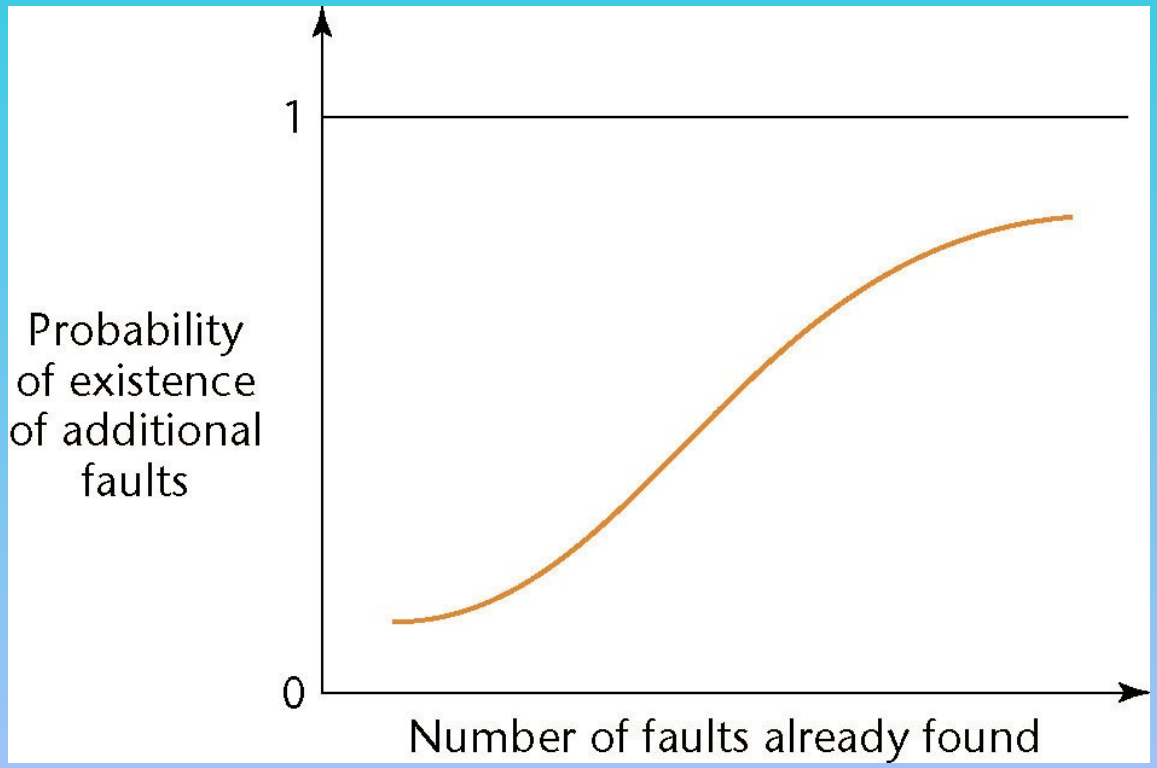
Management Aspects of Unit Testing

- We need to know **when to stop** Testing
- A number of different techniques can be used
  - Cost–Benefit **Analysis**
  - Risk **Analysis**
  - Statistical **Techniques**



When to Rewrite Rather Than Debug

- When a **Code** Artifact has **too many faults**
  - It is cheaper to Re**Design**, then **ReCode**



Fault Distribution in modules Is Not Uniform

[Myers, 1979]

- 47% of the faults in OS/370 were in only 4% of the modules

[Endres, 1975]

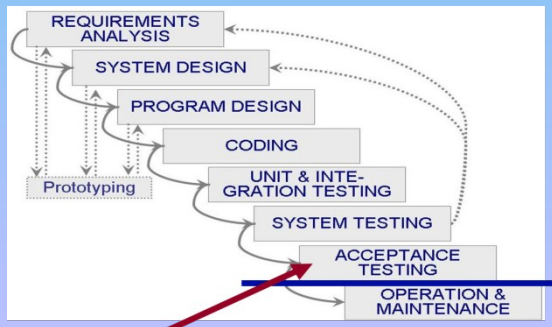
- 512 faults in 202 modules of DOS/VS (Release 28)
- 112 of the modules had only one fault
- There were modules with 14, 15, 19 and 28 faults, respectively
- The latter three were the largest modules in the product, with over 3,000 lines of DOS macro assembler language
- The module with 14 faults was relatively small, and very unstable
- A prime candidate for discarding, reDesigning, reCoding

When to Rewrite Rather Than Debug

- For every **Artifact**, **Management** must predetermine the maximum allowed number of faults during Testing

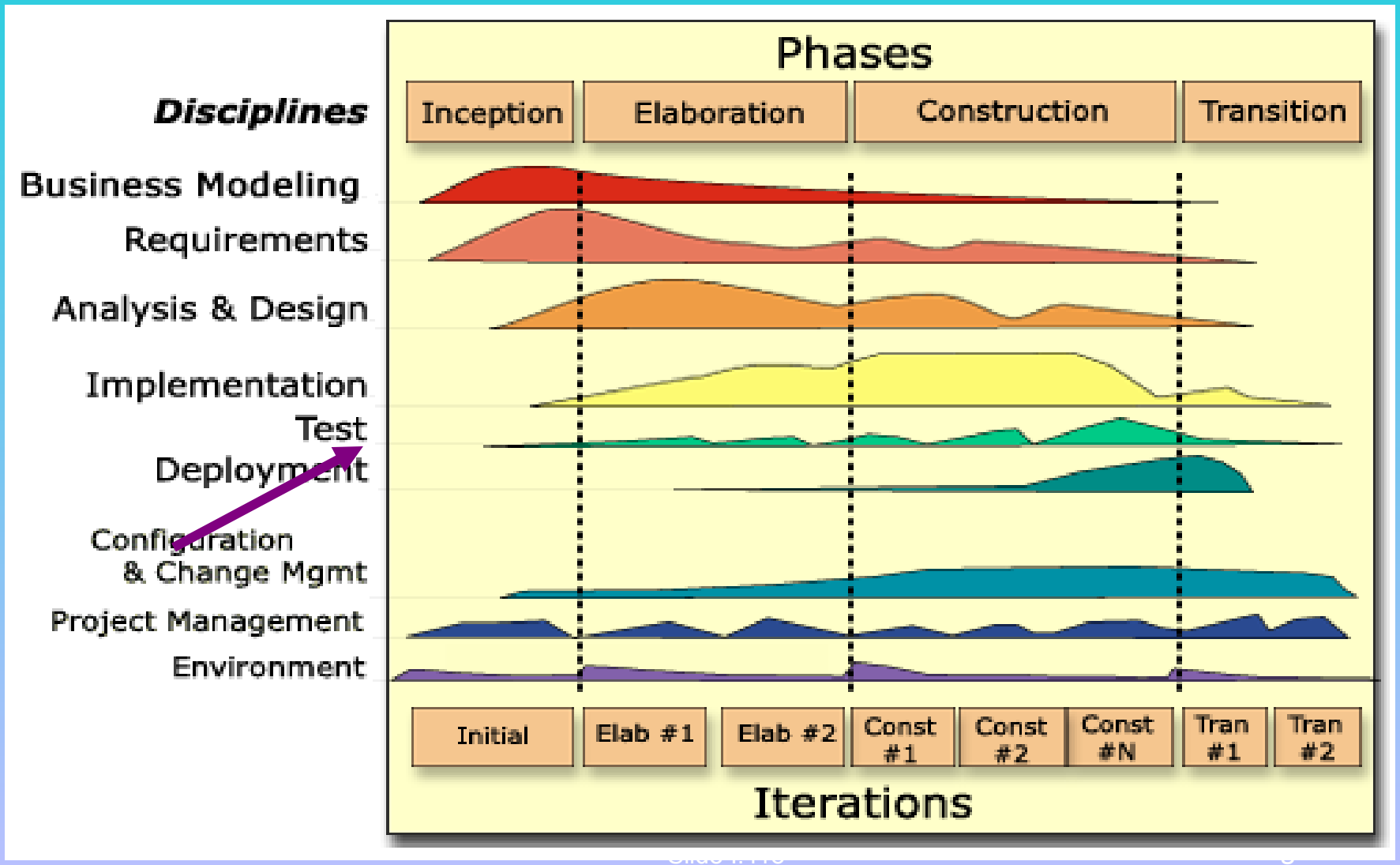
- If **this number** is reached
  - Discard
  - Re**Design**
  - Re**Code**

- The **maximum number of faults** allowed *after* ***delivery*** is **ZERO**



The **Phases** of the **U**nified **P**rocess

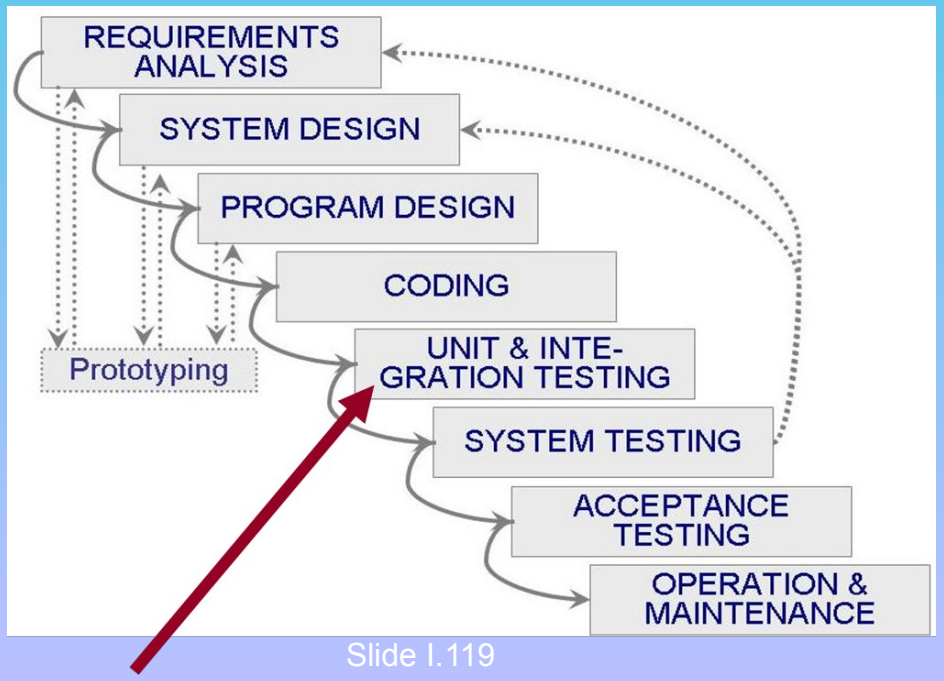
- The **Increments** are identified as **Phases**



Integration Testing

The **Testing** of each **new Code Artifact** when it is added to what has already been **Tested**

Special issues can arise when **Testing** **G**raphical **U**ser **I**nterfaces — see next slide



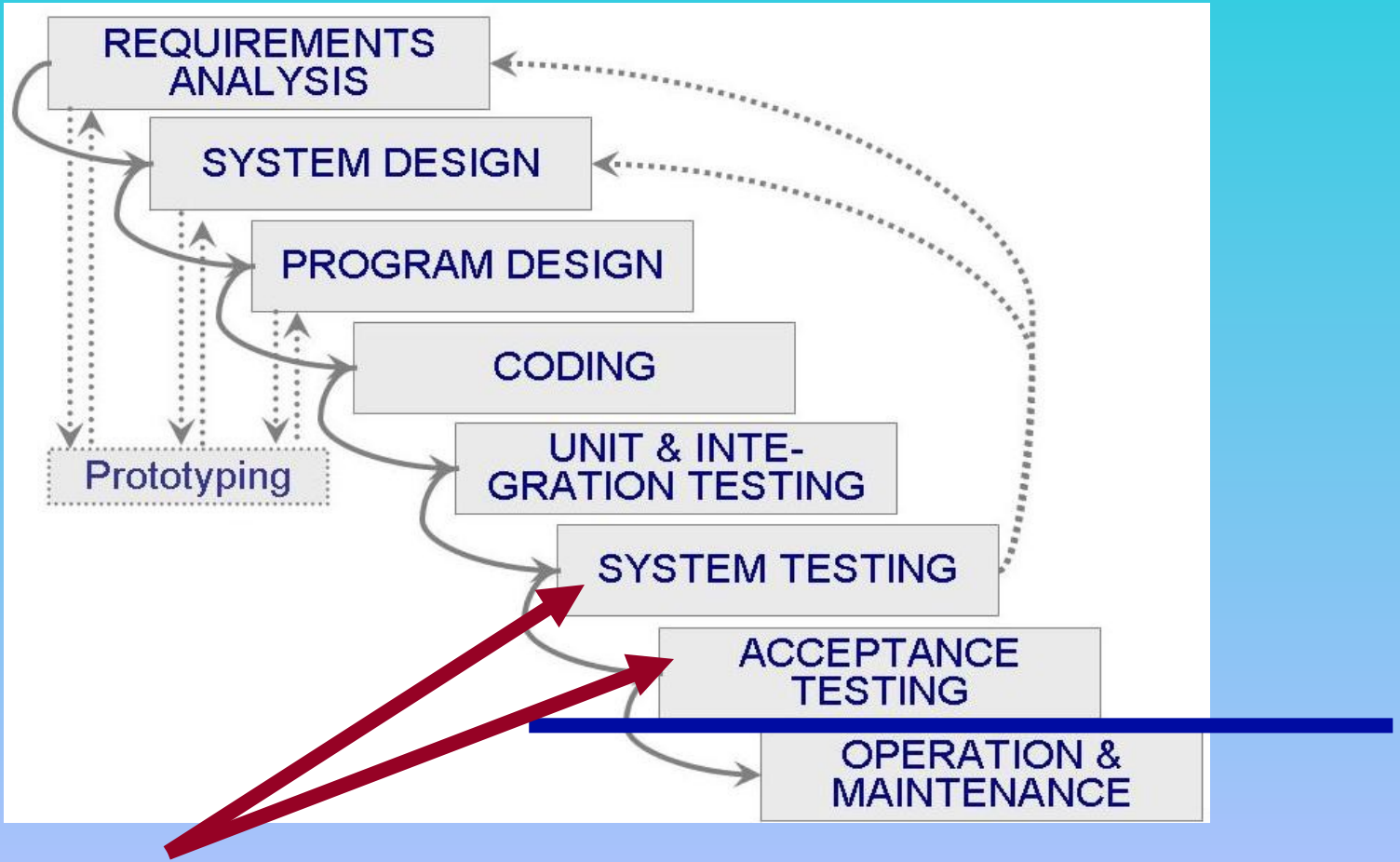
Integration Testing of Graphical User Interfaces

- **GUI** Test Cases include
  - Mouse clicks, and
  - Key presses

- These types of Test Cases cannot be stored in the usual way
  - We need special **CASE Tools**

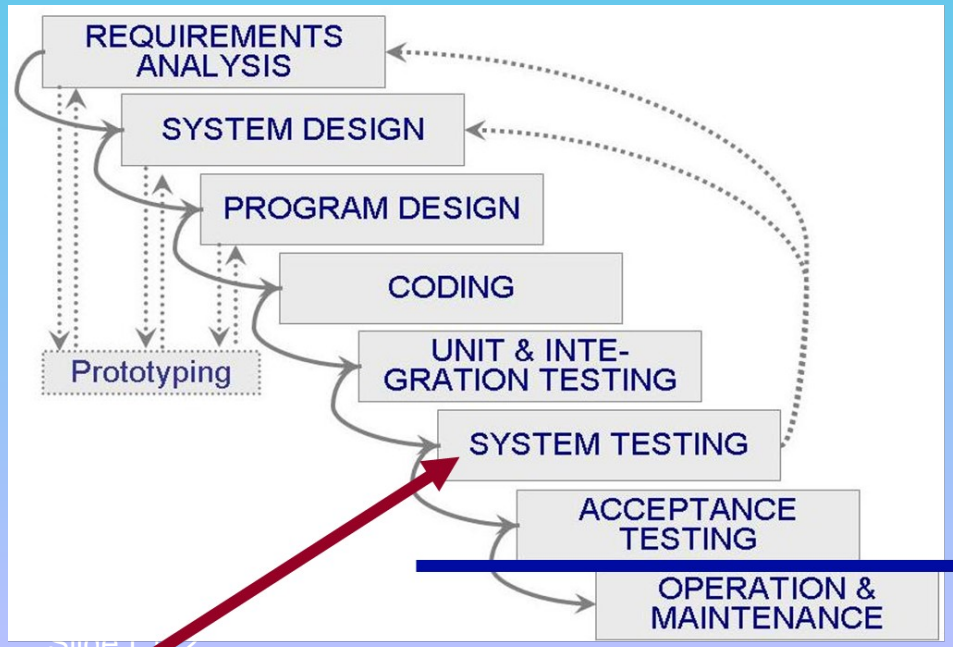
- Examples:
  - **QAPartner**
  - **XRunner**

Waterfall Life Cycle **Model** with Prototyping



Product/System Testing

- **Product Testing** for **custom software**
  - The **SQA Group** must ensure that the **product** passes the **Acceptance Test**
  - **Failing an Acceptance Test has bad consequences for the Development Organization**



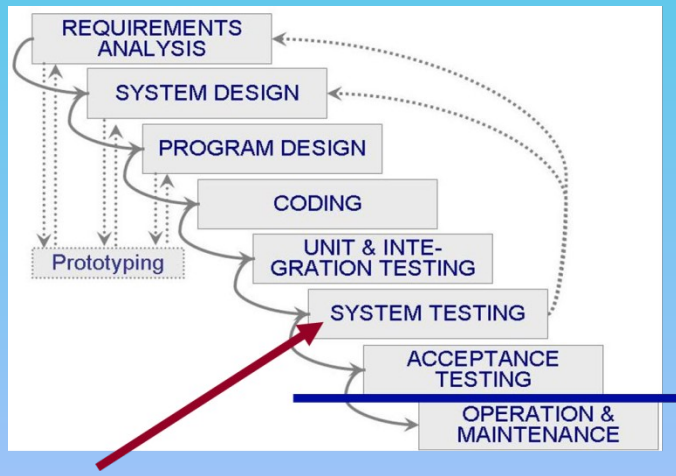
Product Testing for Custom Software

- The **SQA Team** must try to **approximate** the **Acceptance Test**
- **Black Box Test Cases** for the **product** as a whole

- **Robustness** of **product** as a whole
  - *Stress* ***Testing*** (under peak load)
  - *Volume* ***Testing*** (e.g., can it handle large input files?)

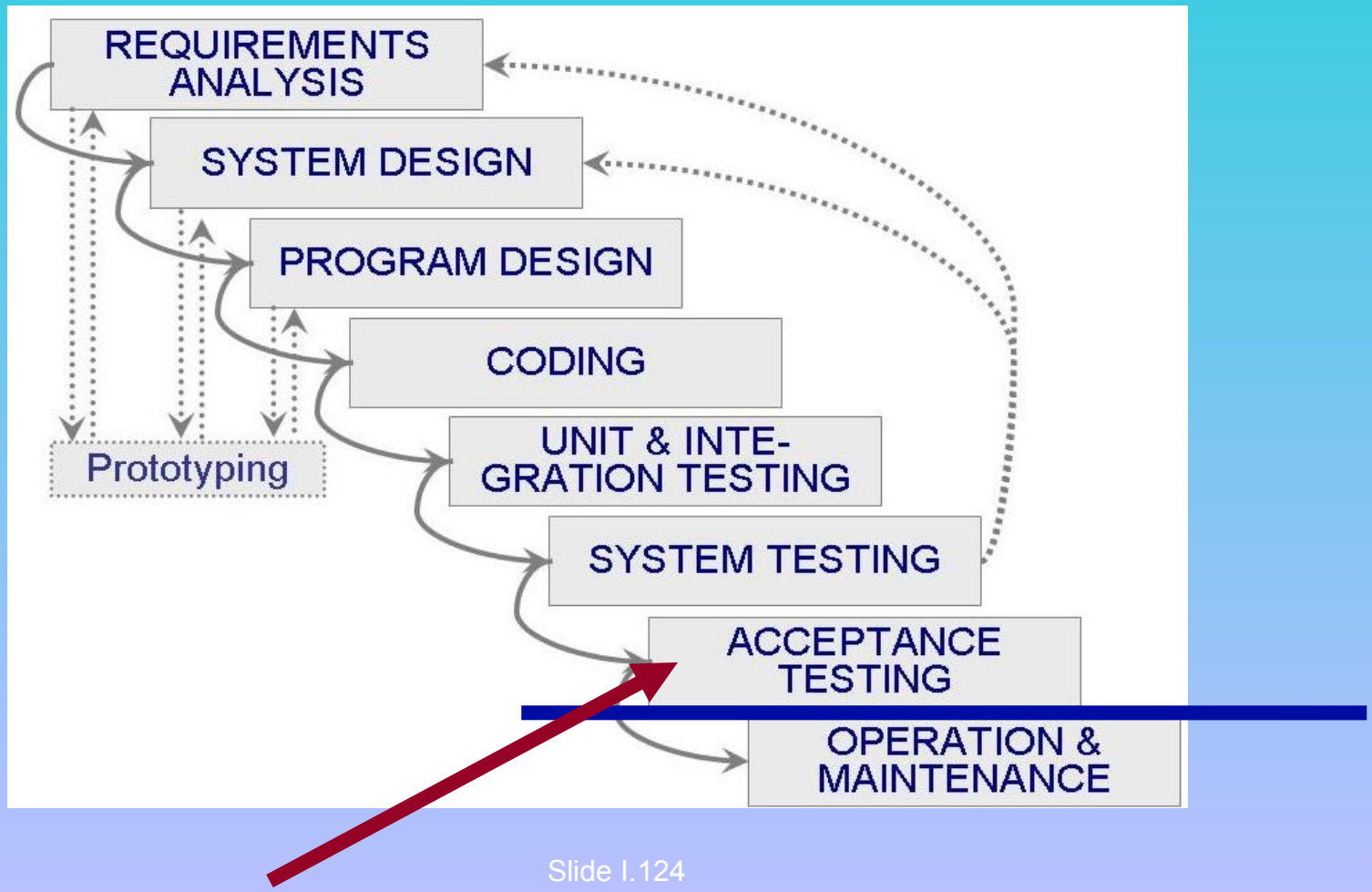
- All **constraints** must be checked

- All **Documentation** must be
  - Checked for **correctness**
  - Checked for conformity with **standards**
  - **Verified** against the current version of the **product**



Product Testing for Custom Software

- The **Product** (**Code plus Documentation**) is now handed over to **the Client Organization** for **Acceptance Testing**

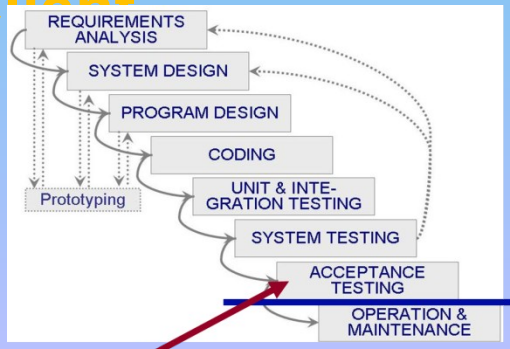


Acceptance Testing

The Client determines whether the **product** satisfies its **Specifications (SRS)**

Acceptance Testing is performed by

- The **Client Organization**, or
- The **SQA Team** in the presence of **Client Representatives**, or
- An **independent SQA Team** hired by **the Client**

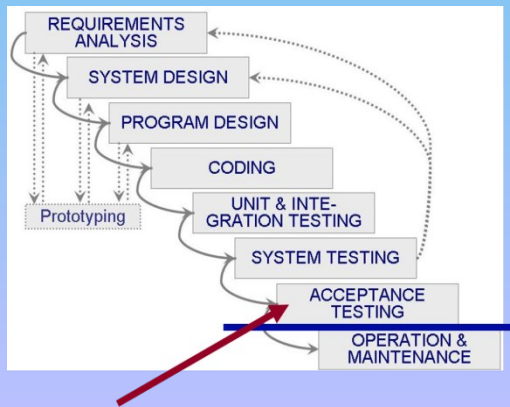


Acceptance Testing

The four major components of **Acceptance Testing** are

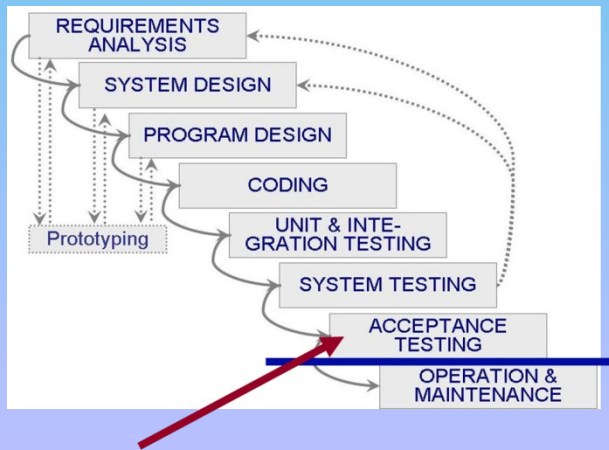
- **Correctness**
- **Robustness**
- **Performance**
- **Documentation**

These are **precisely what** was **Tested** by the **Development Group** during **product /SystemTesting**



Acceptance Testing

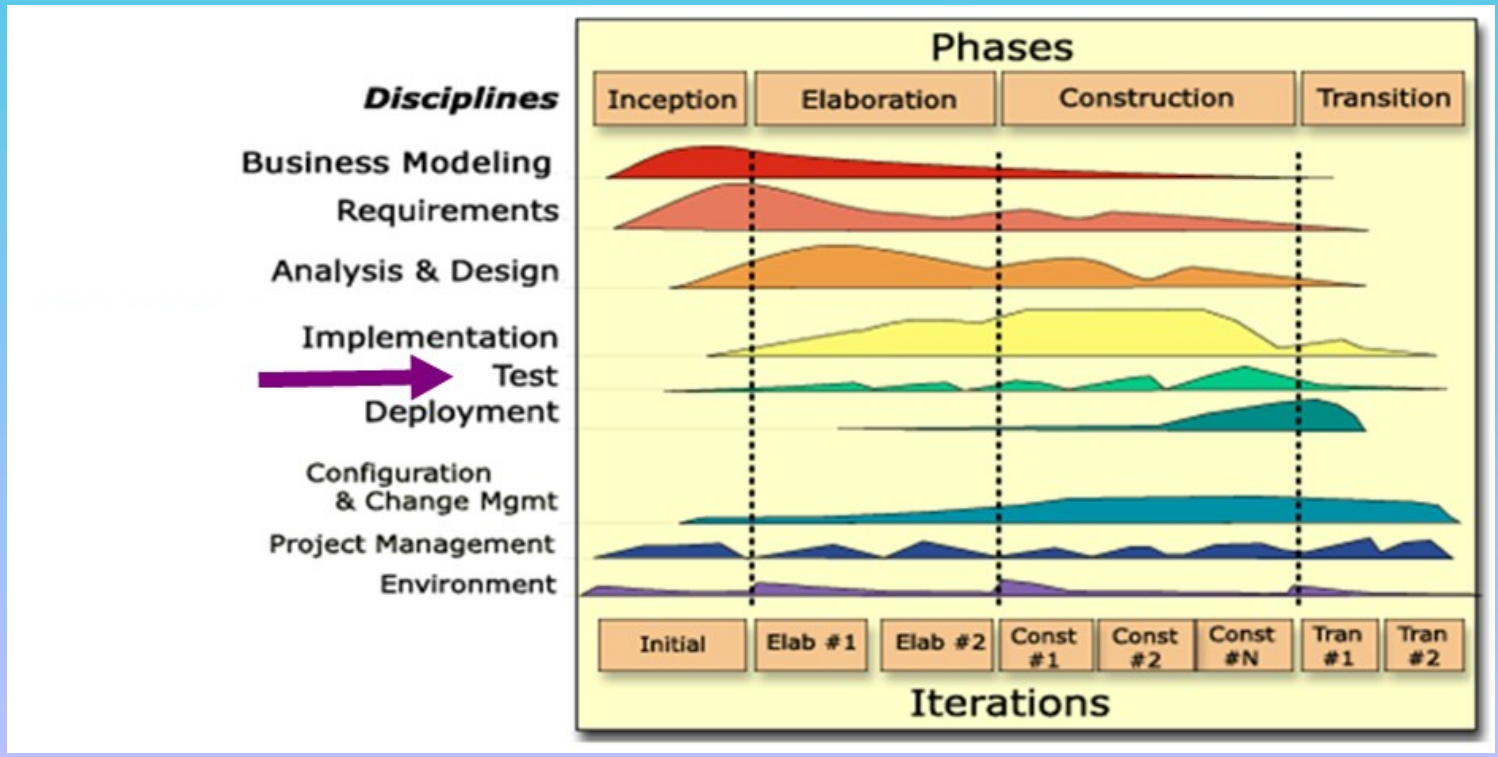
- The key difference between **product/System Testing** and **Acceptance Testing** is
  - **Acceptance Testing** is performed on **actual data**
  - **Product Testing** is preformed on **test data**, which can never be real, by definition



From 4:20 to 5:00 – 40 minutes.

Testing Workflow

THE END



At 5:00 PM.

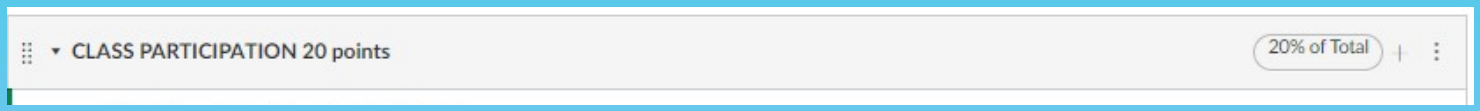
VH, next EXAM 4 REVIEW

VH, next EXAM 4

11.20.2023 (M 4 to 5:30) (26)		EXAM 4 REVIEW (CANVAS)	Download ZyBook: Sections 12-14		
11.27.2023 (M 4 to 5:30) Optional (27)					Q & A Set 4 topics.
11.29.2023 (W 4 to 5:30) (28) LAST CLASS					EXAM 4 (CANVAS)

From 5:05 to 5:15 – 10 minutes.

11.15.2023 (W 4 to 5:30) (25)		Lecture 9: Testing Tutorial 6 TDD			



PASSWORD: IN TEAMS

At 5:15 PM.

End Class 25

VH, Download Attendance Report
Rename it:
11.15.2023 Attendance Report FINAL

VH, upload Class 25 to CANVAS.