Test execution automation

- Write test cases
  - Document intended behavior
  - Execute the program
  - Check observed behavior against intended behavior.
- Frameworks:
  - Unit testing: Junit,
  - GUI testing: Selenim, Robotium, Monkey
  - BDT: Cucumber

Advantages

- Script once, execute multiple times (per day)
- Document oracles
- Measurable in terms of adequacy and effectiveness

Testing with JUnit

- Junit is a **unit test environment** for Java programs developed by *Erich Gamma* and *Kent Beck*.
    - Writing test cases
    - Executing test cases
    - Pass/fail? (expected result = obtained result?)
- Consists in a **framework** providing all the tools for testing.
    - framework: set of classes and conventions to use them.
- It is **integrated into Eclipse** through a graphical plug-in.

Junit (3.x and 4.x)

- Test framework
  - test cases are Java code
  - test case = “sequence of operations +inputs + expected values”

- Production code

```
public int min(…){
  //return the minimum
}
```

- Test code

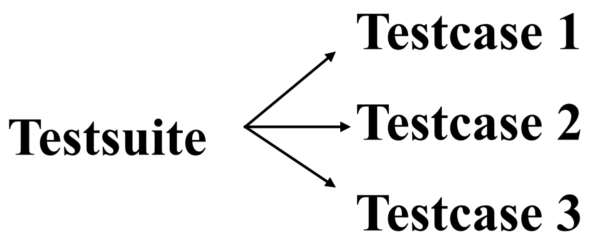
```
void testMin(…) {
   int result= obj.mean(2, 7);
   assertEquals(2, result);
}
```

JUnit 3.x for testing programs

- JUnit tests
  - “substitute the use of **main()** to check the program behaviour”
- All we need to do is:
  - write a sub-class of **TestCase** $\longleftarrow$ **junit.framework.***
  - add to it one or more **test methods**
    - **Method names starting with “test”: testMean()**
  - run the test using JUnit

Framework elements

- **TestCase**
  - Base class for classes that contain tests
- **assert*()**
  - Method family to ***check conditions***
- **TestSuite**
  - Enables grouping several test cases



An example

```
class Stack {
 public boolean isEmpty(){ ...  }
 public void push(int i){ ...  }
 public int pop(){ ...  }
 …
}
```

```
import junit.framework.TestCase;
  public class StackTester extends TestCase {

  public void testIsEmpty() {
    Stack aStack = new Stack();
    System.out.println("Is stack empty: " + aStack.isEmpty());
    // it should be empty.
  }

}
```

Must begin with "test"

Assert*()

- Public methods defined in the base class TestCase
- Their names begin with "assert" and *are used in test methods*
  - es. **assertTrue("stack should be empty", aStack.empty());**
- If the condition is false:
  - test fails
  - execution skips the rest of the test method
  - the message (if any) is printed
- If the condition is true:
  - execution continues normally

An example

```
import junit.framework.TestCase;
  public class StackTester extends TestCase {

  public void testIsEmpty() {
    Stack aStack = new Stack();
    assertTrue(“stack should be empty”,  aStack.empty());
  }

}
```

Assert*()

- for a boolean condition
  - **assertTrue(“message for fail”, condition);**
  - **assertFalse(“message”, condition);**
- for object, int, long, and byte values
  - **assertEquals(expected_value, expression);** obtained
- for float and double values
  - **assertEquals(expected, expression, error);**
- for objects references
  - **assertNull(reference)**
  - **assertNotNull(reference)**
- …

http://junit.org/apidocs/org/junit/Assert.html

Assert: example

```
public void testStack() {
  Stack aStack = new Stack();
  assertTrue("Stack should be empty!", aStack.isEmpty());
  aStack.push(10);
  assertFalse("Stack should not be empty!", aStack.isEmpty());
  aStack.push(4);
  assertEquals(4, aStack.pop());
  assertEquals(10, aStack.pop());
}
```

```
class Stack {
 public boolean isEmpty(){ ...  }
 public void push(int i){ ...  }
 public int pop(){ ...  }
 ...
}
```

One concept at a time …

```
public class StackTester extends TestCase {
 public void testStackEmpty() {
     Stack aStack = new Stack();
     assertTrue("Stack should be empty!", aStack.isEmpty());
     aStack.push(10);
     assertFalse("Stack should not be empty!", aStack.isEmpty());
 }
 public void testPushPop() {
     Stack aStack = new Stack();
     aStack.push(10);
     aStack.push(-4);
     assertEquals(-4, aStack.pop());
     assertEquals(10, aStack.pop());
 }
}
```

Working rule

- For each test case class, JUnit
  - execute all of its test methods
    - i.e. those whose name starts with “test” or annotated with @Test
  - ignores everything else …

TestSuite

- **Groups several test cases:**

junit.framework.*

```
public class AllTests extends TestSuite {

  public static TestSuite suite() {
      TestSuite suite = new TestSuite();
      suite.addTestSuite(StackTester.class);
      suite.addTestSuite(AnotherTester.class);
    return suite;
  }
}
```

Test of “Exceptions”

- There are two cases:
  1. We expect a **normal behavior** and then no exceptions.
  2. We expect **an anomalous behavior** and then an exception.

We expect a normal behavior …

```
try {
   // We call the method with correct parameters
   object.method("Parameter");
   assertTrue(true); // OK
} catch(PossibleException e){
   fail("method should not fail !!!");
}
```

```
class TheClass {
 public void method(String p)
            throws PossibleException
  { /*... */ }
}
```

We expect an exception …

```
try {
    // we call the method with wrong parameters
  object.method(null);
  fail("method should fail!!");
} catch(PossibleException e){
    assertTrue(true); // OK
}
```

```
class TheClass {
 public void method(String p)
            throws PossibleException
  { /*... */ }
}
```

SetUp() and tearDown()

- **setUp()** method initialize object(s) under test.
  - called before every test method
- **tearDown()** method release object(s) under test
  - called after every test case method.

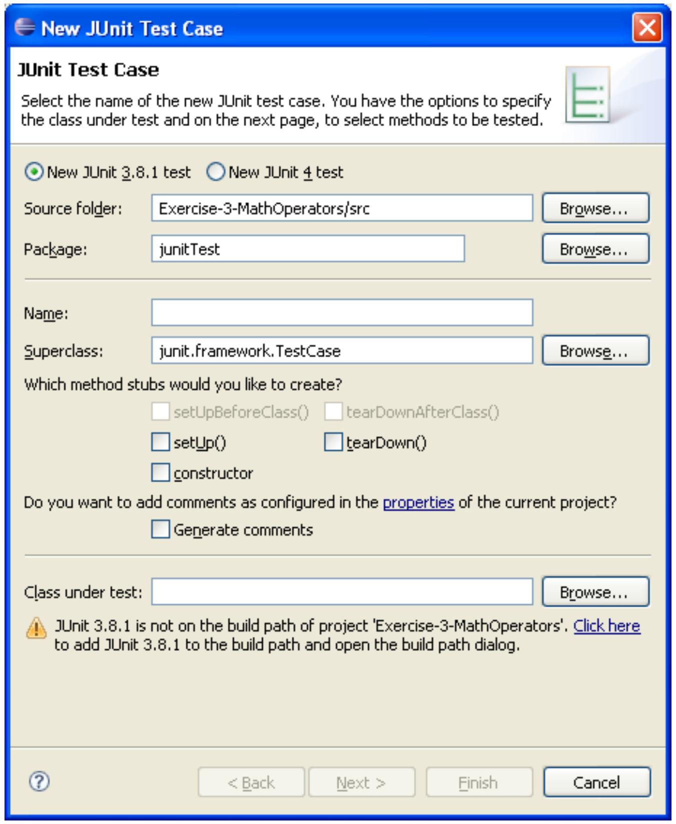
```
ShoppingCart cart;
Book book;

protected void setUp() {
 cart = new ShoppingCart();
 book = new Book(“JUnit", 29.95);
 cart.addItem(book);
}
...
```

Create a new JUnit test case

Eclipse Menu

File Edit Source Refactor Navigate Search Project Run Window Help

- File
  - New
    - Junit Test Case
      - Set the parameters:
        - Junit 3.x or 4.x
        - name of the class
        - etc.
      - Finish


New JUnit Test Case

JUnit Test Case

Select the name of the new JUnit test case. You have the options to specify the class under test and on the next page, to select methods to be tested.

[✓ New JUnit 3.8.1 test] New JUnit 4 test

Source folder: Exercise-3-MathOperators/src Browse...

Package: junitTest Browse...

Name:

Superclass: junit.framework.TestCase Browse...

Which method stubs would you like to create?

☐ setUpBeforeClass() ☐ tearDownAfterClass()

☐ setUp() ☐ tearDown()

☐ constructor

Do you want to add comments as configured in the properties of the current project?

☐ Generate comments

Class under test: Browse...

JUnit 3.8.1 is not on the build path of project 'Exercise-3-MathOperators'. Click here to add JUnit 3.8.1 to the build path and open the build path dialog.

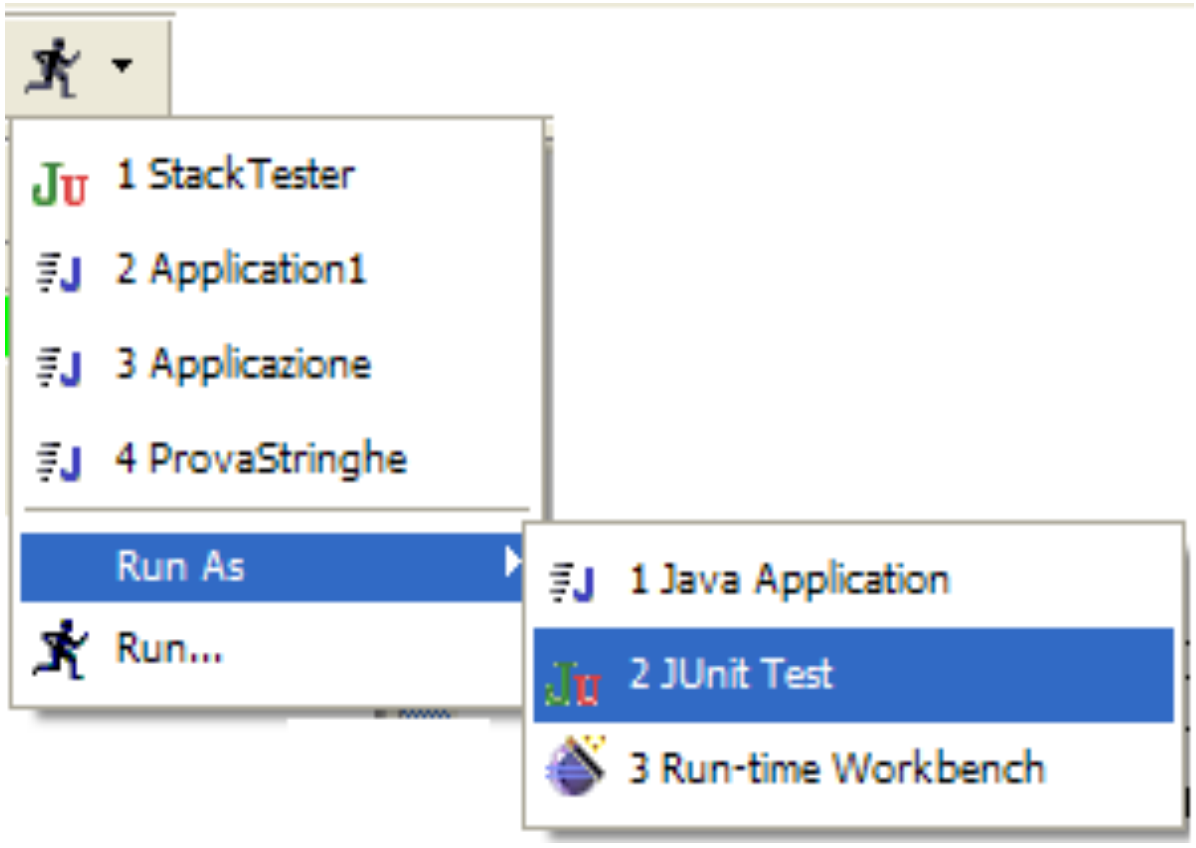
< Back Next > Finish Cancel

Run as JUnit Test

Eclipse Menu

File Edit Source Refactor Navigate Search Project Run Window Help

- Run
  - Run As
    - Junit Test



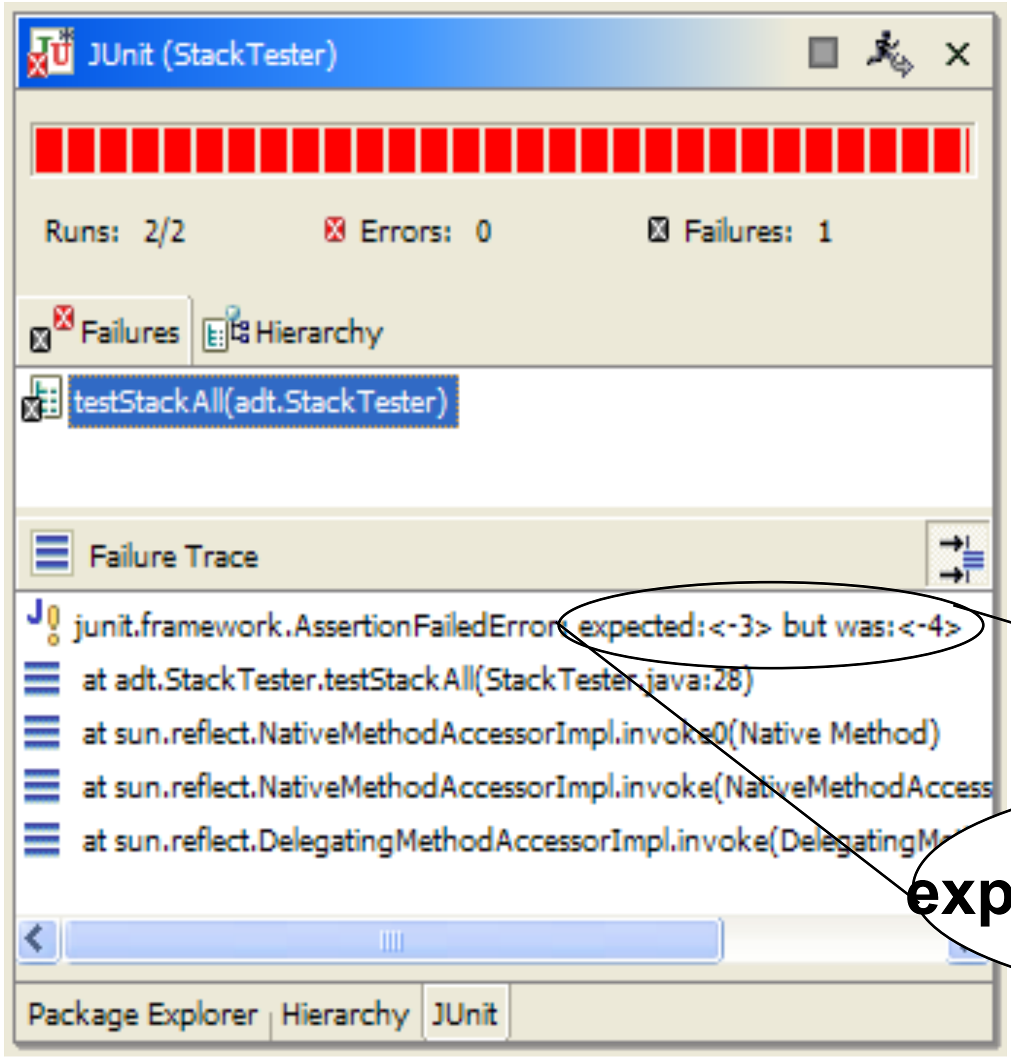
Maven and Unit testing

- All test code should go under:
  - src/test/java
- All test resources should go under:
  - src/test/resources

- Tests can be run (command line):
  - mvn test

Red / Green Bar

Fail



JUnit (StackTester)

Runs: 2/2   Errors: 0   Failures: 1

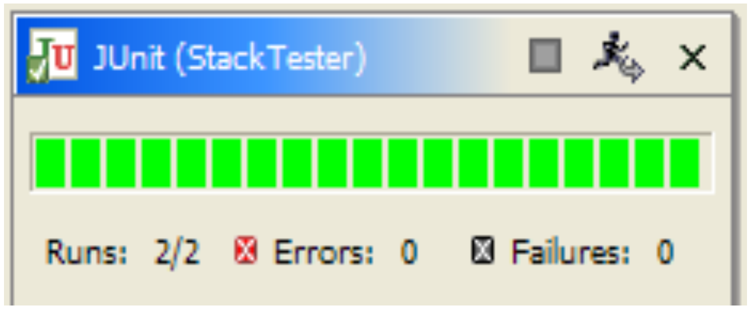
Failures | Hierarchy

testStackAll(adt.StackTester)

Failure Trace

```
junit.framework.AssertionFailedError: expected:<-3> but was:<-4>
at adt.StackTester.testStackAll(StackTester.java:28)
at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccess
at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingM
```

Pass



JUnit (StackTester)

Runs: 2/2   Errors: 0   Failures: 0

expected <-3> but was <-4>

JUnit 3.x and JUnit 4.x

- Most things are about equally easy
  - JUnit 4 can still run JUnit 3 tests
- All the old **assert*XXX*** methods are the same
- JUnit 4 has some additional features
- JUnit 4 provides protection against infinite loops
- Junit 4 uses annotations (@)

From JUnit 3.x to 4.x

- JUnit 4 requires Java 5 or newer
- Don't extend **junit.framework.TestCase**; just use an ordinary class
- Import **org.junit.*** and **org.junit.Assert.***
  - Use a *static* import for **org.junit.Assert.***
  - Static imports replace inheritance from **junit.framework.TestCase**
- Use annotations instead of special method names:
  - Instead of a **setUp** method, put **@Before** before some method
  - Instead of a **tearDown** method, put **@After** before some method
  - Instead of beginning test method names with '**test**', put **@Test** before each test method

Annotations in J2SE

- J2SE 5 introduces the **Metadata** feature (data about data)
- Annotations allow you to add **decorations** to your code (remember javadoc tags: @*author*)
- Annotations are used for code documentation, compiler processing (@Deprecated), code generation, runtime processing
- New annotations can be created by developers

http://java.sun.com/docs/books/tutorial/java/javaOO/annotations.html

Junit 4.x for testing programs

Import the JUnit 4 classes you need

```
import org.junit.*;
import static org.junit.Assert.*;
```

Declare your (conventional) Java class

```
public class MyProgramTest {
```

Declare any variables you are going to use, e.g., an instance of the class being tested

```
MyProgram program;
int [ ] array;
int solution;
```

Junit 4.x for testing programs (2)

If needed, define *one* method to be executed *just once,* when the class is first loaded. For instance, when we need to connecting to a database

```
@BeforeClass
public static void setUpClass() throws Exception {
      // one-time initialization code
}
```

If needed, define *one* method to be executed *just once,* to do cleanup after all the tests have been completed

```
@AfterClass
public static void tearDownClass() throws Exception {
      // one-time cleanup code
}
```

Junit 4.x for testing programs (3)

If needed, define *one or more* methods to be executed <u>before each test</u>, e.g., typically for initializing values

```
@Before
 public void setUp() {
    program = new MyProgram();
    array = new int[] { 1, 2, 3, 4, 5 };
 }
```

If needed, define *one or more* methods to be executed after <u>each test</u>, e.g., typically for releasing resources (files, etc.)

```
@After
 public void tearDown() {
 }
```

Junit 4.x for testing programs (4)

- A test method is annotated with **@Test**
- It takes no parameters, and returns no result.
- All the usual **assert*XXX*** methods can be used

```
@Test
 public void sum() {
    assertEquals(15, program.sum(array));
    assertTrue(program.min(array) > 0);
 }
```

Additional Features of @Test

To avoid infinite loops, an execution time limit can be used. The time limit is specified in milliseconds. The test fails if the method takes too long.

```
@Test (timeout=10)
    public void greatBig() {
       assertTrue(program.ackerman(5, 5) > 10e12);
   }
```

Some method calls should throw an exception. We can specify that an exception is expected. The test will pass if the expected exception is thrown, and fail otherwise

```
@Test (expected=IllegalArgumentException.class)
   public void factorial() {
      program.factorial(-5);
   }
```

Parameterized tests

Using **@RunWith(value=Parameterized.class)** and a method **@Parameters**, a test class is executed with several inputs

```
@RunWith(value=Parameterized.class)
 public class FactorialTest {
    private long expected;
    private int value;

    @Parameters
    public static Collection data() {
       return Arrays.asList( new Object[ ][ ] { { 1, 0 }, { 1, 1 }, { 2, 2 }, { 120, 5 } });
    }

    public FactorialTest(long expected, int value) { // constructor
       this.expected = expected;
       this.value = value;
    }

    @Test
    public void factorial() {
       assertEquals(expected, new Calculator().factorial(value));
    }
 }
```

Parameters used to exercise different instances of the class

Ignoring tests

The @Ignore annotation says to not run a test

```
@Ignore("I don’t want Dave to know this doesn’t work")
@Test
public void add() {
    assertEquals(4, sum(2, 2));
}
```

You shouldn’t use @Ignore without a very good reason!

import **org.junit.runners.Suite**;
import **org.junit.runners.Suite.SuiteClasses**;

Test suites

As before, you can define a suite of tests

```
@RunWith(value=Suite.class)
@SuiteClasses(value={
        value=test1.class,
        value=test2.class
        })
public class AllTests { … }
```

It could be empty

Hamcrest (cool assertions)

- Enhances readability of test code

- Example:
  - assertEquals(theBook, myBook)
  - **assertThat**(theBook, **is**(**equalTo**(myBook)))

See http://code.google.com/p/hamcrest/wiki/Tutorial

Exercise: Test the calculator (JUnit 4)

```
public class Calculator {
   public int add(int a, int b)
   public int divide(int a, int b)
   …
}
```

```
@Test
public void addition() {
    Calculator calculator = new Calculator ();
    int result = calculator.add(2, 3);
    assertEquals(5, result);

    result = calculator.add(-2, -3);
    assertEquals(-5, result);
}

@Test
public void divisionNormal() {
  Calculator calculator = new Calculator ();
   int result = calculator.divide(4, 2);
    assertEquals(2, result);
}
```

```
@Test
public void divisionByZero() {
  Calculator calculator = new Calculator ();
  try {
    int result = calculator.divide(2, 0);
    fail("Division by zero exception was expected!");
  } catch (ArithmeticException success) {
      assertNotNull(success.getMessage()); /* Pass! ☺ */
  }
```

```
public class FizzBuzz {

    private static final int THREE = 3;
    private static final int FIVE = 5;

    public String calculate(int number) {
        if (isDivisibleBy(number, THREE) && isDivisibleBy(number, FIVE)) {
            return "FizzBuzz";
        }

        if (isDivisibleBy(number, THREE)) {
            return "Fizz";
        }

        if (isDivisibleBy(number, FIVE)) {
            return "Buzz";
        }

        return String.valueOf(number);
    }

    private boolean isDivisibleBy(int dividend, int divisor) {
        return dividend % divisor == 0;
    }
}
```

```
@Test
@Parameters({"1", "2", "4", "7", "11", "13", "14"})
public void returnsNumberForNumberNotDivisibleByThreeAndFive(int number) {
    assertThat(fizzBuzz.calculate(number)).isEqualTo("" + number);
}

@Test
@Parameters({"3", "6", "9", "12", "18", "21", "24"})
public void returnFizzForNumberDivisibleByThree(int number) {
    assertThat(fizzBuzz.calculate(number)).isEqualTo("Fizz");
}

@Test
@Parameters({"5", "10", "20", "25", "35", "40", "50"})
public void returnBuzzForNumberDivisibleByFive(int number) {
    assertThat(fizzBuzz.calculate(number)).isEqualTo("Buzz");
}

@Test
@Parameters({"15", "30", "45", "60"})
public void returnsFizzBuzzForNumberDivisibleByThreeAndFive(int number) {
    assertThat(fizzBuzz.calculate(number)).isEqualTo("FizzBuzz");
}
```

Test the calculator (JUnit 4)

```
public class Calculator {
   public int add(int a, int b)
   public int divide(int a, int b)
   …
}
```

```
@Test
public void addition() {
   Calculator calculator = new Calculator ();
   int result = calculator.add(2, 3);
   assertEquals(5, result);

   result = calculator.add(-2, -3);
   assertEquals(-5, result);
}
```

Test the average method

```
public class Calculator {
  private double average(double[] m)
    double sum = 0;
    for (int i = 0; i < m.length; i++) {
        sum += m[i];
    }
    return sum / m.length;
  }
}
```

```
@Test
public void average() {



}
```

Testability is important

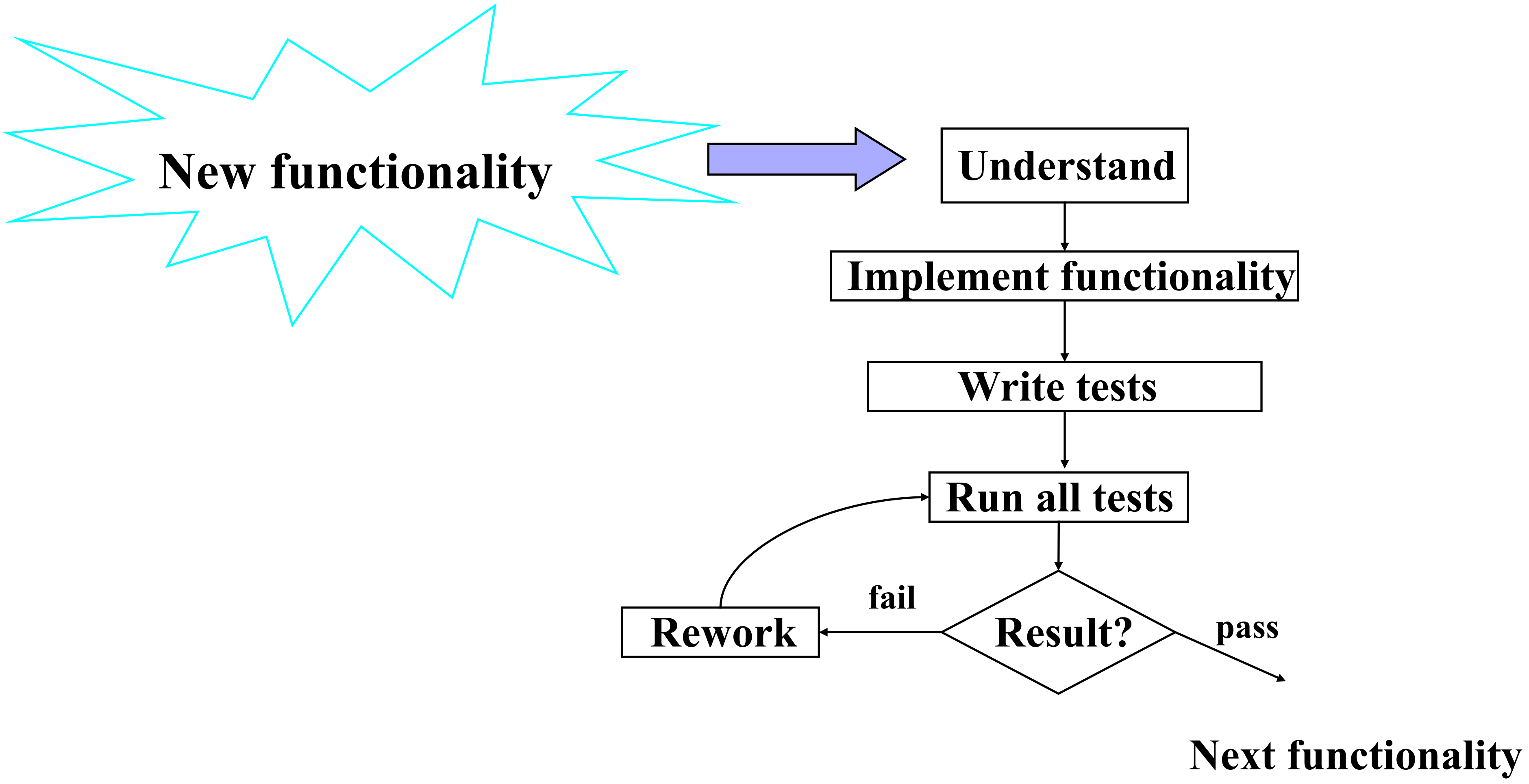
```
public class Calculator {
  private double average(double[] m)
    double sum = 0;
    for (int i = 0; i < m.length; i++) {
        sum += m[i];
    }
    return sum / m.length;
  }
}
```

```
@Test
public void average() {

//  NOT TESTABLE because average is private!

}
```

When testing programs?

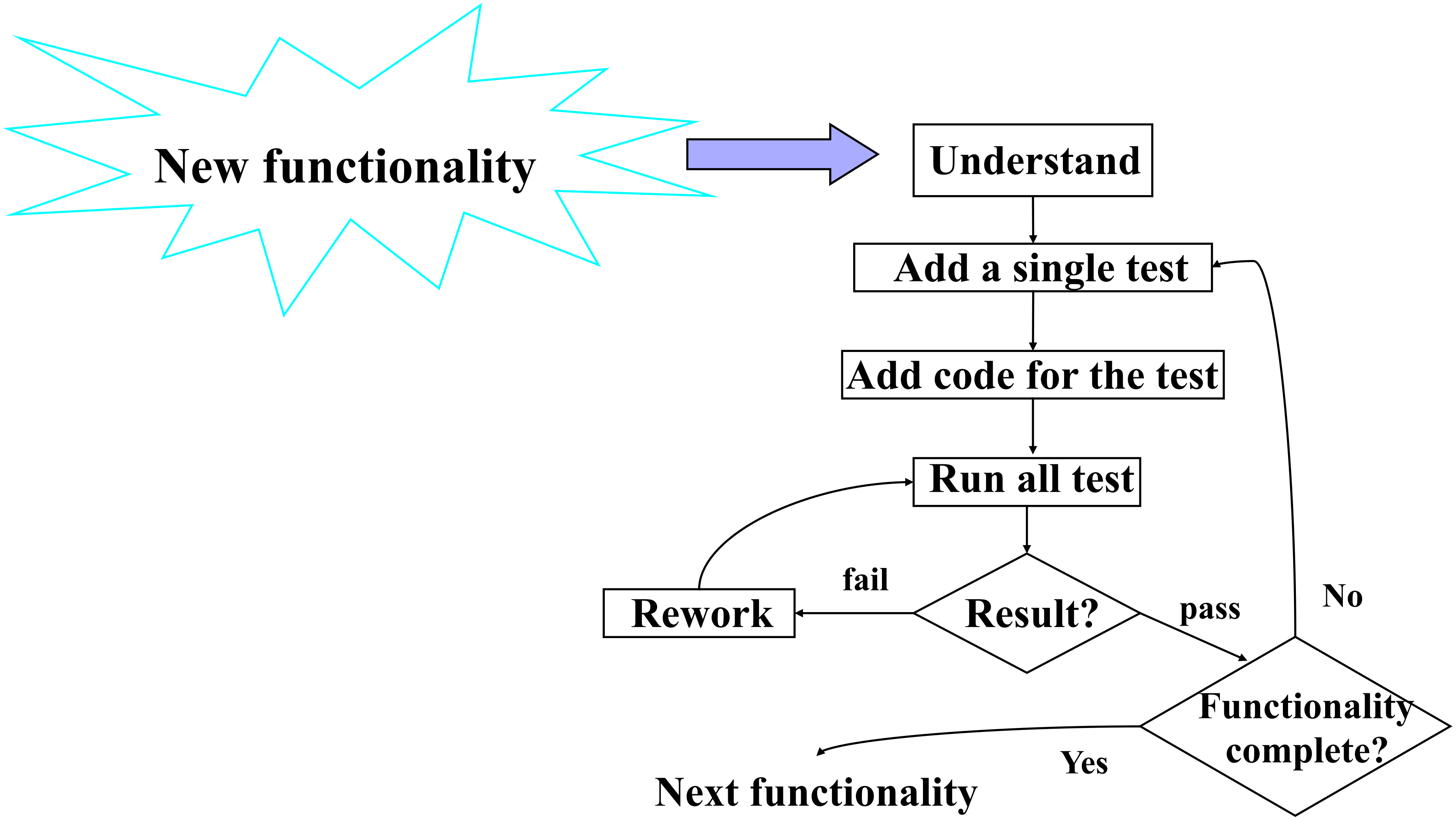
- Test last
  - The conventional way for testing in which testing follows the implementation
- Test first
  - The agile view in which testing is used as a development tool

Test last


New functionality
Understand
Implement functionality
Write tests
Run all tests
Result?
fail
Rework
pass
Next functionality

"Extreme programming" (**XP**) champions the use of tests as a development tool …

Test first



Best Unit Testing Practices

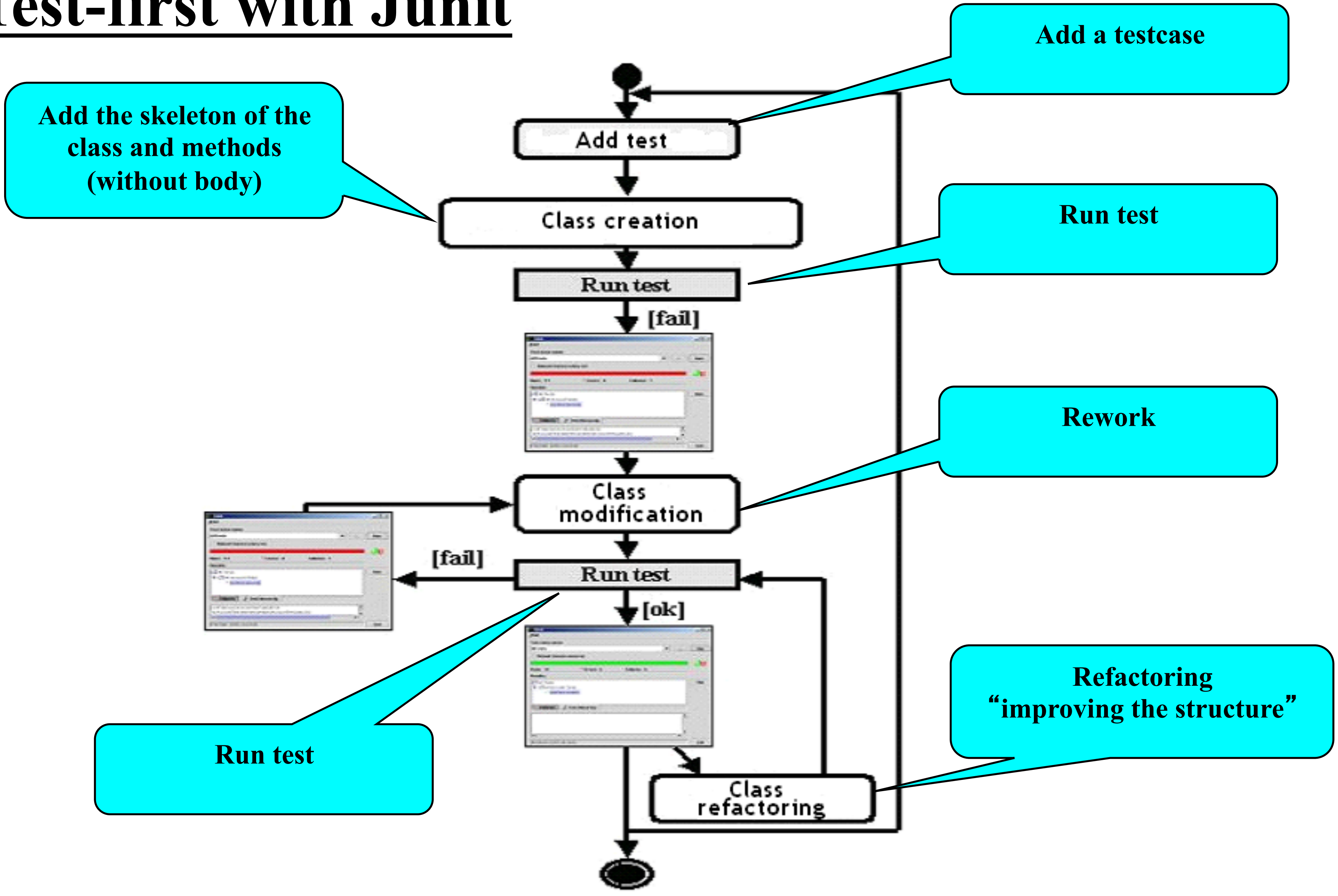
- **During Development:** When you need to add new functionality to the system, write the **tests first**. Then, you will be done developing when the test runs.

- **During Debugging:** When someone discovers a bug in your code, first write a test case that fails (finds the failure). Then debug and repair the code until the test succeeds.

TDD (test first) Advantages

- Each method has an associated testcase
  - the **confidence** of our code increases …
- It simplifies:
  - **refactoring**/restructuring
  - maintenance
  - the introduction of new functionalities
- Test first helps to build the **documentation**
  - testcases are good “use samples”
- Programming is more fun …

Test-first with Junit



When do we have enough testing?

“Each software system should be thoroughly tested”.

- What does thorough mean?
- How can we meaure *test adequacy?*
- When can we stop testing?