



Test automation / JUnit

Building automatically repeatable test suites



Test automation

- Test automation is software that automates any aspect of testing
 - Generating test inputs and expected results
 - Running test suites without manual intervention
 - Evaluating pass/no pass
- Testing must be automated to be effective and repeatable

2



Automated testing steps

- Exercise the implementation with the automated test suite
- Repair faults revealed by failures
- Rerun the test suite on the revised implementation
- Evaluate test suite coverage
- Enhance the test suite to achieve coverage goals
- Rerun the automated test suite to support regression testing

3



Automated testing advantages

- Permits quick and efficient verification of bug fixes
- Speeds debugging and reduces “bad fixes”
- Allows consistent capture and analysis of test results
- Its cost is recovered through increased productivity and better system quality
- More time to design better tests, rather than entering and reentering tests

4



Automated testing advantages

- Unlike manual testing, it is not error-prone and tedious
- Only feasible way to do regression testing
- Necessary to run long and complex tests
- Easily evaluates large quantities of output

5



Limitations and caveats

- A skilled tester can use his experience to react to manual testing results by improvising effective tests
- Automated tests are expensive to create and maintain
- If the implementation is changing frequently, maintaining the test suite might be hard

6

XP approach to testing

- In the Extreme Programming approach
 - Tests are written before the code itself
 - If the code has no automated test cases, it is assumed not to work
 - A testing framework is used so that automated testing can be done after every small change to the code
 - This may be as often as every 5 or 10 minutes
 - If a bug is found after development, a test is created to keep the bug from coming back

7

XP consequences

- Fewer bugs
- More maintainable code
- The code can be refactored without fear
- Continuous integration
 - During development, the program *always works*
 - It may not do everything required, but what it does, it does right

8

JUnit

- JUnit is a framework for writing tests
 - Written by Erich Gamma (of Design Patterns fame) and Kent Beck (creator of XP methodology)
 - Uses Java 5 features such as annotations and static imports
 - JUnit helps the programmer:
 - define and execute tests and test suites
 - formalize requirements
 - write and debug code
 - integrate code and always be ready to release a working version

9

Terminology

- A **test fixture** sets up the data (both objects and primitives) that are needed for every test
 - Example: If you are testing code that updates an employee record, you need an employee record to test it on
- A **unit test** is a test of a *single* class
- A **test case** tests the response of a single method to a particular set of inputs
- A **test suite** is a collection of test cases
- A **test runner** is software that runs tests and reports results

10

Structure of a JUnit test class

- To test a class named **Fraction**
- Create a test class **FractionTest**

```
import org.junit.*;
import static org.junit.Assert.*;
public class FractionTest
{
    ...
}
```

11

Test fixtures

- Methods annotated with `@Before` will execute before every test case
- Methods annotated with `@After` will execute after every test case

```
@Before
public void setUp() {...}
@After
public void tearDown() {...}
```

12

Class Test fixtures

- Methods annotated with `@BeforeClass` will execute once before all test cases
- Methods annotated with `@AfterClass` will execute once after all test cases
- These are useful if you need to allocate and release expensive resources once

13

Test cases

- Methods annotated with `@Test` are considered to be test cases

```
@Test
public void testadd() {...}
@Test
public void testToString() {...}
```

14

What JUnit does

- For *each* test case `t`:
 - JUnit executes all `@Before` methods
 - Their order of execution is not specified
 - JUnit executes `t`
 - Any exceptions during its execution are logged
 - JUnit executes all `@After` methods
 - Their order of execution is not specified
- A report for all test cases is presented

15

Within a test case

- Call the methods of the class being tested
- Assert what the correct result should be with one of the provided [assert methods](#)
- These steps can be repeated as many times as necessary
- An assert method is a JUnit method that performs a test, and throws an [AssertionError](#) if the test fails
 - JUnit catches these exceptions and shows you the results

16

List of assert methods 1

- `assertTrue(boolean b)`
`assertTrue(String s, boolean b)`
 - Throws an [AssertionError](#) if `b` is False
 - The optional message `s` is included in the Error
- `assertFalse(boolean b)`
`assertFalse(String s, boolean b)`
 - Throws an [AssertionError](#) if `b` is True
 - All assert methods have an optional message

17

Example: Counter class

- Consider a trivial “counter” class
 - The constructor creates a counter and sets it to zero
 - The [increment](#) method adds one to the counter and returns the new value
 - The [decrement](#) method subtracts one from the counter and returns the new value
 - The corresponding JUnit test class...

18

```

public class CounterTest {
    Counter counter1;

    @Before
    public void setUp() { // creates a (simple) test fixture
        counter1 = new Counter();
    }

    @Test
    public void testIncrement() {
        assertTrue(counter1.increment() == 1);
        assertTrue(counter1.increment() == 2);
    }

    @Test
    public void testDecrement() {
        assertTrue(counter1.decrement() == -1);
    }
}

```

Note that each test begins with a *brand new* counter
 This means you don't have to worry about the order in which the tests are run

List of assert methods 2

- `assertEquals(Object expected, Object actual)`
- Uses the `equals` method to compare the two objects
- Primitives can be passed as arguments thanks to autoboxing
- Casting may be required for primitives
- There is also a version to compare arrays

20

List of assert methods 3

- `assertSame(Object expected, Object actual)`
 - Asserts that two references are attached to the same object (using `==`)
- `assertNotSame(Object expected, Object actual)`
 - Asserts that two references are not attached to the same object

21

List of assert methods 4

- `assertNull(Object object)`
Asserts that a reference is null
- `assertNotNull(Object object)`
Asserts that a reference is not null
- `fail()`
Causes the test to fail and throw an `AssertionError`
 - Useful as a result of a complex test, or when testing for exceptions

22

Testing for exceptions

- If a test case is expected to raise an exception, it can be noted as follows

```

@Test(expected = Exception.class)
public void testException() {
    //Code that should raise an exception
    fail("Should raise an exception");
}

```

23

```

public void testAnIOExceptionIsThrown {
    try
    {
        // Code that should raise an IO exception
        fail("Expected an IO exception");
    }
    catch (IOException e)
    {
        // This is the expected result, so
        // leave it empty so that the test
        // will pass. If you care about
        // particulars of the exception, you
        // can test various assertions about
        // the exception object
    }
}

```

The assert statement

- A statement such as
`assert boolean_condition;`
will also throw an `AssertionError` if the `boolean_condition` is false
- Can be used instead of the JUnit `assertTrue` method

25

Ignoring test cases

- Test cases that are not finished yet can be annotated with `@Ignore`
- JUnit will not execute the test case but will report how many test cases are being ignored

26

Automated testing issues

- It isn't easy to see how to unit test GUI code
- JUnit is designed to call methods and compare the results they return against expected results
 - This works great for methods that *just* return results, but many methods have side effects

27

Automated testing issues

- To test methods that do output, you have to capture the output
 - It's possible to capture output, but it's an unpleasant coding chore
- To test methods that change the state of the object, you have to have code that checks the state
 - It's a good idea to have methods that test state invariants

28

First steps toward solutions

- You can redefine `System.out` to use a different `PrintStream` with
`System.setOut(PrintStream)`
- You can “automate” GUI use by “faking” events
 - We will see this in more detail later

29

JUnit in Eclipse

- JUnit can be downloaded from <http://junit.sourceforge.net/>
- If you use Eclipse, as in this course, you do not need to download anything
- Eclipse contains wizards to help with the development of test suites with JUnit
- JUnit results are presented in an Eclipse window

Hello World demo

- Run Eclipse
- File -> New -> Project, choose Java Project, and click Next. Type in a project name, e.g. ProjectWithJUnit.
- Click Next
- Click Create New Source Folder, name it test
- Click Finish
- Click Finish

Create a class

- Right-click on ProjectWithJUnit
Select New -> Package
Enter package name, e.g. **eeecs4313**
Click Finish
- Right-click on eeecs4313
Select New -> Class
Enter class name, e.g. **HelloWorld**
Click Finish

Create a class - 2

- Add a dummy method such as
`public String say() { return null; }`
- Right-click in the editor window and select Save

Create a test class

- Right-click on the HelloWorld class
Select New -> JUnit Test Case
- Change the source folder to test as opposed to src
- Check to create a setup method
- Click Next

Create a test class

- Check the checkbox for the say method
 - This will create a stub for a test case for this method
- Click Finish
- Click OK to "Add JUnit 4 library to the build path"
- The HelloWorldTest class is created
- The first version of the test suite is ready

Run the test class - 1st try

- Right click on the HelloWorldTest class
- Select Run as -> JUnit Test
- The results appear in the left
- The automatically created test case fails



Create a better test case

- Import the class under test

```
import eecs4313.HelloWorld;
```
- Declare an attribute of type HelloWorld

```
HelloWorld hi;
```
- The setup method should create a HelloWorld object

```
hi = new HelloWorld();
```
- Modify the testSay method body to

```
assertEquals("Hello World!",  
             hi.say());
```



Run the test class - 2nd try

- Save the new version of the test class and re-run
- This time the test fails due to expected and actual not being equal
- The body of the method `say` has to be modified to

```
return "Hello World!";
```

for the test to pass



Create a test suite

- Right-click on the eecs4313 package in the test source folder
- Select New -> Class. Name the class **AllTests**.
- Modify the class text so it looks like class AllTests on the course website
- Run with Run -> Run As -> JUnit Test
- You can easily add more test classes