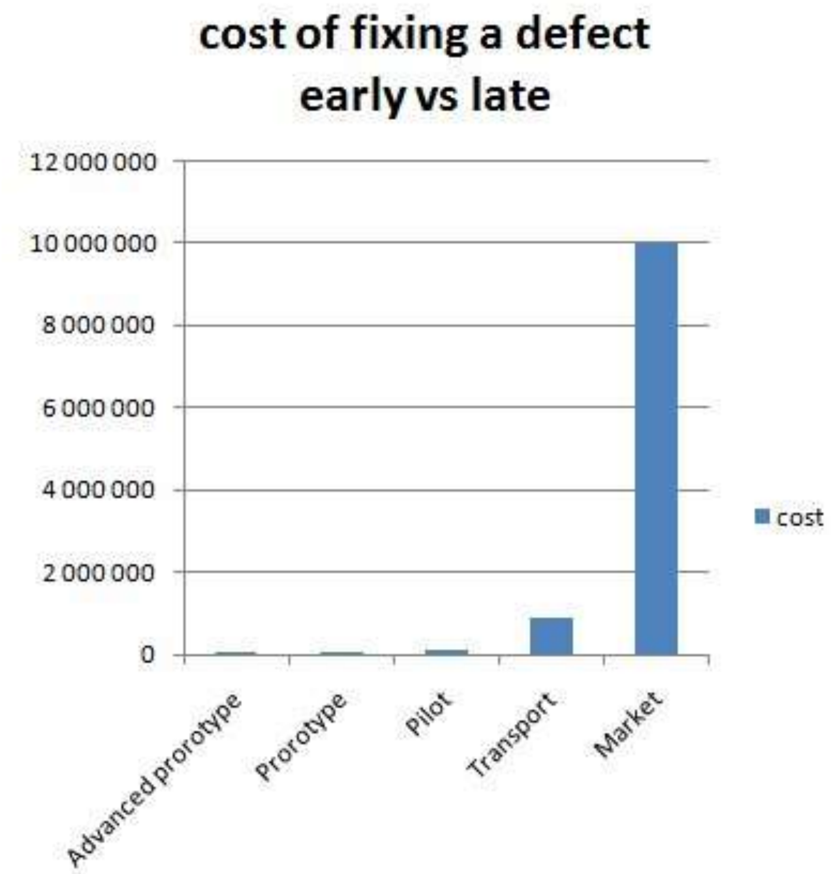
An Introduction to JUnit

Animesh Kumar

Toyota



http://blog.crisp.se/henrikkniberg/2010/03/16/1268757660000.html

Toyota's Prius example

A defect found in the production phase is about 50 times more expensive than if it is found during prototyping. If the defect is found after production it will be about 1,000 – 10,000 times more expensive

Reality turned out to be worse, *a lot* worse! Toyota's problems with the Prius braking systems is costing them over $2 000 000 000 (2 billion!) to fix because of all the recalls and lost sales. "Toyota announced that a glitch with the software program that controls the vehicle's anti-lock braking system was to blame".

Why write tests?

- Write a lot of code, and one day something will **stop** working!
- Fixing a bug is easy, but how about **pinning** it down?
- You never know where the **ripples** of your fixes can go.
- Are you sure the demon is **dead**?

Test Driven Development

An evolutionary approach to development where first you write a **Test** and then add **Functionality** to pass the test.

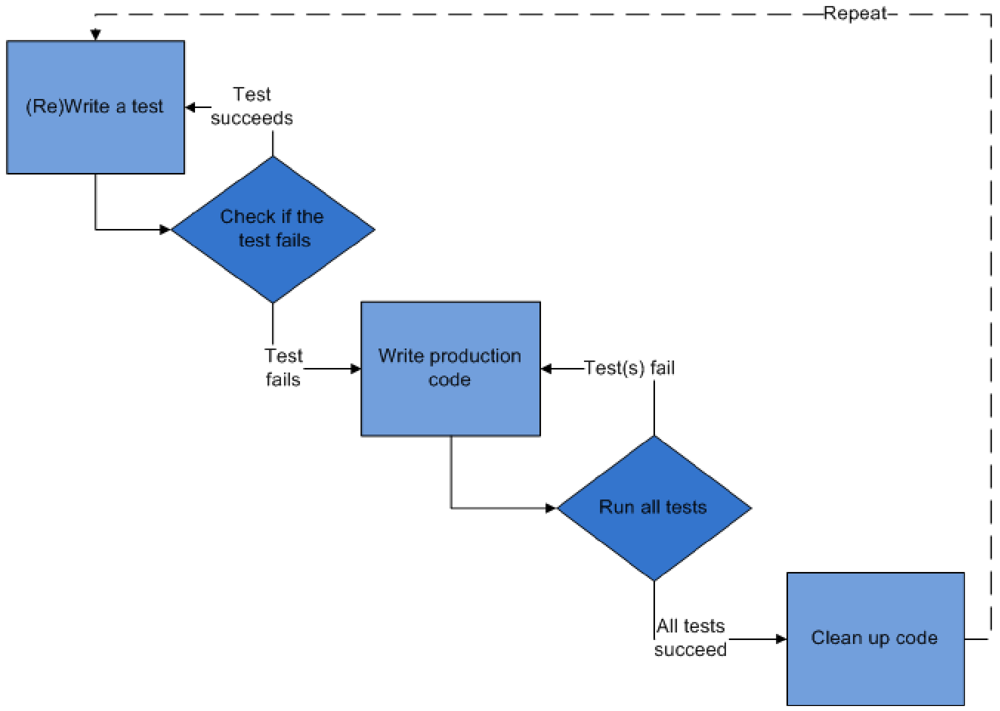


Image courtesy: http://en.wikipedia.org/wiki/Test-driven_development

What is Unit Testing?

- A Unit Test is a procedure to validate a single Functionality of an application.

One Functionality → One Unit Test

- Unit Tests are **automated** and self-checked.
- They run in **isolation** of each other.
- They do **NOT** depend on or connect to external resources, like DB, Network etc.
- They can run in **any order** and even parallel to each other and that will be fine.

What do we get?

Obviously, we get tests to validate the functionalities, but that’s not all…

- One part of the program is isolated from others, i.e. proper separation of concerns.
- An **ecology** to foster Interface/Contract approached programming.
- Interfaces are documented.
- Refactoring made smooth like butter lathered floor.
- Quick bug identification.

JUnit – An introduction

- JUnit is a unit **testing framework** for the java programming language.
- It comes from the family of unit testing frameworks, collectively called **xUnit**, where '**x**' stands for the programming language, e.g. CPPUnit, JSUnit, PHPUnit, PyUnit, RUnit etc.

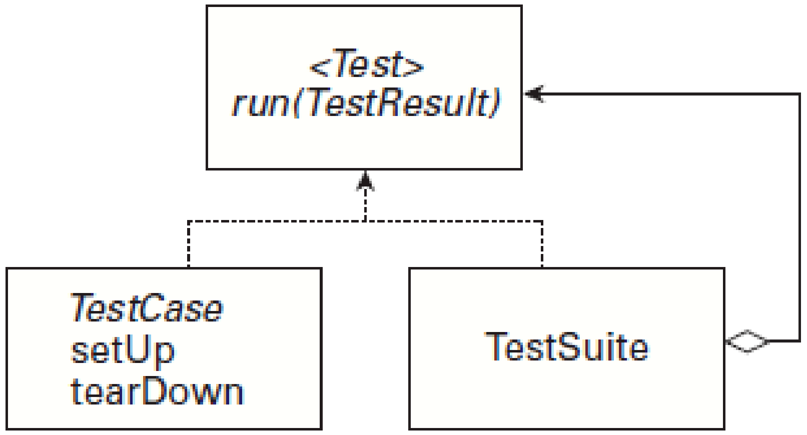
Kent Beck and Erich Gamma originally wrote this framework for 'smalltalk', and it was called SUnit. Later it rippled into other languages.

Why choose JUnit?

Of course there are many tools like JUnit, but none like it.

- JUnit is **simple** and elegant.
- JUnit checks its own results and provide immediate customized feedback.
- JUnit is **hierarchal**.
- JUnit has the widest IDE support, Eclipse, NetBeans, IntelliJ IDEA … you name it.
- JUnit is recognized by popular build tools like, ant, maven etc.
- And… it's **free**. ☺

Design of JUnit



- junit.framework.TestCase is the abstract **command** class which you subclass into your Test Classes.
- junit.framework.TestSuite is a **composite** of other tests, either TestCase or TestSuite. This behavior helps you create hierarchal tests with depth control.

Design of JUnit

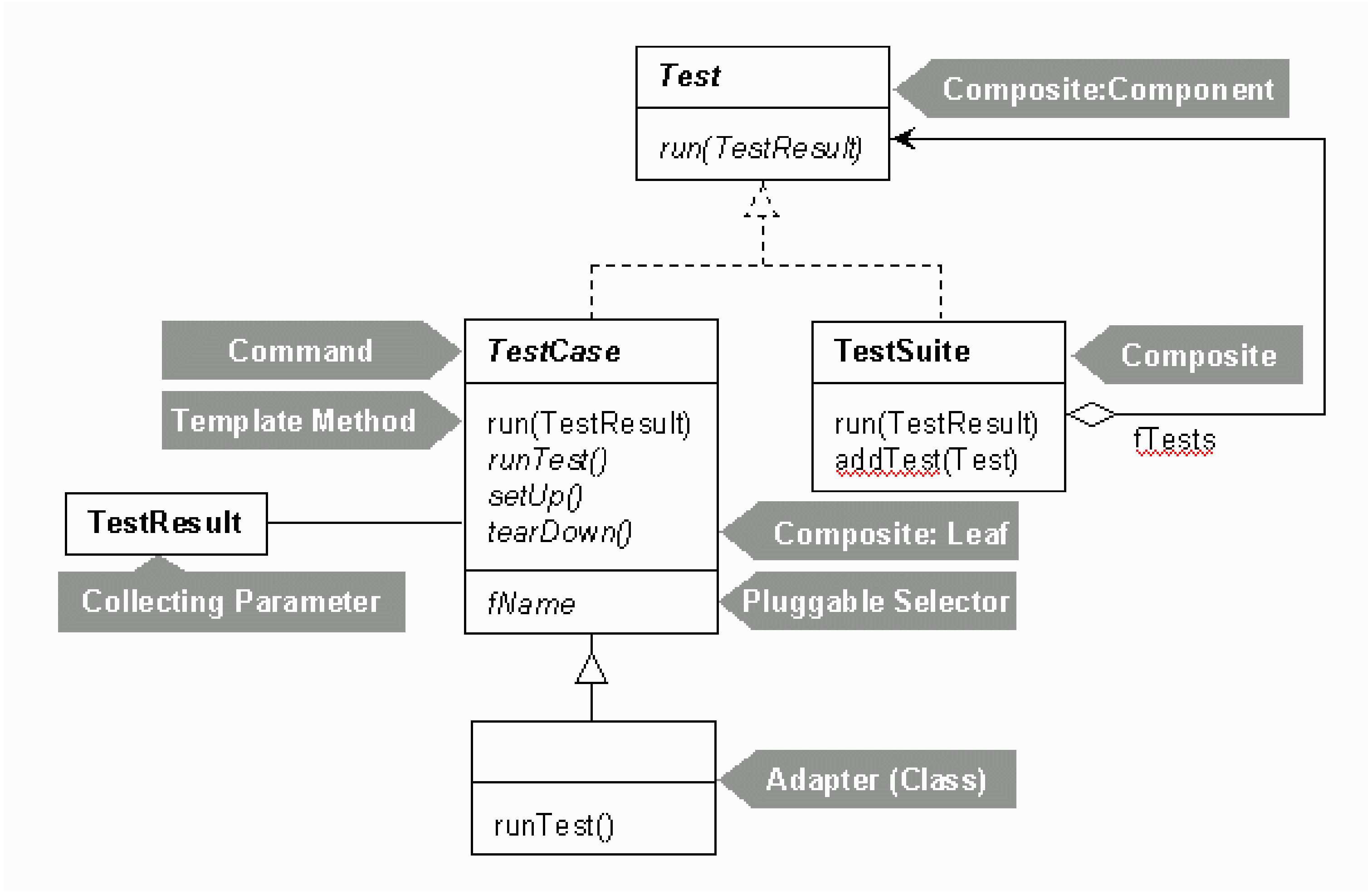


Image Courtesy: JUnit: A Cook's Tour

Write a test case

- Define a subclass of junit.framework.TestCase

```
public class CalculatorTest extends TestCase {
}
```

- Define one or more testXXX() methods that can perform the tests and **assert** expected results.

```
public void testAddition () {
  Calculator calc = new Calculator();
  int expected = 25;
  int result = calc.add (10, 15);
  assertEquals (result, expected); // asserting
}
```

Write a test case

- Override setUp() method to perform initialization.

```
@Override
protected void setUp () {
  // Initialize anything, like
  calc = new Calculator();
}
```

- Override tearDown() method to clean up.

```
@Override
protected void tearDown () {
  // Clean up, like
  calc = null;
}
```

Asserting expectations

- assertEquals (expected, actual)
- assertEquals (message, expected, actual)
- assertEquals (expected, actual, delta)
- assertEquals (message, expected, actual, delta)
- assertFalse ((message)condition)
- Assert(Not)Null (object)
- Assert(Not)Null (message, object)
- Assert(Not)Same (expected, actual)
- Assert(Not)Same (message, expected, actual)
- assertTrue ((message), condition)

Failure?

- JUnit uses the term failure for a test that fails expectedly. That is
  - An assertion was not valid or
  - A fail() was encountered.

Write a test case

```
public class CalculatorTest extends TestCase {
  // initialize
  protected void setUp ()...                      S

  public void testAddition ()...                  T1
  public void testSubtraction ()...               T2
  public void testMultiplication ()...            T3
  // clean up
  protected void tearDownUp ()...                 C
}
```

Execution will be S T1 C, S T2 C, S T3 C in any order.

Write a test suite

Write a class with a static method suite() that creates a junit.framework.TestSuite containing all the Tests.

```
public class AllTests {
   public static Test suite() {
        TestSuite suite = new TestSuite();
        suite.addTestSuite(<test-1>.class);
        suite.addTestSuite(<test-2>.class);
        return suite;
   }
}
```

Run your tests

- You can either run TestCase or TestSuite instances.
- A TestSuite will automatically run all its registered TestCase instances.
- All public testXXX() methods of a TestCase will be executed. But there is no guarantee of the order.

Run your tests

- JUnit comes with TestRunners that run your tests and immediately provide you with feedbacks, errors, and status of completion.
- JUnit has Textual and Graphical test runners.

- Textual Runner

```
>> java junit.textui.TestRunner AllTests
or,
junit.textui.TestRunner.run(AllTests.class);
```

- Graphical Runner

```
>> java junit.swingui.TestRunner AllTests
or,
junit.swingui.TestRunner.run(AllTests.class);
```

- IDE like Eclipse, NetBeans “Run as JUnit”

Test maintenance

- Create 2 separate directories for **source** and **test** codes.
- Mirror source **package** structure into **test**.
- Define a **TestSuite** for each java package that would contain all **TestCase** instances for this package. This will create a hierarchy of Tests.

```
> com                       -
    > impetus               - AllTests
        > Book              - BookTests
            . AddBook           - AddBookTest
            . DeleteBook        - DeleteBookTest
            . ListBooks         - ListBooksTest
        > Library           - LibraryTests
        > User              - UserTests
```

Mocks

- Mocking is a way to deal with third-party dependencies inside TestCase.
- Mocks create ***FAKE*** objects to mock a contract behavior.
- Mocks help make tests more ***unitary***.

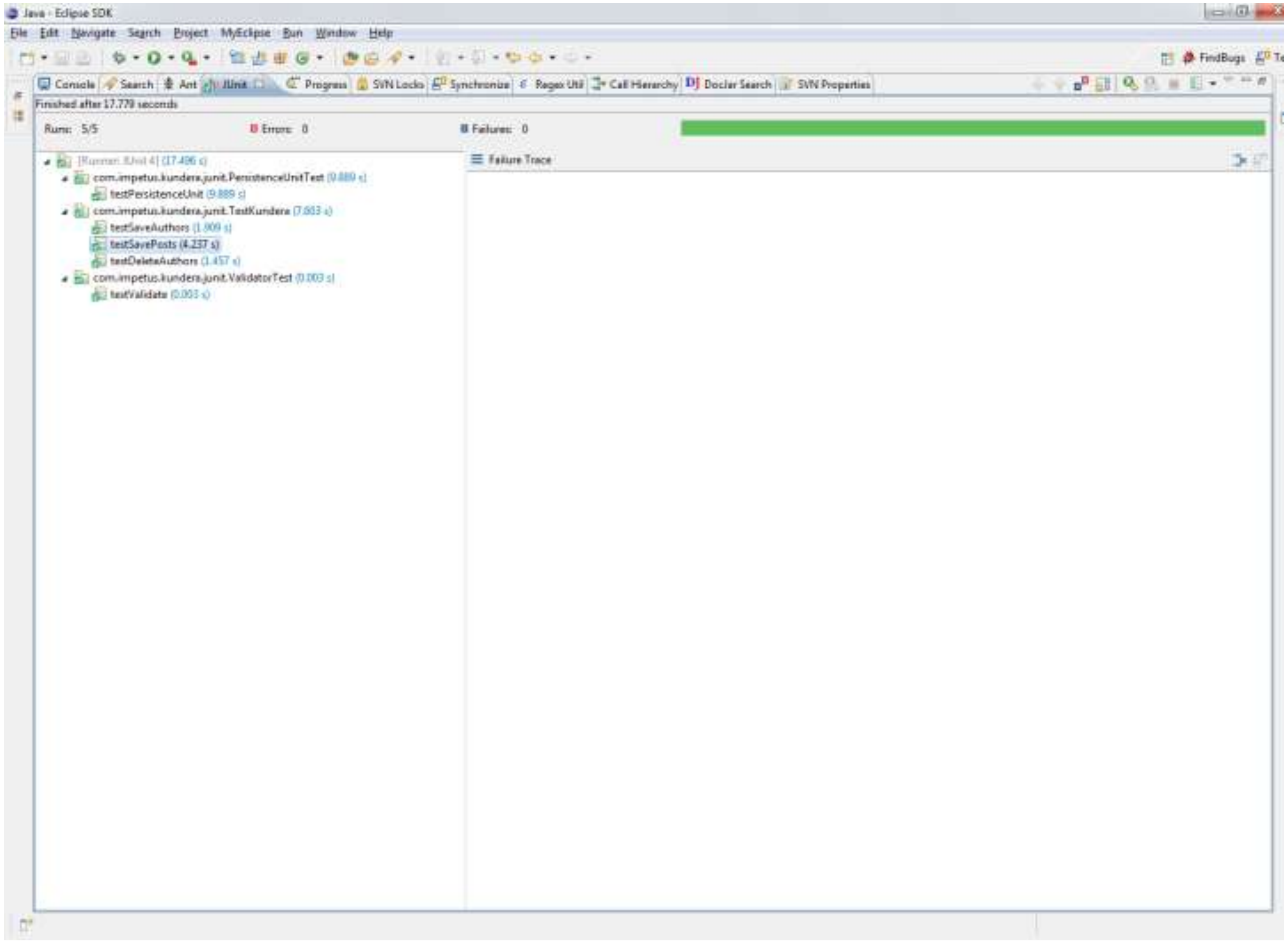
EasyMock

- EasyMock is an open-source java library to help you create Mock Objects.
- Flow: **Expect** => **Replay** => **Verify**

```
userService = EasyMock.createMock(IUserService.class);          1
User user = ... //
EasyMock
   .expect (userService.isRegisteredUser(user))                 2
   .andReturn (true);                                           3
EasyMock.replay(userService);                                   4
assertTrue(userService.isRegisteredUser(user));                 5
```

- Modes: Strict and Nice

JUnit and Eclipse



```
Finished after 17.779 seconds

Runs: 5/5          Errors: 0          Failures: 0

[Runner: JUnit 4] (17.496 s)
  com.impetus.kundera.junit.PersistenceUnitTest (9.889 s)
    testPersistenceUnit (9.889 s)
  com.impetus.kundera.junit.TestKundera (7.603 s)
    testSaveAuthors (1.909 s)
    testSavePosts (4.237 s)
    testDeleteAuthors (1.457 s)
  com.impetus.kundera.junit.ValidatorTest (0.003 s)
    testValidate (0.003 s)
```

JUnit and build tools

- Apache Maven
  - > `mvn test`
  - > `mvn -Dtest=`**`MyTestCase,MyOtherTestCase`**` test`
- Apache Ant

```
<junit printsummary="yes" haltonfailure="yes">
  <test name="my.test.TestCase"/>
</junit>

<junit printsummary="yes" haltonfailure="yes">
  <batchtest fork="yes" todir="${reports.tests}">
    <fileset dir="${src.tests}">
      <include name="**/*Test*.java"/>
      <exclude name="**/AllTests.java"/>
    </fileset>
  </batchtest>
</junit>
```

Side effects – good or bad?

- Designed to call methods.
  - This works great for methods that just return results
  - Methods that change the state of the object could turn out to be tricky to test
  - Difficult to unit test GUI code
- Encourages “functional” style of coding
  - This can be a <u>**good**</u> thing

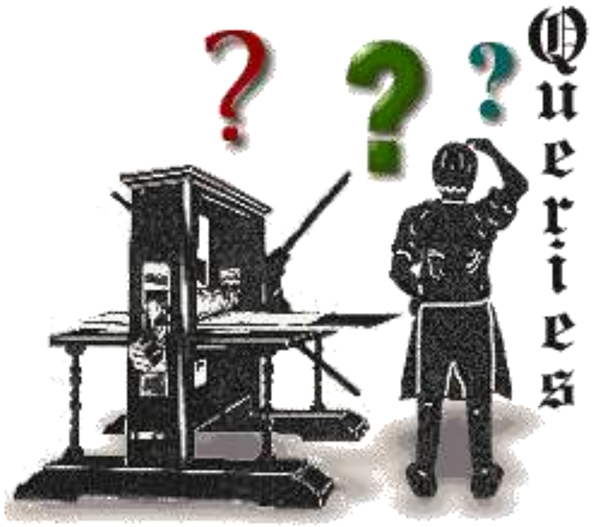
How to approach?

- Test a little, Code a little, Test a little, Code a little … doesn't it rhyme? ;)
- Write tests to validate **functionality**, not functions.
- If tempted to write **System.out.println()** to debug something, better write a test for it.
- If a bug is found, write a test to expose the bug.

Resources

- Book: Manning: JUnit in Action

- http://www.junit.org/index.htm
- http://www.cs.umanitoba.ca/~eclipse/10-JUnit.pdf
- http://supportweb.cs.bham.ac.uk/documentation/tutorials/docsystem/build/tutorials/junit/junit.pdf
- http://junit.sourceforge.net/javadoc/junit/framework/



impetus™