Unit Testing

Stéphane Ducasse
Stephane.Ducasse@univ-savoie.fr
http://www.listic.univ-savoie.fr/~ducasse/
License: CC-Attribution-ShareAlike 2.0

http://creativecommons.org/licenses/by-sa/2.0/

Creative Commons Attribution-ShareAlike 2.0

You are free:

- to copy, distribute, display, and perform the work
- to make derivative works
- to make commercial use of the work

Under the following conditions:

BY: Attribution. You must give the original author credit.

Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

This is a human-readable summary of the Legal Code (the full license).
Goal

• How can I trust that the changes did not destroy something?
• What is my confidence in the system?
• How do I write tests?
• What is unit testing?
• What is SUnit?
Tests

• Tests represent your *trust* in the system
• Build them *incrementally*
  • Do not need to focus on *everything*
  • When a *new* bug shows up, write a test
• Even better write them before the code
  • Act as your *first client*, better interface
• Active documentation always in sync
Testing Style

“The style here is to write a few lines of code, then a test that should run, or even better, to write a test that won't run, then write the code that will make it run.”

- write unit tests that thoroughly test a single class
- write tests as you develop (even before you implement)
- write tests for every new piece of functionality

“Developers should spend 25-50% of their time developing tests.”
But I can’t cover everything!

- Sure! Nobody can but
- When someone discovers a defect in your code, first write a test that demonstrates the defect.
- Then debug until the test succeeds.

“Whenever you are tempted to type something into a print statement or a debugger expression, write it as a test instead.” Martin Fowler
Unit Testing

- Ensure that you get the specified behavior of the public interface of a class
- Normally tests a single class

A test
- Create a context,
- Send a stimulus,
- Check the results
SetTestCase

Class: SetTestCase
    superclass: TestCase

SetTestCase>>testAdd
    | empty |
    empty := Set new.
    empty add: 5.
    self assert: (empty includes: 5).

SetTestCase run: #testAdd

"Context"
"Stimulus"
"Check"
In a subclass of `TestCase`

- Each method starting with `test`*
- Represents a test
- Is automatically executed
- The results of the test are collected in a `TestResult` object
Examples

testExampleRunArray3
"this demonstrates that adjacent runs with equal attributes are merged."

| runArray |
runArray := RunArray new.
runArray
   addLast: TextEmphasis normal times: 5;
   addLast: TextEmphasis bold times: 5;
   addLast: TextEmphasis bold times: 5.
self assert: (runArray runs size = 2).
Failures and Errors

- A failure is a failed assertion, i.e., an anticipated problem that you test.
- An error is a condition you didn’t check for.

SetTestCase>>removeElementNotInSet

self should: [Set new remove: 1]
raise: Error
Good Tests

- Repeatable
- No human intervention
- “self-described”
- Change less often than the system
- Tells a story
_Unit Frameworks

- _Unit is a simple “testing framework” that provides:
  - classes for writing Test Cases and Test Suites
  - methods for setting up and cleaning up test data (‘fixtures’)
  - methods for making assertions
  - textual and graphical tools for running tests

- _Unit distinguishes between failures and errors:
  - A failure is a failed assertion, i.e., an anticipated problem that you test.
  - An error is a condition you didn’t check for.
JUnit

- Junit (inspired by SUnit) is a simple “testing framework” that provides:
  - classes for writing Test Cases and Test Suites
  - methods for setting up and cleaning up test data (“fixtures”)
  - methods for making assertions
  - textual and graphical tools for running tests
The JUnit Framework

A Test can run a number of concrete test cases

```
interface Test
+ countTestCases() : int
+ run(TestResult)
```

```
TestSuite
+ create()
+ create(Class)
+ addTest(Test)

A TestSuite bundles a set of Tests
```

```
TestCase abstract
+ create(String)
+ fail()
+ void runBare()
# void runTest()
# void setUp()
# void tearDown()
+ name() : String
```

```
公用 Assert
+ assertTrue(boolean)
+ assertEquals(Object, Object)
...
```

```
TestResult
+ create()
# void run(TestCase)
+ addError(TestCase, Throwable)
+ addFailure(TestCase, Throwable)
+ errors() : Enumeration
+ failures() : Enumeration
```

All errors and failures are collected into a TestResult.
A Testing Scenario

The framework calls the test methods that you define for your test cases.
SUnit

- Original framework
SetTestCase

Class: SetTestCase
superclass: TestCase

SetTestCase>>testAddition
| s |
s := Set new.
s add: 5; add: 3.
self assert: s size = 2.
s add: 5.
self assert: s size = 2
Duplicating the Context

Set TestCase>>testOccurrences

| empty |
empty := Set new.
self assert: (empty occurrencesOf: 0) = 0.
empty add: 5; add:5.
self assert: (empty occurrencesOf: 5) = 1
Testing Remove

SetTestCase>>testAddition

| s |

s := Set new.
s add: 6.
self assert: s size = 1.
s remove: 6.
self assert: s size = 0.
self should: [s remove: 1000] raise: Error.
res := s remove: 5 ifAbsent: [33].
self assert: (res = 33)
setUp and tearDown

- Executed before and after each test
- setUp allows us to specify and reuse the context
- tearDown to clean after.
Example: Testing Set

- Class: SetTestCase
  superclass: TestCase
  instance variable: ‘empty full’

- SetTestCase>>setUp
  empty := Set new.
  full := Set with: 6 with: 5

- The setUp is the context in which each test is run.
Tests…

SetTestCase>>testAdd
empty add: 5.
self assert: (empty includes: 5).

SetTestCase>>testOccurrences
self assert: (empty occurrenceOf: 0) = 0.
self assert: (full occurrencesOf: 5) = 1.
full add: 5.
self assert: (full occurrencesOf: 5) = 1

SetTestCase>>testRemove
full remove: 5.
self assert: (full includes: 6).
self deny: (full includes: 5)
SUnit Core

A Test Suite bundles a set of tests

TestSuite
- run
- resources
- addTest:

TestResult
- passedCount
- runCount
- failuresCount
- errorCount
- tests

TestCase
- setUp
- tearDown
- assert:
- deny:
- should:raise:
- shouldnt:raise:
- run
- resources

TestResource
- isAvailable
- isUnavailable

A Test verifies certain conditions in a given context

S.Ducasse
TestSuite, TestCase and TestResult

ATestCase represents one test
SetTestCase>>testOccurenceOf

AtestSuite is a group of tests
SUnit automatically builds a suite from the methods starting with ‘test*’

TestResult represents a test execution results
Test Resources

- A Test Resource is an object which is needed by a number of Test Cases, and whose instantiation is so costly in terms of time or resources that it becomes advantageous to only initialize it once for a Test Suite run.
TestResources

A TestResources is invoked once before any test is run.

Does not work if you have mutually exclusive TestResources.
Refactorings and Tests
What is Refactoring?

• The process of changing a software system in such a way that it does not alter the external behaviour of the code, yet improves its internal structure [Fowl99a]

• A behaviour-preserving source-to-source program transformation [Robe98a]

• A change to the system that leaves its behaviour unchanged, but enhances some non-functional quality - simplicity, flexibility, understandability, ... [Beck99a]
## Typical Refactorings

List of refactorings provided by the refactoring browser

<table>
<thead>
<tr>
<th>Class Refactorings</th>
<th>Method Refactorings</th>
<th>Attribute Refactorings</th>
</tr>
</thead>
<tbody>
<tr>
<td>add (sub)class to hierarchy</td>
<td>add method to class</td>
<td>add variable to class</td>
</tr>
<tr>
<td>rename class</td>
<td>rename method</td>
<td>rename variable</td>
</tr>
<tr>
<td>remove class</td>
<td>remove method</td>
<td>remove variable</td>
</tr>
<tr>
<td>push method down</td>
<td>push variable down</td>
<td></td>
</tr>
<tr>
<td>push method up</td>
<td>pull variable up</td>
<td></td>
</tr>
<tr>
<td>add parameter to method</td>
<td>create accessors</td>
<td></td>
</tr>
<tr>
<td>move method to component</td>
<td>abstract variable</td>
<td></td>
</tr>
<tr>
<td>extract code in new method</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Why Refactoring?

“Grow, don’t build software” Fred Brooks

• Some argue that good design does not lead to code needing refactoring,
• But in reality
  - Extremely difficult to get the design right the first time
  - You cannot fully understand the problem domain
  - You cannot understand user requirements, if he does!
  - You cannot really plan how the system will evolve in five years
  - Original design is often inadequate
  - System becomes brittle, difficult to change
• Refactoring helps you to
  - Manipulate code in a safe environment (behavior preserving)
  - Create an environment a situation where evolution is possible
  - Understand existing code
Refactor To Understand

Obvious

• Programs hard to read => Programs hard to understand => Programs hard to modify
• Programs with duplicated logic are hard to understand
• Programs with complex conditionals are hard to understand
• Programs hard to modify

Refactoring code creates and supports the understanding

• Renaming instance variables helps understanding methods
• Renaming methods helps understanding responsibility
• Iterations are necessary

The refactored code does not have to be used!
Test and Refactorings

- Tests can cover places where you have to manually change the code
- Changing 3 by 33, nil but NewObject new
- Tests let you been more aggressive to change and improve your code
Summary

If you are serious about programming
If you do not have time to lose
If you want to have synchronized documentations

Write unit tests...