Unit Testing

Stéphane Ducasse
stephane.ducasse@inria.fr
http://stephane.ducasse.free.fr/

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.              "Context"
    empty add: 5.                  "Stimulus"
    self assert: (empty includes: 5).      "Check"


SetTestCase run: #testAdd
```

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.

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.
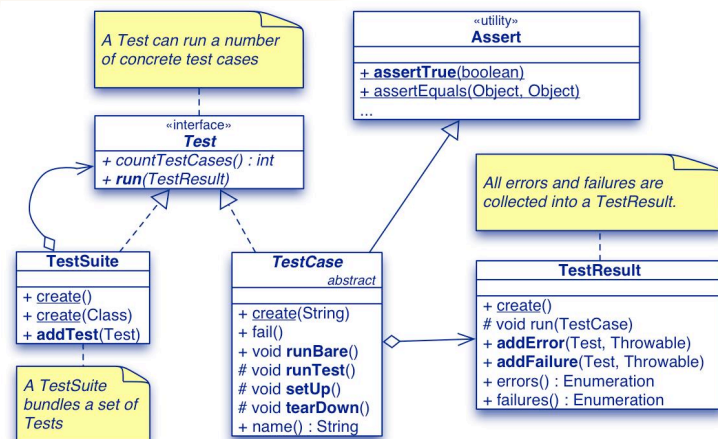   - An error is a condition you didn't check for.
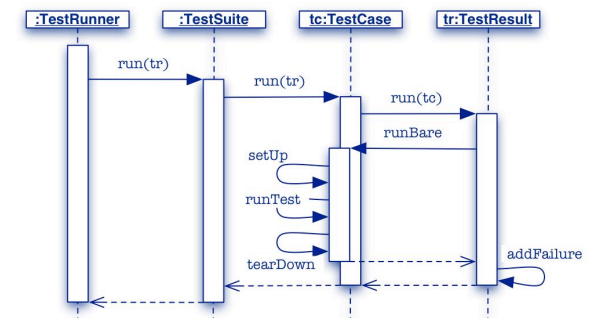
JUnit

JUnit

- 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

«utility»
Assert

+ **assertTrue**(boolean)
+ assertEquals(Object, Object)
...

«interface»
Test
+ countTestCases() : int
+ **run**(TestResult)

All errors and failures are collected into a 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

TestResult

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

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**

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)

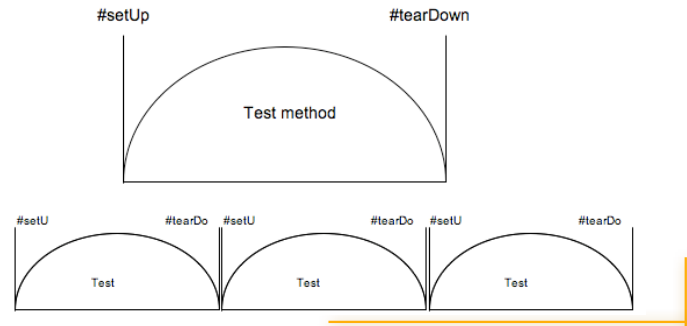
Duplicating the Context

SetTestCase>>testOccurrences

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

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
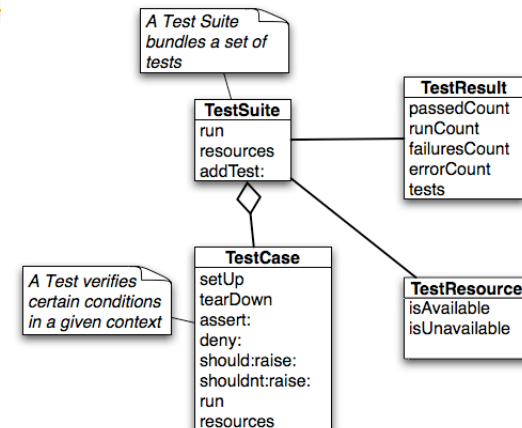
TestSuite, TestCase and TestResult

a TestCase represents one test
SetTestCase>>testOccurenceOf

A testSuite is a group of tests
SUnit automatically builds a suite from the methods
starting with 'test*'

TestResult represents a test execution results

Test Ressources

- 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

Class Refactorings	Method Refactorings	Attribute Refactorings
add (sub)class to hierarchy	add method to class	add variable to class
rename class	rename method	rename variable
remove class	remove method	remove variable
	push method down	push variable down
	push method up	pull variable up
	add parameter to method	create accessors
	move method to component	abstract variable
	extract code in new method	

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...