Unit Testing Explained

- How to support changes?
- How to support basic but synchronized documentation?

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."

Good Tests

- Repeatable
- No human intervention
- "self-described"
- Change less often than the system
- Tells a story

Changes

- Changes are costly
  + Client checking…
  + Documentation
- Introduce bugs, hidden ripple effects
- System "sclerosis"
- Less and less axes of freedom

Unit Testing

- Lot of theory and practices behind tests
  + Black-box, whitebox, paths…
- Put to the light again with XP emergence
- How can I trust that the changes did not destroy something?
- What is my confidence in the system?
- Refactoring are ok but when I change 3 to 5, is my system still working

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 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.

The SUnit Framework

- A Test Suite [function a set of tests]
- A Test Run [invoke the tests]
- A TestResult [return the results of the test run]
- A TestContext [context in which the test results are collected]
The JUnit Framework

Testing Set Addition
- Class: SetTestCase
  superclass: TestCase
  remove: | s |
  s := Set new.
  s add: 5; add: 3.
  self assert: s size = 2.
  s add: 5.
  s assert: s size = 2.

Testing Set Creation
- SetTestCase>>testCreation
  self assert: empty isEmpty.
  self deny: full isEmpty.

Testing Remove
- SetTestCase>>testRemove
  empty remove: 5.
  self assert: (empty includes: 5).

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

- SetTestCase>>testAdd2
  empty add: 5.
  empty add: 5.
  self assert: (empty includes: 5).
  full add: 5
  self assert: (full size = 2).

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.

A Testing Scenario
- The framework calls the test methods that you define for your test cases.

Reusing the Context
- Class: SetTestCase
  superclass: TestCase
  instance variable: empty full

Occurences and Remove
- SetTestCase>>testOccurences
  self assert: (empty occurrenceOf: 0) = 0.
  self assert: (full occurrenceOf: 5) = 1.

- SetTestCase>>testRemove
  full remove: 5.
  self assert: (full includes: 6).
  self deny: (full includes: 5).
Exceptions

Set TestCase >> testRemoveNonExistingElement
	sel should: [empty remove: 5]
raise: Error

Refactorings

* Behavior preserving source code transformation

Synergy between Tests 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