Unit Testing Explained

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

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

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

Good Tests

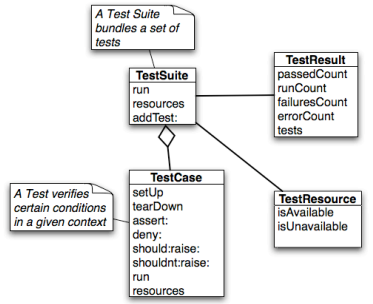
- *Repeatable*
- *No* human intervention
- "*self*-described"
- Change less often than the system
- Tells a story

_Unit Frameworks

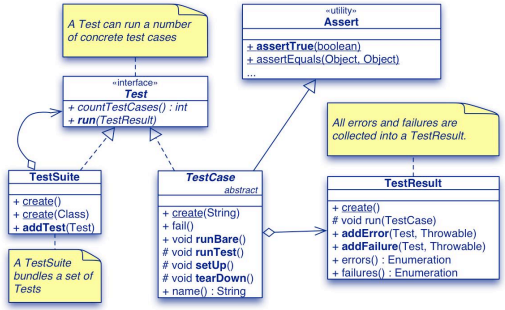
_Unit is a simple "testing framework" that provides:
- classes for writing *Test Cases and Test Suites*
- methods for *setting up and cleaning up test data* ("fixtures")
- methods for *making assertions*
- textual and graphical tools for *running tests*

_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.*

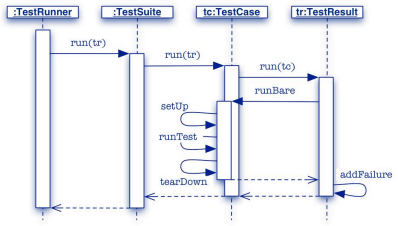
The SUnit Framework



The JUnit Framework



A Testing Scenario



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

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

Testing Set Addition

- Class: SetTestCase
  superclass: TestCase
- SetTestCase>>testAddition

```
| s |
s := Set new.
s add: 5; add: 3.
self assert: s size = 2.
s add: 5.
s 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)
```


Reusing the Context

- Class: SetTestCase
  superclass: TestCase
  instance variable: ***empty full***
- SetTestCase>>***setUp***
  empty := Set new.
  full := Set with: 6 with: 5
- The ***setUp*** method specifies the *context* in which ***each*** test is run.

Testing Set Creation

```
SetTestCase>>testCreation
    self assert: empty isEmpty.
    self deny: full isEmpty
```


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).
```


Occurrences and Remove

```
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)
```

Exceptions

```
SetTestCase>>testRemoveNonExistingElement
    self 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