Testing

“If it’s not tested, it doesn’t work”
Why Unit Testing?

• *If it is not tested, it does not work*

• Tests represent an *executable specification* of what the methods *ought* to do
  
  ▶ non-executable specifications gather dust on shelves.
Why Unit Testing (2)

• The more time between coding and testing:
  ▶ More effort is needed to write tests
  ▶ More effort is needed to find bugs
  ▶ Fewer bugs are found
  ▶ Time is wasted working with buggy code
  ▶ Development time increases
  ▶ Quality decreases
• Without unit tests:
  ▶ Code integration is a nightmare
    ◦ Changing code required more courage than I have!
Why Automated Tests?

• What is wrong with:
  ▶ Using print statements?
  ▶ Writing comments that exercise your code?
  ▶ Writing extra methods that exercise your code?
  ▶ Writing small workspace scripts to run code?
  ▶ Running program and testing it by using it?
A testing method should:

- Work with \( n \) programmers working for \( k \) months (years)
- Help when modifying code 6 months after it was written
- Check impact of code changes on rest of system
- Work in a school project as well as in industry
  - This is probably unrealistic!
- Help to build good habits and skills
We have a QA Team, so why should I write tests?

• How long does it take QA to test your code?
• How much time does your team spend working around bugs before QA tests?
• How easy is it to find & correct the errors after QA finds them?
• Most programmers already have an informal testing process
• With a little more work you can develop a useful and reusable test suite
When to Write Unit Tests

- *First* write the tests — *Test Driven Development*
- *Then* write the code to be tested
- Writing tests first saves time!
  - Makes you aware of the interface & functionality of the code
  - Removes temptation to skip tests
SUnit (and JUnit)

• Free frameworks for Unit testing
• SUnit originally written by Kent Beck 1994
• Built into VisualWorks, Squeak, ...
• JUnit written by Kent Beck & Erich Gamma
Not just for Smalltalk & Java

• Ports are available in:

.NET  Ada  AppleScript  C
C#  C++  Curl  Delphi
Eiffel  Eiffel  Flash  Forte 4GL
Gemstone/S  Haskell  HTML  Jade
LISP  Objective-C  Oracle  Palm
Perl  Php  PowerBuilder  Python
Ruby  Scheme  Smalltalk  Visual Basic
XML  XSLT
How to Use SUnit

1. Create a test class as subclass of *TestCase*

2. Write test methods
   - The framework treats methods starting with 'test' as test methods

3. Run the tests!
   - SUnit *TestRunner* is in the image.
   - TestBrowser can be downloaded from SqueakMap.
Don’t let slow tests bog you down

• Michael Feathers (http://tinyurl.com/87nj2) writes:
• A test is not a unit test if:
  – It talks to the database
  – It communicates across the network
  – It touches the file system
  – It can't run at the same time as any of your other unit tests
  – You have to do special things to your environment (such as editing config files) to run it.
Rationale

• Tests that do these things aren't bad. Often they are worth writing, and they can be written in a unit test harness.

• However, it is important to be able to separate them from true unit tests so that we can keep a set of tests that we can run fast whenever we make our changes.
Acceptance Tests vs. Unit Tests

• Unit tests:
  ▶ capture one piece of functionality
  ▶ make it easier to identify bugs in that functionality

• Acceptance tests (aka Functional tests)
  ▶ represent a scenario in the larger application
  ▶ Tests that break Feathers’ rules may make good acceptance tests.
Acceptance Tests

• Example: a compiler
  ▶ one test for each possible source language statement, makes assertions about the emitted code
  ▶ might exercise many classes, read and write from the file system …

• You can put such tests in SUnit
  ▶ but separate them from the true unit tests (why?)
How to test a client

• So, your job is to write a client that interacts with a database. How do you test it?

• Use *Mock Objects* to simulate the database
  
  
  – *Test Driven Development, A Practical Guide* by David Astels
  
  – Attend my course on XP!
Unit Tests: The Internals

• Write like any other method, but **assert**: what you want to happen

  ▶ **testAppendChar**
    
    | l string l |
    | string := ‘go’. |
    | string append: ‘ banana!’. |
    | self assert: string = ‘go banana!’. |

• Tests are run with TestRunner or the Package Browser

• So let’s write some tests for Strings!
Asserting more things

- **assert**: takes what you expect to be true
- **deny**: takes what you expect to be false
- **should**: takes a block and the kind of error it should raise
- **shouldn’t**: conversely
Unit Tests: More Details

- The setUp method happens before each testX method (the framework ensures this)
- The tearDown method happens after
- Let’s take a look at the official StringTest…
Best Practices

• Test everything that you want to work
• More test methods in your TestCase than in the class you are testing
• Tests should be as fine grained as possible
• Tests should be independent
• Should not take long to run (a few seconds)
• Easy to understand: tests read like a specification
Black's rule of testing

• Clearly:
  ▸ For every important property, there should be a test

• Not so obvious:
  ▸ For every test, there should be a property, such that when the test passes, your confidence in the property increases
  ▸ Multiple tests of same property are bad (why?)

Have a property in mind when you write a test
Tests as Specification

testNewSetIsEmpty
    assert: [set new isEmpty]

testSetsDontContainDuplicates
    a := set new.
    a add: #aThing.
    a add: #aThing.
    self assert: [a size = 1]
    description: ‘sets should not contain duplicates’.
    a remove: #aThing.
    self assert: [a isEmpty]
So why Unit Test?

- Not much work to write or run
- Documents your class
- Gives you / others confidence that your code works
- No need to wait for “testing team”
- Tests are fine-grained – can be run independently
- Tests can be aggregated easily
- Which tests fail give you a hint of where a bug was introduced
- Form a fairly-complete regression test
What is Test-Driven Development?

• A new way to build software

• A strict development method:
  ▶ Add a test.
  ▶ Run the test.
  ▶ Make a small change.
  ▶ Run the tests again. (If they fail, go back to 3)
  ▶ Refactor (while testing)
Where did this come from?

- Test-First Development (+refactoring)
- A practice of Extreme Programming
  - Accept and love change
  - Release early, release often
- There are many supposed advantages, but we’ll discuss those after we try it
So let’s do it…

• We’ll build a little application that represents a network of friends.

• We’ll build incrementally

• Build acceptance/unit tests out of “user stories”

• I’ll be both the customer and lead developer
  ▶ The customer is on site, so you can ask him questions, but he won’t interject

• You’ll be developers, too
So why Test-first?

- You always know what to do next: write a test or make a test pass
- You test code while you are writing it, instead of after you have forgotten about it
- Your tests are always up to date – no backlogs of testing to-do
- You take the customer’s point of view – what do I really want the code to do
- The code you have is exactly what is requested – no more, no less
Patterns for Testing

Simple Smalltalk Testing: With Patterns

Kent Beck,
First Class Software, Inc.
KentBeck@compuserve.com

http://www.xprogramming.com/testfram.htm