Testing.................................................................................................................................................2
Johnson's Law ........................................................................................................................................2
Why Unit Testing .................................................................3
Testing First .................................................................................................4
SUnit ..............................................................................................................6
How to Use the Debugger .....................................................................................10
Some Debugger Magic ..........................................................................................11
Changing Values of Variables ..............................................................................11
Changing a Complex Object ................................................................................12
Changing Code in the Debugger ...........................................................................13
Debugging Code with self halt .............................................................................14
TestCase methods of interest .............................................................................16
XP on Unit Tests ................................................................................................17
Test Everything Example - By Jeffries .................................................................18
Standard Questions & Answer by Jeffries ...........................................................21
  What about GUIs? ...............................................................................................21
  What about real time or multithreading errors? ..............................................22
  What if your tests run really slowly? .............................................................22
  What do you do if you have a body of code, but no tests? .........................23
  How do you test when you have an attached database? .............................23
  How do you know if you have tested everything that could possibly break? 23
  What about errors that in collaborations between classes .........................24
  What if you can't figure out how to test a class? .........................................24

References

Refactoring: Improving the Design of Existing Code, Fowler, Addison-Wesley, 1999 Chapter 4, pp. 89-102


Simple Smalltalk Testing: With Patterns, Kent Beck
http://www.xProgramming.com/testfram.htm

Unit Testing Framework for Various Languages http://www.xprogramming.com/software.htm

Reading

Refactoring, Chapter 4

Extreme Programming Installed, Chapters 13-14
Testing

Johnson's Law

If it is not tested it does not work

Types of tests

• Unit Tests
  Tests individual code segments

• Functional Tests
  Test functionality of an application
Why Unit Testing

If it is not tested it does not work

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 is a nightmare
Testing First

First write the tests

Then write the code to be tested

Writing tests first:

- Removes temptation to skip tests
- Makes you define of the interface & functionality of the code before
Why Automated Tests

Why bother with automated test since one can easily test code in a workspace?

Automated tests

- Allow you to run many tests at once
- Allow others to run the tests
- Allows you to keep the tests for later
SUnit

Testing framework for unit tests in Smalltalk

Local version of SUnit
http://www.eli.sdsu.edu/SmalltalkCode/sunit/

The parts
SUnitSQ271-SUnit.st
SUnitSQ271-SUnitTests.st
SUnitSQ271-SUnitPreload.st
SUnitSQ271-SUnitUI.st
TestRunner27Fixes.1.cs
SunitMenus.1.cs

Ports to other languages can be found at:
http://www.xProgramming.com/software.htm
How to Use SUnit

0. Install SUnit
   File in this order:
   SUnitSQ271-SUnitPreload.st
   SUnitSQ271-SUnit.st
   SUnitSQ271-SUnitTests.st
   SUnitSQ271-SUnitUI.st
   TestRunner27Fixes.1.cs
   SunitMenus.1.cs

1. Make test class a subclass of TestCase

   TestCase subclass: #BankAccountTest
   instanceVariableNames: ""
   classVariableNames: ""
   poolDictionaries: ""
   category: 'Whitney-Examples'

2. Make test methods

   The framework will run methods that start the name 'test'

   testWithdrawl
     | account |
     account := BankAccount name: 'sam'.
     account
       deposit: 100;
       withdrawal: 50.
     self assert: (account balance = 50).
     account withdrawal: 50.
     self assert: (account balance = 0).
How to Use SUnit

3. Start TestRunner

   TestRunner new openAsMorph

Or

   TestRunner new open

Or click on the SUnit item in the open menu

4. Select test class and click on "Run"
   If all your tests run correctly you will get green for go. If one or more of your tests fail then you will get a list of tests that fail and some red
How to Use SUnit

Tests can not pass for two types of reasons:

- **Failure**
  - A test fails

- **Errors**
  - An exception occurred while running your test

Due to some Squeak problem to investigate a failure:

- Click on the test listed in the failure pane
- Click proceed on the first debug window
- Click debug on the second window

To investigate an error:

- Click on the test listed in the failure pane

You will forget the difference between dealing with errors and failures. When you do you will have problems. Just remember the Main Maxim\(^1\) and come back here.

---

\(^1\) The Main Maxim: What you don't know may not hurt you, but what you don't remember always does, Gerald Weinberg, The Secrets of Consulting, 1985, pp. 92
How to Use the Debugger

When you click on the failed test you will get a window like the one below. If the test failed then click on "Proceed" and you will get similar window. Then click on debug. If the test failed due to an exception, just click on Debug.

You will get the Debugger as shown below.

The top pane shows the method calls on the stack. If you click on one of them you will see the source code of the method in the text pane in the middle. The lower-left pane lists the instance variables in the receiver of the method. The third pane from the left on the bottom shows the local variables in the method. When you select any of the variables their value is displayed the pane to the right.
Some Debugger Magic
Changing Values of Variables

You can change the value of variables in the debugger and then continue execution of
the program. To change the value of a variable, first select the variable and then
select its current value as seen below.

Type in the new value. Below I use the value 5. To set account to an
OrderedCollection use the expression "OrderedCollection new" or "account :=
OrderedCollection new"

Accept the change. This can be done with a key command (alt-s) or the pane menu.
Using the pane menu is shown below.
Changing a Complex Object

You can also change the value of an instance variable of a variable. To do this select the variable in the lower pane of debugger. Then get the menu of the pane containing the variable. This is shown below.

In the menu select the "inspect" item. Note that the key command alt-i selects this without using the menus.

You get an inspector window on the object. One can now play the same games to change values of the instance variables of this object.
Some Debugger Magic
Changing Code in the Debugger

Perhaps the most useful thing about the debugger is that one can modify code and then continue running the program. One can change code in the System Browser while the debugger is open, then resume the program. One can also just change the code in the text pane in the debugger. Just edit the code and accept the changes. To resume the program, get the menu for the top pane. Then select the "proceed" item. As you can see there some other useful operations, most of which have key command equivalents.

By the way one can also save and quit with the debugger open. When you restart the image you can continue your debugging session. Can the debuggers in the IDEs for your other languages do all this? These features have been standard in Smalltalk since sometime in the 1970s. Perhaps it really is time to go back to the future.
Debugging Code with self halt

Assume we have the following Counter class with errors. When one uses the class, it does not work properly.

Object subclass: #Counter
  instanceVariableNames: 'count'
  classVariableNames: ''
  poolDictionaries: ''
  category: 'Whitney-Examples'

count
  ^count

decrease
  count ifNil: [count = 0].
  count := count - 1

increase
  count ifNil: [count = 0].
  count := count + 1

To debug the class, insert the line "self halt." in one of the methods that does not seem to work. For example:

decrease
  self halt.
  count ifNil: [count = 0].
  count := count - 1
Now the following will open a debugger:

Counter new decrease

In the debugger you can step through the code to see what is going wrong. When you find out just change the code in the debugger. There is no need to waste time typing code to send values to the Transcript. Just use the debugger.
**TestCase methods of interest**

Methods to assert conditions:

- `assert: aBooleanExpression`
- `deny: aBooleanExpression`
- `should: [aBooleanExpression]`
- `should: [aBooleanExpression] raise: AnExceptionClass`
- `shouldnt: [aBooleanExpression]`
- `shouldnt: [aBooleanExpression] raise: AnExceptionClass`
- `signalFailure: aString`

`setUp`

Called before running each test method in the class.

`tearDown`

Called after running each test method in the class.
XP on Unit Tests

Programmers
• Write unit tests
• Write the test first, then write the code that the tests test

Test everything that could break

What can't break?

   Accessor methods

   Simple methods

       printAccount
       accounts do: [:each | each print]
Test Everything Example - By Jeffries

Task - Build a class **Account** that

- Holds collections of transactions
- Can return the balance of the transactions
Assume we have the class Transaction that is already tested

Object subclass: #Transaction
  instanceVariableNames: 'amount '
  classVariableNames: "
  poolDictionaries: "
  category: 'Whitney-Examples'

Transaction methodsFor:

  setAmount: anAmount
    amount := anAmount

  value
    ^amount! !

Transaction class methodsFor:

  deposit: anAmount
    ^super new setAmount: anAmount

  withdrawl: anAmount
    ^super new setAmount: anAmount negated
The Tests

Write the tests first!

TestCase subclass: #AccountTest
    instanceVariableNames: ""
    classVariableNames: ""
    poolDictionaries: ""
    category: 'Whitney-Examples'

testEmpty
    self assert: Account new balance = 0

Stop here and now go write the code to make the above test to work. We will wait until you get back. The process works much better if you write one test, then make the test work. Writing lots of tests before writing any real code make the process seem painful. Most people do not like to do painful things, so put them off to latter. If you put off writing tests, then there will be too many tests to write when you get to it. You will then make sure that you never get to writing the tests. OK now you can write the second test. Then make it work.

testThree
    | account |
    account := Account new.
    account
        add: (Transaction deposit: 100);
        add: (Transaction withdrawl: 20);
        add: (Transaction withdrawl: 15).
    self assert: account balance = 65
Standard Questions & Answer by Jeffries

- What about GUIs?
- What about real time or multithreading errors
- Is it OK to test a class by just testing the classes that use the class?
- What if your tests run really slowly
- What do you do if you have a body of code, but no tests
- How do you test when you have an attached database
- How do you know if you have tested everything that could possibly break?
- What about errors that only show up in collaborations between classes
- What if you can't figure out how to test a class
- My stuff can not be tested because …

What about GUIs?

Do no processing in your GUI code
GUI code handles just GUI events
Model code contains all the processing
Test the model code

Use non-automated tests on GUI code (rew)

Web applications are very GUI intensive
Try HttpUnit
What about real time or multithreading errors?  

Keep the designs very simple in these situations

Use many eyeballs

Reflective tests can be used to determine if you are doing what you think you are (always using a semaphore…)

Testing a class by testing the classes that use the class

Ron Jeffries:

Used to say there was no need to test class A

Now says things go better if every class has its own tests

Easier to refactor the class

Makes it easier to locate bugs

What if your tests run really slowly?

You need to keep the tests fast

Profile them to find out the problem and correct it
What do you do if you have a body of code, but no tests?

Quit?

Write tests on the part that you need to work on

The alternative to writing the tests is shipping buggy software

How do you test when you have an attached database?

If the problem is performance, fake most of the calls to the database

How do you know if you have tested everything that could possibly break?

Knowing is a matter of conscience and experience

If you start getting a lot of errors in functional testing or in the field, you need more and/or better unit tests
What about errors that in collaborations between classes

If you have a lot of these you probably have a process problem

Find it and fix it

What if you can't figure out how to test a class?

What is hard about it?

Get the team to talk about it

My stuff can not be tested because...

Jeffries does not believe you
Exercises

1. Add the following class to your browser. Call the decrease method and fix the code in the debugger.

```smalltalk
Object subclass: #Counter
    instanceVariableNames: 'count'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Your category here'

count
    ^count

decrease
    self halt.
    count ifNil: [count = 0].
    count := count - 1

increase
    count ifNil: [count = 0].
    count := count + 1
```

2. Install SUnit in your image. The computers in the campus computer labs have SUnit already installed.

3. Write SUnit tests for your BankAccount class from assignment 1