MetaProg*
Quality in Software

# Advanced eXtreme Programming Testing Techniques in Smalltalk

Joseph Pelrine
C*O, MetaProg GmbH
jpelrine@metaprog.com



MetaProg*
Quality in Software

## SUnit is not enough

- **Quality control doesn't stop at development, but should include the whole delivery and deployment process.**
  - It doesn't help to have a running application if you can't package and deliver it to your customers reliably.
- **For this reason, companies who have built their reputation on delivering quality software to customers on time tend to develop strategies for testing the deliverability of their code.**
- **SUnit alone isn't sufficient, so we need additional tools.**

MetaProg*
Quality in Software

## Outline

- SUnit
- Test Resources
- Extensible Test Cases
- Model-vs. view-level testing
- Tool Support
- Hands-on




MetaProg*
Quality in Software

## SUnit

- Originally invented by Kent Beck
  - "Simple Smalltalk Testing", *Smalltalk Report*, October 1994
- Rewritten XP-style using itself in 1998
- Taken over by Camp Smalltalk (Sames Shuster) in 2000

MetaProg*
Quality in Software

## SUnit Situation

- **Was incompatible across dialects**
- **Camp Smalltalk version on SourceForge**
  - http://ansi-st-tests.sourceforge.net/SUnit.html
  - current version is 3.0
- **ANSI standard core**
  - dialect-specific preLoad code
  - dialect-independent tests
  - dialect-specific UI




MetaProg*
Quality in Software

## Test Resources

- **Instantiating test objects can be expensive**
  - Database connections
  - Extremely complex objects
- **We don't want to have to do this for each TestCase**
- **Keeping a test object alive over multiple TestCases**
  - breaks one of the primary rules of unit testing
  - is nevertheless desirable
- **This is why we've developed Test Resources**

MetaProg*
Quality in Software

## Test Resource Object Types

- **What should we use TestResources for?**
- *One final bit of philosophy. It is tempting to set up a bunch of test data, then run a bunch of tests, then clean up. In my experience, this always causes more problems than it is worth. Tests end up interacting with one another, and a failure in one test can prevent subsequent tests from running. The testing framework makes it easy to set up a common set of test data, but the data will be created and thrown away for each test. The potential performance problems with this approach shouldn't be a big deal because suites of tests can run unobserved.* [Beck95]
- **Resources should be solely used for – well, resources (DB connections and the like), helper objects which are not tangented by the tests, or for *read-only* test data. If people start raping resources as a way of getting around #setUp, we haven't gained anything, and may have lost a lot.**




MetaProg*
Quality in Software

## The TestResource class

- **Implemented as an optional singleton**
  - Follows standard singleton #current protocol
  - #new is not overridden to return an error
- **Polymorphic syntax with TestCase**
  - #setUp
  - #tearDown

MetaProg*
Quality in Software

## Initializing Resources

- **All required resources should initialized before a TestSuite runs**
  - This occurs non-deterministically
  - TestResource classes are sent the message #isAvailable
- **TestCases optionally/preferably define required resources**
  - TestCase class>>#resources
  - By not defining a resource in this method, its initialization becomes responsibility of the TestCase itself
- **Future directions**
  - Initialization order
  - Conditional initialization (dependent upon a pre-run TestCase)




MetaProg*
Quality in Software

## Running Tests with Resources

```
TestSuite>>#run
    | result |
    result := TestResult new.
    (self areAllResourcesAvailable]) ifFalse: [
          ^TestResult signalErrorWith: 'Resource could not be initialized'].
    [self run: result] ensure: [
          self resources do: [:each | each reset]].
    ^result

TestSuite>>#areAllResourcesAvailable
    ^self resources
        inject: true
        into: [:total :each | each isAvailable & total]

TestSuite>>defaultResources
        ^self tests
                inject: Set new
                into: [:coll :testCase |
                        coll
                                addAll: testCase resources;
                                yourself]
```

MetaProg*
Quality in Software

## Releasing a Test Resource

- When do you release a Test Resource?
- We let you decide
  - TestRunner sends #reset to the resource class when it is finished
  - This invokes #tearDown on the resource and nils it out




MetaProg*
Quality in Software

## Defining Test Resources

- Define your resource as a subclass of TestResource
- Override #setUp in your TestResource subclass to initialize your resource
- Override #resources on the class side of your TestCase subclass to return a Collection of all resource class names which you would like the system to maintain for you (n.b. you don't have to do this if you want to turn the resources on and off yourself).
- Let 'er rip...

MetaProg*
Quality in Software

## Test Resource Variations

- In my StableSqueak work, for example, I have a resource instance holding a full semantic model of the system environment. It takes > 1 hour to instantiate. Since it's just used as a reference object, I don't have to worry about it being changed. Because of this, I overrode #reset to do nothing, so that the instance was never removed.
- One of my colleagues has overriden #reset in a so-called TimedReleaseTestResource to set off a timer which releases the resource (a GemStone connection) after 5 minutes. He says that normally he can correct the problem and get back up and running in < 5 min. If not, he'll probably need a lot more time, so this solution makes sense to him.




MetaProg*
Quality in Software

## Extensible Test Cases

- Unit test support for external testing tools like ENVY/QA or SmallLint
- The implementation of new Test Cases is based on specialized subclasses of `MedExtensibleTestCase`
- To perform an unit test, you can use the standard `TestRunner`

MetaProg*
Quality in Software

## XTC Implementation in Smalltalk

- XTC's have different behavior at run time and debug time
- They typically are external tools which have their own browsers
- This forces us to separate the running of the test and the evaluation of the results




MetaProg*
Quality in Software

```
MedExtensibleTestCase>>#runCase
    self performTestSetUp.
    [self
        runTestCode;
        performTestAction] sunitEnsure: [self tearDown]

MedExtensibleTestCase>>#runCaseAsFailure
    self performDebugSetUp.
    [[self
        runTestCode;
        performDebugAction] sunitEnsure: [self tearDown]] fork


MedExtensibleTestCase>>#runTestCode
    self perform: testSelector sunitAsSymbol.
    self runTestSilently
```

MetaProg*
Quality in Software

## XTC Methods

- #performTestSetUp – can be used to install a special error handler
- #performDebugSetUp – leaving this empty would use the default error handler
- #test** - sets up the objects needed for the test
- #runTestSilently – runs the actual test
- #performTestAction – evaluates the test result in the test context
- #performDebugAction – evaluates the test result in the debug context




MetaProg*
Quality in Software

## SmallLintTestCase

```
MedExtensibleTestCase>>#performTestAction
          ^self assert: self expectedResult
   MedExtensibleTestCase>>#performDebugAction
       ^rule openEditor


   MedExtensibleTestCase>>#runTestSilently
       ^SmalllintChecker
           runRule: self rule
           onEnvironment: self environment


   MedSmallLintExtensionsExampleTestCase>>#testLintChecksBugs
       rule := CompositeLintRule  ruleFor: BasicLintRule protocol: 'bugs'.
       classes := self defaultClasses
```

MetaProg*
Quality in Software

## Build Your Own XTC

- **Consider which tool you would like to interface to SUnit**
- **Define the XTC subclass**
- **Override the methods necessary to**
  - set up the data
  - call the test engine
  - evaluate the results




MetaProg*
Quality in Software

## GUI Testing

- **Skeleton interfaces**
  - XP has profitted from the proliferation of the Internet, as the Net has gotten users accustomed to GUIs which are suboptimal, not user-friendly, and which change on an unpredictable basis. Users have become less fussy about how the GUI works.
- **Any non-trivial project will tend have a significant amount of code stuffed away in the user interface. This code needs to be tested as stringently as model-level code does.**
  - The quantity of code missed by not doing GUI – level testing can be amazing. In a series of impromptu coverage tests run on two production systems implemented in XP, it was found that barely the half of the total code was covered by the SUnit tests.

MetaProg*
Quality in Software

## View level testing

- **Window geometry and behavior**
  - This area relates to all aspects of the behavior of the window as a whole, and in relation to the underlaying operating system. Does the window resize properly? If a window is minimized and restored, is the appearance and behavior the same?
- **Inter-widget synchronisation**
  - This area relates to all aspects of the behavior of the window internally. When an input field has content, is the OK button enabled?
- **Model-view communication**
  - This area relates to all aspects of the flow of information from the window to the data objects being manipulated in the window. Does information get from the view to the model and back?




MetaProg*
Quality in Software

## Validation

- **We also need to test validation of the data input by the user.**
  - .Syntactic validation – ‚30/foo/1999' can not be a Date
  - .Semantic validation – Feb. 30, 1999 is not a valid Date
  - Contextual validation – Someone born on ‚30/2/1999' can not be a parent, or a parent can not be younger than their child.

MetaProg*
Quality in Software

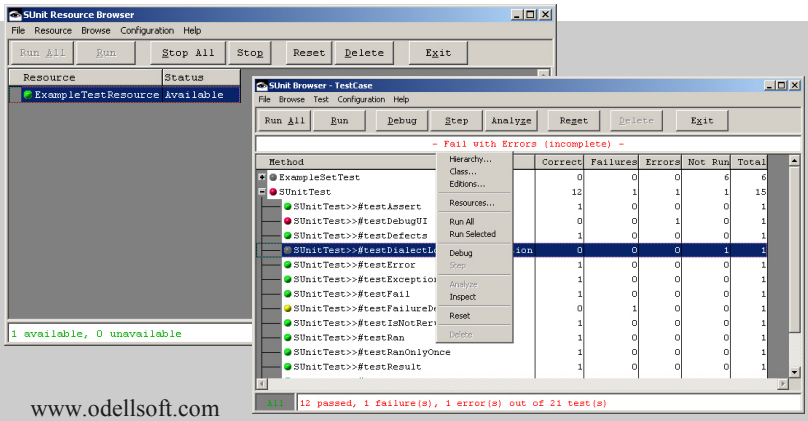
## Model/View Testing Rules of Thumb

- If there is significant latency between appearance of view after model, test the model. Note: Still need to test model-view connection if there is risk in it.
- If other components in addition to a single view depend on a model, test the model
- If a high degree of control is required, test from the model
- If model and view are highly interdependent, test from view (can be considered same subsystem)
- For deep object verification, implement object verification at model level. Note: This may be used to support view testing.




MetaProg*
Quality in Software

## The SUnitBrowser




www.odellsoft.com

MetaProg*
Quality in Software

## Smalltalk Tool Support

- SmallLint (from the Refactoring Browser)
- Test Mentor (Silvermark)
- VA Assist Pro (Smalltalk Systems)
- *Mastering ENVY/Developer* tools

Maybe we need an InstallShieldUnit???




MetaProg*
Quality in Software

## Hands-on Exercises

- Defining new kinds of TestResources
- Defining new kinds of Extensible Test Cases
- SmallLint skin
- Warning 49 skin
- ...

MetaProg*
Quality in Software

## For more Info:

- http://ansi-st-tests.sourceforge.net/SUnit.html
- http://wiki.cs.uiuc.edu/CampSmalltalk
- http://www.xProgramming.com
- http://www.metaprog.com




MetaProg*
Quality in Software

## Thanks to:

- Alan Knight & Adrian Cho
- Sames Shuster & the Camp Smalltalk SUnit group
- S. Sridhar & Jeff Odell
- Kent Beck
- Eric Clayberg
- Don&John for the Refactoring Browser
- Mark Foulkrod & Mike Silverstein from Silvermark