Advanced eXtreme Programming Testing Techniques in Smalltalk

Joseph Pelrine [|]
Daedalos Consulting
jpelrine@acm.org
http://www.daedalos.com/~j_pelrine

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.

Outline

- Model-vs. view-level testing
- SUnit
- Test Resources
- Skins
- Performance testing
- Packaging & delivery testing
- Hands-on

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.

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?

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.

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.

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 Schuster) in 2000

SUnit Situation

- Incompatible across dialects
- Camp Smalltalk version on SourceForge
  - http://ansi-st-tests.sourceforge.net/SUnit.html
  - current version is 2.7
- ANSI standard core
  - dialect-specific preLoad code
  - dialect-independent tests
  - dialect-specific UI

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 TestResources

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

Initializing Resources

- All required resources are 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)

Running tests with resources

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


Releasing a TestResource

- When do you release a TestResource?
- 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
  - You can easily implement a
    - TimedReleaseTestResource
    - ManualReleaseTestResource

Performance Testing

- If performance is an issue, write a story card for it, implement a TestCase, and do it like any other task

```
self assert: ((Time millisecondsToRun: [foo]) < 1000)
```


Performance Testing Tools

- Profilers exist for most Smalltalk dialects
  - ENVY/Stats and Benchmark Workshop for VA
  - Advanced Tools Profilers for VW
  - etc.
- Call profiling tools from SUnit?

Build Testing

- An important aspect of quality control in development projects is to have a stable, reproducible build process from the vendor's base Smalltalk image through the development environment to the final stripped or packaged run-time image.
  - There are numerous projects where the developers are unable to reconstruct their development environment from a virgin Smalltalk image. This is a great (and sometime insurmountable) handicap when upgrading the base Smalltalk dialect or third-party tools.

Package early, package often

Prerequisite Testing

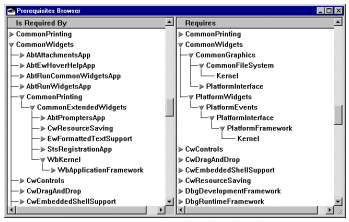
- Prerequisites define dependencies and loading order between classes in VisualAge.
- There are system-level tests to analyze these dependencies.
- SUnit can be extended to include these tests.
  - But what about method-level dependencies?

Sorting Out Prerequisites

- Don't ignore Warning 49
  - The method which has just been compiled references a class which it shouldn't. This is because the class referenced is not "visible" in the prerequisite chain of the application containing the method.
- This can be corrected in two ways:
  - You can move the method to another application which has the class in its prerequisite chain.
  - You can move the class to make it visible, by including the class's defining application into the prerequisite chain for the method's application
- First you have to find out where something's wrong

Prerequisites Browser for VA

- Thanks to Glenn Jones



http://www.netkonect.co.uk/~paget/

Finding Warning 49

- CompiledMethod>>#testVisibilityIn:

```
SubApplication class publicMethods

badPrereqs

    Transcript cr; show: 'Looking for bad prereqs…'.
    self withAllSubApplications do: [ :app |
        Transcript cr; show: 'Checking ', app name.
        app classes do: [ :class |
            (class methodsIn: app) do: [ :method |
                method testVisibilityIn: app]]].
    Transcript cr; show: 'Done'.
```


Nil Globals

- Iterate through the literal frame

```
CompiledMethod publicMethods

testForNilGlobals

    self allLiteralsDo: [:literal |
        literal isAssociation ifTrue: [
            | key |
            key := literal key.
            ((self isClassOrPoolVar: key) not and: [
            self isNilOrUnmanaged: key]) ifTrue: [
                | qualifiedName |
                qualifiedName := ' the nil or unmanaged class or global ', key.
                EmImageSupport errorReporter
                    logError: 49
                    withParms: (Array
                        with: self printString
                        with: qualifiedName)]]]
```


…nil Globals...

```
CompiledMethod>>isClassOrPoolVar: key

    ^(self methodClass
        variableAssociationAt: key
        using: Smalltalk
        ifAbsent: [nil]) notNil

isNilOrUnmanaged: key

    "VisualAge version"
    ^(System image globalNamespace
        at: key
        ifAbsent: [nil]) isNil or: [
            System image globalNamespace unmanagedNamespace includesKey: key]

isNilOrUnmanaged: key

    "VisualWorks version. This may be changed in VW 5 "
    ^(Smalltalk at: key ifAbsent: [nil]) isNil
        or: [Undeclared includesKey: key]
```


…nil Globals

```
SubApplication class publicMethods

badPrereqs

    EmImageSupport errorReporter logDevice
        cr; show: ('Scanning %1...' bindWith: self name).
    self withAllSubApplications do: [ :app |
        Transcript cr; show: 'Checking ', app name.
        app classes do: [ :class |
            (class methodsIn: app) do: [ :method |
                method
                    testVisibilityIn: app;
                    testForNilGlobals]]].
    EmImageSupport errorReporter logDevice
        cr; show: 'Done'.
```


Error Logger

```
Object subclass: #MedSilentErrorLogger
    classInstanceVariableNames: 'current '
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''

nextPutAll: aString
    (aString indexOfSubCollection: 'Warning: 49' startingAt: 1) = 1 ifTrue: [
        self restoreDefaultLogger.
        TestResult exFailure signalWith: aString]

restoreDefaultLogger
    EmImageSupport errorReporter logDevice: System errorLog
```


Prereq Checking Test Case

```
TestCase subclass: #PrereqCheckTestCase
    instanceVariableNames: ''
    classVariableNames: ''
    poolDictionaries: ''

testBadPrereqs
        self should: [MedErrorReporterTestApp badPrereqs]
```


Refactoring Prerequisites

- Although it is not needed by a specific application, a prerequisite can not be removed if it is required by a dependent of the application

How to do it

- Version and release any open class editions
  - You‘re going to reload the current (released) lineup
- Browse Application Editions
- Remove the prereq
- Add the prereq
- From the AppMan, choose both, then reload current

Quality Testing

- SmallLint includes numerous tests for code quality.
  - Methods sent but not implemented
  - Methods sent but not implemented in application
- SUnit can also be extended to include these tests.

Packaging Testing

- Packaging is a pain in the &*/*%
- Suit can be extended to run the VA Packager.
- This increases the feedback loop time unacceptably :-(

Packaging/Delivery Testing Rules of Thumb

- Incorporate prerequisite testing into the SUnit test suite.
- Correct prerequisite problems immediately.
- A clean load into a virgin image, and a packaging run with no errors, are required prior to integration.
- If possible, run view-level tests on the packaged image before integration.

Tool Support

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

Maybe we need an InstallShieldUnit???

Hands-on Exercises

- TestRunner support for TestResources
- SmallLint skin
- Warning 49 skin
- ...

For more Info:

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

Thanks to:

- Alan Knight & Adrian Cho
- Sames Schuster & the Camp Smalltalk SUnit group
- Sridhar & Jeff Odell
- Kent Beck
- Eric Clayberg
- Mark & Mike from Silvermark
- all the guys at Daedalos