XMP  eXtremeMetaProgrammers

Solving the XP Legacy Problem with (Extreme) Meta-Programming

Author(s): Niall Ross and Andrew McQuiggin
Date: 20 April 2002
Reference: XMP/meta-test/pres/0005/1.0

Table of Contents

Publication history

September 2002

Issue 1.1. Minor edit of Notes chapter to correct errors and improve phrasing

April 20th 2002

Issue 1.0. Version presented at Smalltalk Solutions on April 23rd 2002.

April 2002

Issue 0.1. Version provided for inclusion in conference CD.

About this document

Key to XP is that test-driven development also supports refactoring; tests reveal when a refactoring breaks the system. When XP is introduced to a large existing system, writing tests for the legacy is a sysiphesean task which lacks the synergy of writing tests in test-driven development. Without them, however, refactoring is constrained to stay within the area of new XP development or else be unsafe.

We (Niall Ross of eXtremeMetaProgrammers and Andrew McQuiggin of HECM) used meta-programming to help introduce XP into a large financial system. We subclassed the standard SUnit framework and browser to support deep comparison of data captured by tests run in pre- and post-refactor images. By combining these techniques with a feasibly small set of basic application-specific tests, we aim to achieve a test set for the legacy that is sufficient to make refactoring safe. This talk describes our approach, our experience with it to date, and indicates the kind of systems to which these techniques are appropriate.

Intended audience

Smalltalkers who want to introduce XP into legacy systems.

Smalltalkers with an interest in meta-programming.

Conventions used

Meta-data: data describing domain classes and behaviours that, in a conventional system, would be embodied as hard-coded classes and methods; not to be confused with meta-programming

Meta-programming: in this talk, meta-programming means use of the Smalltalk meta-protocol to write methods that walk and manipulate arbitrary object graphs; the term also has other legitimate meanings

XP: eXtreme Programming

VA: VisualAge Smalltalk

VW: VisualWorks Smalltalk

Acknowledgements

Our work benefited from examining utilities produced by Paul Baumann and John Brant.

References

[1] Custom Deep Copies, Paul Baumann, The Smalltalk Report 7 5 March/April 1998 (copy of article plus utility in Smalltalk archive)

[2] The Business Case for Adequate Reflection, Niall Ross, 8th European Smalltalk Summer School, Ghent, 30th August - 3rd September 1999 (navigate from http://www.esug.org/)

[3] XP-rience: eXtreme Programming Experience, Niall Ross, Camp Smalltalk 3 and 10th European Smalltalk Summer School, Essen, 25th August - 1st September 2001 (navigate from http://scgwiki.iam.unibe.ch:8080/SmalltalkWiki/117)

Solving the XP Legacy Problem with (eXtreme) Meta-Programming

Version 1.0 (presented)

Niall Ross, eXtremeMetaProgrammers

Andrew McQuiggin, HECM

nfr@bigwig.net **amcquiggin@yahoo.com**

These are the slides of the talk presented at Smalltalk Solutions 2002. An earlier rough draft version was included on the CD distributed at the conference. This later, much more developed version, with detailed notes in the following chapter, is on the conference website.

Overview

- **Motives for this work and for this talk**
  - **— Meta-Programming: a Smalltalk enabler?**
  - **— XP Legacy: the problem**
- **Our Approach**
- **Implementation**
  - **— Obtaining Tests to Compare**
  - **— Test Browser Framework**
  - **— Deep Comparison Framework**
- **Discussion: Value of this Work**
  - **— SUnit extention ?**
  - **— Standard Meta-Programming frameworks ?**

Meta-Programming

Meta-program *verb* ..., use meta-object protocol to walk and manipulate object graphs, ...

Secondary Subject of this Talk

Pure programs manipulate data. Pure meta-programs manipulate program structure as data. Pure programs are slow to change. Pure meta-programs are slow to deliver.

- **program: code that should be meta-programmed once is replicated as patterns.**
- **quick meta-program: 90% right but 10% wrong (i.e. needing program overrides).**
- **Correct meta-program: too late (must solve specifically many times to build generic).**

Good OO systems grow by refactoring state downwards and behaviour upwards

- **behaviour upward: moving behaviour to meta-program is natural extention**
- **state downward: state includes program behaviour overrides of meta-program**

Smalltalk is exceptionally well-suited to

- **writing mixed program and meta-program systems**
- **incrementally refactoring behaviour between program and meta-program**

Meta-programming patterns and frameworks should be a Smalltalk enabler. This talk refers to one practical use. Who else is working on this? Can we make it happen?

The XP Legacy Problem

Refactor *verb colloq*. to improve (a program)

Refactor *verb* to change implementation without changing (desired) behaviour.

Extreme programming relies on a synergy:

- **Test-driven development: writing tests first speeds coding**
- **Refactoring: pass tests => change was a behaviour-preserving refactor**

Legacy breaks this synergy:

- **No XP test suite => no safe refactoring**
- **Hard to write post-hoc tests for your old code, harder for others' old code**
  - **— tedious labour with no test-driven synergy**
  - **— hard to sell management (and self) on task's value**
  - **— hard to find the assertions, much harder to trust you've found enough**

But much of Smalltalk's survival through the lean years was due to legacy; systems that 'will be rewritten in Java in the next two years' but weren't because they couldn't. Now these programs need refactoring and want to use XP.

Getting a Handle on Legacy

Our system (and other Smalltalk legacy systems ?) has certain features.

- **It *has* been tested**
  - **— standard waterfall-style distinct test phase, test spec documents, etc.**
  - **— non-XP: each refactor invalidates whole effort-heavy test phase**
- **Only a subset of behaviour changes per 3-month release**
  - **— hundreds of products, many options, many permissions**
- **Complex hehaviour passes though small(er) 'narrow' subsystems**
  - **— States: rich specific detail within small generic state model**
  - **— Business Logic Validation: complex errors/warnings raised by small protocol**
  - **— View layer: many forms built from finite set of widgets**
  - **— Persistence layer: complex products realized from compact data in DB**

Our Approach: Fundamentals

Deep Comparison of Tests

Refactor *verb* to change implementation without changing (desired) behaviour.

Behaviour *noun* ... , (in OO) a network of objects existing at the end of an operation, ...

- **The tested behaviour of the latest release is acceptable**
- **All the supported products go through same basic states**

  create, update, validate, save, ...
- **Each state expresses some key behaviour in narrow subsystem(s)**

  — Validation, UI, DB interface
- **same object network within narrow subsystem => same behaviour from user's view**

```
self assert:
   (releaseTest narrowRoot deepCompareTo: refactorTest narrowRoot)
```

(Of course, it's not quite that simple :-)

Obtaining Tests to Compare

- **Write some basic test classes and methods**

```
GenericProductScenarioTC testChosenNarrowSubsystemEtc
```

- **Create test instances for each specific product type from stored data**

```
[1]testSuite addTests:
    persistentProductData collect: [...
      GenericProductScenarioTC keyInstVar: ... productStandingData ...
```

- **Test moves product to scenario's basic state by**
  - **— reusing stored product instance data (synergy with other tests)**
  - **— using 'example instances': generated class/widget-specific example data**

  Assertions check that the scenario's state is reached. (Other assertions added as you know how and have time; deep comparison is main assertion.)
- **Basic test deepCopies appropriate graph from chosen narrow subsystem root, e.g.**
  - **— Business Logic Validation system root**
  - **— Product's top-level view**

1. This code is vastly simplified to give an overview; later code examples are also somewhat simplified

TestBrowser Framework

Subclasses the SUnit and SUnitBrowser frameworks:

- **ComparisonTestResult has a ComparisonTestCase**
  - **— both have instvars: ‘earlierResult laterResult’ holding superclass’ instances**
- **Test Browser is packagable (coy. process required be able to test in packaged images)**
- **Test Browser holds multiple TestResults; one is current TestResult**
  - **— create test result, run selected tests in start (e.g. released) state**
    - **– tests in Browser’s suite copied (with key values), run, added to current Result**
    - **– rerunning overwrites previous copies in current result**
  - **— get new result, effect refactor, rerun (all or some) previously-run tests**
    - **– new copies run and added to new current result**
  - **— create comparison result for these two results**
    - **– invoking ‘run’ on a test now runs a comparison test on its two run copies**
  - **— get more results, do more refactors**
    - **– re-run tests, compare with released, with last, ...**

Test Results in the Test Browser

'Run' when TestResult being viewed: run selected test, show outcome (i.e. as usual)

- **Pass: test completed and satisfied all its assertions**
- **Fail: test completed but did not satisfy all its assertions**
- **Error: test did not complete**

'Run' when ComparisonTestResult being viewed: run comparison test on tests keyed by selected test in earlierResult and laterResult

- **Pass: both tests had same outcome and satisfied the comparison test assertions (i.e. deep comparison of object graphs captured by the two tests)**
- **Fail: both tests had the same outcome but did not satisfy comparison test assertions**
- **Error: tests did not even have same outcome (comparison test assertions not executed in this case as they could be met only by misleading accident)**

Comparisons can be inter-image

— Browser supports dumping and loading of TestResults + contents

or intra-image

— deep copying object graphs on capture avoids trivial comparisons

Base Deep Comparison Framework

(We found few examples[1], none for deep comparison; are we not looking hard enough?)

Key issues in building meta-programming comparison framework

- **collection comparison: backtracking for unordered and sorted comparison**
- **very flexible comparison customisation**

Basic Framework: Strategy class and Object methods

```
Object>>compareStructureTo: anObject
   ^self class == anObject class

Object>>compareContentsTo: anObject using: comparisonStrategy
   ^((comparisonStrategy comparableInstVarIndices: self) allSatisfy:
        [:instVarIndex | comparisonStrategy
             compare: (self instVarAt: instVarIndex)
             to: (anObject instVarAt: instVarIndex ifAbsent: [nil])])...

ComparisonStrategy>>compare: first to: second
   ... self hasAlreadyCompared: first to: second)or:...
     compareStructureBlock value: first value: second)
     and: [visited at: first put: second.
       compareContentsBlock value: first value: second]
```

1. Paul Baumann's CustomDeepCopy in ST Archive, ReferenceFinder in RB

Customisable Deep Comparison Framework

Two routes to customisation

- **(per compared class) Override comparison methods; done in layers**
  - **— Vendor (VA, VW so far): vendor-specific classes, singletons, etc**
  - **— General: default comparison choices for collections, immediates, truncation, etc**
  - **— System: default test comparison choices for system under test**
- **(per comparison) Customise strategy before (re-)using**
  - **— Choose methods via blocks: structure, content and nil-equivalent blocks**
  - **— Class-oriented, via strategy's filtering of object's instVars**

```
aComparisonStrategy
   ignore: BusinessModel atAll: #(#identifier #eventDependents);
   ignore: ProposalSummaryModel at: #savedToDatabase;
   ignore: ValidationCache at: #validationBlock;
   compareRoot: first to: second
```

  - **— Instance-oriented, via pre-populating strategy's visited dictionary**

```
aCopyStrategy ignore: view holder.
aCopyReplaceStrategy at: view model client use: self testClient.
aComparisonStrategy nextDifference.
```

Tools (that helped us *meta*-program[1])

Dynamic type recovery tools speed customising comparison strategy and methods

- **We used SmallTyper (VA)**
- **The Analysis Browser works similarly (VW)**
- **Do other dialects have such tools ?**

Serialization for inter-image test result comparison

- **We used standard VAST dumper / loader**
  - **— needed same customisations as our strategies but used different protocol**
  - **— implemented with primitives so hard to debug (that's why I wrote ConstrainedRF)**

  Is there an easy way to debug? How do other dialects' dumpers compare?

Tools wanted

- **Make Refactoring Browser meta-program-protocol aware? (We may explore this.)**
  - **— safer: e.g. warn or rewrite strategy ovverides affected by 'rename instVar'**
  - **— easier: refactor per strategy override to per class override (a common action)**

1. We also used the RB (massively), MethodWrappers, Greg Hutchinson's code quality tools, etc., etc.

Demo I

Launch browser, demo features[1].

- **load test result, rename it and rerun three selected tests**
- **do trivial refactor and create new test result, rerun selected tests**
- **create comparison result for two results, rerun to show amber and red results**
  - **— Show the difference found by amber comparison using an inspector**
    - **– (if time allows, let ConstrainedReferenceFinder detect amber-comparing tests' non-comparing values from tests' roots, then show paths to it are identical)**
- **fix refactor, rerun amber-comparing test in refactor result, then comparison result**
  - **— formerly-failing comparison test now passes**

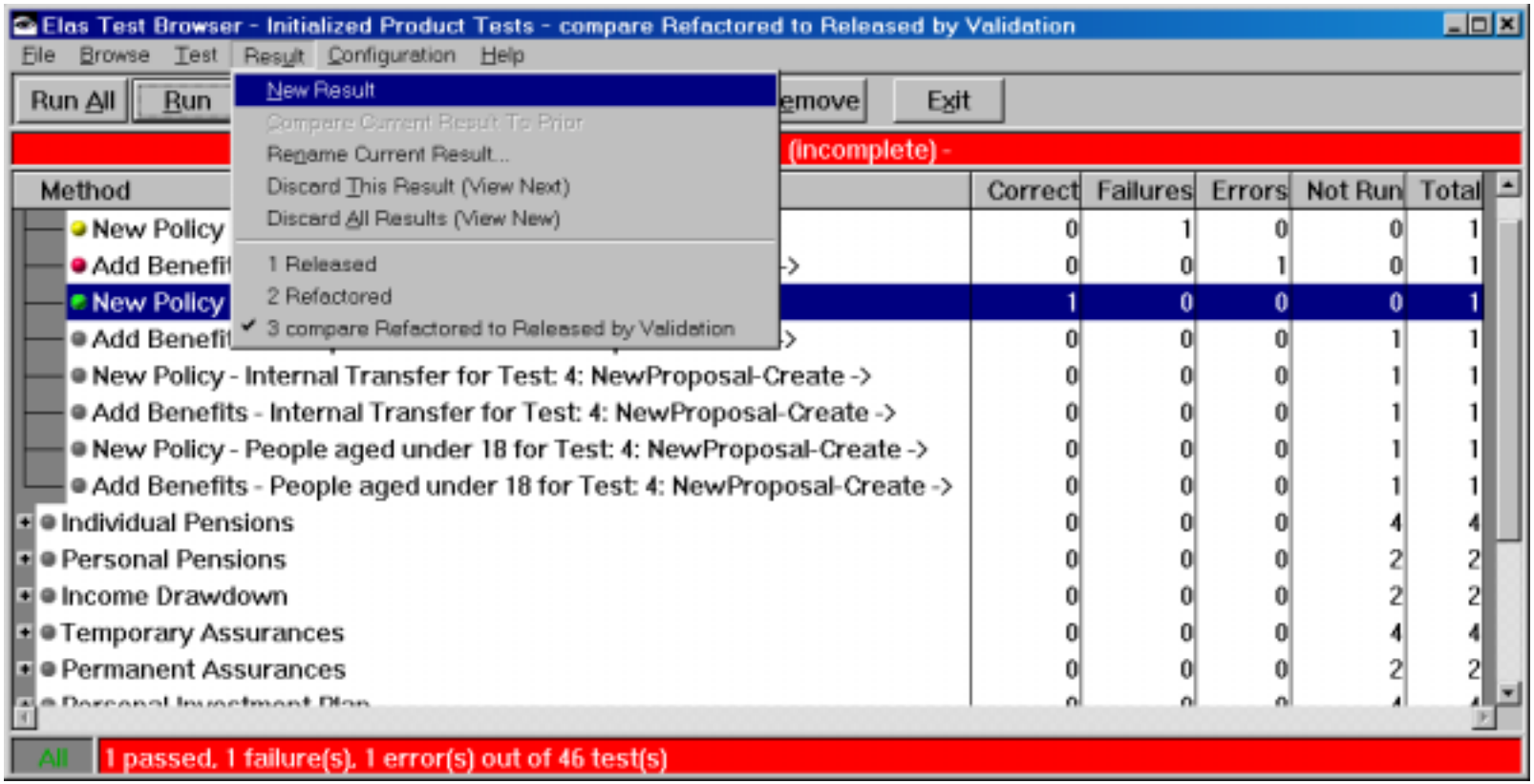
Mention other uses of comparison test framework:

- **user permissions: verify more permissions give more, not different, behaviour**
- **packaging: run tests in development image, export result to package, rerun, compare**
- **external changes: compare results of tests capturing interface subgraphs before/after**

1. No database access from Cincinnati so only new product creation scenarios testable

Demo II

Viewing the Comparison Result at the end of the demo.



Discussion: Value of this Work

For eXtreme Programming ?

- **Test Browser could be made public domain**
  - **— e.g. as SUnit(Browser) add-on at Camp Smalltalk**
- **Key instVars for TestCases**
  - **— arose naturally from system's use of data to define products**
  - **— comparison tests catch key mismatches**
  - **— Is this safe? Is this in accord with SUnit philosophy?**

For Meta-Programming ?

- **Meta-Programming framework(s) can be more powerful and more standard**
  - **— e.g., I subclassed RB's ReferenceFinder to ConstrainedReferenceFinder, using same protocol and implementation as my strategy for the extra features; easy !!!**

Would a common protocol or common framework be used?

'Smalltalk Best Practice Meta-Patterns': Can I buy it? Must I write it (any co-authors)?

Presentation 1 Slide Notes: Solving the XP Legacy Problem with (Extreme) Meta-Programming

The text in sections 1.1 through 1.4 constitutes notes to the slides. Each subsection heading is the title of the corresponding slide.

1.1 Our Approach

Key to XP is that test-driven development also supports refactoring; tests reveal when a refactoring breaks the system. When XP is introduced to a large existing system, writing tests for the legacy is a sysiphesean task which lacks the synergy of writing tests in test-driven development. Without them, however, refactoring is constrained to stay within the area of new XP development or else be unsafe.

1.1.1 Title slide

For all I know, there is a solution to the XP legacy problem already in the public domain. But one thing I've learned from the web is that they can write the stuff faster than I can read it. Today I will present the solution that Andrew and I have pursued in hopes to learn more.

1.1.2 Overview

This talk begins by explaining our motives for this work and for this talk, which are two-fold: an interest in meta-programming and a need to use XP on legacy. Next I describe our approach in outline and then in detail. After a demo, we shall discuss the possible value of this work to XP and/or meta-programming.

1.1.3 Meta-Programming

I have experience of several kinds of meta-programming. The meaning we needed for the work I'm about to describe is simply, 'using Smalltalk's meta-object protocol to walk and manipulate arbitrary object networks built from classes unknown in advance.'

I have a secondary aim in presenting this fairly trivial example of meta-programming. I've often thought that meta-programming frameworks should be more commonly used in production code, and more publicly known, than I've found them. (Of course, it might be that they are and I'm just not looking in the right places.)

Pure meta-programming solutions to commercial problems are hard. A completely correct meta-algorithm can only be developed slowly from several well-understood examples, the exact opposite of the usual commercial scenario where a window of opportunity for a new imperfectly-understood problem must be hit acceptably. Worse, if a complex meta-algorithm supports the current portfolio, one dare not hack a change into it on a short timescale. Quickly-refactored meta-algorithms are typically 90% right and a very-hard-to-understand-and-fix 10% wrong. But this 10% is usually easy to handle acceptably at the programming level, thus a seamless mixed program-and-meta-program framework is commercially viable. As importantly, such a framework allows for the piecemeal refactoring of behaviour to a meta-behaviour on longer timescales.

This is the application to meta-programming of a general truth about OO systems. Ralph Johnson has pointed out that a living OO system evolves to refactor state downwards (as the system becomes able to handle more situations) while refactoring behaviour upwards (as specific behaviours are recognised as examples of more general patterns). Refactoring behaviour into meta-behaviour is a natural extension of this rule.

Why am I delaying the start of the main talk with these reflections? Well, because I think Smalltalk is particularly well suited to building frameworks in which refactoring between the program and meta-program layers can be done in small increments, as XP demands. I shall return to this at the end of the talk.

1.1.4 The XP Legacy Problem

XP needs refactoring. Refactoring needs a means of showing that a change is (or is not) behaviour-preserving. The refactoring browser has a strong means: provable formal equivalence. XP uses a weaker one; if my tests don't break then the behaviour I care about has been preserved. Key to XP's viability is that test-driven development also supports refactoring; the tests that drive development of a feature go on revealing when a refactoring breaks that feature.

When XP is introduced to a large existing system, this synergy is lost. Writing tests for the legacy is a serious and demotivating problem. The original coders are often long gone, leaving the would-be XPer to try and work out what the development tests should have been. Without such tests, however, refactoring is constrained to stay within the area of new XP-style development or else be unsafe.

This would be a problem in any context but I think it's especially important for Smalltalk. In the lean years of Java-hype, a lot of Smalltalkers survived in legacy systems; systems that management decreed were to be replaced but which were just too complex to rewrite in a more fashionable but less productive language and too vital to discard. These systems need XP.

1.1.5 Getting a Handle on Legacy

In trying to solve this problem for our system, Andrew and I decided to exploit certain features it had, features we suspect it shares with other such survivors. Firstly, it was of course tested - in good old-fashioned waterfall style by integration testers at the back-end of the development process. Thus we could always feel reasonably sure that the behaviour exhibited by the previous release

was acceptable. What we couldn't do was have the integration testers rerun their effort-intensive process every time we changed a line of code in the upcoming development stream.

A second feature was that a fairly small subset of the total behaviour was deliberately changed in each 3-month release; a majority of our hundreds of supported products did not have changes requested against a given release. Unfortunately, the integration testers still had to retest all of it, lest we had accidentally altered something.

The third feature was that, considered very abstractly, the system's function was to enforce complex business logic between multiple users' editing and viewing of product data on the one hand, and the database' acceptance of valid data on the other. Thus products went through a sequence of generally comparable states at which the system's status in regard to them was adequately captured in smaller subsystems, such as

- the actual values held in the UI widgets
- the actual model objects saved to the database
- the business logic validation objects raised

These items were not meaningful as individual objects but they did provide pointers into object graphs which adequately expressed the outcome of operations. We saw them as narrow channels through which the system's behaviour momentarily flowed, conveniently built from a finite set of widgets and relationships. (Commercial UI builders usually have a very finite set of widgets and relationships, and while the validation and database models were our own utilities, they shared this feature.)

1.1.6 Our Approach: Fundamentals

Putting all this together, Andrew and I decided to try out an additional definition for what an XP refactoring could be. Let the behaviour whose change we want to detect be represented by the network of objects it creates from a root in some appropriate subsystem (the UI, the validation logic, whatever). Create basic tests that capture these networks after exercising a given product to a given state. Use appropriately-truncated deep comparison to verify whether tests run before and after a refactor captured the same networks. Simple.

Of course, there were a few small details to sort out. :-)

1.1.7 Obtaining Tests to Compare

The work to implement this fell into three parts:

- creating a suite of basic tests
- creating a test browser framework to run them and their deep comparisons
- creating the deep comparison framework

The first of these was the most straightforward. We defined basic tests to exercise a scenario (i.e. move reused or newly-created product to given state) and capture a given network graph.

Product type data and product rules are stored in what our system calls standing data, populating a smaller number of templates, themselves pluggable. (This is an example of another meaning sometimes given to meta-programming but I call it meta-data, not meta-programming, myself.) Thus it was most natural for us to create tests for each product by giving the basic test classes key instVars, (and the behaviour of copying these when their instances were rerun or debugged; a behaviour not in the SUnitBrowser which has no concept of tests having key instVars). We could then create tests for all products by creating instances of our chosen generic test populated from our meta-data. We found this a flexible and cost effective method but arguably it is a departure from the original SUnit philosophy, a point I'll discuss later.

Lots of reusable product instance data existed in the database along with generic methods to display it. This data we could reuse, in the form of entire reused products, or as values to populate new clients and products by deep copy-replace at the model or view layer. We also instrumented each model class to offer example instances that partitioned its type, letting us create new products and jitter existing products. Note that these new products did not have to be valid in terms of business logic; testing whether a refactor changed the handling of invalid product submissions is at least as valuable.

The only essential basic test assertions are those that check the chosen state was reached before network graph capture. We added such others as we knew were valid and had time for. (Arguably, as time passes, much that is now checked by deep comparison may be refactored to specific assertions in product tests as debugging test failures teaches developers what the legacy needs.) These assertions run, the test then captures an appropriately-deep copy of its narrow subsystem root's graph. For example, validation logic (most used to date) also copies any model layer objects that have errors. View layer roots capture editable widgets and all that connect them but, except for widget values, ignore the model layer.

1.1.8 Test Browser Framework

The subclassing of the SUnit and SUnitBrowser frameworks to get what we wanted was reasonably straightforward. The classes in our ComparisonTestCase sub-hierarchy run various flavours of deep comparison assertions between the two tests that each comparison test instance holds. Our test browser (which we made packagable since our release process requires some packaged use of it, done by loading an altered version of SUnitBrowser's superclass) holds multiple instances of TestResult and/or ComparisonResult, viewing one at a time. I'll describe the process verbally and show a brief demo later in the talk.

- Open the browser on your chosen suite of basic tests.
- In your pre-refactor image, run some basic tests to capture a baseline of run test instances. Copies of the tests you select will be run and added to the current TestResult (which it is advisable to name appropriately so you can recognise it later). At this point the browser has few differences from the standard one; you can run tests you don't intend to compare, rerun tests, etc.

- Once the pre-refactor result is as full of tests as you wish, effect your refactor (more on that in a minute) and request a new result with an appropriate name. (For example, if your refactor's development test were called testAnnuityCanHaveMultipleGrantees then your test results might be called AnnuityHasSingleGrantee and AnnuityHasMultipleGrantees, or they might be called Stream-P17 and Stream-P17a, or by the dates on which they were started, or whatever best helps you recognise them.) Run the tests you want to compare again; their run copies will now populate your new result. As before, you can run other tests as well.
- Request a comparison result for the two. Running a selected test now runs a copy of the ComparisonResult's comparison test on the two run copies of the selected test being held in the two results being compared, and puts the result in the ComparisonResult.

You can continue the process, creating further results and comparing tests run in your current state with the original tests, with the most recent tests, whatever.

1.1.9 Test results in the Test Browser

When viewing a standard TestResult, results mean what they usually do: the colour shows which of the three possible outcomes occurred. When viewing a ComparisonResult:

- Pass (green) means both tests had the same outcome (Pass, Fail or Error) and they satisfied the comparison test's deep comparison assertions.
- Fail (amber) means both tests had the same outcome but they did not satisfy the comparison test's assertions
- Error (red) means the tests did not have the same outcome (e.g. one passed and the other errored). The comparison test assertions are not performed in this case as they could be met only by misleading accident.

I spoke earlier of 'effecting the refactoring' between distinct test result runs. The browser supports dumping serialized test results to a file and reloading them into compatible browsers in other images. Comparison can also be done within the same image as ran the tests that populated the pre-refactoring result: run the tests and then either make the change, or load a configuration map that has it. (For example, I have sometimes maintained a developer and a comparison image, regularly releasing my current state into my developer map in the developer image, then (re)loading that map in my comparison image and re-running my tests.) When comparing intra-image, the deep copy that is effected by serialized dumping has to be done explicitly, so we routinely set up the tests to do it anyway.

1.2 Meta-Programming Frameworks

We needed deep comparison, deep copy and deep copy-replace for the main task, with reference tracing for debugging (especially, debugging the dumping of test results). We found few examples of public domain deep-graph-walking frameworks and none for deep comparison or copy-replace. I'd be delighted to learn that we were not looking hard enough and there are dozens; that's one of my motives for giving this talk. Meanwhile, I benefited from Paul Baumann's deep copy framework (and article [1]) and from the Refactoring Browser's ReferenceFinder . Hence I wrote deep comparison and deep copy-replace frameworks, and adapted deep copy and reference tracing to our needs.

1.2.1 Base Deep Comparison Framework

I developed deep comparison test-first in standard XP fashion. Also in standard if non-ideal fashion, subsequent practical use revealed omissions in my initial tests and assertions. The main issues that arose were two.

- firstly, collections require much subtler handling in deep comparison than in deep copy or reference tracing; backtracking and comparison order issues took plenty of fixing
- secondly, we needed all our deep-walking strategies to be both more customisable and customisable from more contexts.

The most basic part of the framework is a comparison strategy paired with methods whereby the objects being compared invoke it. Structural comparison comes first; are the two objects sufficiently alike to make detailed comparison safe and sensible? Content comparison, unless overridden in specific classes, asks the strategy both what to compare and how to compare. A third method (not on slide) truncates comparison.

The comparison strategy's own most basic job is shown in the (simplified) code snippet; to keep a dictionary of what has been or is being matched to what, and to use that when the same node in the graph is encountered again.

1.2.2 Customisable Deep Comparison Framework

There are two ways to customise comparison.

- The first way is by overriding the basic methods that objects invoke in more specialized classes and in a hierarchy of layers. (Each layer calls the one below in class Object if it does not encounter an override.)
  - — Lowest above the Basic Layer is the Vendor Layer, where vendor-specific classes, singletons and other special values are handled, etc. (So far we have layers for VW and VA.)
  - — This is called by the General Layer, where default choices for collection order comparison, immediate-to-non-immediate comparison and acceptable equivalents to nil are set.
  - — Above these, whatever layers a specific application may require call the general layer (or a lower application layer). We've written two distinct application layers for our system to date, one for the legacy testing I'm describing and one to support testing changes to our database and its access protocols. Deep comparison was a particularly convenient way to test the latter since by definition all object graphs produced by all operations on the Smalltalk side were required to be exactly the same under these changes.
- The above applies to every comparison. Both while developing the right comparisons for the system layer to use in general and when a given comparison applies to only one or a few tests, we needed means of customising that applies per comparison call. I provided three ways of doing this.
  - — Firstly, blocks set on strategy instance creation couple the structure, content and truncation methods to the strategy. Thus the caller can

choose which layer to use and wrap the calls with local handling or alternates if they desire.

— Secondly, a strategy can be set to ignore given instVars. The relevant indices are recomputed on each root call so changes to class shapes and subclasses hierarchies are either handled or (if they affect the instVar names) caught.

— Thirdly, a strategy can be set to ignore instances or handle them in particular ways by pre-populating its dictionary. It can also be reused; its dictionary can be manipulated, e.g. to ignore a difference found, and then the comparison rerun.

1.2.3 Tools (that helped us meta-program)

We found dynamic type recovery a useful way to get a first cut on what classes to override in the system layer. In VA, we found SmallTyper a very useful tool for this (once we realized that some combinations of partial type recovery were incompatible, a fact the manual forgot to mention; we now routinely recover all types for any chosen class). We would instrument our system classes of interest, run some tests, examine the generalized types of their instvars and so get a first approximation to which classes might benefit from system-layer comparison overrides. (In VW, the equivalent tool is the Analysis Browser. I'm interested in knowing whether other dialects have similar tools.)

Another utility we used was the dumper/loader. This does graph-walking too, of course. There were two minor bugbears.

- Firstly, whereas all our strategy classes used the same protocol for overrides, the dumper uses its own protocol. Almost invariably, every strategy being used in a given context wants to ignore, handle specially or handle generally the same instVars in the same, or analogous, ways. It was tedious to translate the code used for all our strategies into the different protocol the dumper requires. (Sometimes we didn't bother but used hacks, e.g. shutting down the product's view layer while dumping.) I have thought of writing a custom refactoring for it or else writing wrappers that map my protocols; perhaps I will get around to one of these sometime.
- Secondly, the VA dumper, being implemented with primitives, was much harder to debug than ours. Doubtless it ran faster than if it had not been, but that was not an issue for our usage. I would have welcomed a non-primitive setting. Our most frequent use of ConstrainedReferenceFinder was to track down the causes of dumper errors. If there's some obvious way round this, I'd be glad to learn it. I'd also be interested to know how other dialects' dumpers compare on these issues.

Something we would have liked, though its absence hasn't bitten us yet, is RB awareness of our meta-protocol. It would be good, when renaming an instVar, to be at least warned of strategy customisations that refer to it, and better to have them refactored along with the rest. I may look into this. Custom refactorings to map between the various ways of customising a comparison would also be nice.

These tools specifically helped us meta-program. Of course we also used the RB incessantly, plus method wrappers (about which I'd like to talk to anyone who understands how the VA packager handles CompiledMethods), Greg's quality control tools (he has them invoking Smalllint as well - we requested it - and I've learned not to ignore the 'Guard clauses' warning), and others.

1.3 Demo

See the slides.

1.4 Discussion: Value of this Work

My main reason for giving this talk is that I want to get a feel for the value of this work, if it has any. Tell me your opinion now, catch me over dinner or email me as suits you.

1.4.1 Future Directions

Firstly, what is its value in introducing XP to Smalltalk legacy? I gave my reasons for thinking the Smalltalk legacy issue important near the start of this talk. Do people agree? And if so, is this technique generally viable? If there is sufficient feeling that it is, there are some things that can be put into the public domain on various timescales. I'd be happy to provide a public domain version of the Test Browser in time for this summer's Camp Smalltalk to align with the latest SUnit, port to other dialects or whatever. (And of course, I'd be delighted to provide consultancy on applying this technique to other systems.)

(A secondary issue in this is what do people think about key instVars for TestCases. Is that a legitimate development of SUnit? If not, what should we do instead - generate test case code from our standing data? This touches on SUnit issues that themselves need resolving; whether the test instance itself or a copy of it is executed in run mode and in debug mode differs between basic SUnit and SUnitBrowser. Resources are also differently handled in the two cases.)

Secondly, what is the value of this work in particular and meta-programming in general to Smalltalk? I've given my reasons for thinking that meta-programming frameworks, provided they allow easy program-level overriding and instance-level customisation, should be a key smalltalk enabler. Do people agree? I feel that meta-programming frameworks have a lot in common - quite enough to make it worth thinking of common core implementation and protocol. For example, I found it very straightforward to add my customisation protocol and implementation to a subclass of the Refactoring Browser's ReferenceFinder. So, would anyone use a common framework or protocol? Does one already exist? If I sit down to write 'Smalltalk Best Practice Meta-Patterns', would there be any co-authors to offer examples of patterns or, better still, implementations? Has the book already been written; can I buy it instead of writing it? Questions and comments, please.

1.5 Other Remarks (no slides for these)

These are untried ideas that arose while I was doing this work.

1.5.1 Using Deep Comparison to Refine our definition of 'Behaviour'

A variant approach we have not yet tried would be to deepCopy from the narrow subsystem root before running the test, then after running, run a comparison and note all the differences within a given depth. This information could be passed with the test and the comparison between the two tests run only on these differences i.e. on the objects altered by running the test (of course, a compare general start state test would also be needed). It is possible this would significantly reduce the amount of comparison customisation needed and/or focus in on the actual behaviour elicited by the test.

1.5.2 Other Uses of the Test Browser

No rule limits using our test browser to run only deep comparisons. Are there any other kinds of secondary assertions on single, double or multiple tests that would be worth running? Could it be developed into a test composition browser?

1.5.3 Comparison Framework Optimisation

Much Prolog compiler work addresses backtracking optimisation. My framework is unoptimised at present, my hasty attempts to apply these ideas crudely to my framework having met errors. Can anyone advise, e.g. how does SOUL and other Prolog-in-Smalltalk work handle backtracking?

1.5.4 'Move to Component' Refactoring

Frequently, especially when developing complex test suites, I found myself needing to refactor a method from the class where it was defined to a parameter's (or instVar's) class, e.g.

```
ProductTestCase>>copyClient: aClient
     "Code that after successive refactors now has little
or nothing to do with ProductTestCase"
```

into

```
Client>>copy OR Client>>copyInTest: aTestCase
     "'aClient' replaced by 'self', 'self' replaced by
'aTestCase', and perhaps any directly-accessed instVars of
aTestCase replaced by accessors"
```

with the calls being appropriately rewritten. This often happens when utility methods first defined on the test case are refactored into chunks some of which belong on the classes being tested. Sometimes, as the method develops, the need to dispatch on the parameter's subclass makes it essential to move them there.

The relevant refactoring is 'Move to Component' in the RB. I mention this in case anyone else is as slow as I to realise this.

1.6 Issues

Our tests have key instVars. I've had to overriding methods in SUnitBrowser and alter TestCase>>debugUsing: in SUnit to achieve this. I propose that systems that use meta-data will naturally want to write test classes with key instVars so that their instances can be populated from that meta-data. Any agreement or argument?

1.7 Platforms

Almost all the comparison framework is dialect neutral. A vendor layer for each dialect is essential but fairly straightforward to write from another dialect's example. Layers for VW and VA exist.

The browser currently runs only in VA. A straightforward set of widget tweaks should port it to any dialect that already has SUnitBrowser. Part of any port to a dialect that used another SUnit UI would be making the tests in that UI's test suite distinct from the (copied) tests actually run by it.