# Testing for Real

## ESUG: 2006

# Refactoring Test Code out of Real Data

**Niall Ross, eXtremeMetaProgrammers Ltd, nfr@bigwig.net**
**Massimo Milan, Lifeware SA**
**Massimo Arnoldi, Lifeware SA**

Slides of my talk at the European Smalltalk Users Group Conference in Prague, September 4th - 8th, 2006.

# Overview

**Background**

**What is the problem ?**

- **In theory**
- **In context**

**What is the solution ?**

- **Elements: test-writing framework and domain objects**
- **Approach: refactor from data**
- **Future Work**
- **Discussion**

# Background

**Lifeware provides a system to manage life insurance contracts.**

- **clients design and sell insurance products**
- **Lifeware provide back-end support throught the lifecycle**

**A generic system is customised to specific insurers' needs**

- **benefits of robustness and experience**
- **each insurer sees their unique modus operandi**

**Lifeware's value proposition: pay per contract, not per system**

- **selling one contract commits a client to manage it for decades**
  - **— develop your own system upfront = risk**
  - **— pay for what client sells = no more cost than client can afford = no risk**
- **ability to compute exact IT cost**

**Lifeware uses VW and GemStone.**

# The Problem: Theory

**Test-driven development: the greatest thing since sliced bread !**

- **Tests make you code faster, rewrite to get it right**
- **The tests stay around so your system stays right**

**So the developers all lived happily every after: well ...**

- **Development tests**
  - **— cluster near the initial / normal states**
  - **— describe how the system *should* be used**
  - **— guard against obvious errors**
  - **— are no longer than the developer will write**
  - **— are no more complex than the developer can imagine**
- **Real users**
  - **— use the system as they need to, not as they ‘should’**
  - **— do incredible things to correct incredible mistakes**

**So we should just write better tests? Well ...**

# The Problem: Theory (continued)

**What is a test? Theory in the literature speaks of**

- **Test fixture: an initialized model of domain objects**
- **Test Stimuli: operations applied to this model**
- **Expected results: assertions that should hold after these operations**

**These states are not in fact separable (especially to Smalltalkers)**

- **production code is refactored against tests**
- **tests are refactored against production code**

**Very soon, fixtures, stimuli and assertions all mingle**

- **tests are scripts: building, asserting, reshaping, ...**
- **a domain evolves test frameworks to build these scripts**

**Test frameworks speed test writing but even more important:**

## A test script must be *readable*.

# The Problem: example

**Lifeware was a very early adopter of Kent's test framework**

- **now they have an impressive, distinctive test framework**
- **12,000 tests run whenever a developer integrates; 3,000 more run nightly / weekends**
- **TestBuilder framework classes support writing test fixtures**

**But**

- **Insurance contracts have many configurations**
  - **— customer types and roles: person/company, funder/beneficiary, ...**
  - **— funding patterns: lump sums, scheduled payments, ...**
  - **— investment patterns: types of investment, specific funds, ...**
- **Contracts have complex lifecycles**
  - **— intentionally complex: flexible payments, weighted schedules, ...**
  - **— unintentionally complex: cancellations, missed/restored payments, revisions, ...**
    - **– complex fixes to these unforeseen situations**

**Huge volumes of very complex domain objects accumulate in GemStone.**

# Not the Solution

**We have**

## Framework to create simple readable tests

**Why not write more complex tests to match our complex actual usage?**

- **not enough keystrokes in the working day**
- **not enough neurons in the developers brain**

## Complex persistent domain objects

**Why not use them in tests?**

- **unreadable: the test's meaning is in an inspectable object graph, not in code**
- **brittle: small changes appear as failures, waste investigation time**

**If only we could somehow combine the two.**

# The Solution: Refactor Test Scripts from Data

**Enter the Refactoring Framework (Niall's answer to every question :-)**

**Pre-step: annotate appropriate test builder framework code with value getters. Then**

- **Fault in the object model from GemStone to VW**
  - **— history reified as temporal series of events**
  - **— dependent temporal data reified as pseudo-events: fund values, exchange rates**
  - **— time travel via method wrappers: posting dates, effectives dates, perspectives**
- **Match the basic builder classes for this kind of contract**
- **Match events to builder methods**
  - **— by event class, by data values, ...**
- **Parse methods and rewrite: evaluate some nodes, refactor others**
  - **— recursively[1] inline conditional nodes and loops from their values**
  - **— replace parameter expressions with values (literals or constructors)**
  - **— anti-inline resulting code to convenience methods**

1. Actually, this is more complex than mere recursion; it must mimic smaltalk code execution (see below).

# Example of Refactoring from Partial Evaluation

**Pre-step) Rewrite test fragments ...**

```
builder ...
   fund1: 'Anlagestrategie-SpeedLane' percentage: 100;
   putInForceAs: ...

builder ...
   fund6: 'Franklin US Equity' percentage: 5;
   fund7: 'Vontobel Swiss Stars Equity' percentage: 10;
   putInForceAs: ...

builder ...
   lifelongStrategy: Strategy conservativeStrategy;
   putInForceAs: ...
```

**... to event partial-evaluation fragments**

```
self fundAllocation isManagedStrategy
   ifTrue: [builder lifelongStrategy: self fundAllocation strategy]
   ifFalse
     [self fundAllocation funds doWithIndex:
       [:each :index |
       builder
         fund: each fund displayShortString
         at: index
         percentage: each percent]].
```

### 1) Inline conditionals

```
self fundAllocation funds doWithIndex:
   [:each :index |
   builder
     fund: each fund displayShortString
     at: index
     percentage: each percent]
```

### 2) Expand loops ...

```
builder fund: each fund displayShortString at: 1 percentage: each percent
builder fund: each fund displayShortString at: 2 percentage: each percent
builder fund: each fund displayShortString at: 3 percentage: each percent
```

### ... and anti-inline

```
builder fund1: each fund displayShortString percentage: each percent
builder fund2: each fund displayShortString percentage: each percent
builder fund3: each fund displayShortString percentage: each percent
```

### 3) Evaluate and replace parameters as literals, e.g.

```
builder fund1: 'Threadneedle European Growth' percentage: 50.
builder fund2: 'Franklin US Equity' percentage: 30.
builder fund3: 'Vontobel Swiss Stars Equity' percentage: 20.
```

### or (e.g. if conditional above had evaluated to true) as storeOn: expressions

```
builder lifelongStrategy: Strategy dynamicStrategy.
```

## 4) Inline into overall ‘event’ method

```
TrioContract1001080Test>>application
   self timestamp: (16 jun: 2003) @ '11:10:36'.
   builder
      newApplicationOn: (23 jun: 2003);
      maleBornOn: (12 jun: 1967);
      firstname: 'Peter' lastname: 'Winter';
      street: 'Asylstrasse'
        civicNumber: '55'
        zip: '84030'
        city: 'Zurich';
      insuredJob: 'Research Engineer' jobKey: 7804;
      monthlyPremium: 40 years: 20;
      duration: 25;
      coverage: 9600;
      beneficiary: (Beneficiary text: 'Clara Winter');
      fund1: 'Threadneedle European Growth' percentage: 50.
      fund2: 'Franklin US Equity' percentage: 30;
      fund3: 'Vontobel Swiss Stars Equity' percentage: 20;
      putInForceAs: '1001080'
```

**Superclass of generated class chosen to match that contract’s lifecycle**

**Superclass’ methods denote events in lifecycle (e.g. customer applies for insurance)**

## 5) Finally we place this event in the overall history

```
replay2003Q2Events
   self
      loadBank: 'Sparkasse Südliche Weinstraße in Landau'
      zip: '76831'
      city: 'Billigheim-Ingenheim'
      clearing: 54850010
      checkNumber: '00'
      recordNumber: '012524'.
   self brokerFirstname: 'Udo'lastname: 'Paul'number: '1006400'.
   self application.
   self
      loadPrice: 100
      fund: 'Vontobel Swiss Stars Equity'
      date: (27 jun: 2003)
      timestamp: (27 jun: 2003) @ '07:47:00'.
   self
      loadExchangeRate: 1.1457d
      from: Currency EUR
      to: Currency USD
      date: (27 jun:2003)
      timestamp: (27 jun:2003) @ '10:27:56'.
   self
      collectPremiumOn: (1 jul:2003)
      timestamp: (27 jun:2003) @ '10:27:57'.
```

# Generating Tests

## Map event to partially-evaluable method

```
TestEventMethodGenerator>>generateTestForEvent: anEvent
   ^self
     generateTest: (anEvent class parseTreeFor:
                                          anEvent class buildOnSelector)
     forEvent: anEvent
```

## Overall partial-evaluation refactoring

```
TestEventMethodGenerator>>generateTest
   "Evaluate all conditions and arguments in tree, treating the supplied
event as self.  Thus rewrite the tree to inline all conditions and replace
all event-dependent expressions with literals or independently-evaluable
expressions (latter obtained via storeOn: sent to result of evaluation)."

   self inlineSelfSendsInTree.
   self inlineEventTempsInTree.
   self inlineConditionsInTree.
   self inlineLoopsInTree.
   (self model classFor: event class) compileTree:
     (self mapTemporariesToInstVarsIn:
       (self evaluateLiteralEvaluationsIn:
         (self evaluateParametersIn: self tree))).
   ^self tree arguments: #(); selector: event eventSelector; yourself
```

# Generating Tests (continued)

**Find the data nodes ...**

```
inlineSelfSendsInTree
   "Recursively inline sends to self or super in the tree with implementors
in the intersection of the event class hierarchy with the refactoring's
model's environment (set when I am initialized).  Before inlining, inline
any temps which require instvars of the event for their evaluation."

   self
     matchesAnyOf: #('self `@method: dummyARG')
     refactor: InlineMethodRefactoring
     inMethod: event class buildOnSelector
     forRBClass: (self model classFor: event class)
     do: [:inlineRefactoring |
          self
            inlineEventTempsInMethod: inlineRefactoring inlineSelector
            forRBClass: inlineRefactoring inlineClass].
```

**... and evaluate them**

```
RBProgramNode>>evaluateFor: anObject
   "Primitive evaluation protocol; assumes caller has ensured anObject is
rational for this node. Evaluate the node as if its code were running and
anObject were self."

   ^Compiler evaluate: self formattedCode for: anObject logged: false
```

# Refactored Refactorings

**Refactoring of refactorings**

- **to support recursive evaluation**
  - **— InlineEvaluableTemporaryRefactoring**
  - **— InlineRefactoring and InlineToComponentRefactoring clean-up**
- **to map between template test code and hand-written test code styles**
  - **— InlineEvaluableIteratorRefactoring: small loops v. repetition**
  - **— Anti-inline: detect code an existing method can replace and ‘extract method’ to it**

## Refactored Refactoring Framework

**VW7: resurrected ability to restrict refactoring model’s view of image**

# Future Work: make parse tree fully evaluable

**Today, we combine refactorings with RBProgramNode>>evaluateFor: aSelfObject**

- **evaluate + inline of a conditional is just partial evaluation**
- **handle complex expressions**
  - **— our current mimicing of Smalltalk execution is imperfect**
- **deduce evaluable sections from abstract context**
  - **— or wrap in handler, stop when error raised? (beware unintended polymorphism)**

**Evaluable Abstract Grammar evaluation frameworks exist**

- **Zork-Analysis AG framework (VW, in Cincom OR)**
- **SmallTyper uses a framework for (VA Tool)**
- **others ?**

**We need to reuse or unify with them**

- **next step: partially evaluate the execution path, then refactor the remaining nodes**
- **future ideal: a single unified parse tree for evaluation and refactoring**

**(For this use) Could partial evaluation wholly replace refactoring ?**

# Discussion

**“Extracting realistic tests before beginning implementation is becoming as addictive as writing tests by hand was when the code and data were simpler.”**

**Kent Beck, after remotely pair-programming with Lifeware staff in summer 2006**