



IterEx

Expressive Testing ...and your Code For Free?

Tim Mackinnon
www.iterex.co.uk

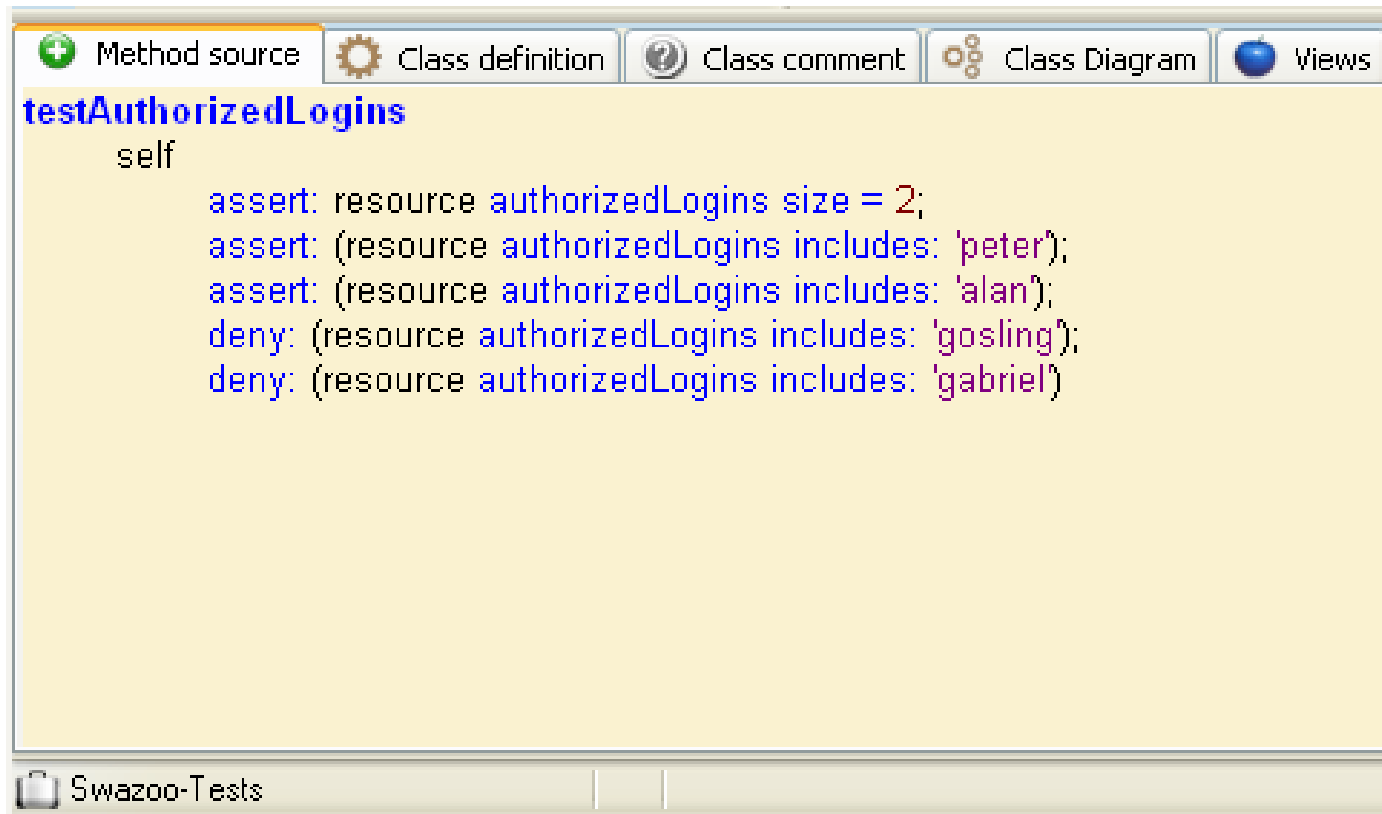
Presentation Outline+Objectives

- SUnit – features and flaws
 - Importance Of Failures – failing is good right?
 - Readability
 - Assertions as Intent
 - More specific testing
 - Code for free
-
- I want to get feedback on these ideas, is there something interesting?

SUnit – 13 Years On

- From the SUnit site:
“SUnit is the mother of all unit testing frameworks, and serves as one of the cornerstones of test-driven development methodologies such as eXtreme Programming”
- Simple model for testing, inspired many X-Unit clones
- Available on all Smalltalk implementations
- Current Smalltalk version is 3.1 – stable but no recent development work
- Many popular frameworks ship with SUnit tests. So how are tests in the wild?

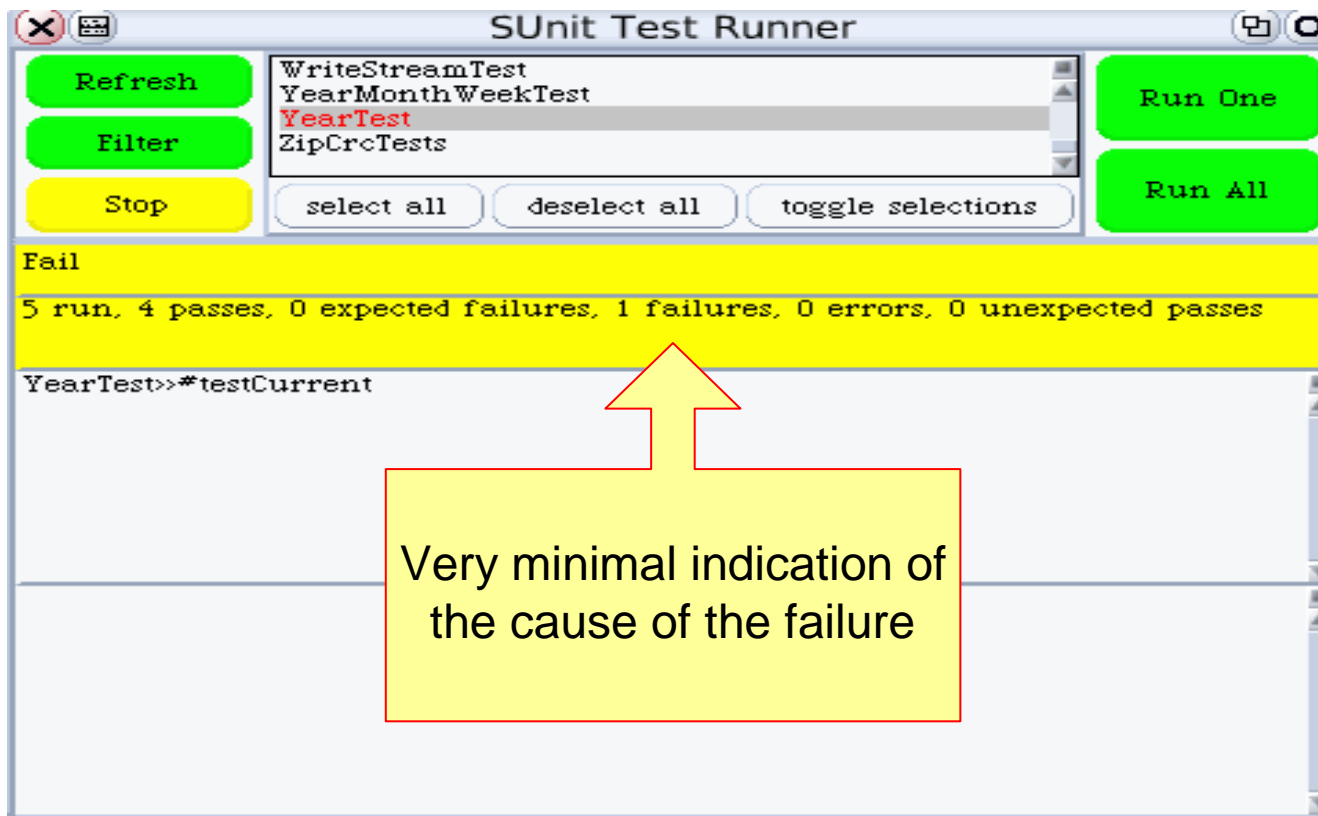
Sample SUnit test



```
Method source | Class definition | Class comment | Class Diagram | Views
testAuthorizedLogins
self
  assert: resource authorizedLogins size = 2;
  assert: (resource authorizedLogins includes: 'peter');
  assert: (resource authorizedLogins includes: 'alan');
  deny: (resource authorizedLogins includes: 'gosling');
  deny: (resource authorizedLogins includes: 'gabriel')
```

Swazoo-Tests

Sample SUnit Result

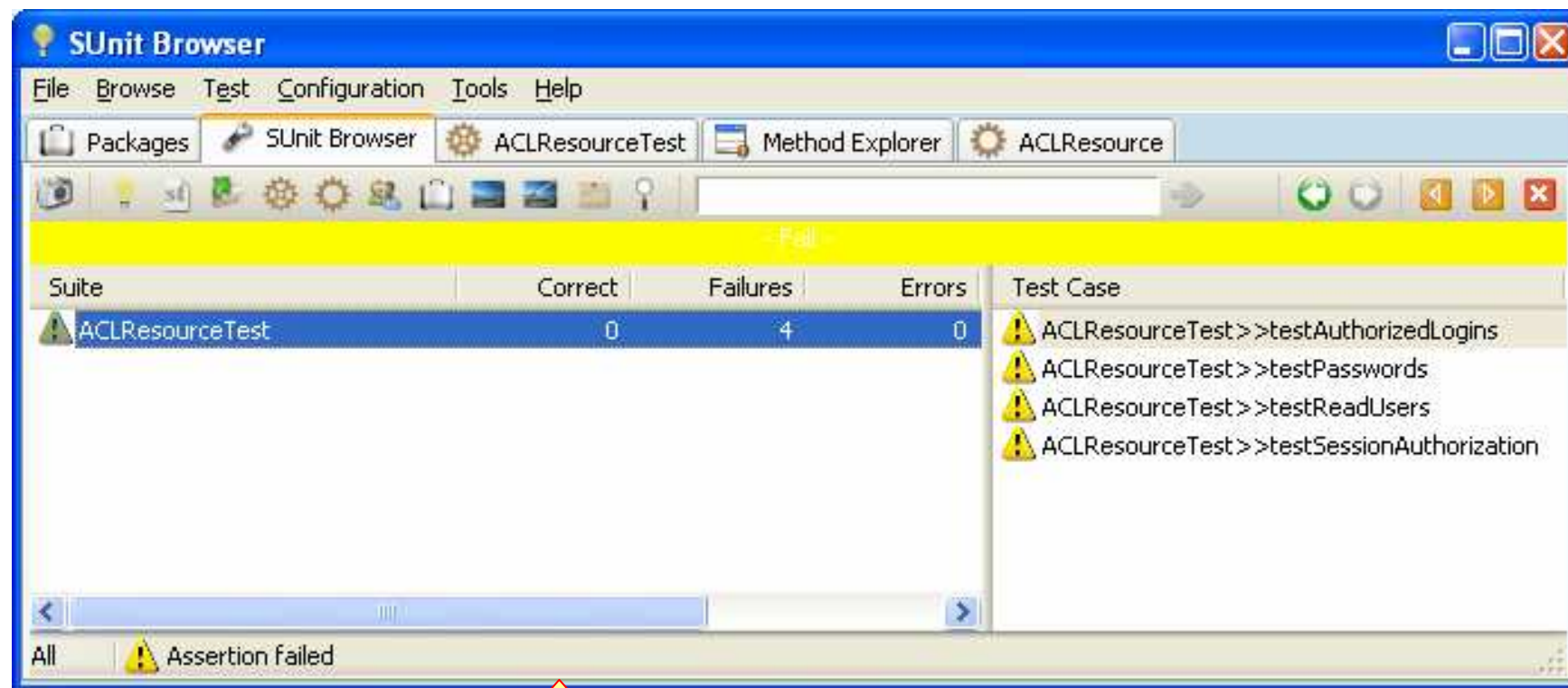


The screenshot shows the SUnit Test Runner window. The title bar reads "SUnit Test Runner". On the left, there are three buttons: "Refresh" (green), "Filter" (green), and "Stop" (yellow). In the center, there is a list box containing "WriteStreamTest", "YearMonthWeekTest", "YearTest" (highlighted in red), and "ZipCrcTests". Below the list box are three buttons: "select all", "deselect all", and "toggle selections". On the right, there are two buttons: "Run One" (green) and "Run All" (green). The main area of the window displays the following text:

```
Fail
5 run, 4 passes, 0 expected failures, 1 failures, 0 errors, 0 unexpected passes
YearTest>>#testCurrent
```

A yellow box with a red border is overlaid on the bottom part of the window, containing the text "Very minimal indication of the cause of the failure". A red arrow points from this box up to the "YearTest>>#testCurrent" line in the output.

Sample SUnit Result



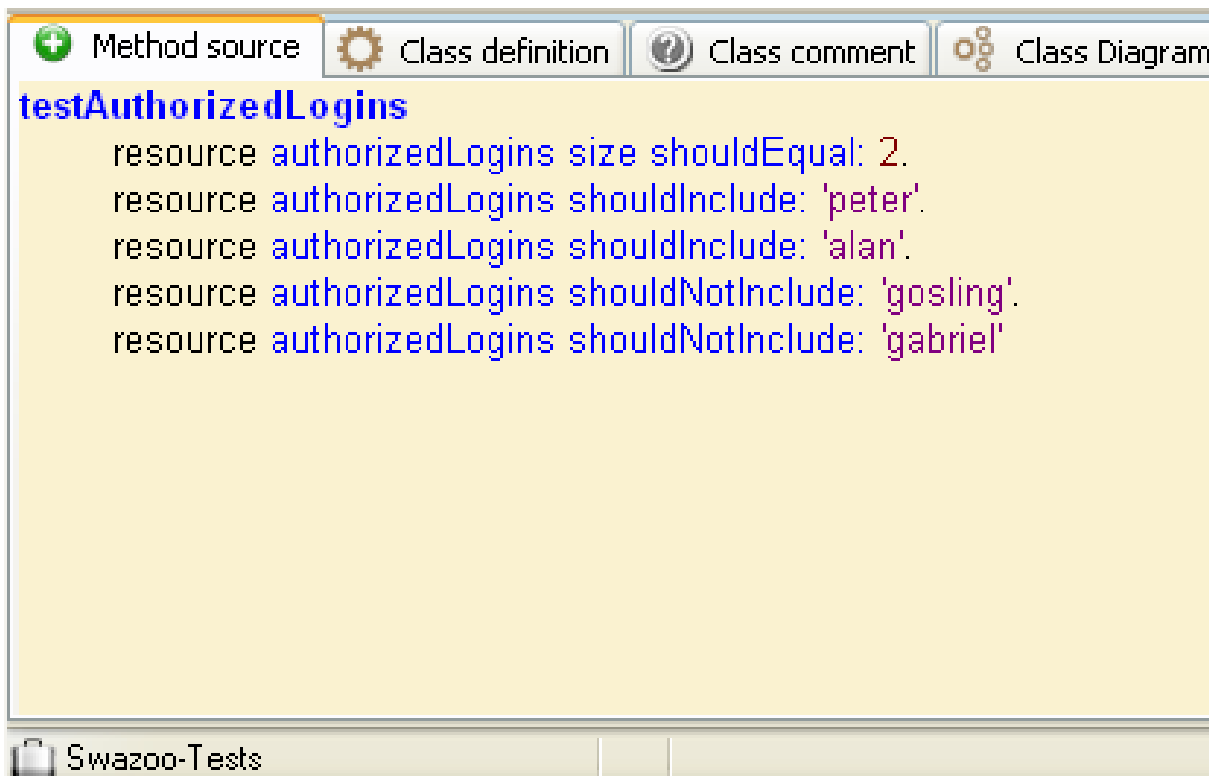
Slightly improved failure indication in the status bar via "Intelli-Dolphin" but still not particularly helpful

So what's wrong with this?

- Generic style tests require using a debugger to find out the problem
- The error displayed the in SUnit Runner is not very descriptive/helpful
 - Not bad when doing initial TDD, but if mass failures afterwards, can be tedious to track down a problem
 - Not always clear that you got the failure you expected unless you take the time to debug
- Erwin Reichstein (Carleton University – undergrad CS)
“If you don't find any errors in your code – you should be very worried”

How about writing tests a different way?

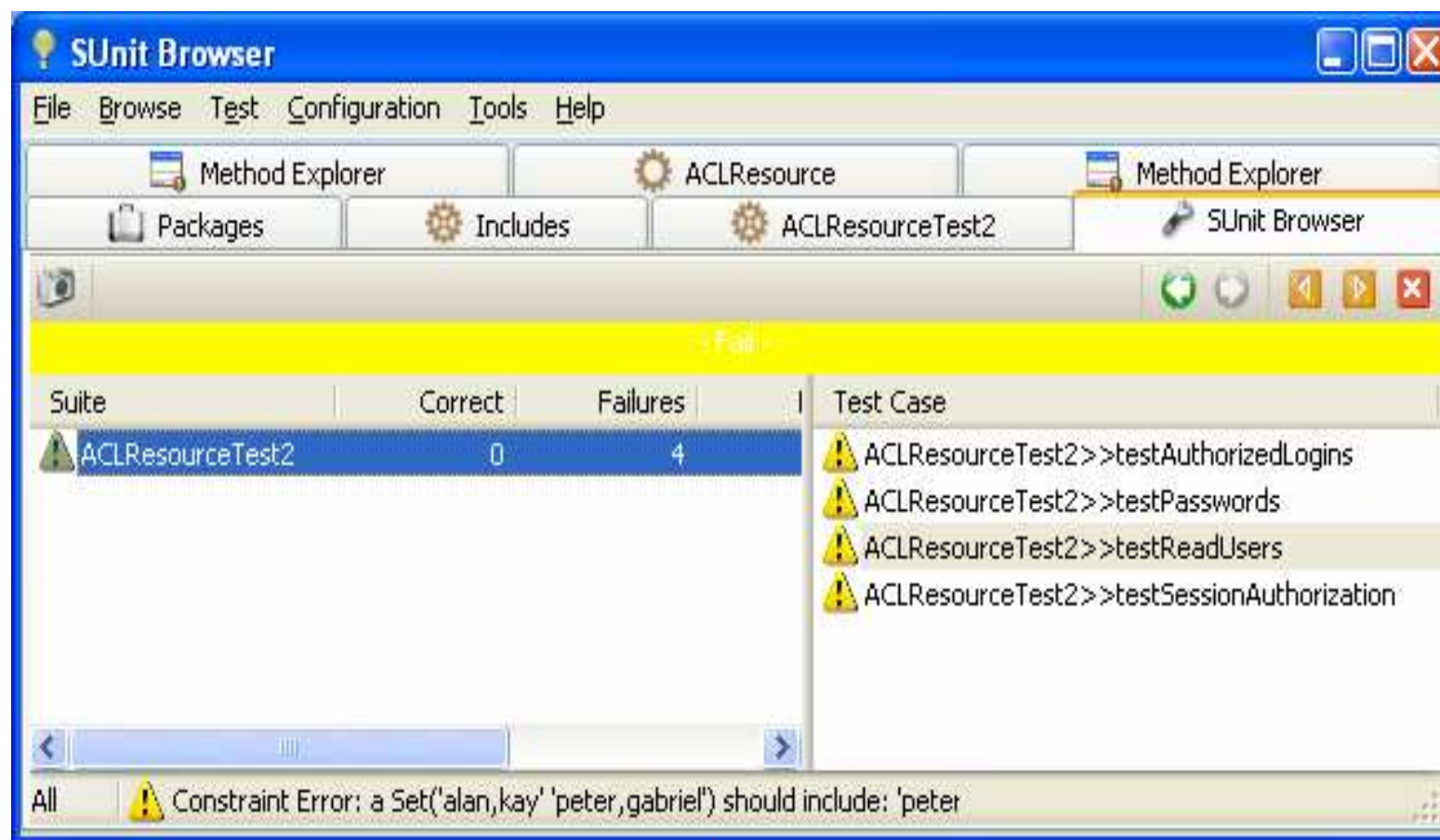
- Concisely indicate your test intent, and leverage this information to give clear messages for inevitable failure...



```
Method source | Class definition | Class comment | Class Diagram
testAuthorizedLogins
resource authorizedLogins size shouldEqual: 2.
resource authorizedLogins shouldInclude: 'peter'.
resource authorizedLogins shouldInclude: 'alan'.
resource authorizedLogins shouldNotInclude: 'gosling'.
resource authorizedLogins shouldNotInclude: 'gabriel'
```

Swazoo-Tests

And presenting test results more usefully?



Constraint provides a much clearer indication of the error

SUnit Issues that cropped up...

- TestResults do not store the exception that causes them
 - Therefore a UI has no additional information to report

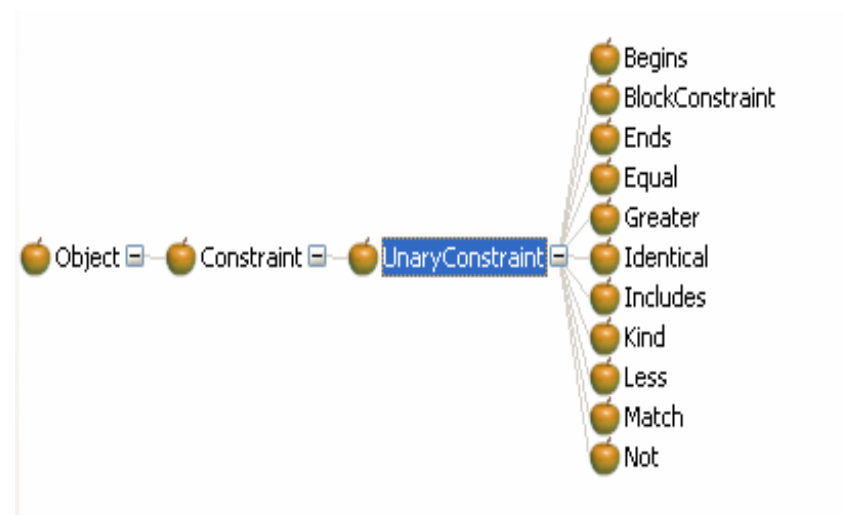
- If you store the exception, how can you get a meaningful message from it?

Expressing Expectations as Objects

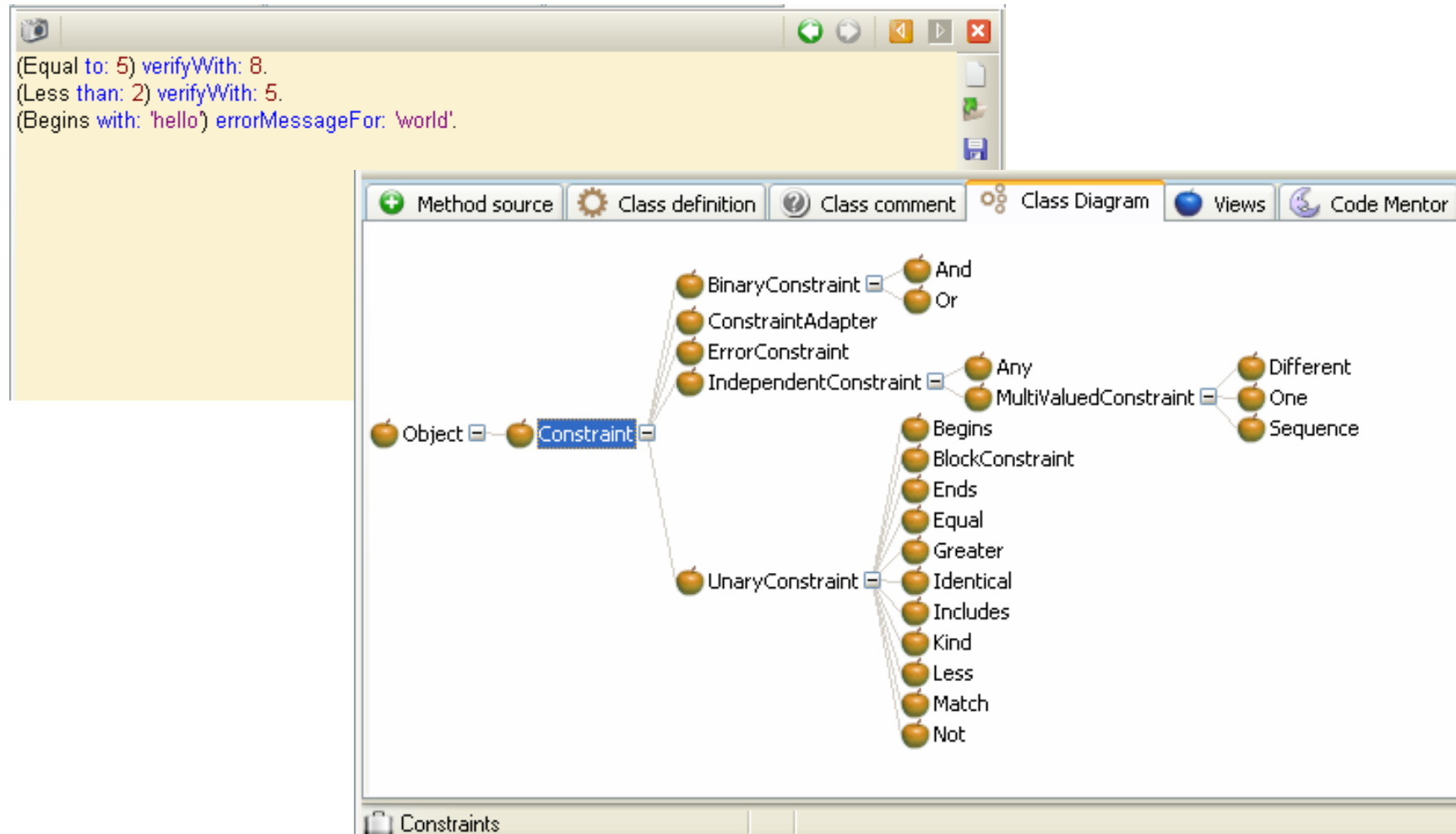
- Create a family of Constraint objects with a protocol:
 - #satisfies:
 - #verifyWith:
 - #errorMessageFor:
 - #printAbbreviationOn:

- Try to use readable terminology for instantiation
 - Equal to:
 - Less than:

- Loose methods for convenience of instantiation
 - #shouldEqual:



Lets explore some code



The screenshot shows an IDE window with a code editor and a class diagram view.

Code Editor:

```
(Equal to: 5) verifyWith: 8.  
(Less than: 2) verifyWith: 5.  
(Begins with: 'hello') errorMessageFor: 'world'.
```

Class Diagram:

- Object — Constraint
 - BinaryConstraint
 - And
 - Or
 - ConstraintAdapter
 - ErrorConstraint
 - IndependentConstraint
 - Any
 - MultiValuedConstraint
 - Different
 - One
 - Sequence
 - UnaryConstraint
 - Begins
 - BlockConstraint
 - Ends
 - Equal
 - Greater
 - Identical
 - Includes
 - Kind
 - Less
 - Match
 - Not

The class diagram is titled "Constraints" at the bottom.

More Complex Constraints

constraint :=

(Begins with: 'p') | (Ends with: 's') & [:i | i size < 5].

- In using constraints, discovered some useful new patterns:

(Begins with: 'p') & Different values.

Only values: #('peter' 'john')

Sequence of: #('peter' 'john' 'harry')

- Leverage these objects to generate more specific error messages:

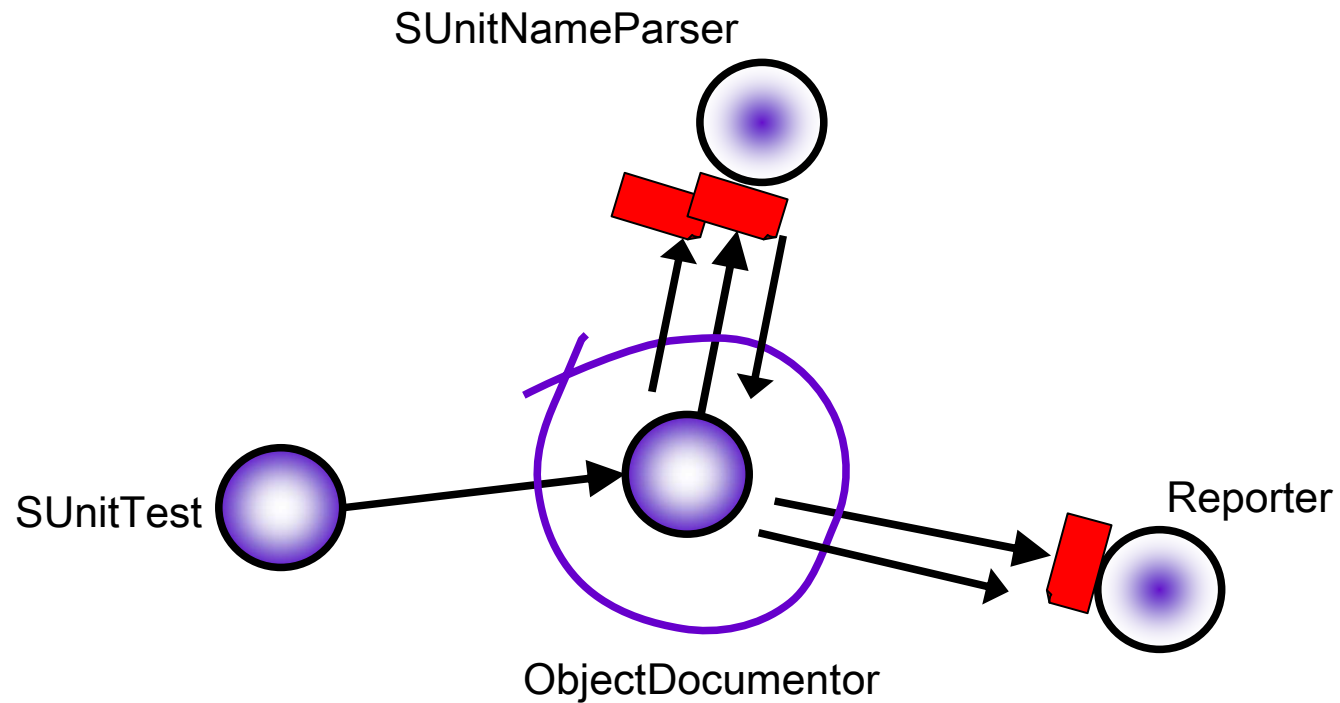
“john not item 1 in #('peter' 'john' 'harry')


Do constraints have other users?

- Yes - Specifying expected values on method calls for testing:
 - MethodWrappers
 - MockObjects

- Code generation

Example of a test using Mocks and Constraints



Use Constraints () to verify each invocation to a proxy object

Example Mock test using constraints

```
testDoesntProcessNonTestMethods
```

```
|report nameParser methods documentor|  
methods := #('setUp' 'testCalculates').
```

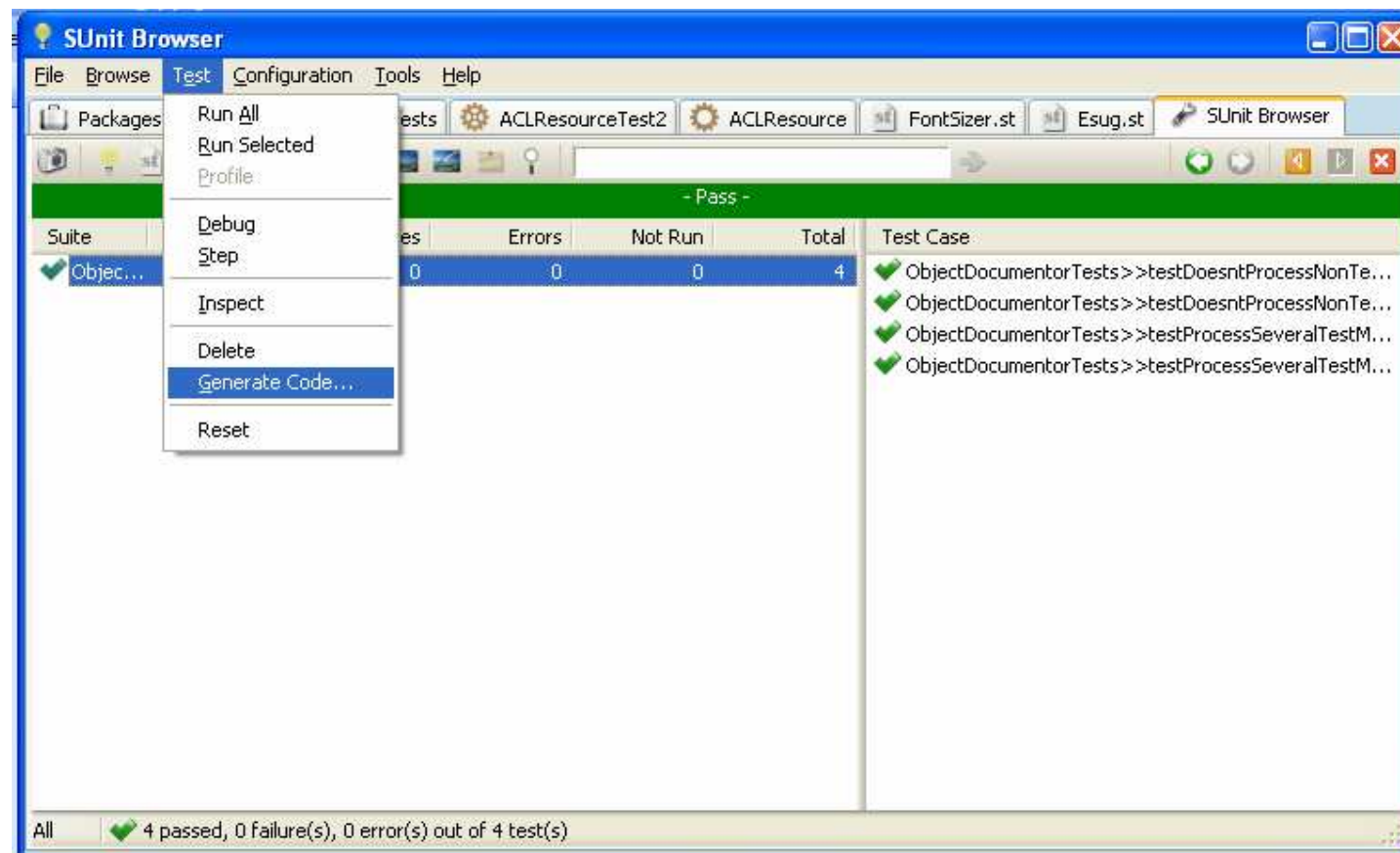
```
nameParser := mockery createMock: #SUnitNameParser.  
report := mockery createMock: #ResponsibilityReport.
```

```
documentor := ObjectDocumentor new.
```

```
[documentor process: methods using: nameParser onto: report]  
  expecting:  
    ([nameParser isTestMethod: (Only values: methods)]  
      answerWith: #(false true))  
  + ([nameParser parse: 'testCalculates']  
      answer: 'Calculates' exactly: 1)  
  + [report printResponsibility:  
      (Kind of: String) &~ (Begins with: 'test')] once
```


Generating Code from tests....

- Run the tests, gather all the mock objects used, ask them to generate code, protocols, comments.



Future Work

- Keep gathering useful constraints (like Sequence, Different etc.)
- Investigate if constraints can improve code generation (beyond simplistic usages)
- Investigate whether constraints can infer missing or conflicting test cases



IterEx

Conclusions

Summary

When a test fails – ask yourself:

Is it telling me everything it can about the failure?

Would expressing it as a test constraint make it clearer?

Speaker:

Tim Mackinnon

<http://www.iterex.co.uk/research>

Related Work

- James Robertson's Daily Smalltalk: ComplexConditions
 - (http://www.cincomsmalltalk.com/casts/stDaily/2007/smalltalk_daily-08-15-07.html)

Acknowledgements

- Blaine Buxton/Brian Rice for encouragement at STS2006
- Nat Pryce for introducing me to the idea of constraints as objects

Tim Mackinnon - Who are you?

- 1996 – OTI
 - Developer on teams credited for early use of agile practices

- 1999 – Connextra
 - Formed one of the first Agile teams in the UK
 - Invented “Mock Objects” test technique
 - Pioneered Iteration/Heartbeat Retrospectives

- 2003 – ThoughtWorks
 - Agile enablement coaching
 - Established hi-level release estimation techniques
 - Developed worldwide QuickStart project workshops

- 2006 – Iterative Excellence
 - Tailored Consulting for Agile projects
 - Iterex Professional – Software helping teams plan and track agile projects