Proceedings of
Expanding the Boundaries of Unit Testing

November 4, 2002
Seattle, Washington

http://www.metaprog.com/OOPSLAWorkshop

Workshop held at OOPSLA 2002:
The ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications

All papers are Copyright © 2002 by the author(s).
OOPSLA'02, November 4-8, 2002,
Seattle, Washington, USA.
2002 ACM 02/0011.

Edited by Joseph Pelrine (jpelrine@metaprog.com)

Contents

OOPSLA Workshop on Expanding the Boundaries of Unit Testing 5

1. Automating the Creation of Unit Tests 7
   Jenny Ahn, Parasoft
2. Expanding the Boundaries of Unit Testing 10
   Andrej Kline, FJA OdaTeam
3. Extending SUnit to Test SASE Events 13
   Blair McGlashan & Andy Bower, Object Arts
4. Implementing and Using Resumable TestFailures in Smalltalk 20
   Joseph Pelrine, MetaProg
5. Java/.NET Unit Testing 23
   David Vydra, Plumtree Software
6. Testing, Measuring, Pairing, Feedback: Variations of a Concept 27
   Hans Wegener, Swiss Re
7. Test-Driven Development Support in Immerjion 31
   Dwight S. Wilson & Matthew Pekar

Workshop on Expanding the Boundaries of Unit Testing

OOPSLA 2002

Themes and goals

One of the major benefits of the Agile Development movement has been to move unit testing to its rightful place as the software development technique. It has also supplied us with a family of unit testing frameworks to assist with this task. Unit testing, however, takes effort and skill to do well, especially for applications such as distributed systems.

Unit testing code is at the core of developing high-quality software, but is not widely understood in practice, and is too important to defer to the end of a project. It serves a very different purpose when used as part of Test-Driven Development, rather than a practice for reducing errors in existing code. We're still discovering new techniques and practices, and find there's lots of scope for discussion and dissension.

This workshop is one of a series concerned with bringing together and sharing experience between people who have experience of (or are just interested in) unit testing of software, particularly for Test-Driven Development.

The workshop will serve not only as a meeting point to exchange ideas about unit testing, but also as a "matchmaking" event between people with testing questions/problems and people with solutions. In this sense, the workshop will focus highly on the social interaction component. In the unit testing workshop which was held in Oxford, the organizers used a parody of the television show "The Dating Game" to match up people best suited to helping each other. We would like to use this and similar games/techniques to keep the atmosphere light and easy-going, to avoid inhibiting participants who would otherwise be shy about sharing their problems and fears.

Pre-workshop activities

There are no pre-workshop activities. Participants will be asked to submit a two-page position paper detailing their experience with unit testing in an agile project, and illustrating what they've done to expand the boundaries of unit testing. Typical examples would include extensions to the xUnit testing framework, automating tests etc. The organizers will accept position statements based on originality, relevance and suitability for discussion.

Post-workshop activities

The results of the workshop, as well as the accepted position papers, will be published on a web site, and will be made available to the community at large.

Topics of Interest

Workshop topics include, but are not limited to:

- Bad smells in testing
- Testing patterns
- Refactoring test code
- Test first design
- Acceptance testing
- Managing your test suite
- Dealing with testing conflicts
- Experience reports

Organizing committee

Organizer	Joseph Pelrine (CH)	jpelrine@acm.org
Co-organizers	Steve Freeman (UK) Tim Mackinnon (UK)	steve@m3p.co.uk tim@connextra.com

Automating the Creation of Unit Tests

When a developer completes a class, function, or other unit, he or she has to make a decision: “Do I perform unit testing to verify that the unit actually works? Or, do I hope that the unit is correct and move on to the next feature, assuming that any problems with the unit will be exposed during later (post-integration) testing?” I’ve found that developers typically choose the second option. Unfortunately, the decision to proceed with implementation without performing unit testing has some serious negative repercussions for both developers and end users.

Development teams who choose the second option typically end up producing a less reliable product and wasting more time and money than teams who choose the first option because delaying testing until the integration phase is both ineffective and inefficient. It’s incredibly difficult and time-consuming to find problems in the units when you test at the system level after the units are integrated. First, it’s difficult to thoroughly exercise individual units with system-level test cases. If you were to measure the unit coverage achieved by system-level tests, you would see that many units are barely even touched by the system-level test suite. With such spotty coverage, it is difficult to expose unit-level errors, and many errors will likely go unnoticed. Second, even if system-level tests do happen to reveal problems, it is much more difficult and time-consuming to pinpoint and repair the cause at this phase than it would have been during unit testing. When a problem is found in system-level testing, the developers need to search through the entire system to diagnose the cause of the problem. Once they determine what units were responsible for the problem, they need to spend time reviewing each unit’s purpose and specification because they have typically forgotten about these units by this point in the development process. As a result, finding and fixing a simple error in this manner can take hours or even days longer than it would with unit testing.

Why, then, do so many intelligent developers make the dumb decision to forgo unit testing? It’s not because they don’t know better. Almost every developer I’ve met understands the benefits of unit testing and recognizes that unit testing should be a part of the development process. However, when push comes to shove, few developers actually perform unit testing every time they complete a unit. Why? Typically, because it demands so much time and effort. To perform unit testing, developers need to:

- Build an environment that allows them to test a unit independently of the system it will eventually join.
- Determine what test inputs will verify the unit’s functionality and construction; ideally, these inputs should represent the inputs the completed, working system will feed to the unit, but it’s difficult to predict actual inputs.
- Code test cases that will feed the inputs to the unit.
- Run the tests.
- Verify whether the test inputs yielded the expected results.

xUnit framework tools such as JUnit and CppUnit help speed up this process by providing developers a framework for adding test cases, running tests, and evaluating results. However, these frameworks do not automate the most difficult and time-consuming part of unit testing: test case design and creation. Developers using these frameworks still need to determine what inputs will accurately verify the unit under test, and they still need to write code to test how the unit responds to these inputs. I’ve seen that when unit testing frameworks are first introduced into a development team, the team members become very excited about unit testing and start using the tool to verify every unit that they write. However, after a couple weeks and several dozen test cases, the honeymoon is over. The developers realize that designing and coding unit tests is still a lot of work, and they start writing fewer and fewer tests until they eventually revert to neglecting unit testing altogether.

My research and experience at Parasoft has shown that if you want developers to consistently perform unit testing to immediately verify each unit they write, you need to provide them with a way to fully automate the process. In other words, you need to provide a technology that allows developers to point to a completed unit, click a button, then sit back and let the tool do all the dirty work (test case design and creation, test execution, and result evaluation). The developers can add additional test cases as needed, but all they are *required* to do is click a button to start the test and then fix the problems found.

Designing technology that automates unit test case creation is challenging, but it is indeed possible, and-- because it encourages developers to truly practice unit testing on a day-to-day basis and reap the benefits it offers-- it is well worth the investment. I believe that with sufficient research and time, it is possible to develop technology that automatically designs and writes test cases for units of every language. Basically, automatic test case creation technology needs to read the source code, then determine how to create inputs that will cover most or all of the unit and exercise it with a wide variety of inputs. These test cases verify a unit's construction. Thanks to the advent of new technologies that help developers integrate specification information into the code (for example, Design by Contract), it is also possible to create test cases that verify a unit's functionality.

At this time, Parasoft has automated unit test creation for Java, C/C++, Web services, and Web applications. To give you an idea of how to automate unit test case creation, I will explain how our popular Jtest tool automatically creates Java unit test cases.

Jtest automatically creates construction test cases after examining the structure of the class under test. These test cases exercise the class's methods with a wide range of expected and unexpected inputs to identify inputs that result in uncaught runtime exceptions. When designing test cases, Jtest tries to create tests that execute every possible branch of every method (for example, if the method contains a conditional statement such as an if block, Jtest will generate test cases that test the true and false outcomes of the if statement). If the class under test references an external resource (for example, databases or external files), Jtest automatically generates the necessary stubs, or gives users the option of calling the actual external method or entering their own stubs. For classes using CORBA, Jtest provides stubs for the Object Request Broker and other objects referenced by the class. For classes using EJB, Jtest invokes bean initialization routines and provides a simulated container context, then verifies whether the bean class will behave correctly.

In addition, Jtest automatically creates functionality test cases when the class under test uses Design by Contract (DbC) to express specification information. DbC is a formal way of using comments to incorporate specification information into the code itself. Basically, the code specification is expressed unambiguously using a formal language that describes the code's implicit contracts. These contracts specify such requirements as:

- Conditions that must hold true before a method can execute (preconditions).
- Conditions that must hold true after a method completes (postconditions).
- Conditions that must hold true any time a client can invoke an object's method (invariants).
- Method body assertions that must evaluate to true (assertions).

Jtest reads each class's DbC specification information, then automatically develops test cases based on this specification. Jtest designs its test cases as follows:

- If the code has postcondition contracts, Jtest creates test cases that verify whether the code satisfies those conditions.
- If the code has assertion contracts, Jtest creates test cases that try to make the assertions fail.

- If the code has invariant contracts, Jtest creates test cases that try to make the invariant conditions fail.
- If the code has precondition contracts, Jtest tries to find inputs that force all of the paths in the preconditions.
- If the method under test calls other methods that have specified precondition contracts, Jtest determines whether the method under test can pass non-permissible values to the other methods.
- If any class under test (with or without contracts) calls a class that contains contracts, Jtest determines whether the class under test can interact with the second class in a way that violates the contract.

Automatic unit test case generation technologies such as Jtest significantly improve software reliability and streamline the development process. First, they help developers create the volume and range of test cases that is required to thoroughly exercise the unit under test; this helps developers spot errors and potential weaknesses (for example, instances where the software will fail if it is given unexpected inputs) that might otherwise go unnoticed. Second, they encourage early and frequent unit testing by allowing developers to test a unit with just the click of a button. Study after study has demonstrated that it is easier, faster, and cheaper to fix a problem at the unit level than it is later in the development process. Moreover, when the team tests each unit as soon as it is completed, each team member finds and fixes problems before someone unwittingly introduces additional errors by adding code that builds upon or interacts with the problematic code. In sum, by automating the test case creation process so that developers perform unit testing early, thoroughly, and consistently, you help developers uncover more bugs, find and fix each bug more easily, and minimize the chances of each bug spawning more bugs. The result: more reliable software, as well as a significant reduction in development time and cost.

These days, most developers recognize the value of unit testing and intend to perform comprehensive unit testing as part of their regular development practices. However, unit testing is rarely practiced as early, frequently, or thoroughly as it should be because most developers simply do not have the resources or desire to manually write test case after test case. This problem can be solved by automating the test case creation process. When this process is automated, unit testing becomes so easy and efficient that developers can realistically integrate it into their daily development practices and reap the benefits it has to offer.

Author: **Andrej Kline**
Current location: FJA OdaTeam company, Slovenia (Europe)
Business value: Insurance software developing and delivering
Current team size: 17
Role: Coaching and playing the proxy customer
In “free-time”: Managing projects, in business and in real life

My position on:

Expanding the Boundaries of Unit Testing

My experience with Unit testing

Me and my team-colleagues starting using it approximately 3 years ago, as a principle (or technique) of XP. We have actually been using it before, occasionally, not giving it a proper place nor the persistence in our development “style”.
The greatest change-in-mind, that the Unit testing triggered in our development team, were the following values of the test:

- Test is the first usage of the object (unit in general)
- Test is documenting a system
- Test shows what is to be done, not how
- Tests enforce simple design
- Difficulties in writing tests reveal the problems in code (the system) itself

Though we differ between unit tests and acceptance tests, for the matter of discussion, I’ll refer to each test as a unit test. But – there again – what are the **boundaries** of a unit test, what are the boundaries of an acceptance test?

What is to be expanded?

Unit testing (along with acceptance testing) is only a framework in the sense of collecting the “cases” that the system - being developed and deployed - can trap into. It’s up to every SW developing team to set the boundaries flexible, extensible and scalable. These imaginary **boundaries should move through time** with the conscious of the development team.

Here are the situations we have lived through until now and the way we coped with them:

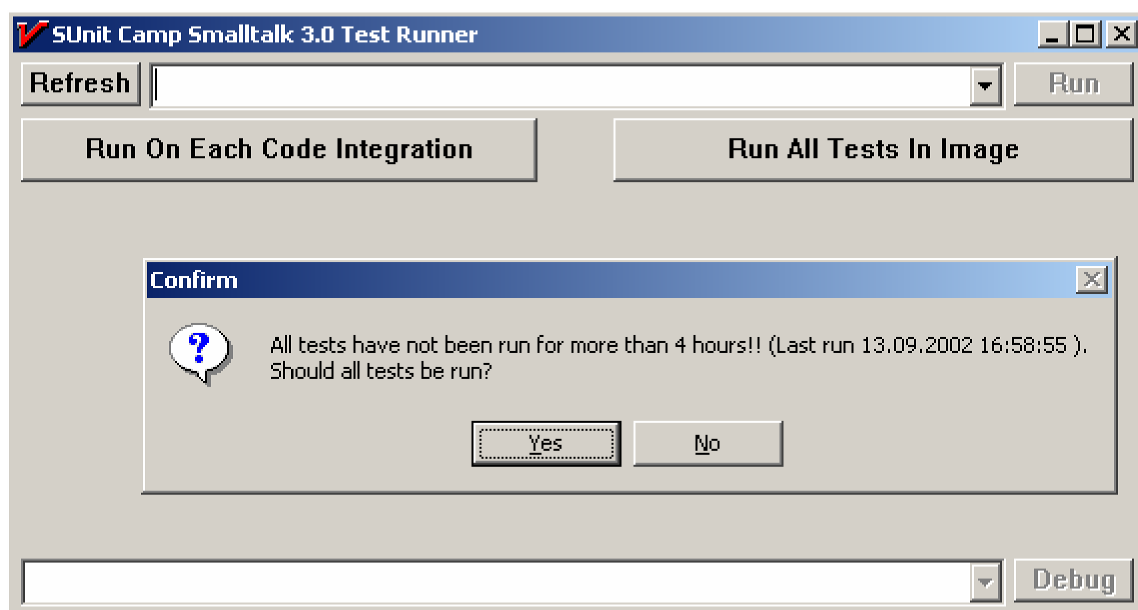
Testing a legacy system

When starting to write automatic tests about 3 years ago, our system already had about 3.000 classes and was deployed to 6 insurance companies. There were quite some tests existing, but none of them was neither repeatable nor automated. They were all performed manually.
Today we have about 2.500 tests, which cover approximately 1/3 of the deployed system. If we hadn’t decided to start **rigorous testing** at that time, we wouldn’t have had so much of the system tested. We know we have a long way to go, before we’ll have all units tested, but we are getting more and more confident and courage from day to day, after each test we write.
Because we have only partially test-covered system, there still exists a need of performing heavy and time consuming manual tests before deploying any update to our system. So we

keep rewarding ourselves with **beer-sessions after each 500 tests** written. The good thing is that the more we test the sooner the beer session comes around ☺.
How do we fill-into the beer-fund? Every developer, who does not run all the tests after each code integration, is "caught" and must contribute.

Automating tests

For making the process of running all the tests as easy as possible, we involved all currently supported xUnit tools, as described in the following chapters.
What I want to point-out, we customized those tools to fit our needs. For example, we "tailored the Smalltalk test unit runner for our – still used Visual Smalltalk – environment, as shown in the picture below. We can choose any particular test or select test suite(s) to run, due to what we want to test. The Runner is accessible globally, from the Smalltalk Transcript window.



We even don't trust ourselves that much, when was the last time we run all the tests. Therefore we are measuring time between test runs. If the warning presented appears the supporter developer is obligated to run all of the tests.

Separating Unit tests from Acceptance tests

This separation is based on the time spent to run all of 2.5000 automatic tests. We split the tests into two groups (as shown in the picture):

- Tests to be run on each code integration
- All tests

The primer is the sub-set of the letter. The sub-set includes mostly the unit tests. The delta are all the acceptance tests, which involve heavy set-up of test resources like several testing databases etc.
What's the cause of doing this? All tests run on the integration computer takes about 10 minutes. Running only the small set takes 3 minutes. So we gain time to test and integrate, but – as written in the picture – **at least twice a working day ALL tests must be run**.

Deploying tests into Runtime environment

This decision was made after we discovered that the testing environment we have established in our company does not fit the one our customers are using. And it is not the configuration differences that caused problems. It is the image we develop in and the one we deploy to our customers. They were different in the sense of the set-up of all testing resources.
So we solved this problem by deploying also all the automatic tests into runtime environment. Now we run all tests in development environment after each code integration and in runtime environment at least **before packing any update**.
Only by performing the "runtime" tests we can be sure that our test reach the goal not only within our development group, but also at our customers.

Managing tests in diverse development environments

Porting our stand-alone application to web, we touched the problem of having the source code in different development environments and/or platforms. Our system encapsulates the following layers:

1. Database server - our own implementation, in Java
2. Business server - in Smalltalk
3. Application dialog server - in Java
4. Web client - JSP's (HTML)

To test each layer properly, we write tests with the help of **SUnit**, **JUnit** and **httpUnit**.
So what we do is collecting the tests from different development environments. We run tests is each environment separately, not being able to run them together, on "one mouse click", to test the **synergy of all the layers** mentioned.

My current concern

The system we are developing and maintaining, is growing. It has become diverse in technological view of the development environments we use. Despite that, the tests must cover the whole system, peace by peace, unit after unit, in a **central-manageable testing framework**.

With these positions I end my test of getting you attracted for a further discussion.
Until then, I wish you all a lot of "green" tests.
Andrej

Extending SUnit to Test SASE Events

Position Paper for OOPSLA 2002 Workshop:
"Expanding the Boundaries
of Unit Testing"

Blair McGlashan, Andy Bower
Object Arts Ltd

Introduction

In this paper we discuss our enhancement to SUnit unit testing framework to extend its coverage to the testing of Self-Addressed Stamped Envelope (SASE) events [Brown95].

Although not included in the ANSI Smalltalk standard, SASE event frameworks have become a de-facto standard feature of modern Smalltalk systems. SASE is a form of the Observer pattern [Gamma95] in which a subscriber requests of a specific publisher that it should notify that subscriber by sending it a specific message when an individual event occurs. This differs from the older dependency mechanism of Smalltalk-80, in which observers could not selectively subscribe to individual events, but instead received notifications of all events published, through a single callback. Both SASE and dependency mechanisms facilitate the construction of loosely coupled system in which the publishers of events need have no prior knowledge of the subscribers, and can have any number of the latter. The SASE mechanism has a number of advantages (and some disadvantages) over the dependency mechanism, which we will not go into here, and has largely superseded it, even in VisualWorks.

SASE implementations differ, but the protocols used are the same, probably being based on the implementation in Digitalk Smalltalk, which we assume to be the original. SASE has a default implementation in Object (the root of the Smalltalk class hierarchy), and consequently any Object may be a publisher or subscriber. A subscription request is made by the subscriber sending the publisher a message of the form "when X send me Y [with arguments]". There are a number of versions of this message implemented in Object, the most commonly used form of which *#when:send:to:*. For example:

```
self model when: #itemUpdated: send: #onItemUpdated: to: self
```

Events can have parameters, as in the above example where the *#itemUpdated:* event is accompanied by the item that has been updated. It is also possible to register "static" arguments at the time of making a subscription request that are sent back with the event notification. This can be useful for closure purposes (for example when handling multiple events in one handler), or to supply default values for the event where the handler expects more arguments than the publisher supplies.

In order to publish an event, an object sends itself one of the *#trigger:[with:…]* family of messages. The first argument is the symbolic name of the event, which must match that used by the subscribers, and subsequent arguments are the parameters to accompany the event. For example:

```
self trigger: #itemUpdated: with: anObject
```

If there are any subscribers to the event, then they will each be notified of the event, in no particular order, by receiving the callback message they previously registered with *#when:send:to:*.

SASE is a very powerful and general mechanism that facilitates the construction of dynamic, loosely coupled systems. It has particular applicability in UI frameworks, such as Dolphin Smalltalk's Model-View-Presenter [Bower00], but can be used in many other situations where Observer is appropriate.

In practice one finds that SASE is extensively used in modern Smalltalks. For example in our current Dolphin development image containing approximately 1800 classes, there are about 100 publishers, and about 140 subscribers. 30 classes are both publisher and consumers of events. Despite this SUnit currently provides no mechanism for testing that the right set of events is published at the right time, and this undoubtedly contributes to its weakness in testing UI components[1]. We propose a simple and

[1] As Smalltalk implementers we also needed a mechanism to test the event framework itself!

portable extension to SUnit that allows one to assert that an operation against a publisher causes it to publish a particular set of events matching certain criteria, and (optionally) that it only publishes those events.

Testing Events

In testing events we want to be able to supply a block of code and:

1. Assert that a specified object (the publisher) raises particular events when that block of code is evaluated.
2. Optionally we may want to examine the events to see that they are correctly formed. This means we need to be able to assert on the event arguments.
3. Optionally we want to be able to deny that the publisher will raise other events during the operation.

Testing that the right events are fired at the right time has similar requirements to being able to test that the right exceptions are raised at the right time, and so we can follow a similar pattern as that used to test exceptions in SUnit. The basic methods for exception testing in SUnit is *TestCase>>should:raise:* and *TestCase>>shouldnt:raise:*[2]. The first argument of each message is the block of code that should (or shouldn't) raise an exception of the class specified as the second argument. For example we might test that Array bounds checking is working as follows:

```
self should: [#('a') at: 2] raise: BoundsError.
self shouldnt: [#('a') at: 1] raise: BoundsError.
```

Applying this pattern for testing events we might want to be able to write in a *TestCase*:

```
self
    should: [publisher  trigger: #testNoArgs]
    trigger: #testNoArgs
    against: publisher
```

This satisfies our basic requirement, but it is often important that the correct arguments accompany the event. In Dolphin we ship a useful extension to *#should:raise:* which allows one to pass a discriminator block to verify not only that an exception is raised, but that it matches certain criteria. For example:

```
self
    should: [#('a') at: 2]
    raise: BoundsError
    matching: [:ex | ex tag = 2]
```

We can apply this approach to test that raised events are correctly formed too:

```
self
    should: [publisher trigger: #testOneArg: with: 1]
    trigger: #testOneArg:
    matching: [:arg | arg = 1]
    against: publisher
```

When testing exceptions in SUnit one must test separately for each individual exception that it is either raised, or not raised. This is generally satisfactory since one can often ensure that the test operations are idempotent, and so the tests can be regarded as independent. In the case of events this is often not possible, especially when testing UI components, and we often need to test atomically that one set of events is raised to the exclusion of another set. For example we might be testing a ListBox widget that is specified to trigger both *#selectionChanging:* and *#selectionChanged* notifications when the user changes the selection with mouse or keyboard. The selection-changing event gives the program the opportunity to reject the users selection change request, perhaps following a prompt to save changes.

---

[2] In Dolphin we ship a useful extension to *#should:raise*: which allows one to pass a discriminator block to verify not only that an exception is raised, but that it matches certain criteria. For example:

```
self should: [#('a' at: 2] raise: BoundsError matching: [:ex | ex tag = 2].
```

However the selection-changing event may not be expected in response to a programmatic change that causes a mandatory change of selection (e.g. the programmatic deletion of the selected item). In such cases we need to be able to perform the "should trigger" and "shouldn't triggers" tests in one, for example:

```
presenter selection: item.
self
    should: [presenter model remove: item]
    trigger: #selectionChanged
    butNot: #selectionChanging:
    against: presenter.
```

Combining all this together, the most general form we need is:

```
should: operation
    triggerAllOf: expectedEvents
    matching: discriminator
    butNoneOf: unexpectedEvents
    against: publisher
```

Where:

- *operation* is a zero argument block that defines the operation being tested
- *expectedEvents* is a *Collection* of *Symbols*, being the names of the events that the operation should trigger.
- *discriminator* is a one argument block that is passed the *Message* [3] formed from the subscribers event registration and the arguments supplied by the publisher when it triggered the event
- *unexpectedEvents* is a *Collection* of *Symbols*, being the symbolic names of the events that the operation should not trigger.
- *publisher* is the *Object* expected to trigger the event.

The Implementation

Our extension is implemented entirely by the addition of a few new methods to the *TestCase* class. As mentioned the most general form is:

```
#should:triggerAllOf:matching:butNoneOf:against:
```

Various shorter, more convenient, forms can be implemented simply by layering on top of this (see Convenience Messages below).

In order to be able to test that an object triggers a particular set of events (and not some others) when the operation is evaluated, we need to register to receive each of these events through the normal subscription mechanism, that is using one of the *#when:send:to:* family of messages. We also need to tear down the subscriptions when the test operation has finished to leave the object in the same state (with respect to observers) that it was before the test started. The easiest way to do this is to save the subscription state on entry, and restore it on exit.

The main difficulty in providing a generic implementation is that the events may take any number of arguments. Although the event subsystem allows the number of arguments expected by the subscriber to be fewer than the number published (with arguments being dropped from the right accordingly), we need to be able to receive all the arguments in order to be able to perform tests against them in the discriminator block. If we vector all event notifications through a single method, or statically defined set of methods, then we must at some point limit the maximum number of arguments. We can in fact implement this simply and generically in Smalltalk by making use of the *#doesNotUnderstand:* mechanism.

---

[3] A *Message* is the reified form of a polymorphic function call, specifying the selector (symbolic function name), and an array of the objects passed as arguments by the sender (caller).

Since Smalltalk is a dynamically typed language, checks to see whether an object implements a particular method are deferred until runtime. Should an object not implement a method to match a message it is sent, then the virtual machine detects this case and reifies the message into a *Message* object and invokes instead the objects implementation of *#doesNotUnderstand:*, passing the original *Message* as its argument.

The *#doesNotUnderstand:* mechanism is useful for the implementation of generic proxies, and similar forwarders, but we can also use it to detect the events that an object triggers by registering have all those events sent to an object as messages which it will not understand. In an ANSI Smalltalk environment the default implementation of *#doesNotUnderstand:* will raise a *MessageNotUnderstood* exception. We can catch these exceptions, examine them to determine the event that was triggered, and resume execution[4].

Putting all this together our current implementation is:

```
TestCase>>should: aZeroArgBlock triggerAllOf: aCollectionOfSymbols matching: aOneArgBlock butNoneOf:
aCollectionOfSymbols2 against: anObject
    | fired allEvents |
    fired := Bag new.
    allEvents := aCollectionOfSymbols union: aCollectionOfSymbols2.
    "The expected and unexpected event sets should be disjoint"
    self assert: allEvents size = (aCollectionOfSymbols size + aCollectionOfSymbols2 size).
    [allEvents do:
            [:each |
            "If the event selector is part of nil's protocol, then no MNU will be raised,
            and the test will be invalid"
            self deny: (nil respondsTo: each).
            anObject when: each sendTo: nil].
    aZeroArgBlock on: MessageNotUnderstood
        do:
            [:mnu |
            (mnu receiver ~~ nil or: [(allEvents includes: mnu selector) not]) ifTrue: [mnu pass].
            ((aCollectionOfSymbols2 includes: mnu selector) or: [aOneArgBlock value: mnu message])
                ifTrue: [fired add: mnu selector].
            mnu resume: mnu receiver]]
            ensure: [anObject removeEventsTriggeredFor: nil].
    "If this assertion fails, then the object did not trigger one or more of the expected events"
    self assert: (aCollectionOfSymbols difference: fired) isEmpty.
    "If this assertion fails, then the object triggered one or more events that it should not have"
    self assert: (aCollectionOfSymbols2 intersection: fired) isEmpty
```

Some points to note:

- We subscribe for all of the events, both expected and unexpected, on behalf of the undefined object (*nil*) since this is the simplest approach. *nil* is the nearest thing to a 'null' pointer in Smalltalk, but it is still derived from Object, and may therefore implement quite a substantial interface. In Dolphin Smalltalk *nil* responds to approximately 150 messages, and in some Smalltalks this may run into the several hundred. Since we register to receive messages with the same selector as the associated event, it is possible that an event name may correspond with a message that *nil* does understand. We include an assertion so that the test will fail should this occur, but this is still not entirely satisfactory. Alternative approaches would be to have the messages sent to a proto-object[5], rather than nil, but this would not be portable, or we could construct selectors that we could guarantee would not be understood. The latter would increase complexity, and in practice we have not found name clashes to be a problem.

---

[4] Smalltalk exceptions are quite unusual in that when exception handlers are executed the stack has not yet been unwound. This is in contrast to C++ and Java, but is a more powerful model as it allows one to support resumable exceptions.

[5] By "proto-object" we mean an object of a class not derived from *Object,* which therefore implements only the very minimal protocol required by the environment for a thing to be a valid object.

- We are careful to ignore *MessageNotUnderstood* exceptions that do not relate to the absence of event handlers, since we don't want to suppress other errors.
- The implementation uses certain non-standard set-theoretic Collection messages (*#union:*, *#difference:*, *#intersection:*). These could be included in ports to other Smalltalks, or replaced with more long-winded statements using the standard messages.

Convenience Messages

We hardly ever use the most general form of test, but instead use one of the following convenience messages we have also added to the *TestCase* class:

```
should: operation trigger: eventName against: publisher
should: operation trigger: eventName1 butNot: eventName2 against: publisher
should: operation trigger: eventName matching: discriminator against: publisher6
should: operation triggerAllOf: eventNames against: anObject
should: operation triggerAllOf: eventNames matching: discriminator against: publisher
shouldnt: operation trigger: eventName against: publisher
shouldnt: operaiont triggerAnyOf: eventNames against: publisher
```

The full implementation of these is included in the appendix.

We use all of these in our tests, except *#should:triggerAllOf:matching:against:*, which is probably not needed.

Conclusion

SASE events are a de-facto standard facility in modern Smalltalks, and increasingly widely used to construct dynamic, loosely coupled systems, especially in UI frameworks. SUnit currently has no built-in facilities for testing events, and therefore this important part of an object's interface goes untested. In order to adequately test UI components we have added some extension methods to the *TestCase* class that provide a similar (though more sophisticated) mechanism for testing events as SUnit already provides for testing exceptions. The basic implementation of this exception is relatively simple, and runs to a few tens of lines of code.

Using this extension we have been able to employ SUnit to test UI components in our MVP user-interface framework. Although we still cannot test that the visual appearance is correct, we are able to test that the components behave correctly in terms of both their internal state (which we could already do), and that they generate the right events at the right time (which we could not).

We believe this extension to be an essential addition to SUnit, and that since it should be relatively easy to port it to other dialects, that it should be adopted into the standard SUnit distribution.

References

[Bower00] "Twisting the Triad, The Evolution of the Dolphin Smalltalk MVP Framework", Andy Bower and Blair McGlashan, Object Arts Ltd, (http://www.object-arts.com/Papers/TwistingTheTriad.PDF)

[Brown95] Understanding inter-layer communication with the SASE pattern", Kyle Brown, Smalltalk Report, November-December 1995 (http://members.aol.com/kgb1001001/Articles/SASE/sase2.htm)

[Gamma95] "Design Patterns: Elements of Reusable Object-Oriented Software", Gamma, Helm, Johnson and Vlissides, Addison-Wesley, 1995

[6] In the case of *#should:trigger:matching:against:* the discriminator block is not passed the entire *Message*, but each of the individual arguments to the event (i.e. the block must have the same number of arguments as the event). This simplifies writing the discriminator block for this common case of testing a single event's arguments.

Appendix: Complete Implementation

All methods extend *TestCase*.

```
should: aZeroArgBlock trigger: aSymbol against: anObject
    self
        should: aZeroArgBlock
        triggerAllOf: (Array with: aSymbol)
        against: anObject

should: aBlock trigger: eventSymbol1 butNot: eventSymbol2 against: anObject
    self
        should: aBlock
        triggerAllOf: (Array with: eventSymbol1)
        matching: [:message | true]
        butNoneOf: (Array with: eventSymbol2)
        against: anObject

should: aZeroArgBlock trigger: aSymbol matching: discriminatorBlock against: anObject
    self assert: aSymbol argumentCount = discriminatorBlock argumentCount.
    self
        should: aZeroArgBlock
        triggerAllOf: (Array with: aSymbol)
        matching: [:message | discriminatorBlock valueWithArguments: message arguments]
        against: anObject

should: aZeroArgBlock triggerAllOf: aCollectionOfSymbols against: anObject
    self
        should: aZeroArgBlock
        triggerAllOf: aCollectionOfSymbols
        matching: [:message | true]
        against: anObject

should: aZeroArgBlock triggerAllOf: aCollectionOfSymbols matching: aOneArgBlockOrNil against: anObject
    self
        should: aZeroArgBlock
        triggerAllOf: aCollectionOfSymbols
        matching: aOneArgBlockOrNil
        butNoneOf: #()
        against: anObject

should: aZeroArgBlock triggerAllOf: aCollectionOfSymbols matching: aOneArgBlock butNoneOf: aCollectionOfSymbols2
against: anObject
    | fired allEvents |
    fired := Bag new.
    allEvents := aCollectionOfSymbols union: aCollectionOfSymbols2.
    "The expected and unexpected event sets should be disjoint"
    self assert: allEvents size = (aCollectionOfSymbols size + aCollectionOfSymbols2 size).
    [allEvents do:
            [:each |
            "If the event selector is part of nil's protocol, then no MNU will be raised,
            and the test will be invalid"
            self deny: (nil respondsTo: each).
            anObject when: each sendTo: nil].
    aZeroArgBlock on: MessageNotUnderstood
        do:
            [:mnu |
            (mnu receiver ~~ nil or: [(allEvents includes: mnu selector) not]) ifTrue: [mnu pass].
            ((aCollectionOfSymbols2 includes: mnu selector) or: [aOneArgBlock value: mnu message])
                ifTrue: [fired add: mnu selector].
            mnu resume: mnu receiver]]
```

```
            ensure: [anObject removeEventsTriggeredFor: nil].
    "If this assertion fails, then the object did not trigger one or more of the expected events"
    self assert: (aCollectionOfSymbols difference: fired) isEmpty.
    "If this assertion fails, then the object triggered one or more events that it should not have"
    self assert: (aCollectionOfSymbols2 intersection: fired) isEmpty

  shouldnt: aBlock trigger: aSymbol against: anObject
    self
        shouldnt: aBlock
        triggerAnyOf: (Array with: aSymbol)
        against: anObject

shouldnt: aZeroArgBlock triggerAnyOf: aCollectionOfSymbols against: anObject
    self
        shouldnt: aZeroArgBlock
        triggerAllOf: #()
        matching: [:message | true]
        butNoneOf: aCollectionOfSymbols
        against: anObject
```

Implementing and Using Resumable TestFailures in Smalltalk

Joseph Pelrine
MetaProg GmbH

Position paper for
Expanding the Boundaries of Unit Testing
OOPSLA 2002, Seattle, WA

The high performance aspect of extreme Programming derives in part from the rapid feedback cycles in unit testing. Collection testing and validation, however, can be very time-intensive, and can slow down the development process to the point where the advantages of test-driven programming are lost. Through the implementation of ”resumable” test failures, though, this deficit can be compensated for. The ResumableTestFailure (to be introduced in SUnit 3.1) offers a flexible implementation of this in Smalltalk.

The new SUnit release 3.1 adds more functionality at little cost to both Smalltalk’s and extreme Programming’s premier testing framework. In addition to the #assert:description: family of methods (well-known from JUnit), which allow you to attach arbitrary description strings to assertions, and the implementation of basic logging facilities, the major change is the introduction of a resumable TestFailure.

Why would you need a resumable TestFailure? Take a look at this example from a typical test case method:

```
aCollection do: [ :each | self assert: each isFoo]
```

In this case, as soon as the first element of the collection isn’t Foo, the test stops and returns a failure. Although this information is necessary for test-driven development, it normally isn’t sufficient. In most cases, we would like to continue, and see both how many elements and which elements aren’t Foo. It would also be nice to log this information. You can do this in this way:

```
aCollection do: [ :each |
      self
            assert: each isFoo
            description: each printString, 'is not Foo'
            resumable: true]
```

This will print out a message on the Transcript for each element that fails. It doesn't cumulate failures, i.e., if the assertion fail 10 times in your test method, you'll still only see one failure.

Implementation

As a result of SUnit being extremely lightweight, it required only minimal effort to implement the functionality required to support ResumableTestFailures.

1. The class ResumableTestFailure was created as a subclass of TestFailure, which itself is defined in the SUnitPreload package. (This package contains all dialect-specific Classes and Methods for SUnit, and makes it possible for the core SUnit package to be dialect-independent).
2. The method Exception>>#isResumable was overwritten to return **true**.
3. The method Exception>>#sunitExitWith: , which normally returns from the exception, was overwritten to resume execution.

While running the test cases, it was noticed that the SUnit framework had a conceptual inconsistency which was overlooked in the original implementation. The method TestResult>>#failures, which returns the collection of failures for a test run, was implemented to be an OrderedCollection. This led to each triggering of a ResumableTestFailure adding yet another failure to the collection. The implementation was changed to be a Set, based on the fact that a test case method is a failure regardless of how many assertions in the method are false. Also, implementing the failure collection as a Set reflects the fact that test cases should be non-deterministic, i.e., the order in which the test cases are executed is irrelevant.

The change in TestResult>>#failures led to a slight change in TestResult>>#defects, which was dependent on the failures being contained in an OrderedCollection. This change was minor, and will not be discussed further.

The implementation also required a method for triggering both regular and resumable TestFailures. The basic method, TestCase>>#assert:description:resumable: is illustrated below:

```
assert: aBoolean description: aString resumable:
resumableBoolean

      | exception |
      aBoolean ifFalse: [
            self logFailure: aString.
            exception := resumableBoolean
                  ifTrue: [ResumableTestFailure]
                  ifFalse: [TestResult failure].
            exception sunitSignalWith: aString]
```

Once again, the implementation of SUnit has proven to be very efficient and flexible when it comes to adding or extending behavior without changing the base packages. Of course, being in Smalltalk helps too – YMMV.

Joseph Pelrine wrote the reference implementation of SUnit 3.0. He is (together with Jeff Odell) currently maintainer of the Camp Smalltalk SUnit distribution on Sourceforge, and has just released SUnit 3.1 to the world.

He can be reached at:

Joseph Pelrine
MetaProg GmbH
Bachlettenstrasse 41
CH-4054 Basel
Switzerland
Email: jpelrine@metaprog.com

Single-Source Unit Testing for Applications that Ship on Java and .NET

David Vydra
Tools Team
Plumtree Software
david@vydra.net

Position paper for
Expanding the Boundaries of Unit Testing
OOPSLA 2002, Seattle, WA

Abstract

Two of the fasted growing software platforms today are Java and .NET. Many companies are not taking sides and are developing products for both platforms. The cost of writing and maintaining unit tests separately for each platform can be substantial. We present an approach for writing and executing java unit tests (JUnit [1]) on .NET by using the J# compiler from Microsoft.

Acquiring the Tools

The choice of J# can be troubling at first, since it only claims jdk 1.1.4 compatibility and most development today is done with jdk 1.3 and above, but upon closer investigation the situation is not that bleak. There are several mitigating features: J# ships with Java 1.2 collections framework and it has a nifty feature that allows you to override most of the core classes implementations (for example we were able to add java.util.Properties implementation from the jdk 1.3 source*). Of course your JUnit tests compiled with J# will probably not link with your .NET code since in all likelyhood it will use .NET specific types. This can be remedied by writing a wrapper that adapts java tests for .NET types. This wrapper can either be written manually or dynamically generated during the build process.

We wanted to run .NET tests in a GUI runner and from Ant. To build these tools with J# infrastructure we had to modify the JUnit core framework, AWT runner, and JUnit Ant tasks. An open-source port of JUnit AWT TestRunner is available at http://vydra.net/jsharp-unit.

Proving the Approach

To test our approach we wrote a wrapper for the SampleMoney example that ships with both JUnit and NUnit. This wrapper allows JUnit tests to run against the C# implementation of SampleMoney that ships with NUnit 1.0. See Appendix for code samples.

Lessons Learned

When developing our internal APIs we found it convenient to write a foundation layer that abstracts the differences between Java and .NET core classes. All platform specific client code and core unit tests were written against that API.

When writing a testing wrapper for our external APIs we learned to be careful about the semantics of collections used. It is important to choose Java and .NET collections that offer the same logical contract.

When testing the HTML UI and our custom http-based protocol we used Http Request/Response mock objects. Unfortunately .NET HttpRequest/Response classes do not implement an interface, as in java, and are also 'sealed' (not subclassable). To overcome this deficiency we created mock objects that wrappered the 'real' HttpRequest/Response objects and modified their state via reflection.

Conclusion

The approach presented here can substantially reduce the cardinal ‘smell’ of duplicate code [2] in your unit tests for cross-platform development. It may not work for all tests, but if it fits, it will pay off handsomely.

* Assumes a clean room implementation that does not violate Sun’s copyright.

Appendix

Java to .NET wrapper of SampleMoney

```
package junit.samples.money;
public class Money implements IMoney
{
      NUnit.Samples.Money.Money _money;
      public NUnit.Samples.Money.IMoney getAdaptee()
      {
                  return _money;
      };
      public Money(int amount, String currency)
      {
            _money = new NUnit.Samples.Money.Money(amount,currency);
      }
      Money(NUnit.Samples.Money.Money money)
      {
            _money = money;
      }

      public IMoney add(IMoney m)
      {
            return
MoneyAdapterHelper.adapt(_money.Add(m.getAdaptee()));
      }
      public IMoney addMoney(Money m)
```

```
	{
		return MoneyAdapterHelper.adapt(_money.AddMoney(m._money));
	}
	public IMoney addMoneyBag(MoneyBag s)
	{
		return
MoneyAdapterHelper.adapt(_money.AddMoneyBag(s._moneyBag));
	}
	public int amount()
	{
		return _money.get_Amount();
	}
	public String currency()
	{
		return _money.get_Currency();
	}
	public boolean equals(Object obj)
	{
		if(obj instanceof IMoney)
		{
			IMoney m = (IMoney)obj;
			return _money.Equals(m.getAdaptee());
		}
		return super.equals(obj);
	}
	public int hashCode()
	{
		return _money.GetHashCode();
	}
	public boolean isZero()
	{
		return _money.get_IsZero();
	}
	public IMoney multiply(int factor)
	{
		return MoneyAdapterHelper.adapt(_money.Multiply(factor));
	}
	public IMoney negate()
	{
		return MoneyAdapterHelper.adapt(_money.Negate());
	}
	public IMoney subtract(IMoney m)
	{
		return
MoneyAdapterHelper.adapt(_money.Subtract(m.getAdaptee()));
	}
	public String toString()
	{
		return _money.ToString();
	}

}

package junit.samples.money;
public interface IMoney {
	public abstract NUnit.Samples.Money.IMoney getAdaptee();
	public abstract IMoney add(IMoney m);
```

```
	IMoney addMoney(Money m);
	IMoney addMoneyBag(MoneyBag s);
	public abstract boolean isZero();
	public abstract IMoney multiply(int factor);
	public abstract IMoney negate();
	public abstract IMoney subtract(IMoney m);
}

public class MoneyAdapterHelper
{
	static public IMoney adapt(NUnit.Samples.Money.IMoney m)
	{
		if(m instanceof NUnit.Samples.Money.Money)
			return new Money( (NUnit.Samples.Money.Money)m);
		else if(m instanceof NUnit.Samples.Money.MoneyBag)
			return new MoneyBag((NUnit.Samples.Money.MoneyBag)m);
		else
			throw new IllegalArgumentException("Has to be Money
or MoneyBag");
	}
}
```

References

[1] Beck, K., Gamma, E., The JUnit Testing Framework v3.7, www.junit.org
[2] Fowler, M.,Refactoring: Improving the Design of Existing Code, Addison-Wesley, 1999

Testing, Measuring, Pairing, Feedback: Variations of a Concept

Hans Wegener

Swiss Re, Mythenquai 50/60, 8022 Zürich, Switzerland

Phone: +41 (43) 285 27 28, Fax: +41 (43) 282 27 28, E-Mail: Hans_Wegener@swissre.com

1. Introduction

When Extreme Programming (XP) saw the light of day [2], testing was perceived as the task of two groups: programmers and customers; its primary focus was testing the software continuously, be it individual units or functional acceptance. Others such as stress tests were considered reasonable only from time to time.

The Agile Manifesto [1] makes no explicit reference to testing. However, it states that working software is the primary measure of progress, i.e. that compliance with specific criteria is indeed measured. Customers and developers work together to agree on the requirements and how they be tested for compliance.

In his book on agile development, Cockburn [3] lists methods of jumping communication gaps. To him, synchronicity and lack of latency are contributors to more effective and efficient communication. The validation of requirements is based on finding mutual understanding.

Improving the ways to form consensus on requirements is based on the artifact in vivo, i.e. the concrete product in use by customers. Validation is a process whose purpose is an intervention of the user into a solution proposal [4]. It is a constructive activity with the aim to improve the appropriateness of both the system attributes and the user requirements.

How do all these concepts fit together? What can be their rightful place in agile development? This position paper is intended to clarify the relationships between them, and to show how they can be used together to improve results in a software project.

2. Experiences

In a recent project (2000-2002), we developed a web-based search, navigation, and administration frontend to data warehouse metadata. As the team lead, I decided that unit testing should become mandatory. Furthermore, the tests should be run as part of software integration, i.e. before a CVS commit. The idea was that no one should be able to commit erroneous code. We practiced things like those outlined in [5]:

- unit tests,
- functional tests,
- conversational tests,
- mock objects,
- setup tests, or
- worst and average case performance tests.

A very specific problem in our project was that I was engaged in several other tasks. Hence, I often lacked the necessary time to work together with the team, and the problem could not be solved organizationally. The internal quality of the software was at stake, so we had to find a way to get automated feedback about the quality of the code. Specifically, this meant measuring things such as these (tools used in brackets):

- test code coverage (JProbe),
- code complexity (homegrown solution, JavaNCSS),
- dependency constraints (homegrown solution, iDarwin), or
- completeness of documentation for public and protected methods (homegrown solution, iDoc).

Finally, our customers were dispersed across the entire city, and they had very different backgrounds. No single customer was on site, and of course, they didn't speak with one voice. Naturally, it was difficult to get unanimous,

instant feedback. We helped ourselves by discussing critical issues with them via email and formulating tests and visual design by using a best-guess strategy. Only once a month did we get to actually present the running software to them, at which occasion they also uttered requests for improvement.

The results were mixed, and the reasons were diverse. First of all, the original idea was to control the results people produced. That wasn't such a bad idea in itself. For example, JProbe allows a very specific analysis of the code covered by tests. In many cases I was able to discover code that hadn't been tested at all. However, it is reasonable to only test those pieces of software that are non-trivial. Since we had quite many getter and setter methods, this caused effort that didn't increase quality in any substantial way. Also, our version of JProbe proved to have an unnerving bug, which meant that one had to restart the program each time the test was run. If that didn't happen, code coverage was printed correctly, but the source code was marked incorrectly (i.e., some lines that had actually been covered were shown as uncovered). Lastly, when one colleague took over code written by someone else, the statement coverage was comparatively low. Since I demanded high coverage, this caused frustration, because this person had to fix what another had done wrong. We finally agreed to tolerate the lower coverage temporarily and slowly increased it. Additional time was allocated for that work.

Testing the completeness of the code documentation was another hurdle, because iDoc cannot cope with the complexity of life. For example, Javadoc is able to copy the documentation of an inherited method to an overwriting method, while iDoc is incapable of recognizing this alleviating factor. Complex additional scripting was necessary to cope with this behavior, and again, tool bugs like problems with inner class documentation. On the other hand, iDoc proved very helpful in identifying missing documentation. We were also able to adapt our coding styleguide such that less documentation was necessary (e.g., by putting the tests into the same package as the tested code, leading to less public and protected methods). In total, there was some value in using the tool, but the overall value was much less than expected.

As a third example, dependency constraint enforcement was a very successful tool and helped to simplify the design substantially. Our software was distributed over several tiers, and referencing the wrong packages meant unnecessary code bloat, because they dragged a host of other code with them. In fact, one reason why we started checking dependencies was the fact that our deployment process had become unstable and ensuing software crashes ("class not found") difficult to debug. The impact of not catching constraint violations was so big that it became inevitable to enforce them. Also, it was possible to check the constraints comparatively easy and precisely.

Finally, code complexity measurement proved to be a double-edged sword, too. (We measured lines per class, lines per method, and methods per class.) On the positive side, overly long methods or classes could be easily spotted. Most of the time that happened, an extract-method refactoring was highly recommended. In this regard, the method worked very well. However, in some cases it was not possible to refactor the code so that it would become simpler (in the terms of our measures), such as when an anonymous inner class with more than one simple method was created within another method. In these cases, we refrained to maintaining exception lists, which listed the places that were deemed ok, even though they didn't stick to the standard. This worked fine, although there was an additional effort.

3. Sources, Types, and Goals of Feedback

For practical purposes we distinguish different types of feedback. *Immediate feedback* is given within seconds of an error, *delayed feedback* is given at the end of the problem. *Minimal feedback* just states whether one did right or wrong on a task; *non-minimal feedback* provides hints to goals and rules or completion levels. Van Lehn [6] cites experiences from tutoring, suggesting that students learn faster and more from immediate, non-minimal feedback than from delayed, minimal feedback.

Whether feedback can be immediate or must be delayed depends on the lowest possible *latency*. It is sometimes difficult or expensive to gather feedback immediately, for example when the customer is not on site or when the software only runs in an embedded system. Practices like *colocation* help to reduce latency, as does removing modalities [3]. Finally, feedback can be *automated* or not.

3.1 What Kind of Feedback You Get Where

Unit tests provide minimal feedback, because the result can be either ok or not ok. The rule is that all unit tests should be 100% ok. Unit tests are used as immediate feedback, because developers are requested to code them before the actual implementation and run them immediately after implementation is completed. By the same token, acceptance tests can be considered minimal, immediate feedback as well. Stability tests are a form of delayed, minimal feedback, because they usually require elaborate setups, such as dedicated hardware and time windows. Unit and stability tests are automated.

Measurements provide non-minimal feedback; they state where the solution is located along a dimension continuum. Developers interpret the figure in the light of their experience and the problem at hand to decide over their future conduct. Depending on the complexity of the measures the feedback may be more or less delayed. Usually they are delayed because of the demand in computational power. Measurement is usually automated, depending on the complexity of the problem (the more complex it is, the more likely it will be).

Pairing is a form of colocating people for the benefit of minimizing latency. It is immediate feedback, and it can be both minimal and non-minimal. Developers discuss the solution implementation, developers and customers discuss the problem statement. Minimal feedback may come in the form of a binary statement ("This is not what I meant.") or an elaborate description ("I meant to foo that bar as a baz."). Pairing is inherently not automated feedback.

3.2 Facilitation and Stimulation

Feedback *facilitates* understanding when it gives answers to formulated questions. These questions are used to determine whether the task is complete and the individual can proceed with different tasks, or to determine how to proceed when there is more than feasible solution. Often, the question posed will be: "Is this what I wanted?" for a customer, and: "Is this what I understood it should be?" for a developer. The means to answer this questions are unit, acceptance, and usability tests. On other occasions, developers may focus on internal factors and ask themselves: "Should we refactor?" or "Are we going too fast?" or "How do I implement this in an efficient manner?". The measure to use here is pairing, but it could be measurements as well.

Feedback *stimulates* understanding when it gives answers to unasked questions. These questions are not at the focus of task execution, but their answers may serve as information radiators [3] to the developer. They can provide *trend information* to the developer or help to gain *insight* into structural properties unrecognized before. A question that may be asked is: "Has the performance of this service improved or dropped after my change?" or: "What are the component dependencies in my system?" The answer is often only a hint, the further conduct is still under control of the developer. Interpretation of the measurements is at his or her discretion.

4. Places for (Unit) Testing

If people learn faster and more from immediate, non-minimal feedback than from delayed, minimal feedback, what does this mean for the rightful place for (unit) testing?

One goal of agile development, no surprise there, is to remain flexible. The software design must be as simple as possible—that's why we refactor. The process must be malleable—that's because we may want to change it when we see the necessity. However, this must be balanced with the interest in quality—that's why we test, measure, and pair. How does this translate into the role for testing?

At certain points during the development process, we must test our product for compliance with formulated requirements. We need facilitating feedback. Programming ceases and we subject the product to rigorous, automated tests formulated beforehand. This rigidity, like it was formulated by XP, constrains the liberties of the developer. Therefore, if the testing phase is heavyweight, the whole development process becomes heavyweight, much also because XP demands that tests should be performed almost continuously. Feedback latency increases, thereby disturbing the learning process and reducing confidence about the product.

However, sometimes it is necessary to tolerate a high latency, because the risk taken justifies the waiting period. For example, the initial formulation of the key elements in an application (XP calls that metaphor) is a phase without any coding, feedback is based on abstraction alone. Yet, we could not go all the way (finishing the first release) without knowing where to go in the first place, without having spent any thought on how things might fit together and reaching consensus with the customers.

For reasons like the above, the reason to develop a fault model is based on a consideration like this:

> *The place for testing depends on two factors, namely the latency to get it and the risks taken when feedback is not gotten. Low latency is best, high risk is worst. The project team must decide where to place the emphasis based on its latency and risk characteristics.*

Stimulating feedback like it is exemplified by measuring, is usually geared towards getting feedback about issues that are not so important, i.e. not so risky. As long as the performance of the application is within the specified range we do not have to take any action. We want to learn even about properties of the software not yet asked for. However, this must not come at a price that compromises our ability to stay lightweight.

The according consideration is this:

The place for measuring lies in helping the team to better understand code properties that may be of importance. It depends on the ratio between the value of the information created and the impact the creation has on the agility of the development process. To reduce information overload, we must explicitly allow for exceptions.

The two above considerations, and the Agile Manifesto's principle of the value of face-to-face communication help us rank pairing, testing, and measuring with respect to their value for gaining feedback. Pairing is by far the best way to support the learning process between people; testing is the best way to learn about the important code properties, and it may not be regarded irrespective of the risk run without testing; measuring can be a helpful way of supporting further learning, but it may not interfere (without reason) with the structure of the development effort.

In an agile project, there is no such thing as too much learning. In a way this means: test as often as necessary, measure as often as possible. How much we test may not depend on how much it costs, but on what we may lose. How much we measure, on the other hand, should only be limited by the impact it has on our ability to stay lightweight.

As a final practical example, we decided to abandon measuring as a must-do after having drawn the above conclusions. Instead, we made measurements a feature integrated behind the scenes. Anyone is invited to use this information, but no one must do so. The results we have seen so far are promising.

5. References

1. Agile Alliance: The Agile Manifesto. Available from http://www.agilemanifesto.org
2. Kent Beck: Extreme Programming Explained. Reading 2000 (Addison-Wesley)
3. Alistair Cockburn: Agile Software Development. Reading 2001 (Addison-Wesley)
4. Wolfgang Dzida, Regine Freitag: Making Use of Scenarios for Validating Analysis and Design. IEEE Transactions on Software Engineering 24(12):1182-1196
5. Dierk König, Hans Wegener: Testing Weblications. Available from http://www.jugs.ch
6. Kurt van Lehn: Cognitive Architectures and Tutoring. 5th Annual ACT-R Workshop and Summer School, Carnegie Mellon University, 18-21 July 1998. Available from http://act-r.psy.cmu.edu

Test-Driven Development Support in Immerjion

Dwight S. Wilson

Matthew Pekar

Abstract

Immerjion is an open-source Java development tool supporting several supporting XP practices particularly test-driven development. We describe the TDD support and look for feedback on improving these features.

1 Introduction

Immerjion [4] is an open-source Java development system with an interface inspired by Smalltalk systems, particularly Squeak [7]. It was originally written by the first author and used as the basis of a class project in the course "Software Development using XP" taught at the Johns Hopkins University [8]. Currently a small group of students and former students are continuing development of the system.

The environment supports some of the Extreme Programming [1, 5] practices, including test-driven development [2], refactoring [3], and continuous integration. Support for distributed pair-programming is under development.

Immerjion uses a "code browsing" approach to viewing and editing code rather than a file based approach. The system contains a variety of browsers, starting with a with a "System Browser" similar to that found in Smalltalk. Additional browsers include a package browser, class browser, hierarchy browser, method browser, and class and method list browsers. The system also supports fast, powerful code navigation, including commands to find subclasses of a class, implementations of a method, and usage of classes, fields, and methods.

2 TDD Support

Current support for test-driven development in Immerjion is based on JUnit and Pounder [6]. Pounder is a JUnit extension for automated GUI testing. Basic TDD support includes:

1. Automated generation of skeletons for JUnit classes and methods as code is written.

2. An integrated test runner and result browser. The result browser is similar in function to the other code browser and allows navigation from a failed test.

3. Several commands to execute subsets of tests including:

   (a) Execute all tests.

   (b) Select and execute an explicit set of tests.

   (c) Execute tests using a selected class.

   (d) Execute all tests in a package.

   (e) Execute tests using uncommitted code.

   (f) Execute tests using any code that has been modified since it was last successfully tested.

4. Test-first support - as test methods are written, stubs for new production classes and methods are generated by the system.

A difficult task for developers is creation of automated tests for GUIs. Pounder is a utility that allows automated GUI testing using recording and playback mechanisms. A sequence of user actions may be recorded by customers, while developers write code to verify that correct changes have been made to the underlying model.

Currently, testing through Pounder is limited to playback of recorded scripts. This makes it more difficult to develop automated acceptance tests in parallel with feature implementation as enough of the GUI must be in place to allow recording. Furthermore, the resulting scripts may be platform dependent. For example, the trigger for popup menus may not be the same on different platforms, making it difficult or impossible to play back scripts involving popup menus on platforms other than the one used for for recording. An alternative higher-level scripting language is planned for a future version of Pounder. This will support platform independence and facilitate parallel development of acceptance tests.

3 Integration with Other Features

Unit-tests support is integrated with support for refactoring and source-code control. Currently CVS is the only source-code control system supported, though it would not be difficult to support other systems. Unit-test execution may optionally be specified as part of the check-in process. In that case, the unit-tests will automatically be executed, and the check-in will only continue if they pass at 100%.

Immerjion supports several refactorings, though currently not as many as some commercial IDEs. Optionally, unit-tests may be run automatically before and after a refactoring.

References

[1] Beck, K., “Extreme Programming Explained,” Addison-Wesley, 1999

[2] Beck, K., “Test-Driven Development By Example,” pre-publication draft, 2002

[3] Fowler, M., “Refactoring: Improving the Design of Existing Code,” Addison-Wesley, 1999

[4] Immerjion, http://www.immerjion.org

[5] Jeffries, R., Anderson, A., Hendrickson, C., and Beck, K., “Extreme Programming Installed,” Addison-Wesley, 2000

[6] Pounder, http://pounder.sourceforge.net

[7] Squeak, http://www.squeak.org

[8] Wilson, D., “Teaching XP: A Case Study,” XP Universe 2001