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of advantages (and some disadvantages) over the dependency mechanism, which we will not go into here, and has largely superseded it, even in VisualWorks.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: publisher[6]
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
```