Embracing Change with Squeak: Extreme Programming (XP)

J. Sarkela, P. McDonough, D. Caster

Introduction

In the world of sports, one often hears the adjective "extreme" applied to activities that involve high risk, a sense of daring, and a participant who constantly pushes the envelope to achieve the next level of success. Similarly, in the programming world, "extreme" describes a methodology in which teams collaborate on projects that entail risk, explore uncharted territories, and constantly push the envelope of productivity. This is the world of Extreme Programming (XP).

The XP practices embody a set of heuristics for recognizing and adapting to change, for change is the only constant in XP. The XP process values learning as a basic skill for individuals and the team, as the tensions inherent in development stimulate evolutionary growth. In the XP view, setbacks and failures provide essential feedback on which the software development process thrives, where risk is something to be understood and managed, not merely avoided.

In this chapter we explore the XP methodology and consider how it empowers the Squeak programmer. We introduce enough of the XP process to keep the discussion somewhat self-contained, before exploring a few chosen aspects such as planning and testing in more depth, to facilitate a Squeak developer's first few steps toward adopting XP.[1] We also describe simple ways to use XP in Squeak, exploiting the development tools present in a standard Squeak distribution, as well as the SUnit testing framework. [2] Finally, we conclude with some observations based on our experience with Squeak and the XP process.

XP Overview

Extreme Programming encompasses a set of mutually supporting practices which encourage communication, learning, and a constant emphasis on "the simplest thing that could possibly work". Teamwork among the customer, the development staff, and related business interests is essential to the success of any project. XP concentrates on building an evolving, shared vision of the project rather than promulgating detailed rules of architecture,

---

[1] A thorough treatment of XP can be found in *Extreme Programming Explained*, written by Kent Beck, published by Addison Wesley, 1999.

[2] The SUnit framework is available for download at www.xProgramming.com, among other places.

documentation, and the like. In a sense, all participants agree to a social contract when they decide to "do XP."

XP is an integrated approach, which cannot easily be broken out into its components without losing its essence. Nevertheless, several distinguishing practices within XP stand out: a focus on testing and repeatability, a rapid and iterative approach to software development and delivery, and programming in pairs (two people, one computer terminal, keyboard, and mouse). After the initial planning stage of a project, these quickly become the team's basic, day to day working environment. They are also transferable and beneficial to smaller teams or individuals, or to more informal projects which may not be able to access XP in its entirety.

XP's advocacy of close collaboration between technical and business interests does not, of course, change the fact that it is the programmers who write the code. Facilitating the production of code is the most important job, and XP devolves both power and responsibility onto the programmers. An iterative approach, with short production cycles from one code release to the next, leaves the programming team plenty of latitude to revisit and improve existing code. At the same time, this brings a working (if rudimentary) system to the customers as quickly as possible, giving them the opportunity to start with a minimal feature set, and then guide the requirements over time in response to actual business needs.

In order to make this work, XP imposes upon programmers the requirement that each new piece of code added to the system must come with a piece of test code to prove its functionality – and that the test code be written prior to the writing of the code it will test. Over time, this yields a growing suite of regression tests that represents the expected functionality of the program at any given time. Running the full suite of tests after each introduction of any new code serves both to assure programmers that, in making changes, they have not damaged existing features, and to assure the end user that the quality of the system will only improve over time. More subtly, this use of iteration and testing gives teams the courage to revisit and improve their earlier work, a crucial freedom for the long-term health of the code base.

Pair programming is the most visible manifestation of XP's overall emphasis on communication and mutual understanding of the project. Generally, one partner will 'drive', typing in the code while the other partner reads along. This gives the first partner the confidence that another pair of eyes is watching for any mistakes, and the second partner can also think ahead and be ready to propose a next course of action when the person doing the typing comes to a stopping point. More broadly, the programming pair is the atomic unit of an XP team, and crucial to building a shared understanding of the project's code base. This knowledge will transfer across the code base as team members switch partners from time to time, and as the various project tasks are passed from pair to pair – for in XP, nobody owns code, and each programmer will likely visit at least two parts of the system (different programming tasks) each week. By participating actively in ongoing planning and acceptance testing activities, among others, the end user will also share in the overall understanding of the project and its aims.

By keeping simplicity, reliability, and the rapid delivery of useful functionality at the forefront, XP weaves these practices together into an

efficient, practical methodology. In addition to the direct benefits of its component practices, XP also serves a project by eliminating the need for many time-consuming activities prescribed by most approaches to software development. With commonly understood and well-factored code, internal documentation becomes unnecessary. Similarly, as all programming is done in pairs and as each piece of code is revisited repeatedly over the life of the project, formal code reviews are mere redundancies in the XP world. Finally, by sticking to a simple, shared story of how the system works during each planning/development iteration, and by including a customer at all times in the project's day-to-day activities, XP smoothes and rationalizes the management-level communication, allowing programmers to get on with the job at hand.

Planning in XP

XP, as the name implies, focuses first and foremost on programming. Experience has shown it nearly impossible to determine at the beginning of a significant software development project precisely what features will be in the final system, so XP does not attempt it.

Nevertheless, XP begins with a plan. The plan aims not to detail and control software development far into the future, but rather to set the course for development. In his recent book *Extreme Programming Explained*, Kent Beck compares the practice of XP to the act of driving an automobile – and like a driver at the steering wheel, an XP team expects to correct its course on an ongoing basis. At the outset of development, the plan will be speculative. As the project progresses, collective understanding of the deliverable is refined, and adjustments take place as needed. By revisiting the plan throughout the life of the project, the team also gains a basis against which to calibrate its expectations of itself.

The primary medium of planning and documentation in XP is the humble index card. Customer plans take the form of story cards, and then development tasks appear on task cards. Each story describes a way in which the end user wishes to use the system, and many of the development tasks will correspond directly to one or more user stories. Since the technical plan takes into account internal quality issues as well as external features, however, other tasks will be related to refinement of the underlying design as much as to the implementation of the system. The cards themselves are short-lived, serving only to set out the aims of a single project iteration (a matter of a few weeks); in future iterations, the source code and the tests form a complete, accurate record of the current state of the project.

XP transforms the process of software development into a continuous learning experience for both customer and developer, and planning serves learning. The team learns what it is capable of, and what needs must be met to produce a system of value. It does this by being courageous, by taking small, measured steps, and by carefully evaluating the results it gets. Working in discrete iterations rather than attempting to drive the entire effort from a predefined "master plan" allows the team to accommodate the inevitable changes that take place in technical and business requirements. Taking frequent, small steps insures against making large investments of effort based on assumptions which turn out to be false in actual practice.

Plan Only as Long as Needed

The common practice of finalizing a design before attempting to write code often results in "over-frameworking"—constructing solutions to problems before the problems are actually discovered. In practice, this can lead to brittle, complicated programs that do not address the system requirements. (Since the actual requirements tend to vary over the life of a project anyway, one cannot expect to avoid this problem with ever more exact, detailed requirement specifications.) XP turns the process upside down, spending only enough effort on a given activity during a given production cycle to enable its dependent activities.

In XP, a system release is first planned by identifying a set of "user stories". This activity is the Planning Game. In this game, the objective is to identify (from the customer perspective) an increment of system function that either (initially) represents a minimum useful system, or (subsequently) adds value to a working system.

One of the purposes of planning is to make schedules predictable. During the Planning Game, each story is estimated by the development team in terms of ideal engineering time—time spent devoted entirely to the task. Projection of a completion date is determined by summing the ideal engineering time represented by the customer selected stories and multiplying by the reciprocal of the loading factor. From this result, a projected delivery date can be determined. A loading factor (always less than or equal to .5) is determined for the team by experience. If there is no calibrated (experience-based) value for the loading factor, something less than .5 must be used. Why? Because if the loading factor of individuals exceeds .5, they are not helping each other enough. Remember that all programming in XP is done in pairs.

The next level of planning, called the Iteration Planning Game is now played: the selected stories are decomposed into engineering tasks and estimated in essentially the same way as in the Planning Game. Iterations are increments of work towards completion of the stories selected in the Planning Game. Iteration planning is crucial to tuning the loading factor, and providing early feedback on team progress towards completion of the stories. Individual iterations tend to be short, about one to three weeks, whereas release cycles may take several months. The goal is to keep iterations short enough so that steering can take place in a controlled and informed manner.

Testing

In many popular approaches to software development, activities proceed in a linear fashion: from analysis and requirements gathering, to design, to coding, to testing, and finally to code release. XP regards these phases as artificial barriers, erected to compartmentalize an inherently fluid process. XP, in contrast, leverages the support each "phase" makes available to the others to adopt an iterative approach based upon rapid, repeated deployment of the software.

Once the activities of an iteration are identified, tests are written to validate the features that comprise that deliverable. In order to write tests, enough design is done to identify the key classes and behaviors of the system under construction. When the tests have been implemented, the

simplest code that will pass those tests is written. In later iterations, (or if the team decides to adopt XP in the course of an ongoing project), this may lead to the discovery of an opportunity to eliminate redundancy and redesign existing code. Such re-factoring facilitates future modification and reuse. As a direct consequence, all activities stay focused upon the group's fundamental objective: a working, usable system.

Test Thoroughly and Integrate Continuously

Unit tests give the programmer courage to do the right thing. At times in development, it will become clear that part or all of the system needs radical refactoring (reorganization, simplification, reassignment of responsibilities or other cleanup). An experienced developer normally feels some trepidation when embarking upon what could turn out to be a significant change. Automated unit tests assure that (tested) program features can be relied upon. After radical refactoring, a click of a button will confirm the fidelity with which we have preserved the system function that predated our changes. In the event that we were not entirely successful in maintaining system quality, the tests point to the areas that require immediate attention before we proceed with new feature implementation.

In modern programming, almost never do we find ourselves working alone on a software project of any significant size. Furthermore, in an XP project, no code is "owned" by a single individual (or pair of individuals). It is therefore essential that new work be integrated with the rest of the work of others as often as practical, which is to say, at the end of an episode of development, usually a few hours, or at most a day's worth of work. Integration is not complete until all the tests run correctly. Such integration episodes happen (serially) one at a time. This way, if testing reveals a regression, it is clear which set of changes "broke the build".

Using XP on a Squeak Project

Most of what became XP grew out of the experience of many individuals building object-oriented systems in Smalltalk, making XP a natural for use with Squeak. We now present a sketch of how one might proceed to adopt XP practices on a Squeak-based project.

One of the essential notions to keep in mind when using the XP process is that of scale. XP scales over a large continuum of project sizes. Such scaling may entail reducing or eliminating individual practices for smaller projects, but one must do this carefully. For instance, the planning phases may not require the two-tier approach of Planning Game and Iteration Planning Game. Perhaps a particular project requires only about a week's worth of work, and the stories are few. It may be the case that stories and tasks are in 1-to-1 correspondence. The iteration period may be shortened to a few days or even hours. These are all examples of projects at the small end of the size spectrum

If you are working alone, consider getting someone who understands what you are doing to pair program with you. Pair programming skills are good to have, and the other person just might have a suggestion that makes what you are doing better.

Test and Integrate

Probably the best way in which to become familiar with XP is to adopt the practices one at a time. For a software developer, the practice of testing is the place to start. That means making testing the cornerstone of all coding activities.

By writing the unit tests first, even before writing the code to be tested, we are led to consider what the public interface of our object should be, in a form of what Bertrand Meyer has called "Programming by Contract".[3] The tests verify that the object under test does indeed provide the services that we expect of it. The next step is to consider the conditions under which we expect the object to provide service to clients, and what would be reasonable responses to the agreed queries (api). Later on, when we actually begin writing code, we start by running the tests and seeing what works and what doesn't (prior to writing code, obviously, we expect nothing to work at all – which means things can only get better!). We take each success, failure, and error one at a time and extend and modify the code until all of the tests pass. Then and only then does it make sense for us to consider integrating this object or objects into larger systems.

Testing and frequent integration should never be skipped. Even on the smallest project, start by writing tests for the capability you plan to implement. Write tests while fleshing out the design for whatever it is you want the new capability to be. Implement the behavior tested for and ensure that the tests all pass. If you are adding capability to existing code, write tests to verify the continued correct operation of whatever you are extending or reusing.

Frequent integration ensures that collective code updates occur in a sequenced and repeatable manner. While there are many possible approaches, this is one simple suggestion. Identify a machine as the build machine. This machine should be one of the faster machines available to the development team. Create a build directory on the build machine. At least two images are kept in the build directory. The first is a raw, untouched image that is used as the basis for performing full system builds on a periodic basis. The second is the current development base image. This directory also contains a file that has the names of the released change sets—we'll call this a build list file. The released change sets themselves should be stored in a subdirectory.

At the beginning of a code development episode, the programmers copy the current development image from the build machine to the development machine they intend to use. When the image is first started, the programmers create a new change set with a name that identifies the programming task being addressed.

When the code has reached a point where all of the unit tests pass, the change set is filed out. The pair submits this change set file to the system builder, who files it into the current development base image. All of the unit tests for the project are run. If all of the tests pass, then that image is saved as the new development base image and the change set is placed in a

---

[3] Meyer, Bertrand. *Object-Oriented Software Construction*. Prentice Hall, 1988.

subdirectory of released code changes. The file name of this change set is added as the last line of the build list file.

If any less than 100 percent of the unit tests succeed, the image is not saved. The code submission is rejected and the responsible team grabs the latest development build, loads their changes, and works until all of the tests pass. They then resubmit the new change set to the system builder.

A full system build should be performed after every development iteration, in order to archive a benchmark as a starting point for the next iteration. A full system build consists of starting with a raw image and filing in the changes that are recorded in the build list file. It is important for reasons of repeatability to file in these changes in the order that they are called out in the build list file. After all of the released change sets have been filed in, all tests are run. If any tests fail, the fixes are placed in updated change sets, the build list file updated accordingly, and the build restarted. The next iteration should not begin until all of the unit tests pass, immediately after a complete build is run.

Reusing Previously Developed Code

Squeak comes with a large class library. Very often, the simplest thing that could possibly work implies reusing code that is part of the base class library. Since the Squeak base class library has few SUnit tests, what should a conscientious programming team do?

The answer is, it depends. Squeak is an open source project. This fact is both a blessing and a (mild) curse. Because Squeak is undergoing constant change, individual system behavior may be altered subtly or radically over short intervals of time. Writing tests for the capability being reused will help to defend the integrity of present and future system function added by an XP team using Squeak.

If the code being reused is known to be stable and to have survived extensive reuse, then there may be no compelling need to take the time required to develop an SUnit test. Examples of this would be the core Collection and Magnitude classes.[4] On the other hand, one of the purposes of SUnit tests is to help give us courage when reusing and refactoring code. In the presence of reuse we would like to have our SUnit test serve as a contract for service with the code being considered for reuse. Rather than writing a comprehensive unit test suite for the classes being reused, it is sufficient to write a test that defines a contract for service. We only need to test for the capabilities of the interface that we need to reuse.

These observations may provoke controversy between the pragmatist and the purist. It is certainly in the interest of all Squeak programmers to have an extensive test suite for the underlying classes of the base distribution. This is true if for no other reason than to insure that ports to other platforms work correctly. Furthermore, an extensive test suite built up over time would help the developers and maintainers of Squeak to improve the base class hierarchy and give them courage when faced with the

[4] They are not included in the standard Squeak distribution, but the ANSI Smalltalk unit tests released under the aegis of Camp Smalltalk address significant portions of these class hierarchies, among others.

inevitable need to make fundamental changes. Finally, when packaging Squeak applications for deployment, such tests could help to insure that needed system function is present after the packaging process is completed.

Testing User Interface Functionality

A basic tenet of XP is that anything that does not have an automated test does not exist. That is an interesting theory in principle, but it is less than satisfactory in practice, especially in a media rich environment like Squeak. This is a time when the 20-80 rule comes strongly into play. There are times when we cannot afford a solution that addresses 100 percent of our needs. The 20-80 rule asserts that 80 percent of the benefit comes from 20 percent of the work. Thus, even though we cannot effectively automate 100 percent of the user interface components, automating even a small measure of the user interface capability yields tangible benefit.

For a first level of testing, there is a distinct benefit to just programmatically creating user interface components, opening them and deleting them. This level of testing will identify gross errors. More sophisticated tests can test whether allocated resources and component models are returned to the system after the component has been deleted.

A more comprehensive test may be constructed using the `EventRecorderMorph`. These tests should ensure exact location and extent of the interface component under test. Once the components are placed, event tapes may be played back that exercise the actual code paths. A caveat is that popups, like notifiers and confirmers, will appear under the active hand and not the hand playing back the remote events. For development that is heavily user interface-centric, it would be well worth defining a programming task to allow these popups to open under the playback hand rather than the user hand.

Example: an event framework

How do we adopt XP practices, then? We jump right in and start writing tests, of course. Well, soon, but not quite yet. In order to write tests, we need to understand enough about our design to be able to expect to reuse the end result. We must discover the essential classes involved, the messages they implement, and how these objects collaborate. In this section, we will look at an actual development episode, to introduce the planning and testing practices in the course of retro-fitting XP on to an existing project.

Specifically, we will look at how one might test and (if necessary) modify an event signaling and handling mechanism for Squeak.

The event mechanism: stories and tasks

The Squeak 2.8 update stream adds an event based dependency mechanism to the base image. In our example, we will test a similar event mechanism. We intend to reuse this latter event mechanism, an application of the Observer pattern[5], and depend heavily upon its correct behavior. The

[5] Gamma et al., *Design Patterns,* Addison-Wesley, 1995.

mechanism allows a model object to trigger synchronous event notifications, and observer objects may register an action which they will perform when that model object triggers a particular event notification – conceptually, akin to a broadcast form of message send. An event is identified by a unary or keyword symbol. The model object triggering a keyword event will be expected to supply appropriate argument objects for that particular event. On the other side, the observer object's action is typically specified in terms of a receiver object, a selector, and argument bindings either from a keyword event notification or supplied with the registration. Our prior experience led us to begin with a different interface, so we expect to add method selectors for portability reasons, and we may also have to make some adjustments in the implementation.

The key behaviors for the event mechanism are introduced as extensions to the class Object. These behaviors may be categorized as relating to registering, triggering and releasing events, and we will reflect this in the method protocols we create. As tests reveal missing selectors, we will implement the necessary behavior.

At this point, we might want to describe the operation of the event mechanism on a task card. This particular event mechanism is probably about as complex as a single task should get, so we would probably consider it one task for planning purposes.

We intend to implement these semantics:

Registering

These messages allow an observer to register an action to be performed when an event is triggered. When the receiver triggers <anEvent>, the message <aSelector> is sent to <anObject>. If <anEvent> offers arguments, these will be bound to <aSelector> prior to forwarding to <anObject>. If multiple actions are specified, they will be performed in the order in which they were registered.

when: anEvent send: aSelector to: anObject

when: anEvent send: aSelector to: anObject with: anArgument

when: anEvent send: aSelector to: anObject withArguments: aSequenceOfArgs

Triggering

These messages allow an object to signal that an event named <anEvent> has occurred. All registered actions will have completed before these messages return. The return value of the trigger method is the value returned by the last action performed.

triggerEvent: anEvent

triggerEvent: anEvent with: anArgument

triggerEvent: anEvent withArguments: aSequenceOfArgs

triggerEvent: anEvent ifNotHandled: aBlock

triggerEvent: anEvent with: anArgument ifNotHandled: aBlock

triggerEvent: anEvent withArguments: aSequenceOfArgs ifNotHandled: aBlock

Releasing

These messages allow an observer to release event dependencies upon the receiver.

removeAllActionsWithReceiver: anObject

removeAllActionsWithReceiver: anObject forEvent: anEvent

removeActionsForEvent: anEvent

Writing SUnit tests for the event mechanism

Our task is to build a Unit test that exercises this interface. To build a unit test, we first subclass TestCase. To provide reference to the initial conditions which must exist in order for our test to have meaning, we add instance variables to this TestCase subclass. We override two methods, #setUp and #tearDown, in order to support the corresponding initialization and finalization (releasing) behavior. With this information at hand, we create a new change set, KernelEventTests, to track the development of the test case. As we begin to write the tests for event triggering, we soon discover that the message selector, #triggerEvent:, is not implemented. This is something of a surprise. Upon examination of the class Object, we discover that in the Squeak implementation, the designers chose the selector #trigger: rather than #triggerEvent:.

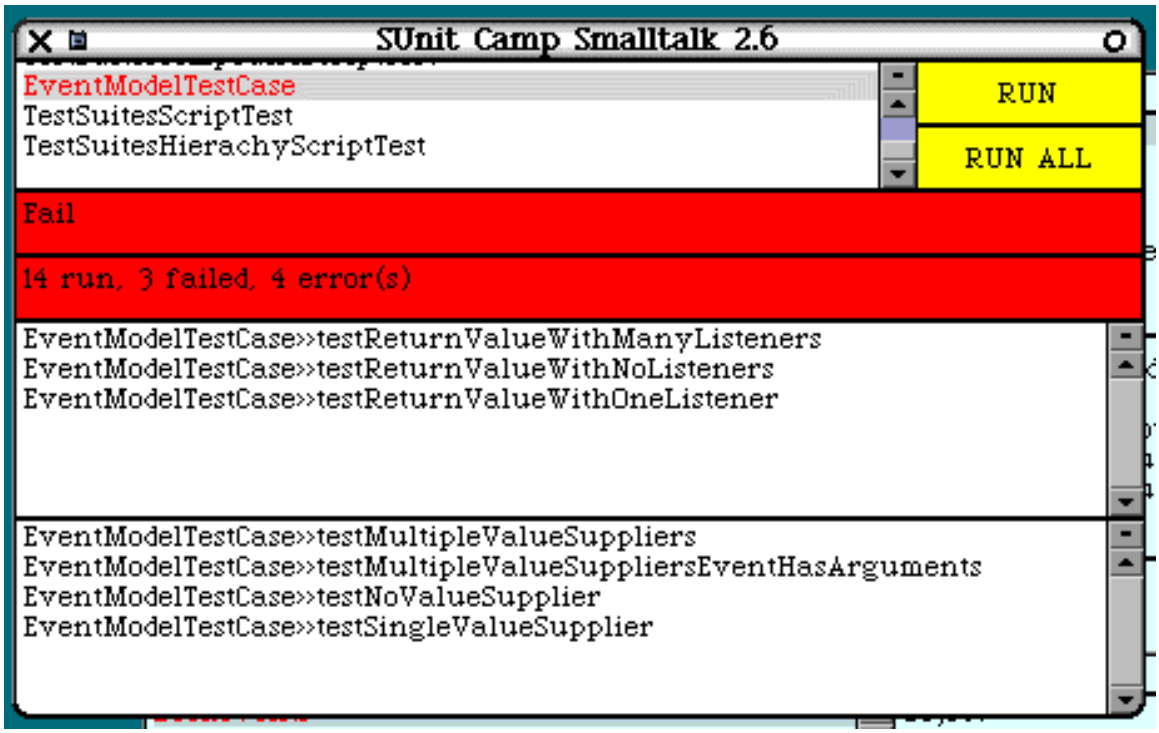
At this point, we pause and put on our re-designers' hats. The folks who made this choice are themselves familiar with yet another implementation of this mechanism that uses slightly different message selectors. On this project, we value portability. If we implement the other selector form, #triggerEvent:, by delegating its implementation to #trigger:, then we can satisfy both interfaces. Further, if we test the outer wrapper methods, we will know for certain that we have tested the inner methods.

As soon as we recognize the need for wrapper methods, that task becomes fairly obvious, and as developers we naturally want to jump right in and create those methods – after all, they're trivially simple, so why not? Well, at this point it behooves us to make a conscious decision. Writing the wrapper methods is a coding task. Developers are always faced with the temptation of blasting ahead and jumping into coding. Fortunately, XP supplies an alternative – remember, there are always two of you at a keyboard. So rather than going off track, the programmer who is not at the keyboard grabs a handy index card and starts a list of things that must be implemented when the tests are done. At the top of that list is the task of writing wrapper methods for the #trigger: family of selectors. The presence of this list will pay off eventually, because the pair of programmers will not have to stop in order to discuss what to do next once the task at hand is complete.

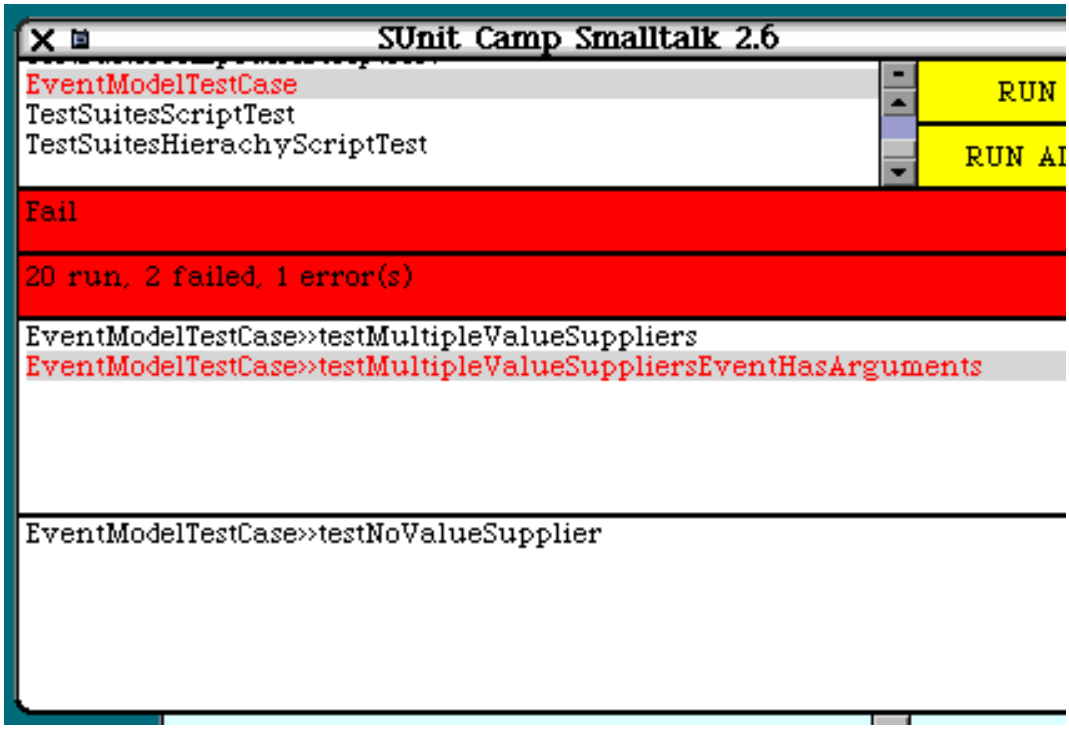
In the course of the coding episode which resulted in the creation of this event system and test suite (see Appendix, below), we made the following notations on our to-do task card:

```
wrap trigger:* methods with triggerEvent:* methods
need *:ifNotHandled: methods
need to implement removeActionsForEvent:
need to implement removeAllActionsWithReceiver
need to implement removeAllActionsWithReceiver:forEvent:
```

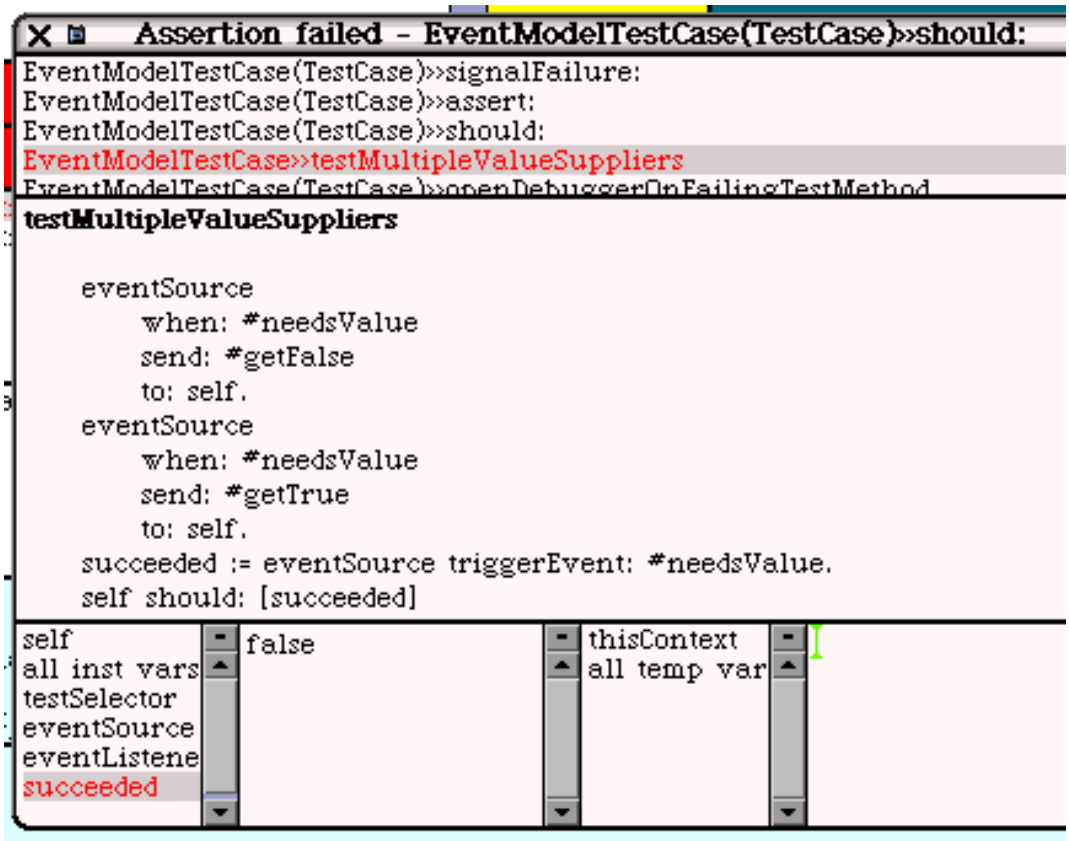
Meanwhile, back at test central, we have continued to implement the unit tests for the event mechanism. In that process we noted that the event framework also uses divergent selectors when releasing event dependencies, and we further note the absence of #trigger:ifNotHandled: selectors. In the spirit of our previous decisions, we note on an index card that these selectors need to be implemented for portability reasons. When the tests are completed, we run through the cards of work to be done and implement the missing methods that remove actions.



Here we see the results of running our SUnit test after implementing the missing selectors that we noted when writing the unit test. Not quite the results we may have hoped for, but very good information. We start by looking at the simpler of the two errors, #testReturnValueWithNoListeners. Upon examination we see that the test expected a nil return value and got instead the object that triggered the event. We read the code in question and see that the mechanism as implemented does not include supporting the notion of events that request information from the dependent. This means that we must rewrite the core #trigger: selectors so that they return the value of the last event handler. In this process, we reassess our earlier design decision to wrap the existing #trigger: methods with our #triggerEvent: methods. If we instead wrap the new triggerEvent: methods with the #trigger: methods, we can remove redundant code in our implementation. The time is taken to refactor the code to reflect this new understanding. We then rerun the tests and see a small improvement.



We need to examine this one more closely. Why exactly did this test fail? Selecting the failure will let us open up a debugger on the test in question.



From the debugger, we see that the value being returned is not the value returned from the last action registered. When we look at the code that implements #trigger: we see that it iterates over the collection of registered event actions and returns the value of the last event action. While looking at this code we see temporary variables that suggest that the collection of event handlers is a Set. So, we conclude that the underlying collection holding event actions is some kind of unordered collection. Upon examination of the code (see Object>>when:perform: in Squeak 2.8), we discover that it is indeed using a Set. We change this to be an OrderedCollection and rerun the tests. All of the tests pass and we are ready to use the event mechanism.

What did we learn from this? First, that even though we may see message selectors that we have been using for over a decade in a different dialect of Smalltalk, there may be subtle but important differences in the semantics of the implementation. We found no bugs in the implementation, just that it did not conform to the expectations that we had based upon prior experience. By codifying the contract for service that we expected as an SUnit test, we discovered this fact earlier and were able to write a compatibility layer that met our expectations and did not perturb the fundamental operation of the system as released.

How much project time was used by writing the tests before coding the implementation? That is hard to assess prior to the actual completion of the project. Our development was very focused and wasted very little effort because the test framework gave us a list of what was not yet working. We could have built some very large systems that appeared to work before discovering the more subtle use of unordered collections to hold event actions. It may have been very much more time consuming to debug a complex system that depended heavily upon events and was failing because of them in ways that pointed away from the event mechanism itself. Our test discovered the discrepancy before any other code could come to depend upon it. So in the last analysis, we did spend more time up front writing tests that reflect our understanding of the reusable component. However, as a consequence we had a strongly directed implementation effort that revealed deep but subtle inconsistencies earlier rather than later.

Programming in Pairs: Two Heads Are Better Than One

Each of the XP practices serves a concrete purpose from the technical point of view. What is not as obvious, is the underlying social engineering inherent in XP practices. For example, focusing on testing not only directs us to our goal, but also defuses territorial behavior amongst team members. When all of the code is owned by all of the team, all code ownership questions disappear. If someone changes some code and makes it simpler without breaking any tests, everyone benefits. By making unit tests a first priority, the real issues are brought into focus. Is the code base simpler? Do all of the tests pass?

Since XP is a comprehensive approach to software production, involving managers as much as architects and programmers, it impacts all aspects of a team's work. In effect, the XP approach becomes an unspoken language connecting all stakeholders in a project. The practice of pair programming, more than any other, leaps out to visibly distinguish XP from other methodologies.

It is probably safe to say that the experience of programming with another person at your side is going to be different as a function of who your partner is. Since each partner's responsibility (and personality) is a little different, it is useful to consider some of the skills that make pair programming work, and some of the obstacles you may need to overcome to become a good pair programmer.

Most of us are accustomed to working alone when we program. We don't have to worry about someone else following what we are doing when we are typing away at our keyboards, operating browsers, evaluating expressions in a workspace, and all the other activities we engage in when we are "Squeaking". Programming with another individual at your side watching (nearly) your every move might take a little getting used to, but it is worth it.

In XP, the person that has the keyboard and mouse has the immediate responsibility for the programming task, and we will call him or her the driver. The other partner, whom we will refer to as the navigator, has the responsibility of thinking strategically and understanding the story arc represented by the particular code written and entered by the driver. Most of us are not mind readers, and at times it can be difficult to remain coupled to the same train of thought, especially when the partners may at times be thinking at different levels of abstraction. This is where communication, be it visual, spoken, or unspoken becomes particularly important.

A running dialog between the partners is essential. It will be much easier to see the same pictures in your respective heads if you talk between yourselves about what you are doing. This not only serves the purpose of staying in touch with each other's thoughts, but also leads both individuals to think through what they are doing. Frequently, verbalizing what you think you are doing makes you think more clearly about what you are actually doing. Weaknesses as well as strengths of particular approaches to problems can be considered quickly, resulting in swifter convergence on working code.

If code is a kind of literature, than reading is probably the most important part of communication between a pair. Most of the time we will be reading code, and probably executing it in our heads. The person that's "driving" has to allow the partner to read and understand what is being written—in fact its essential. Flashing between windows as fast as you can move your mouse is a sure way to lose somebody if you are not in constant communication some other way. Quick tangential thoughts can distract us for a moment and cause us to lose contact with our partner. Avoid this by verbalizing the thought at the proper moment or making a note on an index card. If one of you believes exploration is needed, be sure that you say so, whether you are driving or not.

This is another place where courage is essential. Each partner must be unafraid to verbalize their respective feelings about what is being done. If you are afraid of admitting that you are not following what is going on, then you will not be able to recover from those brief lapses of attention to which we fall prey. If you do not understand part of the system that you are working on, admit that, find someone who does, and get them to help you. In the culture of XP, it is a rule that you cannot refuse to help others on the team when they ask for it.

Learning is essential, too. Take the time to understand what someone else is proposing before becoming predisposed to your own approach. Listen carefully and be observant so that you know who to go to when you need to know something, then take the time to learn as much as you can from them. Pair-programming is a great way to do that.

XP and Squeak

Squeak is ideally suited to serve the emerging world of remote, media rich software applications. In such an environment, far more than on a desktop, the efficiency and reliability of the software becomes a crucial consideration. The Squeak programming and deployment environment is rich, powerful, and almost entirely flexible, and further, its nature as an open source project lends it that dynamic character which spawns experimentation and rapid advancement of the state of the art. But there is a price to be paid for this energy. In such a rapidly changing world, anyone who wishes to deliver truly production quality software over more than one or two release cycles ignores notions such as code simplicity and flexibility, not to mention reliability, only at considerable peril.

In navigating these shoals, Squeak will not be sailing uncharted waters. The notion of using defined software practices to address real-world project complexity is hardly new. As a lightweight, but comprehensive methodology which addresses such concerns, XP is ideally suited for use by Squeakers. Beyond its clear applicability to object-oriented systems and its proven success, it shares with Squeak a fundamental emphasis on the acts of learning and sharing as the background to all activities. More than most formally specified methodologies, XP takes into account the natural creative power inherent in the individual team members and their interactions. And XP gets straight to the point. It aims simply to facilitate the fundamental act of software production – delivering working programs – and recognize that most of anything else that goes on is ultimately wasted effort.

It seems almost axiomatic that every process invented to improve the chances of success for completing a software project must be remembered for a key feature of that process. To pick just one, or even several, of the practices that make up XP, however, would miss the point entirely. Those practices synergize in a way that causes us to look at the systems development effort in an altogether different light: as a holistic, organic and yet ultimately quite rational process. Such an understanding demands an approach to project direction which aims to set out a clear mechanism for assuring the robustness of the end product, and then guiding and facilitating rather than seeking to proscribe the team's creative efforts.

For perhaps most of all, XP recognizes that each real world situation will differ slightly from any possible model, so it knows where to stop. Once the team has learned to work together efficiently, without unnecessary noise or backtracking, they have become experts on the problem at hand, and XP does not presume to tell them exactly how to proceed in every situation – the only absolute rule is, no rule is absolute. Besides all the obvious efficiency and impeccable Smalltalk heritage, in its careful accounting for the human factor, XP reveals the world view it shares with Squeak – computers are there to serve people, not vice versa, and by the way, there's no shame in having fun.

Appendix – Source Code

Unit Tests for the Event System

```
TestCase subclass: #EventModelTestCase
        instanceVariableNames: 'eventSource eventListener succeeded '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'Kernel-Tests-EventModel'!

!EventModelTestCase methodsFor: 'running-dependent action' stamp:
'jwsdlc 7/29/2000 13:29'!
testNoArgumentEvent

        eventSource when: #anEvent send: #heardEvent to: self.
        eventSource triggerEvent: #anEvent.
        self should: [succeeded]! !

!EventModelTestCase methodsFor: 'running-dependent action' stamp:
'jwsdlc 7/29/2000 13:29'!
testOneArgumentEvent

        eventSource when: #anEvent: send: #add: to: eventListener.
        eventSource triggerEvent: #anEvent: with: 9.
        self should: [eventListener includes: 9]! !

!EventModelTestCase methodsFor: 'running-dependent action' stamp:
'jwsdlc 7/29/2000 13:29'!
testTwoArgumentEvent

        eventSource when: #anEvent:info: send: #addArg1:addArg2: to:
self.
        eventSource triggerEvent: #anEvent:info: withArguments: #( 9 42
).
        self should: [(eventListener includes: 9) and: [eventListener
includes: 42]]! !

!EventModelTestCase methodsFor: 'running-broadcast query' stamp: 'jwsdlc
7/29/2000 13:37'!
testMultipleValueSuppliers

        eventSource
                when: #needsValue
                send: #getFalse
                to: self.
        eventSource
                when: #needsValue
                send: #getTrue
                to: self.
        succeeded := eventSource triggerEvent: #needsValue.
        self should: [succeeded]! !

!EventModelTestCase methodsFor: 'running-broadcast query' stamp: 'jwsdlc
7/29/2000 13:39'!
testMultipleValueSuppliersEventHasArguments
```

```
        eventSource
                when: #needsValue:
                send: #getFalse:
                to: self.
        eventSource
                when: #needsValue:
                send: #getTrue:
                to: self.
        succeeded := eventSource triggerEvent: #needsValue: with:
'kolme'.
        self should: [succeeded]! !

!EventModelTestCase methodsFor: 'running-broadcast query' stamp: 'jwsdlc
7/29/2000 14:25'!
testNoValueSupplier

        succeeded := eventSource
                triggerEvent: #needsValue
                ifNotHandled: [true].
        self should: [succeeded]! !

!EventModelTestCase methodsFor: 'running-broadcast query' stamp: 'jwsdlc
7/29/2000 13:37'!
testNoValueSupplierHasArguments

        succeeded := eventSource
                triggerEvent: #needsValue:
                with: 'nelja'
                ifNotHandled: [true].
        self should: [succeeded]! !

!EventModelTestCase methodsFor: 'running-broadcast query' stamp: 'jwsdlc
7/29/2000 13:40'!
testRemoveActionsForEvent

        eventSource
                when: #anEvent send: #size to: eventListener;
                when: #anEvent send: #getTrue to: self;
                when: #anEvent: send: #fizzbin to: self.
        eventSource removeActionsForEvent: #anEvent.
        self shouldnt: [eventSource hasActionForEvent: #anEvent]! !

!EventModelTestCase methodsFor: 'running-broadcast query' stamp: 'jwsdlc
7/29/2000 13:40'!
testSingleValueSupplier

        eventSource
                when: #needsValue
                send: #getTrue
                to: self.
        succeeded := eventSource triggerEvent: #needsValue.
        self should: [succeeded]! !

!EventModelTestCase methodsFor: 'running-dependent value' stamp: 'jwsdlc
7/29/2000 13:38'!
testReturnValueWithManyListeners

        | value newListener |
```

```
        newListener := 'busybody'.
        eventSource
                when: #needsValue
                send: #yourself
                to: eventListener.
        eventSource
                when: #needsValue
                send: #yourself
                to: newListener.
        value := eventSource triggerEvent: #needsValue.
        self should: [value == newListener]! !

!EventModelTestCase methodsFor: 'running-dependent value' stamp: 'jwsdlc
7/29/2000 13:38'!
testReturnValueWithNoListeners

        | value |
        value := eventSource triggerEvent: #needsValue.
        self should: [value == nil]! !

!EventModelTestCase methodsFor: 'running-dependent value' stamp: 'jwsdlc
7/29/2000 13:38'!
testReturnValueWithOneListener

        | value |
        eventSource
                when: #needsValue
                send: #yourself
                to: eventListener.
        value := eventSource triggerEvent: #needsValue.
        self should: [value == eventListener]! !

!EventModelTestCase methodsFor: 'running-dependent action supplied
arguments' stamp: 'jwsdlc 7/29/2000 13:38'!
testNoArgumentEventDependentSuppliedArgument

        eventSource when: #anEvent send: #add: to: eventListener with:
'boundValue'.
        eventSource triggerEvent: #anEvent.
        self should: [eventListener includes: 'boundValue']! !

!EventModelTestCase methodsFor: 'running-dependent action supplied
arguments' stamp: 'jwsdlc 7/29/2000 13:38'!
testNoArgumentEventDependentSuppliedArguments

        eventSource
                when: #anEvent
                send: #addArg1:addArg2:
                to: self
                withArguments: #('hello' 'world').
        eventSource triggerEvent: #anEvent.
        self should: [(eventListener includes: 'hello') and:
[eventListener includes: 'world']]! !

!EventModelTestCase methodsFor: 'running' stamp: 'jwsdlc 7/29/2000
13:27'!
setUp
```

```
        super setUp.
        eventSource := EventModel new.
        eventListener := Bag new.
        succeeded := false! !

!EventModelTestCase methodsFor: 'running' stamp: 'jwsdlc 7/29/2000
13:29'!
tearDown

        eventSource removeEventsTriggeredFor: eventListener.
        eventSource removeAllEventsTriggered.
        eventSource := nil.
        eventListener := nil.
        super tearDown.! !

!EventModelTestCase methodsFor: 'private' stamp: 'jwsdlc 7/29/2000
13:26'!
addArg1: arg1
addArg2: arg2

        eventListener
                add: arg1;
                add: arg2! !

!EventModelTestCase methodsFor: 'private' stamp: 'jwsdlc 7/29/2000
13:26'!
getFalse

        ^false! !

!EventModelTestCase methodsFor: 'private' stamp: 'jwsdlc 7/29/2000
13:26'!
getFalse: anArg

        ^false! !

!EventModelTestCase methodsFor: 'private' stamp: 'jwsdlc 7/29/2000
13:26'!
getTrue

        ^true! !

!EventModelTestCase methodsFor: 'private' stamp: 'jwsdlc 7/29/2000
13:27'!
getTrue: anArg

        ^true! !

!EventModelTestCase methodsFor: 'private' stamp: 'jwsdlc 7/29/2000
13:27'!
heardEvent

        succeeded := true! !
```

The Events Implementation

```
!Object methodsFor: 'events' stamp: 'jwsdlc 7/29/2000 13:57'!
hasActionForEvent: anEventSymbol
      "Return true if the receiver has a handler registered for
<anEventSymbol>."

      | eventTable |
      eventTable := self myEvents.
      eventTable ifNil: [^false].
      ^self myEvents includesKey: anEventSymbol! !

!Object methodsFor: 'events' stamp: 'jwsdlc 7/29/2000 13:57'!
removeActionsForEvent: anEventSymbol

      | scratchEventTable |
      scratchEventTable := self myEvents copy.
      scratchEventTable ifNil: [^self].
      scratchEventTable
            removeKey: anEventSymbol
            ifAbsent: [].
      self myEvents: scratchEventTable! !

!Object methodsFor: 'events' stamp: 'jwsdlc 7/29/2000 14:15'!
trigger: anEventSymbol
      "Evaluate all message sends registered for anEventSymbol."

      ^self triggerEvent: anEventSymbol! !

!Object methodsFor: 'events' stamp: 'jwsdlc 7/29/2000 14:17'!
trigger: anEventSymbol with: aParameter
      "Evaluate all message sends registered for anEventSymbol
      and pass aParamter to the registered actions."

      self triggerEvent: anEventSymbol with: aParameter! !

!Object methodsFor: 'events' stamp: 'jwsdlc 7/29/2000 14:16'!
trigger: anEventSymbol withArguments: anArray
      "Evaluate all message sends registered for anEventSymbol
      and pass anArray to the registered actions."

      ^self triggerEvent: anEventSymbol withArguments: anArray ! !

!Object methodsFor: 'events' stamp: 'jwsdlc 7/29/2000 14:14'!
triggerEvent: anEventSymbol
      "Evaluate all message sends registered for anEventSymbol."

      ^self triggerEvent: anEventSymbol ifNotHandled: [^nil]! !

!Object methodsFor: 'events' stamp: 'jwsdlc 7/29/2000 14:13'!
triggerEvent: anEventSymbol ifNotHandled: aBlock
      "Evaluate all message sends registered for anEventSymbol."

      ^self triggerEvent: anEventSymbol withArguments: #() ifNotHandled:
aBlock! !
```

```
!Object methodsFor: 'events' stamp: 'jwsdlc 7/29/2000 14:15'!
triggerEvent: anEventSymbol with: aParameter
      "Evaluate all message sends registered for anEventSymbol
      and pass aParamter to the registered actions."

      ^self triggerEvent: anEventSymbol with: aParameter ifNotHandled:
[^nil]! !

!Object methodsFor: 'events' stamp: 'jwsdlc 7/29/2000 14:13'!
triggerEvent: anEventSymbol with: aParameter ifNotHandled: aBlock
      "Evaluate all message sends registered for anEventSymbol
      and pass aParamter to the registered actions."

      ^self triggerEvent: anEventSymbol withArguments: (Array with:
aParameter) ifNotHandled: aBlock! !

!Object methodsFor: 'events' stamp: 'jwsdlc 7/29/2000 14:15'!
triggerEvent: anEventSymbol withArguments: anArray
      "Evaluate all message sends registered for anEventSymbol
      and pass anArray to the registered actions."

      self triggerEvent: anEventSymbol withArguments: anArray
ifNotHandled: [^nil]! !

!Object methodsFor: 'events' stamp: 'jwsdlc 7/29/2000 14:12'!
triggerEvent: anEventSymbol withArguments: anArray ifNotHandled: aBlock
      "Evaluate all message sends registered for anEventSymbol
      and pass anArray to the registered actions."

      | result |
      (self hasActionForEvent: anEventSymbol)
            ifFalse: [^aBlock value].
      (self myEvents at: anEventSymbol)
            do:
                  [:each |
                  result := each valueWithArguments: anArray].
      ^result! !

!Object methodsFor: 'events' stamp: 'jwsdlc 7/29/2000 14:22'!
when: anEventSymbol perform: aMessageSend
      "Register aMessageSend as action for anEventSymbol."

      | events |
      (events := self myEvents) ifNil:
            [self myEvents: (events := IdentityDictionary new)].
      (events
            at: anEventSymbol
            ifAbsentPut: [OrderedCollection new]) add: aMessageSend! !

!Object methodsFor: 'events' stamp: 'sma 2/29/2000 20:47'!
when: anEventSymbol send: aSelector to: anObject
      self
            when: anEventSymbol
            perform: (MessageSend receiver: anObject selector:
aSelector) ! !

!Object methodsFor: 'events' stamp: 'sma 2/29/2000 20:49'!
when: anEventSymbol send: aSelector to: anObject with: anArgument
```

```
	self
		when: anEventSymbol
		perform: (MessageSend receiver: anObject selector: aSelector
argument: anArgument) ! !

!Object methodsFor: 'events' stamp: 'sma 2/29/2000 20:48'!
when: anEventSymbol send: aSelector to: anObject withArguments: anArray
	self
		when: anEventSymbol
		perform: (MessageSend receiver: anObject selector: aSelector
arguments: anArray) ! !
```