

# Embracing Change with Squeak: Extreme Programming (XP)

*J. Sarkela, P. McDonough, D. Caster*  
*The Fourth Estate, Incorporated*

---

## Introduction

In the sports world, we often hear 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 only constant in XP is change. The XP practices embody a set of heuristics for recognizing and adapting to change. Just as personal evolution and growth can be difficult and challenging, project evolution and growth present a set of challenges both technical and social in nature. The XP process values learning as a basic skill for individuals and the team. It values the tensions inherent in development as a source of stimulation for evolutionary growth and change. It views setbacks and failures as essential aspects of the development process, as they provide the ongoing feedback on which that process thrives.

In this chapter we explore the XP methodology and consider how it may empower the Squeak programmer. We introduce enough of the XP process to keep the discussion somewhat self-contained.<sup>1</sup> We also describe simple ways to use XP in Squeak, exploiting the myriad development tools present in a standard Squeak distribution. Finally, we conclude with some observations based on our experience with Squeak and the XP process.

---

## XP Overview

Extreme Programming is defined in terms of a set of mutually supporting practices. These practices encourage communication, learning, and the simplest possible things that could work. Teamwork between the customer, related business interests, and the development staff, is essential to the success of any project. In XP, it is a social contract that all participants have agreed to.

Planning in XP is a necessary first step. A plan in XP lays out a course of action. The purpose of the plan is not to detail and control

---

<sup>1</sup> A thorough treatment of XP can be found in [Extreme Programming Explained](#), written by Kent Beck, published by Addison Wesley, 1999.

---

## Embracing Change with Squeak: Extreme Programming (XP)

development, but rather to set the course for development. It is expected that course corrections will be a continuing activity. At the outset of development, the plan will be speculative. As the project progresses, collective understanding of the deliverable is refined, and adjustments will be made through a practice called steering. Ultimately, the plan provides a basis against which to calibrate the team's expectations of itself.

Plans are laid first for the customer, then for the developer. Customer plans are generated from user story cards that describe the ways in which the system will be used. Developer plans are expressed in terms of development tasks. Many of these tasks will be tied directly to one or more user stories. However, the technical plan also takes into account internal quality issues. Many of the tasks will be related to refinement of the design as well as implementation of the system.

The kinesthetic exercise of using index cards in XP planning is the start of the process that identifies what a system under development will do. Experience has shown it nearly impossible to determine at the beginning of a significant software development project just what features will be in the final system. XP translates the development process into a continuous learning process for both customer and developer.

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 iteratively accommodates the inevitable changes in technology and business needs. Taking frequent, small steps ensures against making large investments in wrong assumptions.

---

### *XP in Practice*

Extreme programming will seem to be backwards to someone experienced with most traditional methodologies. In commonly applied approaches to software development, activities run from analysis and requirements gathering, to design, to coding, to testing, and finally code release. Artificial barriers between these phases are often erected even though the work of a real world development project rarely compartmentalizes that rigidly.

#### Practice Only as Long as Needed

Finalizing a design before building concrete experience with the system under development often results in highly abstract frameworks. The results are often brittle, hard to change implementations that do not address fundamental requirements. In contrast, XP turns the process upside down, and only does an activity long enough to enable its dependent activities.

In XP, the system release is first planned. Then tests are written. In order to write tests, enough design is done to identify the key classes and behaviors of the system being built. When the tests have been coded, the simplest code that will pass those tests is written. This may lead to the discovery of the need to eliminate redundancy and exploit opportunities to redesign existing code that makes it easier to change or reuse. A direct consequence of this approach is that all of our activities stay focused upon the fundamental objective: a working, usable system.

---

## Embracing Change with Squeak: Extreme Programming (XP)

### Test Thoroughly and Integrate Continuously

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. Focusing on testing not only directs us to our goal, but also defuses territorial behavior amongst team members. If someone changes “my code” and makes it simpler without breaking any tests, I can't let my sense of ownership lead to conflict. 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?

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). There is a trepidation that the developer feels when embarking upon what could turn out to be a significant change. Automated unit tests assure (tested) program features can be relied upon. After radical refactoring, a click of a button will confirm or deny the fidelity with which we have preserved the system function that predated our changes. In the event that we were not entirely successful, the tests point to the areas that require immediate attention before we proceed with new feature implementation.

Aside from the fact that pairs of programmers (two people, one computer, keyboard, and mouse) do all programming, almost never do we find ourselves working alone on a software project of any significant size. Furthermore, no code is “owned” by a single individual (or even a pair of individuals) on an XP project. 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 one at a time. This way, 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 doing 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.

---

### *Scale*

One of the essential notions to keep in mind when using the XP process is the concept 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 so 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 in number. 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.

---

## Embracing Change with Squeak: Extreme Programming (XP)

If you are working alone, consider getting someone that 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

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 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.

Squeak is an “open source” project. This fact is both a blessing and a 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.

---

### *Continuous Integration*

Continuous integration is a process ensuring 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 of the machines that are available to the development team. A build directory is created on this machine. At least two images are kept in this 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 call this a build list file. The released change sets 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. When the image is first started up, the programmers create a new change set with a name that identifies the programming task that is 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.

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

If 100 percent of the unit tests do not pass, 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.

---

***Full System Build***

A full system build should be performed after every 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. The next iteration should not begin until all of the unit tests pass.

---

***Reusing Previously Developed Code***

Squeak comes with a ponderously huge class library. Very often, the simplest possible thing that could work involves reusing code that is part of the base class library. Since the Squeak base class library has no SUnit tests, what should a conscientious programming team do?

The answer is, it depends. If the code being reused is known to be stable and have survived extensive reuse, then there probably is no compelling need to slow development to develop an SUnit test. Examples of this would be the core Collection and Magnitude classes. 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 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. Over time, these test cases should be aggregated into a full SUnit test suite.

---

***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 in practice is less than satisfactory 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 any 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 pop up under the

---

## Embracing Change with Squeak: Extreme Programming (XP)

active hand and not the hand playing back the remote events. For development that is heavily user interface-centric it, it would be well worth defining a task to allow these popups to open under the playback hand rather than the user hand.

[A section containing concrete examples of reuse and code development will be included here]