Java™ Extreme Programming Cookbook

By Eric M. Burke, Brian M. Coyner

Publisher: O'Reilly
Pub Date: March 2003
ISBN: 0-596-00387-0
Pages: 288

Brimming with over 100 "recipes" for getting down to business and actually doing XP, the *Java Extreme Programming Cookbook* doesn't try to "sell" you on XP; it succinctly documents the most important features of popular open source tools for XP in Java-- including Ant, Junit, HttpUnit, Cactus, Tomcat, XDoclet-- and then digs right in, providing recipes for implementing the tools in real-world environments.

Printed in the United States of America.

Published by O'Reilly & Associates, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly & Associates books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (http://safari.oreilly.com). For more information, contact our corporate/institutional sales department: (800) 998-9938 or corporate@oreilly.com.

designations have been printed in caps or initial caps. The association between the image of a bison and the topic of Java programming is a trademark of O'Reilly & Associates, Inc.

While every precaution has been taken in the preparation of this book, the publisher and authors assume no responsibility for errors or omissions, or for damages resulting from the use of the information contained herein.

Dedication

For Jennifer, Aidan, and Tanner

—Eric M. Burke

For Mom and Dad

—Brian M. Coyner

Preface

Anyone involved with the open source community or using open source software knows there are tons of tools available on the market. Keeping up with these tools, and knowing which tools to use and how to use them, is an intimidating road to travel. We hope to simplify your journey by showing concise, useful recipes for some of the more popular open source Java tools on the market today.

We show you tools like JUnit, JUnitPerf, Mock Objects (more of a concept), and Cactus for testing Java code. We show how to generate EJB files using XDoclet, too. All tools discussed in this book are completely executable through Ant, allowing for a complete and stable build environment on any Java-enabled platform.

This is also a book about Extreme Programming (XP), which led us to choose the tools that we did. The XP software development approach does not depend on a particular set of tools; however, the right tools certainly make following XP practices easier. For instance, test-first development is a cornerstone of XP, so most of the tools in this book are testing frameworks. XP also demands continuous integration, which is where Ant fits in. We are big fans of automation, so we cover the XDoclet code generator as well as describe ways to automate deployment to Tomcat and JBoss.

Audience

This book is for Java programmers who are interested in creating a stable, efficient, and testable development environment using open source tools. We do not assume any prior knowledge of XP or the tools covered in this book, although we do assume that you know Java. The chapters generally open with simple recipes and progress to more advanced topics.

About the Recipes

This book is a collection of solutions and discussions to real-world Java programming problems. The recipes include topics such as writing JUnit tests, packaging and deploying server-side tests to application servers, and generating custom code using XDoclet. Each recipe follows the same format. A problem and brief solution is presented, followed by in-depth discussion.

You can find the code online at http://www.oreilly.com/catalog/jextprockbk/.

Organization

This book consists of 11 chapters, as follows:

Chapter 1

> This chapter provides a quick overview of each tool covered in this book. It also explains how the tool selection relates to XP.

Chapter 2

> This chapter explains many key concepts of XP.

Chapter 3

> This chapter is a beginner's overview to Ant, a portable Java alternative to *make* utilities.

Chapter 4

> This chapter provides in-depth coverage of JUnit, the most popular testing framework available for Java.

Chapter 5

> This chapter shows how to use HttpUnit for testing web applications.

Chapter 6

> This chapter explains techniques for using mock objects for advanced testing.

Chapter 7

> This chapter describes how to test servlets, filters, and JSPs running in a servlet container. This is the only tool in this book devoted to *in-container testing* (tests that execute in a running server).

Chapter 8

> This chapter shows how to use JUnitPerf, a simple and effective tool for writing performance tests around existing JUnit tests. This chapter also discusses how to use JUnitPerfDoclet, which is a custom XDoclet code generator created specifically for this book.

Chapter 9

> This chapter shows how to use the XDoclet code generator. In addition to showing how to generate EJB code, we show how to create a custom code generator from the ground up. This code generator is used to generate JUnitPerf tests and is aptly name JUnitPerfDoclet.

Chapter 10

This chapter shows how to incorporate Tomcat and JBoss into an XP build environment. Tomcat is the official reference implementation of the servlet specification and JBoss is arguably the most popular open source EJB container.

Chapter 11

This chapter introduces additional open source tools that are gaining popularity but were not quite ready for their own chapters.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Used for Unix and Windows commands, filenames and directory names, emphasis, and first use of a technical term.

`Constant width`

Used in code examples and to show the contents of files. Also used for Java class names, Ant task names, tags, attributes, and environment variable names appearing in the text.

`Constant width italic`

Used as a placeholder to indicate an item that should be replaced with an actual value in your program.

`Constant width bold`

Used for user input in text and in examples showing both input and output.

Comments and Questions

Please address comments and questions concerning this book to the publisher:

O'Reilly & Associates, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
(800) 998-9938 (in the United States or Canada)
(707) 829-0515 (international/local)
(707) 829-0104 (fax)

There is a web page for this book, which lists errata, examples, or any additional information. You can access this page at:

http://www.oreilly.com/catalog/jextprockbk/

To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

For information about books, conferences, Resource Centers, and the O'Reilly Network, see the O'Reilly web site at:

http://www.oreilly.com

Acknowledgments

This is my third book, and I find myself consistently underestimating how much time it takes to write these things. I have to extend a big thanks to Brian for helping bring this book to completion. Without his help, I don't think I could have done this.

My family is the most important part of my life, and I want to thank Jennifer, Aidan, and Tanner for supporting and inspiring me, even though I spend way too many hours working. I love you all.

—Eric Burke, December 2002

I would like to thank Eric for bringing me on board to write this book. Without his support and trust, I would not have received this wonderful opportunity.

My Mom and Dad have provided years of support and encouragement. I appreciate everything you have done for me, and I love you both very much. Without you I would not be where I am today.

And Grandpa, you are in my heart and prayers every day. You would be so proud. I wish you were here to see this.

—Brian Coyner, December 2002

We both want to thank our editor, Mike Loukides, for helping mold this book into reality. An infinite amount of thanks goes to the tech reviewers: Kyle Cordes, Kevin Stanley, Bob Lee, Brian Button, Mark Volkmann, Mario Aquino, Mike Clark, Ara Abrahamian, and Derek Lane.

Chapter 1. XP Tools

This is a book about open source Java tools that complement Extreme Programming (XP) practices. In this chapter, we outline the relationship between programming tools and XP, followed by a brief introduction to the tools covered in this book. Our approach to tool selection and software development revolves around three key concepts: automation, regression testing, and consistency among developers. First, let's explain how tools relate to XP.

1.1 Java and XP

XP is a set of principles and practices that guide software development. It is an agile process in that it makes every effort to eliminate unnecessary work, instead focusing on tasks that deliver value to the customer.[1] XP is built upon four principles: simplicity, communication, feedback, and courage, all described in Chapter 2. The four XP principles have nothing to do with programming languages and

tools. Although this book shows a set of Java tools that work nicely with XP, you are not limited to Java and these tools. XP is a language-independent software development approach.

[1] Check out http://www.agilealliance.com to learn more about agile processes.

While XP works with any language, we believe it works well with Java for a few reasons. Most important is the speed with which Java compiles. XP relies on test-first development in which programmers write tests for code before they write the code. For each new feature, you should write a test and then watch the test run and fail. You should then add the feature, compile, and watch the test run successfully. This implies that you must write a little code, compile, and run the tests frequently, perhaps dozens of times each day. Because Java compiles quickly, it is well suited to the test-first approach.

The second reason Java is a good choice for XP development is Java's wealth of tools supporting unit testing and continuous integration. JUnit, covered in Chapter 4, provides a lightweight framework for writing automated unit tests. Ant, the premier build tool for Java, makes continuous integration possible even when working with large development teams. You will also find more specialized testing tools such as Cactus and HttpUnit for server-side testing.

Java's power and simplicity also make it a good language when writing code using XP. Many features of the tools outlined in this book, such as Ant tasks and JUnit's test suites, are built upon Java's reflection capability. Java's relatively easy syntax makes it easier to maintain code written by other team members, which is important for XP's concepts of pair programming, refactoring, and collective code ownership.

1.2 Tools and Philosophies

Creating great software is an art. If you ask a dozen programmers to solve a particular problem, you are likely to get a dozen radically different solutions. If you observe how these programmers reach their solutions, you will note that each programmer has her own favorite set of tools. One programmer may start by designing the application architecture using a UML CASE tool. Another may use wizards included with a fancy IDE, while another is perfectly content to use Emacs or vi.

Differences in opinion also manifest themselves at the team and company level. Some companies feel most comfortable with enterprise class commercial tools, while others are perfectly happy to build their development environment using a variety of open source tools. XP works regardless of which tools you choose, provided that your tools support continuous integration and automated unit testing. These concepts are detailed in the next chapter.

We are very skeptical of the term "enterprise class." This tends to be a marketing ploy, and actually means "the features you really need," such as integrated support for free tools like JUnit, Ant, and CVS.

1.2.1 The IDE Philosophy

Many commercial IDEs focus very heavily on graphical "wizards" that help you automatically generate code, hide complexity from beginners, and deploy to application servers. If you choose such a tool, make sure it also allows for command-line operation so you can support continuous integration and automated unit testing. If you are forced to use the graphical wizards, you will be locked into that vendor's product and unable to automate your processes. We strongly recommend XDoclet, Covered

in Chapter 9, for automated code generation. This is a free alternative to wizard-based code generation and does not lock you into a particular vendor's product.[2]

[2] XDoclet allows you to generate any kind of code and thus works with any application server.

This book does not cover commercial development environments and tools. Instead, we show how you can use free tools to build your own development environment and support XP practices. Perhaps in a sign of things to come, more and more commercial development environments now provide direct support for the open source tools covered in this book. With free IDEs like Eclipse and Netbeans growing in popularity and functionality, you will soon be hard-pressed to justify spending thousands of dollars per developer for functionality you can get for free.[3]

[3] Both authors use IntelliJ IDEA, a commercial IDE available at http://www.intellij.com. Although it costs around $400, we feel its refactoring support easily adds enough productivity to justify the cost.

1.2.2 Minimum Tool Requirements

Regardless of whether you choose to use open source or commercial tools, XP works most effectively when your tool selection supports the concepts mentioned at the beginning of this chapter. These three concepts are automation, regression testing, and consistency among developers.

1.2.2.1 Automation

XP requires automation. In an XP project, programmers are constantly pairing up with one another and working on code from any given part of the application. The system is coded in small steps, with many builds occurring each day. You simply cannot be successful if each programmer must remember a series of manual steps in order to compile, deploy, and test different parts of the application.

People often talk about "one-button testing" in XP, meaning that you should be able to click a single button—or type a simple command like `ant junit`—in order to compile everything, deploy to the app server, and run all tests.

Automation also applies to repetitive, mundane coding tasks. You increase your chances of success by identifying repeating patterns and then writing or using tools to automatically generate code. Automation reduces chances for human error and makes coding more enjoyable because you don't have to spend so much time on the boring stuff.

1.2.2.2 Regression testing

Automated regression testing is the building block that makes XP possible, and we will spend a lot more time talking about it in the next chapter. Testing, most notably unit testing, is tightly coupled with an automated build process. In XP, each new feature is implemented along with a complementary unit test. When bugs are encountered, a test is written to expose the bug before the bug is fixed. Once the bug is fixed, the test passes.

Tools must make it easy for programmers to write and run tests. If tests require anything more than a simple command to run, programmers will not run the tests as often as they should. JUnit makes it easy to write basic unit tests, and more specialized testing frameworks make it possible to write tests for web applications and other types of code. JUnit has been integrated with Ant for a long time, and most recent IDEs make it easy to run JUnit tests by clicking on a menu and selecting "run tests."

1.2.2.3 Consistency among developers

In a true XP environment, developers are constantly shuffling from machine-to-machine, working with new programming partners, and making changes to code throughout a system. To combat the chaos that might otherwise occur, it is essential that tools make every developer's personal build environment identical to other team members' environments. If Alex types `ant junit` and all tests pass, Andy and Rachel should also expect all tests to run when they type the same command on their own computers.

Providing a consistent environment seems obvious, but many IDEs do not support consistent configuration across a team of developers. In many cases, each developer must build his own personal project file. In this world it becomes very difficult to ensure that Andy and Rachel are using the same versions of the various JAR files for a project. Andy may be using an older version of *xalan.jar* than everyone else on the team. He may then commit changes that break the build for everyone else on the team for a few hours while they figure out the root of the problem.

1.3 Open Source Toolkit

Open source tools have been with us for a long time, but did not always enjoy widespread acceptance within the corporate environment. This has all changed in the past few years, as free tools became increasingly powerful and popular. In many cases, open source tools have no commercial equivalent. In others, commercial tools have embraced open source tools due to popular demand—although you may have to purchase the most expensive enterprise edition to get these features. This is ironic because Ant and JUnit are free.

In this section, we introduce the tools used throughout this book. While we have no reason to suggest that you avoid commercial tools, we believe you can achieve the same end result without an expensive investment in tools.

1.3.1 Version Control

Version control tools are an essential building block of any software development project, so much so that we assume you are familiar with the basic concepts. We do not cover any specific tool in this book; however, we do want to spend a few moments pointing out how tools like CVS fit into an XP toolkit.

CVS keeps a master copy of each file on a shared directory or server, known as the repository. The repository keeps a history of all changes to each file, allowing you to view a history of changes, recover previous revisions of files, and mark particular revisions with tag names. In a nutshell, tools like CVS allow an entire team to update and maintain files in a predictable, repeatable way.

Each programmer has a copy of the entire code base on his computer, and makes changes locally without affecting other programmers. When the current task is complete, the programmer commits the modified files back to the CVS repository. The newly revised files are then visible to other programmers, who can choose to update to the new revisions whenever they are ready.

Regardless of whether you are using CVS or some other tool, two key principles always apply. These principles are:

- Work in small steps.
- Stay in sync with the shared repository.

Because of the pair programming required by XP, working in small steps is a necessity. You cannot switch programming partners several times per day if you work on tasks that take several days to complete. A key to XP success is your ability to break down a project into smaller tasks that can be completed within a few hours. Working in small steps also helps when using CVS (or any other version control tool) because your personal workspace does not get too far out of sync with the repository.

With CVS, multiple programmers on the team may work on the same files concurrently. When this happens, you must merge changes and resolve conflicts before committing your modified code to the repository. The best way to minimize potential for conflicts is to perform frequent updates. If a programmer does not get the latest code for days and weeks at a time, she increases the likelihood of conflict with work done by other team members.

While CVS allows concurrent edits to the same files, other version control tools force programmers to lock files before making changes. While exclusive locking seems safer than concurrent editing, it can impede development if other team members are unable to make changes. Again, working in small steps is the best way to avoid problems when working with locking version control tools. If each programmer only locks a few files at a time, the likelihood of lock contention is greatly reduced.

1.3.2 Ant

Ant, covered in Chapter 3, is a Java replacement for platform-specific *make* utilities. Instead of a Makefile, Ant uses an XML buildfile typically named *build.xml*. This buildfile defines how source code is compiled, JAR files are built, EAR files are deployed to servers, and unit tests are executed. Ant controls all aspects of the software build process and guarantees that all developers have a common baseline environment.

In the XP model, all programmers on the team share code. Programmers work in pairs that constantly shuffle. XP shuns the idea of certain individuals owning different frameworks within a system. Instead, any programmer is allowed to work on any piece of code in the application in order to finish tasks. Shared code spreads knowledge and makes it easier for people to swap programming partners. Sharing code also coerces people into writing tests, because those tests provide a safety net when working in unfamiliar territory.

Ant is important to XP because you cannot afford to have each developer compiling different parts of the system using different system configurations. Individual classpath settings might mean that code compiles for one team member, but fails to compile for everyone else. Ant eliminates this class of problem by defining a consistent build environment for all developers.

Ant buildfiles consist of targets and tasks. Targets define how developers use the buildfile, and tasks perform the actual work, such as compiling code or running tests. You generally begin by writing a basic Ant buildfile with the following targets:

prepare

: Creates the output directories which will contain the generated *.class* files.

compile

: Compiles all Java source code into the executable.

clean

Removes the build directory and all generated files, such as *.class* files.

junit

Searches for all unit tests in the directory structure and runs them. Tests are files following the *Test*.java* naming convention.[4]

[4] You can adopt whatever naming convention you wish; we chose *Test*.java* for this book.

This is a good start, and will certainly be expanded upon later. A critical feature of this Ant buildfile is the fact that it should define its own classpath internally. This way, individual developers' environment variable settings do not cause unpredictable builds for other developers. You should add the Ant buildfile to your version control tool and write a few simple classes to confirm that it runs successfully.

The other developers on the team then get the Ant buildfile from the version control repository and use it on their own computers. We also recommend that you place all required JAR files into version control,[5] thus allowing the Ant buildfile to reference those JAR files from a standard location.

[5] You normally don't put generated code into CVS, but third-party JAR files are not generated by you. Instead, they are resources required to build your software, just like source files.

1.3.3 JUnit

Automated unit testing is one of the most important facets of XP and a central theme throughout this book. JUnit tests are written by programmers and are designed to test individual modules. They must be designed to execute without human interaction. JUnit is not intended to be a complete framework for all types of testing. In particular, JUnit is not well-suited for high-level integration testing.

Instead, JUnit is a programmer-level framework for writing unit tests in Java. Programmers extend from the `TestCase` base class and write individual unit tests following a simple naming convention. These tests are then organized into test suites and executed using a text or graphical test runner.

JUnit is a simple framework for writing tests, and it is easily extended. In fact, JUnit is the starting point for several of the other testing tools covered in this book. From one perspective, the JUnit API is a framework for building other, more sophisticated testing frameworks and tools.

Tools like CVS, JUnit, and Ant become more powerful when everyone on the team uses them consistently. You might want to talk to the other programmers on your team and come up with a set of guidelines for adding new features to your application. The following list shows one such approach for adding a new feature to a system:

1. Update your PC with the latest source files from the CVS repository. This minimizes the chance of conflicts once you are finished.
2. Write a unit test using JUnit. Try to execute one facet of the new functionality that does not yet exist. Or, write a test to expose a bug that you are trying to fix.
3. Run the test using JUnit and Ant by typing **`ant junit`**. The `junit` Ant target is defined with a dependency on the `compile` target, so all code is automatically compiled.

4. After watching the test fail, write some code to make the test pass.
5. Run the test again by typing `ant junit`. Repeat steps 2-5 until the task is complete and all tests pass. The task is complete when you have written tests for anything you think might break and all tests pass.
6. Perform another CVS update to ensure you are up-to-date with all recent changes. This is a critical step because the CVS repository may have changed while you were working on your task.
7. Run `ant clean junit` in order to perform a full build.
8. If all code compiles and all tests pass, commit changes to CVS and move to the next task. Otherwise, go back and fix whatever broke before committing changes to CVS.

It is important for every developer to follow these steps, because you are using XP and practicing pair programming. Each of the team members takes turn pair programming with another person and each of you is allowed to make changes to any part of the code. Because you are all constantly updating a shared code base, you rely on the suite of automated unit tests along with a consistent build environment to immediately catch errors.

Provided everyone follows the process, it is generally easy to fix problems when a test starts failing. Because all tests passed before you made your changes, you can assume that the most recent change broke the test. If you work in small steps, the problem is usually (but not always!) easy to fix.

On the other hand, things get really ugly when a programmer commits changes to CVS without first running the tests. If that programmer's change broke a test, then all other programmers on the team begin to see test failures. They assume that they broke the test, and waste time debugging their own code. For a team of ten programmers, this may mean that ten programmers spend one hour each tracking down the problem, only to find that it wasn't their fault in the first place. Had that first programmer run the tests locally, he may have been able to fix the problem in a few minutes rather than wasting ten hours.[6]

[6] If the shared code base breaks frequently, programmers may begin to ignore the errors. This causes a snowball effect when they quit running tests and check in even more bad code. Pair programming helps avoid these breakdowns in diligence.

1.3.4 HttpUnit and Cactus

HttpUnit, covered in Chapter 5, is a framework for testing web applications. HttpUnit isn't built with JUnit; however, you do use JUnit when testing with HttpUnit. HttpUnit tests execute entirely on the client machine and access a web server by sending HTTP requests. In this fashion, HttpUnit simulates a web browser hitting a web site. Although you typically use JUnit when working with HttpUnit, the tests you write are more correctly considered "functional" tests rather than "unit" tests. This is because HttpUnit can only test a web application from the outside view, instead of unit-testing individual classes and methods.

A closely related tool is Cactus, covered in Chapter 7. Cactus is significantly more complicated than HttpUnit, and is built on top of JUnit. Cactus tests allow for true unit testing of web applications, with specific types of tests for servlets, JSPs, and servlet filters. Unlike HttpUnit, Cactus tests execute on both client and server—simultaneously. The client portion of the test acts something like a web browser, issuing requests to the server portion of the test that acts more like a unit test. The server then sends a response back to the client portion that then interprets the results.

Cactus can also make use of the HttpUnit library for parsing the HTML output from web applications. We'll see how this works in Chapter 7.

1.3.5 JUnitPerf

JUnitPerf, as you might expect, is a performance-testing tool built on top of JUnit. In Chapter 8, we show how to use JUnitPerf to ensure that unit tests execute within certain time limits and can handle expected usage loads. JUnitPerf does not help you find performance problems. Instead, it ensures that tests run within predefined performance limits.

You will often use JUnitPerf to complement commercial profiling tools. You may use a profiling tool to isolate performance bottlenecks in your code. Once you have fixed the bottleneck, you write a JUnitPerf test to ensure the code runs within acceptable time limits. The JUnitPerf test is then automated, and will fail if someone changes code and makes the code slow again. At this point, you probably go back to the profiling tool to locate the cause of the problem.

1.3.6 Application Servers

We round out our overview of tools with a brief mention of two open source server tools, JBoss and Tomcat. JBoss is a free application server supporting EJB, while Tomcat is a servlet and JSP container. The recipes in this book do not show how to use these tools in detail. Instead, we describe how to configure JBoss and Tomcat in order to support automated testing and continuous integration.

The kind of automation we are interested in occurs when you compile code and run tests. As mentioned earlier, you should strive for a simple command that compiles your code and then runs all of your tests. When working with an application server, typing a command like `ant junit` may actually do the following:

1. Compile all code.
2. Build a WAR file.
3. Start Tomcat if it is not already running.
4. Deploy the new WAR file.
5. Run all unit tests, including those written using HttpUnit.
6. Display a summary of the test results.

1.3.7 Setting Up a Build Server

At some point, your team will probably decide to create a build server. This is a shared machine that performs a clean build of the software on a continuous basis. The build server ensures that your application always compiles and that all tests run. The build server is easy to set up if you have been using CVS and Ant all along. For the most part, the build server operates exactly like each developer's PC. At various intervals throughout the day, the build server gets a clean copy of the code, builds the application, and runs the test suite.

Over time, however, you may want to make the build server more sophisticated. For instance, you might want the build server to monitor the CVS repository for changes. The build can start after some files have changed, but should not do so immediately. Instead, it should wait for a brief period of inactivity. The server can then get a clean copy of the sources using a timestamp from sometime during the inactive period. This process ensures that the build server is not getting code while programmers are committing changes to the repository.

You might also want to keep a change log of who changes what between each build, in order to notify the correct developers whenever the build succeeds or fails. We have found that notifying the entire team whenever a build fails is not a good idea because people begin to ignore the messages. With a

change log, you can notify only those people who actually made changes. The developer process then begins to look like this:

- Make a change, following all of the steps outlined earlier.[7]

  [7] In theory, the build shouldn't break if every developer follows the process before checking in code. We have found, however, that people occasionally "break the build" no matter how careful they are. An automated build server helps catch problems right away.

- Commit changes to CVS.
- Wait for an email from the build server indicating whether the build succeeds or fails.

There is a tool that does everything just described. It is called Cruise Control, and is available at http://cruisecontrol.sourceforge.net. Cruise Control works in conjunction with Ant, CVS, and JUnit to perform continuous builds on a server machine. The exact mechanics of setting up a build server vary widely depending on what version-control tool you use, whether you are using Linux or Windows, and what steps are required to build your particular application. The important thing to keep in mind is that builds should become a seamless part of everyday activity on an XP project, ensuring that developers can work without constantly stopping to figure out how to compile software and integrate changes with other team members.

Chapter 2. XP Overview

This chapter provides a quick introduction to the key programming-related XP activities. These activities are the aspects of XP that affect programmers the most.

XP encompasses much more than programming techniques. XP is a complete approach to software development, including strategies for planning, gathering user requirements, and everything else necessary to develop complete applications. Understanding these strategies is essential if you wish to base an entire project on XP.

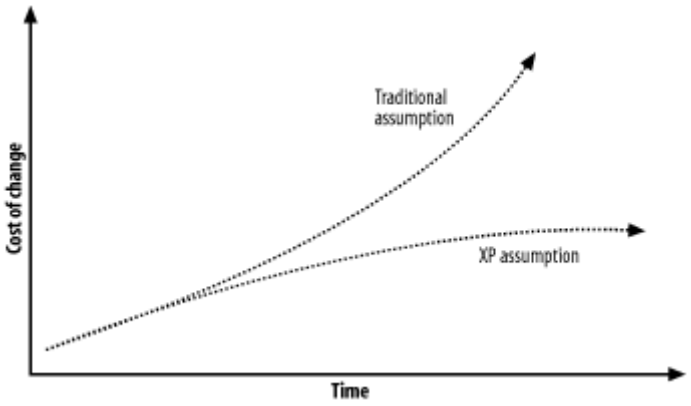
2.1 What Is XP?

XP is based on four key principles: simplicity, communication, feedback, and courage. This section introduces each principle, and the remainder of this chapter touches on each concept where appropriate.

2.1.1 Simplicity

Simplicity is the heart of XP. Applying the principle of simplicity affects everything you do, and profoundly impacts your ability to successfully apply XP. Focusing on simple designs minimizes the risk of spending a long time designing sophisticated frameworks that the customer may not want. Keeping code simple makes changing code easier as the requirements inevitably change. In addition, adopting simple techniques for communicating requirements and tracking progress maximizes chances that the team will actually follow the process. Most importantly, focusing on simple solutions to today's problems minimizes the cost of change over time. Figure 2-1 shows that the intended result of XP practices is to tame the cost of change curve, making it increase much less over time than we would otherwise expect.

Figure 2-1. Cost of change on an XP project



Traditional theory argues that software becomes increasingly expensive to change over the lifetime of a project. The theory is that it is ten times harder to fix a mistake of requirements when you are in the design phase, and 100 times harder to make changes late in a project during the coding phase. There are many reasons. For one, there is more code to change as time goes on. If the design is not simple, one change can affect many other parts of the system. Over time, as more and more programmers change the system, it becomes increasingly complex and hard to understand.

The XP approach recognizes that the cost of change *generally* increases like one would expect, but this increase is not *inevitable*. If you constantly refactor and keep your code simple, you can avoid ever-increasing complexity. Writing a full suite of unit tests is another tool at your disposal, as described later in this chapter. With complete regression testing, you have the ability to make big changes late in the development cycle with confidence. Without these tests, the cost of change does increase because you have to manually test everything else that you may have just broken.

There are other forces in XP projects that balance the rising cost of change. For example, collective code ownership and pair programming ensure that the longer an XP project goes, the better and deeper understanding the whole team has of the whole system.

2.1.2 Communication

Communication comes in many forms. For programmers, code communicates best when it is simple. If it is too complex, you should strive to simplify it until the code is understandable. Although source code comments are a good way to describe code, self-documenting code is a more reliable form of documentation because comments often become out of sync with source code.

Unit tests are another form of communication. XP requires that unit tests be written for a vast majority of the code in a system. Since the unit tests exercise the classes and methods in your application, source code for the tests become a critical part of the system's documentation. Unit tests communicate the design of a class effectively, because the tests show concrete examples of how to exercise the class's functionality.

Programmers constantly communicate with one another because they program in pairs. Pair programming means that two programmers sit at a single computer for all coding tasks; the two share a keyboard, mouse, and CPU. One does the typing while the other thinks about design issues, offers suggestions for additional tests, and validates ideas. The two roles swap often; there is no set observer in a pair.

The customer and programmer also communicate constantly. XP requires an on-site customer to convey the requirements to the team. The customer decides which features are most important, and is always available to answer questions.

2.1.3 Feedback

Having an on-site customer is a great way to get immediate feedback about the project status. XP encourages short release cycles, generally no longer than two weeks. Consider the problems when the customer only sees new releases of your software every few months or so. With that much time in between major feature releases, customers cannot offer real-time feedback to the programming team. Months of work may be thrown away because customers changed their minds, or because the programmers did not deliver what was expected.

With a short release cycle, the customer is able to evaluate each new feature as it is developed, minimizing the necessity to rework and helping the programmers focus on what is most important to the customer. The customer always defines which features are the most important, so the most valuable features are delivered early in the project. Customers are assured that they can cancel the project at any time and have a working system with the features that they value the most.

Code can offer feedback to programmers, and this is where good software development tools shine. In XP, you use unit tests to get immediate feedback on the quality of your code. You run all of the unit tests after each change to source code. A broken test provides immediate feedback that the most recent change caused something in the system to break. After fixing the problem, you check your change into version control and build the entire system, perhaps using a tool like Ant.

2.1.4 Courage

The concepts that were just described seem like common sense, so you might be wondering why it takes so much courage to try out XP. For managers, the concept of pair programming can be hard to accept—it seems like productivity will drop by 50%, because only half of the programmers are writing code at any given time. It takes courage to trust that pair programming improves quality without slowing down progress.[1]

[1] Check out http://www.pairprogramming.com for more information on the benefits of pair programming.

Focusing on simplicity is one of the hardest facets of XP for programmers to adopt. It takes courage to implement the feature the customer is asking for today using a simple approach, because you probably want to build in flexibility to account for tomorrow's features, as well. Avoid this temptation. You cannot afford to work on sophisticated frameworks for weeks or months while the customer waits for the next release of your application.[2] When this happens, the customer does not receive any feedback that you are making progress. You do not receive feedback from the customer that you are even working on the right feature!

[2] The best frameworks usually evolve instead of being designed from scratch. Let refactoring be the mechanism for framework development.

Now, let's look at several specific concepts behind XP in more detail.

2.2 Coding

Coding is an art, and XP acknowledges that. Your success at XP depends largely on your love of coding. Without good code, the exponential cost of change as shown in Figure 2-1 is inevitable. Let's look at some specific ways that XP helps keep code simple.

One of the most frustrating misconceptions about XP is that it is a chaotic approach to software development that caters to hackers. The opposite is true. XP works best with meticulous, detail-oriented programmers who take great pride in their code.

2.2.1 Simplicity

Just getting code to work is not good enough, because the first solution you come up with is hardly ever the simplest possible solution. Your methods may be too long, which makes them harder to test. You may have duplicated functionality, or you may have tightly coupled classes. Complex code is hard to understand and hard to modify, because every little change may break something else in the system. As a system grows, complexity can become overwhelming to the point where your only remaining option is to start over.

When compared to beginners, expert programmers typically implement superior solutions using fewer lines of code. This is a hint that simplicity is harder than complexity, and takes time to master.

Simple code is self-documenting because you pick meaningful names, your methods are concise, and your classes have clearly defined responsibilities. Simple code is hard to achieve, and relies on knowledge in the areas of object-oriented programming, design patterns, and other facets of software engineering.

2.2.2 Comments

If code is self-documenting, do you need source code comments? In short, there will always be cases where you need comments, but you should never write comments simply for the sake of commenting. If the meaning of a method is completely obvious, you do not need a comment. An abundance of comments in code is often an indication that the code is unclear and in need of refactoring. Let's look at a method that needs a comment, and see how to eliminate this need.

```
/**
 * Sets the value of x.
 * @param x the horizontal position in pixels.
 */
public void setX(int x) {
    this.x = x;
}
```

This method needs a comment because the meaning of "x" is not entirely clear. Over time, the comment might not be kept in sync if someone changes the method's implementation or signature. But what if we rename things to make the code more clear? How about this:

```
public void setXPixelPosition(int xPixelPosition) {
```

```
    this.xPixelPosition = xPixelPosition;
}
```

This code no longer needs a comment because it is self-documenting. As a result, we end up typing a little bit more for the method declaration, but save a few lines of comments. This helps us out in the long run because we don't have to spend time and effort keeping the comment in sync with the code. Long method names do not degrade performance in any appreciable way, and are easy to use thanks to code-completion features found in any modern IDE.

2.2.3 Pair Programming

As mentioned earlier, XP teams work in pairs. These pairs of programmers share a single computer, keyboard, and mouse. Having dual monitors is a good idea because both programmers can then see the screen clearly, although this is not a requirement. Desks should be configured so that two people can sit side-by-side comfortably, and the entire team should work in the same room.

Here is how pair programming works:

1. You pick out a user story[3] for your next task.

   [3] A *user story* is a requirement from the customer. Stories are typically written on index cards, and the customer decides which stories are the most important.

2. You ask for help from another programmer.
3. The two of you work together on a small piece of functionality.
   - Try to work on small tasks that take a few hours.
   - After the immediate task is complete, pick a different partner or offer to help someone else.

By working on small tasks, partners rotate frequently. This method facilitates communication between team members and spreads knowledge. As mentioned earlier, writing simple code is hard, and experienced programmers are generally better at it. By pairing people together, beginners can gain valuable coding experience from the experts.

Pair programming is critical because XP requires a very high degree of discipline in order to be successful. As we will learn in the next section, programmers must write unit tests for each new feature added to the application. Writing tests takes a great deal of patience and self-discipline, so having a partner often keeps you honest. When you start to get lazy about writing tests, it is the partner's job to grab the keyboard and take over.

When you have control of the keyboard, you are thinking about the code at a very fine-grained level of detail. When you are not the partner doing the typing, you have time to think about the problem at a higher level of abstraction. The observer should look for ways to simplify the code, and think about additional unit tests. Your job is to help your partner think through problems and ultimately write better code.

2.2.4 Collective Code Ownership

XP teams do not practice individual code ownership. Every team member is able to work on any piece of code in the application, depending upon the current task. The ability to work on any piece of code in an application makes sense when pairs of programmers are constantly shuffling and re-pairing