

/THEORY/IN/PRACTICE

O'REILLY®

Practical Development Environments



Matthew B. Doar

Project Basics

T HIS CHAPTER BRIEFLY DESCRIBES THE DIFFERENT PARTS OF A PROJECT and then introduces the main activities involved in software development and the corresponding tools that make up a development environment. The activities are software configuration management (SCM), building software, testing software, tracking bugs, writing documentation, releasing products, and maintenance. This chapter ends with some personal recommendations of tools for three different types of development environments.

Whether you are starting a project from scratch or looking to improve an existing development environment, my opinion is that *you should consider the different parts of the environment in the same order used in this chapter* and in this book. That is, SCM is the most important part of an environment; next in importance are the build tools, then testing and bug tracking, and so on. This is because your choice of SCM tool is likely to have the largest impact on your environment. This is not to say that any of the parts are unimportant, just that improving how your SCM tool and build tool are used will probably improve your environment more than improving the bug tracking tool or release process. Similarly, if you are creating a new project, an SCM tool should be chosen before a bug tracking tool.

The Parts of a Project

Figure 2-1 shows the different parts of a single project. This project has source files that become products, and those products have customers who use the products. The project has project members (developers, testers, technical writers, toolsmiths, managers, and product marketing staff), who are also customers for their own products. The development environment for this project is made up of both the tools and the local processes and policies for using those tools.

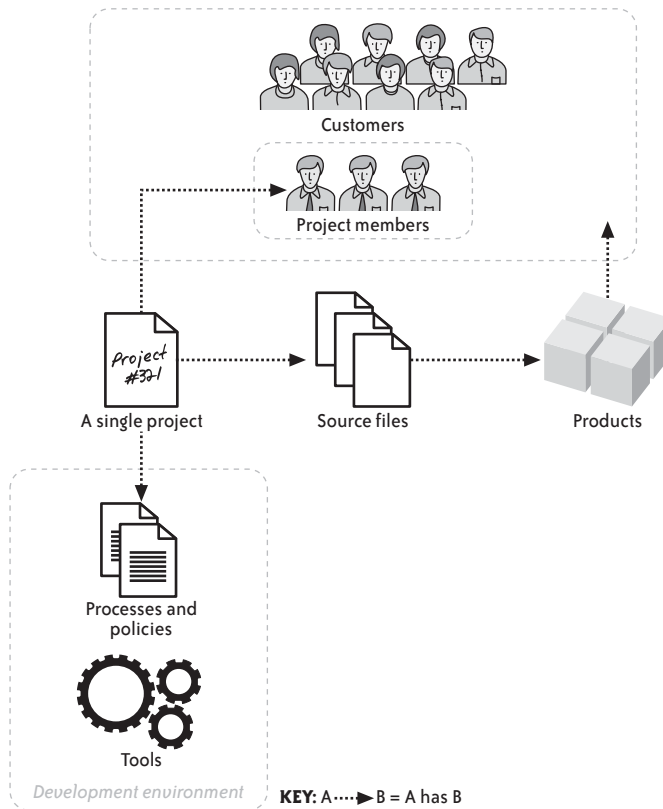


FIGURE 2-1. Different parts of a project

If there are many projects within an organization, then each project will have some areas that overlap and some that don't. For instance, two projects may have different policies about how to use the same bug tracking tool. The processes and tools are chosen to work with the source files but need to be able to operate even when the same source files are used by different projects with different policies.

Software Configuration Management

SCM, the subject of Chapter 4, is the ability to keep track of different versions of the source files that are used to create software products. Even when just one person is changing these source files, it's very useful to be able to see how a file evolved over time or even to undo some changes that in hindsight you regret making. As projects grow in size and complexity, effective SCM tools become vital. To put it another way, projects that don't use an SCM tool won't grow successfully.

SCM tools also provide a good way to share your work with other people in a controlled manner. Rather than just using a common location such as a directory to exchange files with other people, with an SCM tool you can make sure that interdependent files are changed together and you can control who is allowed to make changes. SCM tools also allow you to save messages about what changed in the source files, and also why the files were changed, which can be used to work out which releases a particular bug was fixed in.

Many SCM tools provide ways to support one or more existing releases of a product, while still allowing the team to develop the next release using different versions of the same source files. For instance, after a product is released, all the source files for that release can be marked (or even *branched*) to allow future bug fixes for that release, while the next release is developed independently.

Chapter 4 describes all of the above in more detail and examines seven of the most commonly used or promising SCM tools: CVS, Subversion, Arch, Perforce, BitKeeper, ClearCase, and Visual SourceSafe.

Building Software

Software products are built from their source files, also known as *source code*, which is often the collection of files stored in an SCM tool. A build tool uses the source files and follows some specified build rules to run other tools such as compilers to create the product from the source files. These build rules are usually specified in configuration files known as *build files*, which are part of the source files for a product.

Build tools have to be aware of which products can be built from a given set of source files and other files such as libraries. They also have to know which parts of the product depend on which other parts, so that if one source file is changed by a developer, then all the other affected parts will also be rebuilt. Build tools should be able to execute the correct commands for building the same product on different platforms, while hiding the idiosyncrasies of each platform from the product's developers as much as possible. Build tools are also used to generate executables that run on platforms other than the platform that the build tool is actually run on; this is known as *cross-compiling*.

Chapter 5 describes what to look for in a build tool and examines some commonly used build tools such as *make*, Ant, Jam, and SCons in more detail.

Testing Software

Testing a product spans the range from individual developers writing unit tests that check small parts of an application, to system tests that use the whole application, to customers giving feedback about how they really want the product to work. Chapter 6 focuses on testing environments for unit tests and system tests.

One of the cornerstones of the XP (extreme programming) methodology is the importance of extensive unit tests, written even before the functionality that they test has been written. No matter what your methodology or style of programming is, a healthy test suite is good for reassuring yourself that your latest changes haven't broken some other distant part of the product.

Of course, running all those different tests and interpreting their results quickly becomes tedious; if testing is not automated, it is often postponed and then finally abandoned. So it's important when developing and maintaining a product to have a test environment where tests can be added easily and run easily, and where the test results can be clearly understood.

Tracking Bugs

Testing a product provides information about which parts of it are working and which are not. This information needs to be made available to developers, other testers, managers, the people that decide when a product is ready for release, and also to those who support a product. Bug tracking tools are commonly used to do this. Bugs are sometimes referred to as *issues*, because they are often requests for changes or some other category; some other terms, particularly *defect* and *incident*, can have legal implications and are best avoided if possible. The term *bug* is used colloquially throughout this book to refer to all these categories.

Bug tracking tools often store information about bugs in a database and then provide convenient GUIs and command-line interfaces (CLIs) for adding information about bugs to the database, changing the information recorded about bugs, and creating reports about different kinds of bugs. At a minimum, a recorded bug has a description of the bug (what happened, what should have happened) and an identifier that is unique for each different bug. Other information that is frequently recorded with each bug includes who found it, the steps to reproduce it, who is working on it now, which releases the bug exists in, and which releases it has been fixed in.

Many bug tracking systems define a number of states and allow each bug to be in just one state at a time. This is intended to help guide the workflow of the team when they are working on different bugs. For instance, a new bug may be in the Open state, then move to an Accepted state, then to a Fixed state, and then to a Closed state. This is part of applying a *change management* process (see "What SCM Is and Is Not" in Chapter 4) to how you want bugs to be fixed.

Tracking bugs is a lot more than just not forgetting what still needs fixing in a product. For good and bad, it becomes a way to measure development and testing progress toward the next version. Observing the numbers of bugs in different states over time can play a part in deciding when a product is stable enough to ship. It is also a rather simplistic way to measure how busy individuals are—for instance, by the number of bugs they have assigned to them. As such, bug totals often take on a meaning far beyond a simple record of the problems in a project. Bug tracking tools can even become a way to avoid communicating directly with other project members, as described in “Twisted Communications” in Chapter 12.

Chapter 7 discusses many more aspects of bug tracking and examines some of the more commonly used bug tracking systems: spreadsheets, Bugzilla, GNATS, FogBugz, JIRA, and TestTrack.

Writing Documentation

An ideal product is so transparent that it needs (almost) no documentation. Software products that are as well-designed and well-implemented as this Platonic ideal are rare indeed, so documentation is an expected part of all products. Whether the documentation is a simple *README* file, a large manual that ships with the product, or interactive help available when the product is used, the contents still have to be written by somebody, usually a technical writer.

Chapter 8 takes the viewpoint that the documentation is part of the product and that there are plenty of similarities between writing software and writing documentation. For instance, large documents are made up of smaller ones; different parts of a document depend on other parts of the document; and both source code and natural-language documents can benefit from tools, whether they are compiler warning flags or spellcheckers.

Another similarity between software and documentation is that both are transformed from one file format into other file formats as part of being released. A source file is compiled to an executable. A document is often written in one source file format (such as Microsoft Word or FrameMaker) and then converted to another release format (such as HTML or PDF, Adobe’s Portable Document Format) for use by the customer. Some commonly used documentation formats are discussed in “File Formats for Documentation” in Chapter 8 (HTML, PostScript, and PDF) and “More File Formats” in Chapter 8 (TEX, Texinfo, *troff*, and POD).

A different aspect of documentation is when it is intended for use by a project itself. An example of this is when the APIs (application programming interfaces) of different parts of a product are documented to help other developers use them. There are a number of tools, such as Javadoc, to help with this, and these are also examined in “Internal Project Documentation” in Chapter 8.

Chapter 8 also examines in more detail a variety of file formats and their related documentation environments, including raw text, FrameMaker, XML in DocBook and OpenOffice, and Microsoft Word.

Releasing Products

Once a product has been developed, tested, and documented, it eventually becomes ready to release. You want to release the software according to some predetermined plan. Too often, however, products just escape into the hands of customers because no one considered issues such as release numbers and license keys before releasing the product. Chapter 9 describes all of this.

With the increase in malicious software, assuring customers that the files they download are actually the same as the ones that you released is becoming an essential part of releasing software products. The use of digital signatures and checksums to help with this is discussed further in “Securing Your Releases” in Chapter 9.

Since installing a product is often a customer’s first experience of the software, and first impressions count, the installation process is important. Chapter 9 also examines the most common packaging formats and installation tools for a variety of platforms, as well as some common irritations with installation tools and the installers that they produce.

Maintenance

Maintenance of a product after it has been released takes up a large part of a product’s life span. Chapter 10 describes some typical product maintenance activities and how the tools in a development environment can help you with them.

How to maintain a development environment is also discussed, including what kinds of things stop working as an environment ages and how to know when to throw tools and files away.

Recommended Tools

This section contains my personal recommendations for tools for development environments. The recommendations are intended for projects with less than 1 million lines of source code and under 200 people involved in developing, testing, documenting, and releasing the product. The annual budget for tools probably ranges from zero to \$100,000. These choices are purely personal ones made from the tools available in 2005, with no undue influences from any individual companies or projects.

NOTE

If you use a tool that you feel is much better than one of the tools I’ve recommended, feel free to send me email about it via bookquestions@oreilly.com. My own contact details are available at <http://www.pobox.com/~doar>.

IDE recommendations are also welcome, but rants about editors (the programs, not the people) are generally unproductive—use one that does the job for you, and learn it well.

If these recommendations are enough for you to make progress with a development environment, that's great! Reading the sections about each tool later in the book is still a good idea to get some more background, especially "Choosing New Tools" in Chapter 3.

However, a development environment is more than just its tools. The discussions of the best practices and annoyances of each area in the chapters that follow will help you use each of these tools in a more productive manner.

Modern Environments

This list of tools is for environments that can afford the effort of using tools that are still themselves being developed. Some reading of mailing lists and weblogs, and possibly some local development of the tools by a toolsmith, may be necessary.

SCM tool

Subversion ("Subversion" in Chapter 4) with FishEye (<http://www.cenqua.com/fisheye>)

Build tool

Ant ("Ant" in Chapter 5) for projects using Java™; SCons ("SCons" in Chapter 5) for most projects and other languages

Test environment

xUnit ("xUnit" in Chapter 6)

Bug tracker

JIRA ("JIRA" in Chapter 7)

Documentation

Anything that uses an XML source file format with an open DTD or schema; examples include OpenOffice and DocBook ("XML: DocBook and OpenOffice" in Chapter 8)

Classic Environments

This list of tools is for environments that want to use tools that have been stable for a number of releases and have an extensive support network. These are the tools that you can buy more than one book about.

SCM tool

CVS ("CVS" in Chapter 4) with FishEye (<http://www.cenqua.com>); alternatively, Perforce ("Perforce" in Chapter 4)

Build tool

Ant ("Ant" in Chapter 5) for projects using Java; otherwise, *make* ("make" in Chapter 5)

Test environment

xUnit ("xUnit" in Chapter 6)

Bug tracker

FogBugz ("FogBugz" in Chapter 7), but only if the preconfigured settings work for you; otherwise, TestTrack ("TestTrack" in Chapter 7)

Documentation

FrameMaker ("FrameMaker" in Chapter 8)

Future Environments

This is a list of how I foresee development environment tools changing in the next five years. Most of these tools don't exist yet, though the foundations for them certainly do. An important question for future development environments will be how well each of the tools is integrated with the other tools—SCM with bug tracking is one example—and how much of the process of using them can be automated.

SCM tool

BitKeeper (“BitKeeper” in Chapter 4) or Arch (“Arch” in Chapter 4), but with better integration with bug tracking systems and a clearer view of recent changes.

Build tool

SCons (“SCons” in Chapter 5), with mappings from other languages so that you can write Perl or Java build files as well as Python build files. Support for parallel builds on multiple machines as well.

Test environment

More extensions to the xUnit architecture (“xUnit” in Chapter 6) to support system tests and historical reports of test results.

Bug tracker

A bug tracking system based on an SCM tool, so that every single change to the whole system is recorded. Better built-in support for a bug that appears in multiple releases of a product is also long overdue. Better integration with build tools, so that it's easier to know which bugs were fixed in which releases. Support for changing bugs while disconnected from a network could also be useful.

Documentation

I would hope that a real typesetting tool such as T_EX and all the concepts of literate programming will come back into style one day. Until then, anything that uses an XML source file format with an open DTD or schema.

Release

Better automatic deployment of software products, along with any other pieces of required software. Doing this both atomically and efficiently. Better integration of installation tools with build processes.

As an imaginative finale, with the increased importance of licensing for software, one useful option for a compiler might be the ability to understand various common licenses in source files. Functions from each source code file could then be identified by their license type, and only fully license-compatible libraries and executables would be created. In fact, there are already companies such as Black Duck Software (<http://blackducksoftware.com>) and Palamida (<http://palamida.com>) that provide tools to enforce what is referred to as “software compliance management.”