**Chapter 7. Design and Programming**

**There's a famous quote attributed to Kent Beck,** a widely respected software engineer who's responsible for many advances in the field: "I'm not a great programmer; I'm just a good programmer with great habits." This chapter is about introducing some of those great habits. A good programmer who adopts these habits will build better software.

Programmers spend their time designing and building software, and all of their project work revolves around the source code. But many programming teams find that they lose control of their own code. Sometimes they lose track of the changes that they make; new additions might occasionally disappear, and old bugs routinely pop up. They might lose control of the design of the code, finding that no matter how much care they put into designing the software well, they still end up with messy code that's difficult to maintain. Some programmers have never known any other way, and don't realize that these problems can be eased. A project manager can improve the code by helping the team adopt good programming practices.

While many development problems originate outside of the programming team, there are some basic changes that the programmers can make that will improve the quality of the code they produce. Most teams, even ones with skilled and talented programmers, are vulnerable to the same design and programming problems. These problems can be addressed with a few basic tools and techniques—which can often be put in place entirely within the programming team, without involving anyone else in the organization.

A programming team that adopts the tools and techniques in this chapter will avoid many common project pitfalls. This chapter only covers their most fundamental use—there are many advanced operations that can be done with all of them that are not covered here. With these practices in place, the team will find that they have a much firmer grasp on their own source code. If these tools are not yet in place, working with the programmers to implement them should be high on the project manager's priority list.

Much of this chapter is an introduction to refactoring and unit testing. Many experienced programmers will recognize that refactoring and unit testing are important skills that require practice, reading, and training. We tried to write this book in such a way that someone could read each chapter and immediately use the tools and techniques on a project. Unlike the other practices, however, we don't expect the reader to be able to fully develop unit tests or refactor code after reading this chapter, any more than we would expect to be able to give a thorough presentation of object-oriented programming in one section of a chapter. Instead, our goal is to give the reader an introduction to the theory and practice of both refactoring and unit testing, illustrated with simple examples of each.

NOTE

We encourage anyone interested in developing unit testing and refactoring skills to read these excellent books:

* *Pragmatic Unit Testing in Java with JUnit* or *Pragmatic Unit Testing in C# with NUnit* by Andy Hunt and Dave Thomas (The Pragmatic Programmers, 2004).
* *Refactoring: Improving the Design of Existing Code* by Martin Fowler (Addison Wesley, 1999), and Martin Fowler's web site [http://www.refactoring.com](http://www.refactoring.com/).

**Review the Design**

In many project teams, the programmers begin coding as soon as they have a software requirements specification that everyone agrees on. Typically, the first programming tasks usually involve building a user interface that supports each use case, and creating an object model that implements each of the functional requirements.

The designers and programmers have several options:

* The programmers can simply start building the code and create the objects and user interface elements.
* Designers can build a user interface prototype to demonstrate to the users, stakeholders, and the rest of the team. Any code used to develop the prototype is typically thrown away once the design has been finalized.
* Pictures, flow charts, data flow diagrams, database design diagrams, and other visual tools can be used to determine aspects of the design and architecture.
* An object model can be developed on paper, either using code, simple class diagrams, or Unified Modeling Language (UML) diagrams.
* A written design specification is written, which includes some or all of these tools.

In order to ensure that the software is designed well, the project manager works with the team during the creation of the work breakdown structure and estimates (see [Chapter 3](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch03.html)) to determine which of these options is appropriate for the project. The design tasks should be estimated and included in the project schedule; this requires that the team agree on a single design approach from the outset of the project.

The design tasks should always include reviews, even when there is no written design specification. If the programmers dive immediately into the user interface design, they should hold a walkthrough with the users. This may involve taking screenshots of the UI and turning them into a slide presentation, or it may mean that the programmers give a thorough demonstration of the user interface.

Any written documentation should be reviewed and, if possible, inspected. However, it is important that the reviews and inspections reach the correct audience. Many users who have important input for the user interface may be uninterested or confused by object models and UML diagrams. If all of these elements are bundled into a single design specification, it is important that the scope of each review is clear—it's reasonable to have individual people review only the part of the design specification that they are interested in.

# Version Control with Subversion

The purpose of a version control system is to bring a project's source code under control.[[\*](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch07s02.html" \l "ftn.appliedprojectmgmt-CHP-7-FN1)] The main element of the version control system is the repository , a database or directory that contains each of the files that make up the system. Bringing a group of files under control means that someone can pick a point at any time in the history of the project and see exactly what those files looked like at the time. It is always possible to find the latest version of any file by retrieving it from the repository. Changing a file will not unexpectedly overwrite any previous changes to that file; any change can be rolled back, so no work will accidentally be overwritten. Modern version control systems can identify exactly what changed between two different versions of a file, and allow the team to roll back those changes—even if they were made a long time ago, and the affected files have had many modifications since then.

This is very important for source code. When source code files are stored in a shared folder, for example, it is easy for changes to be lost when more than one programmer is working on the software. Even a single programmer, working on source code that resides in a folder on his computer, can run into problems. He might make a large change, only to realize that he needs to roll it back. Or he may find that he's got several copies of the code on his hard drive and laptop, and that each of them contains a different set of changes that all need to be integrated.

Subversion is a free and open source version control system. It is available from [http://subversion.tigris.org](http://subversion.tigris.org/) for many operating systems and platforms, including Linux, BSD, Solaris, BeOS, OS/2, Mac OS X, and Windows. Subversion provides many advanced features for bringing source code under control, but it takes only a few basic commands for a programming team to use a simple version control repository effectively.

The examples below are based on Version 1.2, the latest version available at the time of this writing. The remainder of this chapter assumes that this version (or later) is installed on the programmer's machine. The Subversion installation procedure does not install a graphical user interface—all functions are accessed through command-line programs. There are graphical utilities and web interfaces that work with Subversion, including TortoiseSVN ([http://tortoisesvn.tigris.org](http://tortoisesvn.tigris.org/)), ViewCVS ([http://viewcvs.sourceforge.net](http://viewcvs.sourceforge.net/)), and SmartSVN (<http://www.smartcvs.com/smartsvn>).

This is the only section in this book in which a specific software package is recommended. However, Subversion is not the only option for version control. Other popular version control systems include:

* CVS (<http://www.nongnu.org/cvs>)
* RCS (<http://www.gnu.org/software/rcs>)
* Arch (<http://www.gnu.org/software/gnu-arch/>)
* Aegis (<http://aegis.sourceforge.net/>)
* Perforce (<http://www.perforce.com/>)
* Visual SourceSafe (<http://msdn.microsoft.com/vstudio/previous/ssafe/>)

## Multiple People Can Work on One File

Programming teams that do not use a modern version control system usually have a rule that only one person at a time can work on any given file. Sometimes this rule is enforced by using an antiquated version control system, which allows only one person to check out a file. Some teams use a master folder to store all of the source code, and individual programmers must rename files that they are using or move them temporarily to a "checkout" folder.

There are version control systems that essentially function as an automated checkout folder. A system like this requires that a programmer check out a file from the repository before modifying it, and keeps track of the checkout status of each file in the repository. The system prevents anyone from modifying any file until it is checked back in. This is known as the "lock-modify-unlock" model, and it is very intuitive to many programmers. However, it is a very restrictive way to manage code, and can cause delays and problems for the team.

One worst-case scenario that some teams encounter is a schedule delay caused by having many programming tasks that must occur on a single piece of code, when that code can be updated by only a single person at a time. For example, programmers might run into trouble when they must add behavior to a complex window in the software that contains many tabs and controls, all of which reside in a single file. If they are only using a folder to store the code, only one person can work on the file at a time. Even if the modifications themselves are relatively straightforward, this process could take a lot of time, and may even require a senior developer to be pulled in to perform the work as quickly as possible (which could lead to extra delays because he can't be assigned to any other task until this one is done.)

When a team has to work with large files that contain a lot of code that is not under control, or that is checked into a lock-modify-unlock version control system, it runs into "unavoidable" delays because only one person can edit each file at a time. Adopting a modern version control system like Subversion is one effective way to fix this problem. Subversion allows multiple people to work on a single file at the same time, using an alternative model called "copy-modify-merge." In this model, a file can be checked out any number of times. When a programmer wants to update the repository with his changes, he retrieves all changes that have occurred to the checked out files and reconciles any of them that conflict with changes he made before updating the repository. (See below for details about how copy-modify-merge works.)

Many programmers who have never worked with a copy-modify-merge version control system will find it counterintuitive. The idea that multiple people can use their own working copies to work on a snapshot of the code may seem like an invitation to disaster. It seems intuitive to him that programmers will step on one another's toes all the time, constantly overwriting changes and eventually creating an unmanageable mess.

In practice, the opposite is true: it turns out that it is usually easy to merge changes, and that very few changes conflict. Code is almost always built in such a way that the functionality is highly encapsulated in functions. And even within individual functions, there are small, independent blocks of code. Even if two neighboring blocks are altered at the same time, it is rare for a conflict to be introduced.

Copy-modify-merge is very efficient. Teams lose much more time from schedule bottlenecks caused by waiting for files that are checked out than they do from trying to figure out how to merge the changes. Giving multiple people the ability to merge changes into a shared repository is an important way to eliminate those bottlenecks. Version control systems based on the copy-modify-merge model have been used for years on many projects. This is especially true for large teams or teams in which the members are distributed over a large geographical area—their work would grind to a halt waiting for people to check code back in.

## Understanding Subversion

The Subversion repository contains a set of files laid out in a tree structure (similar to the file systems in most operating systems), with folders that contain files and other folders, as well as links or shortcuts between files. The main difference is that the Subversion file system tracks every change that is made to each file stored in it. There are multiple versions of each file saved in the repository. The files in the repository are stored on disk in a database, and can only be accessed using the Subversion software.

A Subversion repository has a revision number. This revision number gets incremented every time a change is made to the repository—this way, no revision number is used more than once. The standard convention for writing revision numbers is a lowercase "r" followed by the number of the revision; all output from Subversion will adhere to this convention.

The Subversion repository keeps track of every change that is made to every file stored in it. Even if a file is modified by adding or removing lines, or if that file is removed entirely, all previous revisions still can be accessed by date or number. Subversion can also generate a list of differences between any two revisions of a file.

For example, consider a repository that contains Revision 23 of the file cookies.txt in [Table 7-1](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch07s02.html#appliedprojectmgmt-CHP-7-TABLE-1), which was checked in on March 4.

*Table 7-1. cookies.txt r23 as it looked on March 4*

|  |  |  |
| --- | --- | --- |
| 2 1/4 cups flour | 1 tsp baking soda | 1/2 tsp salt |
| 1 cup butter | 3/4 cup sugar | 3/4 cup brown sugar |
| 1 tsp vanilla | 2 eggs | 2 cups chocolate chips |
| Mix butter and sugars. Beat until creamy. Add vanilla. Beat in eggs, one at a time. Mix in rest of ingredients. Drop by spoonfuls on cookie sheets. Bake at 350 degrees. | | |

Subversion uses global revision numbers, which means that the revision number applies to the entire tree in the repository. Any time that a change is made to any file in the repository, the global revision number of the repository is incremented. [Table 7-1](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch07s02.html#appliedprojectmgmt-CHP-7-TABLE-1) shows a file as it looked on March 4, when the repository was at Revision 23. Some people may refer to this as "Revision 23 of cookies.txt," but what they really mean is "cookies.txt as it appeared in Revision 23 of the repository." If someone updates a different file in the repository on March 5, the repository's revision number will increment to 24. When that happens, there is now an r24 of cookies.txt—it just happens to be identical to r23.

Now, let's say that time has passed since the file was checked in. It's May 13, and other people have added several other files to this repository, so the repository is now at r46. If someone (let's call her Alice) decides to check out cookies.txt and add the line that calls for chopped nuts, when she commits her change the repository increments to r47, and this revision, r47, will contain the new version of cookies.txt in [Table 7-2](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch07s02.html#appliedprojectmgmt-CHP-7-TABLE-2).

*Table 7-2. cookies.txt r47 committed by Alice on May 13 (with chopped nuts added)*

|  |  |  |
| --- | --- | --- |
| 2 1/4 cups flour | 1 tsp baking soda | 1/2 tsp salt |
| 1 cup butter | 3/4 cup sugar | 3/4 cup brown sugar |
| 1 tsp vanilla | 2 eggs | 2 cups chocolate chips |
| 1 cup chopped nuts |  |  |
|  |  |  |
| Mix butter and sugars. Beat until creamy. Add vanilla. Beat in eggs, one at a time. Mix in rest of ingredients. Drop by spoonfuls on cookie sheets. Bake at 350 degrees. | | |

The next time someone checks the tree that contains cookies.txt out of the repository, they will get r47 of the file. However, they can specifically ask for a previous version in one of several ways:

* Specifically ask for r23 of cookies.txt.
* Ask for any revision of cookies.txt between, say, 21 and 43. For example, requesting r36 will retrieve the revision associated with r36 of the repository—and since the file has not changed since r23, this is the revision that will be retrieved.
* Request a copy of cookies.txt as it looked on April 6.

Any of these requests will yield the version of the recipe without the nuts in [Table 7-1](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch07s02.html#appliedprojectmgmt-CHP-7-TABLE-1). Even if the file is later deleted from the Subversion repository, previous revisions are still available and can be retrieved by revision number or date. (This means that the file is never permanently deleted from the repository. When a file is "deleted," the repository simply no longer lists it in the current revision. If a user checks out an old revision from before the file was removed, the version of the file associated with that revision is still available.)

## Check Out Code into a Working Copy

Before a Subversion repository can be accessed, it must be checked out. Checking out a repository simply means retrieving a snapshot of the repository and copying it to the user's local machine. The repository is not altered in any way when files are checked out. The local copy of the repository is called a working copy. This is the core of the copy-modify-merge model: any changes that the user needs to make are done in the working copy, which must be brought up to date before it can be checked back in.[[\*](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch07s02.html#ftn.appliedprojectmgmt-CHP-7-FN2)]

Everybody gets their own working copy; one person can have any number of working copies. When files are checked out from a repository, the Subversion client creates a new working copy for those files. Usually a programmer will check out only one copy of a given directory or tree at a time, but there are occasions when a programmer will have several working copies that contain different snapshots of different parts of the repository. (Only directories or trees can be checked out—Subversion does not allow a programmer to check out a single file in a directory.)

There can be many working copies of the same code, even on the same machine. Each working copy keeps track of both who checked out the files and when they were checked out. The user can retrieve the differences between the working copy and the revision that he had originally checked out at any time, even if the user's machine does not currently have access to the server from which the working copy was checked out.

Additionally, a programmer can check out all of a repository, or just part of it. The checkout can include a single folder or an entire branch including all subfolders. (It can also include the entire repository, but in Subversion this is almost never done, because it can cause an enormous amount of data to be retrieved.) When the files are checked out, the latest revision of each file is copied to the working copy. When the programmer looks in the working copy, she sees all of the files and folders that she checked out, plus another hidden folder called .svn. This folder contains all of the data that Subversion uses to check for differences and to keep track of the state of the working copy.

The programmer edits the files in the working copy, and, when she is satisfied that the files in the working copy are ready, she can commit them back to the repository. It is only during the commit that the repository is updated to look like the working copy. The commit is the step that the programmer takes to finalize the changes and integrate them back into the source code stored in the repository. When someone commits changes to the repository, Subversion updates only those files that have changed since the working copy was checked out. Once changes have been committed, they will be accessible to anyone looking at the repository.

Subversion performs an atomic commit , which means that either all of the changes go into the repository completely, or none of them make it at all. This way, the programmer does not need to worry about only half of the changes making it into the repository. (Some older version control systems suffered from that problem.)

Since multiple people can each check out their own working copies, they can all work on the same file simultaneously. There is no limit to the number of times a file can be checked out at once; any part of the repository can be checked out into multiple working copies on multiple machines at the same time. If several people have files checked out of a repository, it is likely that there are earlier versions of some files in some of their working copies, and later versions in other ones.

If the repository has not changed since the working copy was checked out, then the user can simply commit the change, and Subversion will update the repository. However, if the repository has changed, Subversion will not allow the user to commit the changes just yet—it will tell the user that the working copy is out-of-date. At this point, it is up to the user to bring the working copy up-to-date by telling Subversion to update the working copy. Subversion will then retrieve any changes that have been committed to the repository since the working copy was checked out.

Often, most of the changes that have been committed to the repository since the working copy was checked out are mergeable, meaning that none of the lines that changed in the repository have changed in the working copy. When this happens, Subversion will update all of the files in the working copy to reflect all of the changes that have been committed to the repository.

However, if a conflict occurs, it is up to the user to resolve it. A conflict occurs when two people (let's call the second person Bob) introduce different changes to the same line in the same file in their separate working copies. A conflict occurs when Alice and Bob both need to make different changes to the same lines in the same file. They both check out the file and make changes to it, and Alice commits it back to the repository first. But when Bob attempts to commit the change to the repository, Subversion discovers that there is a conflict. That conflict must be resolved in Bob's working copy before it can be committed to the repository.

To illustrate this idea, consider what would happen to the previous example (cookies.txt r47 in [Table 7-2](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch07s02.html#appliedprojectmgmt-CHP-7-TABLE-2)) if Alice committed her changes under different circumstances. Suppose she checked out r23 (see [Table 7-1](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch07s02.html#appliedprojectmgmt-CHP-7-TABLE-1)), but Bob checked in r38 (shown in [Table 7-3](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch07s02.html#appliedprojectmgmt-CHP-7-TABLE-3)) before she could commit her changes.

*Table 7-3. cookies.txt r38 committed by Bob on April 26 (with M&Ms added)*

|  |  |  |
| --- | --- | --- |
| 2 1/4 cups flour | 1 tsp baking soda | 1/2 tsp salt |
| 1 cup butter | 3/4 cup sugar | 3/4 cup brown sugar |
| 1 tsp vanilla | 2 eggs | 2 cups chocolate chips |
| 1 cup chopped M&Ms |  |  |
|  |  |  |
| Mix butter and sugars. Beat until creamy. Add vanilla. Beat in eggs, one at a time. Mix in rest of ingredients. Drop by spoonfuls on cookie sheets. Bake at 350 degrees. | | |

This version contains a change: Bob added chopped M&Ms to the recipe. This change occurred on April 26, a few weeks before Alice checked in r47. So when it was committed, Subversion did not complain; at the time, there were no conflicts. It simply allowed the change and was able to update the repository without requiring any input from the user.

When May 13 rolls around and Alice tries to commit r47 in [Table 7-2](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch07s02.html#appliedprojectmgmt-CHP-7-TABLE-2), Subversion discovers that the working copy is out of date (because the revision number of the repository is greater than it was when the working copy was checked out). She updates the working copy to incorporate any changes. When she does this, Subversion detects the conflict in line 4: the version in her working copy contains chopped nuts, while the version in the repository contains M&Ms. Alice will then be required to decide which version of that line should be kept; once she makes that decision, the file can be committed to the repository. (See below for more details about how she specifies this.)

This is how the working copy allows two people to check out the same file, and even alter the same lines in that file. They did not have to coordinate with each other, and the first person was never even aware that a conflict occurred. It's up to each person to resolve the individual conflicts that arise. What's more, a full audit trail is available, so if it turns out that someone made the wrong choice, then that choice can be undone later and it will be possible to figure out who made the mistake.

## Access the Subversion Repository Using a URL

A Subversion repository exists on a computer as a folder. It's easy to recognize a Subversion repository: it usually contains subfolders named conf, dav, db, hooks, and locks. These folders contain different important elements of the repository: for example, the conf folder contains configuration files, the db folder contains a database that stores all of the files and revision history, and the hooks folder contains scripts that can be triggered automatically when Subversion performs certain actions.

There are two commands that are installed with Subversion that programmers will typically use to access and maintain the repository. The repository administration tool svnadmin performs actions directly on a repository. It can create a new repository, verify its contents, make a copy of an entire repository, dump the repository to a portable format, recover from a database error, and perform other administrative tasks. svnadmin always works on a local directory—when a user executes it, the repository is always passed as a path to a local folder, in whatever format the operating system expects.

The command-line client tool svn is the main interface that most programmers use to check out and update files in the repository. The svn command-line client can access a repository using any of several different methods: accessing the folder directly on the hard drive or a shared drive, connecting to the svnserve server, using svnserve server over SSH, or using Apache to serve the repository via HTTP or HTTPS.

The svn client uses a URL instead of a simple path, in order to differentiate between the different kinds of access methods. This way, the programmer using it does not need to change the way she works in order to accommodate different kinds of access methods. [Table 7-4](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch07s02.html#appliedprojectmgmt-CHP-7-TABLE-4) shows the schema for each access method.

*Table 7-4. Repository access URLs (reprinted with permission from Version Control with Subversion)*

| **Schema** | **Access method** |
| --- | --- |
| file:/// | Direct repository access (on local disk)  Example: file:///usr/svn/repos/ |
| http:// | Access via WebDAV protocol to Subversion-aware Apache server |
| https:// | Same as http://, but with SSL encryption |
| svn:// | Access via custom protocol to an svnserve server |
| svn+ssh:// | Same as svn://, but through an SSH tunnel |

For the most part, Subversion's URLs use the standard syntax, allowing server names and port numbers to be specified as part of the URL. One difference is that the file:/// access method can be used only for accessing a local repository—if a server name is given, it should always be file://localhost/. It should always be followed by the full path of the repository:

file:///usr/local/svn/repos

If the repository contains folders, those folders can be appended to the repository. (These are the virtual folders contained within the repository, not the physical folder that contains the repository database.) For example, to access /recipes/cookies.txt in the repository /usr/local/svn/repos, the client must be passed the following URL:

file:///usr/local/svn/repos/recipes/cookies.txt

The file:/// URL scheme on Windows platforms uses the unofficially "standard" syntax for specifying a drive letter by using either X: or X|. Note that a URL uses ordinary slashes, even though the native (non-URL) form of a path on Windows uses backslashes. It is important to use quotes so that Windows does not interpret the vertical bar character as a pipe:

C:\>svn checkout file:///C:/Documents and Settings/Ed/Desktop/repo

C:\>svn checkout "file:///D|/svn/repository/path/to/file"

Most of the Subversion examples in this chapter will use UNIX-style command syntax and pathnames.

## Create a Repository

The Subversion installation procedure does not create a repository—that must be done using the svnadmincommand.

Use svnadmin create to create an empty repository:

svnadmin create REPOSITORY\_PATH

The svnadmin command requires that the repository path be specified in the operating system's native format. On a UNIX system, the following commands will create a repository in the /usr/local/svn/repos/ directory:

$ mkdir /usr/local/svn

$ svnadmin create /usr/local/svn/repos

On a Windows system, svnadmin uses a Windows-style path to create the repository. The following commands will create a repository in the c:\svn\repos\ directory. Note that this is a raw path, and not a file:///URL:

C:\>mkdir c:\svn

C:\>svnadmin create c:\svn\repos

A shared folder on a network is all it takes to set up a repository that multiple programmers can use. A single programmer can do the same on her own machine to store her own work in a local repository.

Once the empty repository is created, an initial set of files can be added to it using the svn import command, which allows files to be added to a repository without checking it out:

svn import [PATH] URL

There are several ways that the directories in a repository can be structured. One easy way to do it is to have a root-level folder for each project. Each project folder contains three subfolders: trunk, tags, and branches. The working version of the source code is stored in the trunk folder. (The other two folders are reserved for more advanced version control activities, which are beyond the scope of this book.)

For example, to import a project called hello-world that contains two files, hello-world.c and hello-world.h, they should be put into a folder:

$ mkdir tempimport

$ mkdir tempimport/hello-world

$ mkdir tempimport/hello-world/trunk

$ mkdir tempimport/hello-world/tags

$ mkdir tempimport/hello-world/branches

**[copy hello-world.c and hello-world.h into the import/hello-world/trunk directory]**

$ svn import tempimport/ file:///usr/local/svn/repos/ -m "Initial import"

Adding tempimport/hello-world

Adding tempimport/hello-world/trunk

Adding tempimport/hello-world/trunk/hello-world.c

Adding tempimport/hello-world/trunk/hello-world.h

Adding tempimport/hello-world/branches

Adding tempimport/hello-world/tags

Committed revision 1.

Note that a log message reading "Initial import" was passed to svn using the -m flag. If this is omitted, Subversion will launch the default editor to edit the log message. The environment variable SVN\_EDITOR can be used to change the default editor. (Many Windows users prefer to set this variable to notepad.)

The Subversion repository is now ready for use. Additional files can be added either by using the svnadmin import command or by checking out a branch of the repository and using the svn add command.

## Share the Repository

A Subversion repository can be accessed by programmers using any of the access methods listed in [Table 7-4](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch07s02.html#appliedprojectmgmt-CHP-7-TABLE-4). The simplest way to do this is to use a shared folder and the file:/// URL schema. However, this is not secure and may be problematic to share on an intranet or over the Internet.

Running a server is very straightforward using the svnserve program. Many people think that the word "server" is synonymous with "administrative headache." In fact, setting up and running a Subversion server is simple, and can be done on any machine where a repository is installed. The following command will run a read-only server:

$ svnserve -d -r /usr/local/svn/repos/

It works equally well on a Windows platform:

C:\>svnserve -d -r c:\svn\repository\

If this is running on a machine called [http://mybox.example.com](http://mybox.example.com/), the repository can be accessed using the URL <svn://mybox.example.com>. (Note: At the time of this writing, svnserve does not work with Windows 95/98/ME.)

By default, the server is read-only, which means that it only allows users to retrieve files, not to commit changes. However, it does not take much work to turn on password authentication. First a password file must be created in the conf folder in the Subversion repository. In this example, the file passwords.txt contains the following lines:

[users]

andrew = mypassword

jenny = anotherpw

Then the following lines must be added to the end of svnserve.conf in the same folder:

[general]

anon-access = read

auth-access = write

password-db = passwords.txt

Now when the svnserve command listed above is executed, it will run a server that supports authentication. The username and password can be passed to svn on the command-line with the —username and —passwordflags. When the repository is checked out, the working copy remembers the authentication information, so the username and password only need to be supplied when files are checked out.

$ svn checkout svn://servername/trunk --username andrew --password mypassword

A trunk/hello-world.c

A trunk/hello-world.h

Checked out revision 1.

Note that in this example, since no destination folder name is given, it checks out the working copy into a folder named trunk, which corresponds to the path given in the URL.

A third way to share files is using svnserve tunneled over SSH. All that is required here is that Subversion be installed on a computer that is running an SSH server. A programmer just needs to pass the correct URL schema to svn. Subversion establishes an SSH connection and executes svnserve on the server to connect to the specified repository. Note that this example uses the abbreviation co on the command line:

$ svn co svn+ssh://user@server/usr/local/svn/repos/hello-world/trunk/ hello-world

Password: \*\*\*\*\*

A hello-world/hello-world.c

A hello-world/hello-world.h

Checked out revision 1.

The http:// and https:// URL schemas require that an Apache server be set up to work with Subversion using WebDAV and mod\_svn. (Configuring an Apache server to support access to a Subversion repository is beyond the scope of this book.)

## The Subversion Basic Work Cycle

The authors of Version Control with Subversion (O'Reilly) recommend a basic work cycle that programmers use on a day-to-day basis to do their work. This work cycle is described in the script in [Table 7-5](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch07s02.html#appliedprojectmgmt-CHP-7-TABLE-5).

*Table 7-5. Subversion basic work cycle script*

| **Name** | **Subversion basic work cycle** |
| --- | --- |
| Purpose | Programmers use this work cycle for their day-to-day programming tasks. |
| Summary | Subversion has numerous features and options, bells, and whistles, but most programmers use only the most important ones in the course of a day's work. |
| Work Products | Input  Source code in a Subversion repository  Output  A new revision of the source code has been added to the repository. |
| Entry Criteria | The programmer needs to modify code in the repository. |
| Basic Course of Events | 1. Update the working copy of the code using svn update. 2. Make changes to the code by altering the files in the working copy and using svn add, svn delete, svn move, and svn copy. 3. Examine all changes that were made using svn status and svn diff. 4. Use svn update to incorporate any changes that were made to the repository since the working copy was checked out. Resolve changes made by other people since the working copy was checked out by using svn resolved. 5. Commit the changes by using svn commit. |
| Alternative Paths | 1. In Step 1, if there is no working copy yet, a new copy is checked out by using svn checkout. 2. In Step 3, changes can be undone by using svn revert. This does not alter the repository, only the working copy. 3. The programmer can discard all changes in the working copy by simply never committing them to the repository and deleting the working copy. |
| Exit Criteria | The repository is updated with the modified code. |

### UPDATE THE WORKING COPY OF THE CODE

When a programmer needs to modify the code for a project, the first thing to do is to bring his working copy up to date so that it reflects all the changes that are in the latest revision. If he does not yet have a working copy, he can check one out of the repository with the svn checkout command.

The programmer supplies the URL to the repository and, optionally, a path to check it out into. If no path is specified, Subversion makes a new subdirectory in the current directory and checks it out there. The following action will check out the "trunk" branch from the example above:

$ svn checkout svn://servername/hello-world/trunk hello-world --username andrew

A hello-world/trunk

A hello-world/trunk/hello-world.c

A hello-world/trunk/hello-world.h

A hello-world/branches

A hello-world/tags

Checked out revision 1.

Since the username andrew was given during the checkout, the working copy will remember that username. Before any changes are committed to the repository, it will ask the user for a password. To avoid this, the —password flag can be used to specify the password. (It is important for the URL to contain the trunkfolder—otherwise, Subversion will grab a copy of every branch and tag in the repository, which could be an enormous amount of data.) The third parameter passed to the svn command is the checkout folder—in this case, the user specified hello-world in order to check out the contents of the trunk into a folder called hello-world. Without this parameter, Subversion would instead check out the working copy into a folder called trunk.

The svn update command brings the working copy up-to-date. If someone has committed the file hello-world.c since it was checked out and that file has not been altered in the working copy, the following command will update it in the working copy:

$ svn update hello-world

U hello-world/trunk/hello-world.c

Updated to revision 2.

The svn update command can also be called from anywhere inside the working copy. In that case, the path should be omitted from the command line:

$ cd hello-world/trunk

$ svn update

U hello-world.c

Updated to revision 2.

The letter next to hello-world.c is a code to indicate what action Subversion performed for that file:

* A for a file that was added to the working copy
* U for a file that was updated in the working copy
* D for a file that was deleted
* R for a file that was replaced (meaning that the file was deleted and a new one with the same name was added—which will be considered a new file with a distinct revision history)
* G for a file that was changed and those changes were successfully merged into the working copy
* C for a file that could not be merged because of conflicting changes

### MAKE CHANGES TO THE CODE

Once the working copy is up to date with the latest revision, the programmer can make changes. Generally, these changes will be done using whatever editor or IDE the programming team has always used. Subversion does not require that the working copy is up to date in order for the user to make changes. The repository will generally change while the user is making changes; the programmer will merge these changes into the working copy before committing it back to the repository.

Sometimes files need to be added. For example, the programmer might add a file called Makefile to the hello-world project. Subversion won't recognize the file if it is simply added, so the programmer must also let Subversion know that the file is there:

$ cd hello-world/trunk

$ svn add Makefile

A Makefile

This tells Subversion that the file Makefile has been added to the working copy. The programmer must commit the working copy in order to add the file to the repository. The delete, copy, and move commands all work in a similar manner (see [Table 7-6](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch07s02.html#appliedprojectmgmt-CHP-7-TABLE-6)).

*Table 7-6. Commands to make changes to the working copy (reprinted with permission from Version Control with Subversion)*

|  |
| --- |
| svn add foo  Schedule file, directory, or symbolic link footo be added to the repository. When you next commit, foowill become a child of its parent directory. Note that if foo is a directory, everything underneath foo will be scheduled for addition. If you only want to add foo itself, pass the —non-recursive (-N) switch.  svn delete foo  Schedule file, directory, or symbolic link foo to be deleted from the repository. If foo is a file or link, it is immediately deleted from your working copy. If foo is a directory, it is not deleted, but Subversion schedules it for deletion. When you commit your changes, foo will be removed from your working copy and the repository.  svn copy foo bar  Create a new item bar as a duplicate of foo. bar is automatically scheduled for addition. When bar is added to the repository on the next commit, its copy history is recorded (as having originally come from foo). svn copy does not create intermediate directories.  svn move foo bar  This command is exactly the same as running svn copy foo bar; svn delete foo. That is, bar is scheduled for addition as a copy of foo, and foois scheduled for removal. svn move does not create intermediate directories. |

### EXAMINE ALL CHANGES

When the programmer is ready to commit all of his changes to the repository, he should use the svn statuscommand to ensure that his working copy contains only the changes he intends.

$ svn status hello-world/

M hello-world/trunk/hello-world.c

A hello-world/trunk/Makefile

The svn status command generates a list of all of the files in the working copy that have changed since it was checked out. It does not connect to the repository to do this. Instead, it uses the reference copy of the checked-out revision that was stored as part of the working copy.

Each file listed by svn status has a one-letter code next to it. [Table 7-7](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch07s02.html#appliedprojectmgmt-CHP-7-TABLE-7) contains a list of codes that svn status will return. (If the programmer passes a specific path to svn status, it only returns the status of that file.)

*Table 7-7. Output codes for*svn status*command (reprinted with permission from Version Control with Subversion)*

|  |
| --- |
| A item  The file, directory, or symbolic link item has been scheduled for addition into the repository.  C item  The file item is in a state of conflict. That is, changes received from the server during an update overlap with local changes that you have in your working copy. You must resolve this conflict before committing your changes to the repository.  D item  The file, directory, or symbolic link item has been scheduled for deletion from the repository.  M item  The contents of the file item have been modified.  X item  The directory item is unversioned, but is related to a Subversion externals definition.  ? item  The file, directory, or symbolic link item is not under version control. You can silence the question marks by either passing the —quiet (-q) switch to svn status  ! item  The file, directory, or symbolic link itemis under version control but is missing or somehow incomplete. The item can be missing if it's removed using a non-Subversion command. In the case of a directory, it can be incomplete if you happened to interrupt a checkout or update. A quick svn update will refetch the file or directory from the repository, or svn revert file will restore a missing file.  ~ item  The file, directory, or symbolic link itemis in the repository as one kind of object, but what's actually in your working copy is some other kind. For example, Subversion might have a file in the repository, but you removed the file and created a directory in its place, without using the svn delete or svn add command.  I item  The file, directory, or symbolic link itemis not under version control, and Subversion is configured to ignore it during svn add, svn import, and svn status operations. Note that this symbol only shows up if you pass the —no-ignore option to svn status--otherwise the file would be ignored and not listed at all! |

In addition to svn status, the programmer can use svn diff to see the specific changes that have been made to each file. The svn diff command can take a file parameter to generate a list of differences for that file; if no file is given, then it generates a list of differences for every file that was changed. The differences are displayed using the unified diff format.

The svn revert command can be used to roll back changes. If the programmer issues that command and gives it the filename of a file in the working copy, then that file is overwritten with a "pristine" copy that is identical to one in the revision that was checked out of the repository.

### MERGE ANY CHANGES MADE SINCE THE WORKING COPY WAS CHECKED OUT

Before the changes can be checked in, the user should update the working copy. This causes any changes made to the repository since the working copy was checked out to be changed in the working copy as well. The programmer does this by using the svn update command.

Most of the time, the svn update will automatically merge any changes that were made. But occasionally a programmer will have a change in his working copy that overlaps with another change that was made to the repository since he checked out.

In the example above, Alice wanted to update cookies.txt and checked out r23 of the recipe (which contained neither nuts nor M&Ms—see [Table 7-1](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch07s02.html#appliedprojectmgmt-CHP-7-TABLE-1)). She planned on adding nuts to the recipe but, before she could commit that change to the repository, Bob added M&Ms and committed r38 (see [Table 7-2](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch07s02.html#appliedprojectmgmt-CHP-7-TABLE-2)). When Alice attempted to commit her new revision (see [Table 7-3](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch07s02.html#appliedprojectmgmt-CHP-7-TABLE-3)), Subversion gave her the following message:

$ svn commit -m "Added nuts"

Sending cookies.txt

svn: Commit failed (details follow):

svn: Out of date: 'cookies.txt' in transaction '46-1'

This told Alice that she needed to update her working copy in order to integrate any changes by issuing the svn update command:

$ svn update

C cookies.txt

Updated to revision 46.

When Subversion detects a conflict, it updates the file and marks the conflict using a string of greater-than and less-than signs. [Table 7-8](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch07s02.html#appliedprojectmgmt-CHP-7-TABLE-8) shows the conflicts marked in cookies.txt after the update.

*Table 7-8. Subversion found conflicts in cookies.txt*

|  |  |  |
| --- | --- | --- |
| 2 1/4 cups flour | 1 tsp baking soda | 1/2 tsp salt |
| 1 cup butter | 3/4 cup sugar | 3/4 cup brown sugar |
| 1 tsp vanilla | 2 eggs | 2 cups chocolate chips |
| <<<<<<< .mine |  |  |
| 1 cup chopped nuts |  |  |
| ======= |  |  |
| 1 cup chopped M&Ms |  |  |
| >>>>>>> .38 |  |  |
|  |  |  |
| Mix butter and sugars. Beat until creamy. Add vanilla. Beat in eggs, one at a time. Mix in rest of ingredients. Drop by spoonfuls on cookie sheets. Bake at 350 degrees. | | |

The text between <<<<<<<.mine and ======= indicates the changes that were found in the working copy. The text between ======= and >>>>>>>.r38 indicates the conflicting changes that were found in r38 (see [Table 7-3](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch07s02.html#appliedprojectmgmt-CHP-7-TABLE-3)). It is up to the user to choose one or the other of these changes. The user can also come up with a way to use both of them—for example, indicating that the chopped M&Ms are optional.

Once the user has resolved the changes, the svn resolved command is used to indicate that the conflict has been resolved:

$ svn resolved cookies.txt

Resolved conflicted state of 'cookies.txt'

Now Alice can commit the changes to the repository.

### COMMIT THE CHANGES

Once the working copy has been updated and all of the conflicts have been resolved, it is time to commit the changes by using svn commit. In the example above, Alice would issue the following command:

$ svn commit -m "Updated the recipe to add nuts"

Sending cookies.txt

Transmitting file data .

Committed revision 47.

# Refactoring

To refactor a program is to improve the design of that program without altering its behavior.[[\*](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch07s03.html" \l "ftn.appliedprojectmgmt-CHP-7-FN3)] There are many different kinds of improvements—called refactorings— that can be performed.

Every programmer knows that there are many ways to write code to implement one specific behavior. There are many choices that do not affect the behavior of the software but that can have an enormous impact on how easy the code is to read and understand. The programmers choose variable names, decide whether certain blocks of code should be pulled out into separate functions, choose among various different but syntactically equivalent statements, and make many other choices that can have a significant impact on how easy the software is to maintain.

Many programmers think of coding as a purely constructive task, for which the only reason to add, remove, or change the source code is to alter the behavior of the software. Refactoring introduces a new concept: adding, removing, or changing the source code for the sole purpose of making it easier to maintain. There are many different refactorings, or techniques, through which programmers can alter their code to make it easier to understand.

Refactoring is a new way of thinking about software design. Traditionally, software is designed first and then built. This is especially true of object-oriented programming, where the programmers might be handed a complex object model to implement. But most programmers who have worked on a reasonably complex project have run across instances when they discover ways that an object could have been designed better. They could not have predicted most of these improvements because they only became apparent during the construction of the code. Refactoring provides them with a way to incorporate those improvements in a structured, repeatable manner.

Because each refactoring is a change to the design, it may impact the design review process. If the software design has already been reviewed by project team members, then any changes that arise from refactoring activities should be communicated to the people who reviewed it. This does not necessarily mean that design specification must be reinspected after each refactoring; since refactoring changes the design without altering the functionality, it is usually sufficient to distribute just the changes to the design and have the team members approve those changes. In general, people do not object very often to refactoring, but they appreciate being given the opportunity to discuss it and suggest alternatives.

Each refactoring has a set of steps—similar to the scripts we use to describe the tools in this book—which makes it much less likely for the programmer to introduce defects. It also has two design patterns that show what the code looks like before and after the refactoring. There are dozens of refactorings, each with its own particular pattern and steps. The example below demonstrates four of these refactorings: Extract Method, Replace Magic Number with Symbolic Constant, Decompose Conditional, and Introduce Explaining Variable. A comprehensive catalog of refactorings can be found at <http://www.refactoring.com/catalog>.

## Refactoring Example

In this example, a programming team in an investment bank is reviewing a block of code for a feature that calculates fees and bonuses for brokers who sell a certain kind of investment account to their corporate clients. The programmer who wrote the code uses refactoring to clarify problems that were identified during the code review.

The inspection team performed a code review on this block of Java code, which included the class Account and a function calculateFee from a different class:

1 class Account {

2 float principal;

3 float rate;

4 int daysActive;

5 int accountType;

6

7 public static final int STANDARD = 0;

8 public static final int BUDGET = 1;

9 public static final int PREMIUM = 2;

10 public static final int PREMIUM\_PLUS = 3;

11 }

12

13 float calculateFee(Account accounts[]) {

14 float totalFee = 0;

15 Account account;

16 for (int i = 0; i < accounts.length; i++) {

17 account = accounts[i];

18 if ( account.accountType == Account.PREMIUM ||

19 account.accountType == Account.PREMIUM\_PLUS ) {

20 totalFee += .0125 \* ( account.principal

21 \* Math.exp( account.rate \* (account.daysActive/365.25) )

22 - account.principal );

23 }

24 }

25 return totalFee;

26 }

At first, the code seemed reasonably well designed. But as the inspection team discussed it, a few problems emerged. One of the inspectors was not clear about the purpose of the calculation that was being performed on lines 20 to 22. The programmer explained that this was a compound interest calculation to figure out how much interest was earned on the account, and suggested that they use the Extract Method refactoring to clarify it. They performed the refactoring right there during the code review. Since this calculation only used data that was available in the Account class, they moved it into that class, adding a new method called interestEarned (in lines 12 to 15 below):

1 class Account {

2 float principal;

3 float rate;

4 int daysActive;

5 int accountType;

6

7 public static final int STANDARD = 0;

8 public static final int BUDGET = 1;

9 public static final int PREMIUM = 2;

10 public static final int PREMIUM\_PLUS = 3;

11

12 **float interestEarned() {**

13 **return ( principal \* (float) Math.exp( rate \* (daysActive / 365.25 ) ) )**

14 **- principal;**

15 **}**

16 }

17

18 float calculateFee(Account accounts[]) {

19 float totalFee = 0;

20 Account account;

21 for (int i = 0; i < accounts.length; i++) {

22 account = accounts[i];

23 if ( account.accountType == Account.PREMIUM ||

24 account.accountType == Account.PREMIUM\_PLUS )

25 totalFee += .0125 \* account.interestEarned();

26 }

27 return totalFee;

28 }

An inspector then asked what the number .0125 in line 25 was, and if it could ever change in the future. It turned out that each broker earned a commission fee that was equal to 1.25% of the interest earned on the account. They used the Replace Magic Number with Symbolic Constantrefactoring, replacing it with the constant BROKER\_FEE\_PERCENT and defining that constant later in line 31 (and adding a leading zero to help people read the code quickly):

1 class Account {

2 float principal;

3 float rate;

4 int daysActive;

5 int accountType;

6

7 public static final int STANDARD = 0;

8 public static final int BUDGET = 1;

9 public static final int PREMIUM = 2;

10 public static final int PREMIUM\_PLUS = 3;

11

12 float interestEarned() {

13 return ( principal \* (float) Math.exp( rate \* (daysActive / 365.25 ) ) )

14 - principal;

15 }

16 }

17

18 float calculateFee(Account accounts[]) {

19 float totalFee = 0;

20 Account account;

21 for (int i = 0; i < accounts.length; i++) {

22 account = accounts[i];

23 if ( account.accountType == Account.PREMIUM ||

24 account.accountType == Account.PREMIUM\_PLUS ) {

25 **totalFee += BROKER\_FEE\_PERCENT \* account.interestEarned();**

26 }

27 }

28 return totalFee;

29 }

30

31 **static final double BROKER\_FEE\_PERCENT = 0.0125;**

The next issue that was raised in the code review was confusion about why the accountType variable was being checked in lines 23 and 24. There were several account types, and it wasn't clear why the account was being checked for just these two types. The programmer explained that the brokers only earn a fee for premium accounts, which could either be of the type PREMIUM or PREMIUM\_PLUS.

By using the Decompose Conditionalrefactoring, they were able to clarify the purpose of this code. Adding the isPremium function to the Account class (lines 17 to 22) made it more obvious that this was a check to verify whether the account was a premium account:

1 class Account {

2 float principal;

3 float rate;

4 int daysActive;

5 int accountType;

6

7 public static final int STANDARD = 0;

8 public static final int BUDGET = 1;

9 public static final int PREMIUM = 2;

10 public static final int PREMIUM\_PLUS = 3;

11

12 float interestEarned() {

13 return ( principal \* (float) Math.exp( rate \* (daysActive / 365.25 ) ) )

14 - principal;

15 }

16

17 **public boolean isPremium() {**

18 **if (accountType == Account.PREMIUM || accountType == Account.PREMIUM\_PLUS)**

19 **return true;**

20 **else**

21 **return false;**

22 **}**

23 }

24

25 float calculateFee(Account accounts[]) {

26 float totalFee = 0;

27 Account account;

28 for (int i = 0; i < accounts.length; i++) {

29 account = accounts[i];

30 **if ( account.isPremium() )**

31 totalFee += BROKER\_FEE\_PERCENT \* account.interestEarned();

32 }

33 return totalFee;

34 }

35

36 static final double BROKER\_FEE\_PERCENT = 0.0125;

The last problem found during the inspection involved the interestEarned() method that they had extracted. It was a confusing calculation, with several intermediate steps crammed into a single line. When that behavior was buried inside the larger function, the problem wasn't as glaring, but now that it had its own discrete function, they could get a clearer look at it.

The first problem was that it wasn't exactly clear why there was a division by 365.25 in line 13. The programmer explained that in the Account class, daysActive represented the number of days that the account was active, but the rate was an annual interest rate, so they had to divide daysActive by 365.25 to convert it to years. Another programmer asked why principal was being subtracted at the end of the interest calculation. The explanation was that this was done because the fee calculation was based only on the interest earned, regardless of the principal that initially was put into the account.

The refactoring Introduce Explaining Variablewas used to introduce two intermediate variables, years on line 13 and compoundInterest on line 14, to clarify the code:

1 class Account {

2 float principal;

3 float rate;

4 int daysActive;

5 int accountType;

6

7 public static final int STANDARD = 0;

8 public static final int BUDGET = 1;

9 public static final int PREMIUM = 2;

10 public static final int PREMIUM\_PLUS = 3;

11

12 float interestEarned() {

13 **float years = daysActive / (float) 365.25;**

14 **float compoundInterest = principal \* (float) Math.exp( rate \* years );**

15 **return ( compoundInterest - principal );**

16 }

17

18 public boolean isPremium() {

19 if (accountType == Account.PREMIUM || accountType == Account.PREMIUM\_PLUS)

20 return true;

21 else

22 return false;

23 }

24 }

25

26 float calculateFee(Account accounts[]) {

27 float totalFee = 0;

28 Account account;

29 for (int i = 0; i < accounts.length; i++) {

30 account = accounts[i];

31 if ( account.isPremium() ) {

32 totalFee += BROKER\_FEE\_PERCENT \* account.interestEarned();

33 }

34 }

35 return totalFee;

36 }

37

38 static final double BROKER\_FEE\_PERCENT = 0.0125;

After these four refactorings, the inspection team agreed that the new version of this code was much easier to understand, even though it was almost 50% longer.

The code after refactoring must behave in exactly the same way it did beforehand. In general, every refactoring should be combined with an automated test to verify that the behavior of the software has not changed, because it is very easy to inject defects during refactoring. A framework of automated unit tests can ensure that the code behavior remains intact. Luckily, the team already had a set of unit tests. They had to add tests to verify the new Account.isPremium method, but the new code passed all of the other unit tests and the new version of the code was checked in (along with the new tests).

## Refactoring Pays for Itself

Many people are initially uncomfortable with the idea of having programmers do tasks that don't change the behavior of the code. But, like time spent on project planning and software requirements engineering, the time spent refactoring is more than recouped over the course of the project. In fact, refactoring can help a team recover code that was previously written off as an unmaintainable mess, and can also help to keep new code from ever getting to that state.

Refactoring makes intuitive sense, when one considers the main reasons that code becomes difficult to maintain. As a project moves forward and changes, code that was written for one purpose is often extended and altered. A block of code may look pristine when it's first built, but it can evolve over time into a mess. New functionality or bug fixes can turn clear, sensible code into a mess of enormously long and complex loops, blocks, cases, and patches. Some people call this spaghetti code (a name that should make intuitive sense to anyone who has had to maintain a mess like that), but it is really just code whose design turned out not to be all that well suited to its purpose.

The goal of refactoring is to make the software easier for a human to understand, without changing what it does. Most modern programming languages are very expressive, meaning that any one behavior can be coded in many different ways. When a programmer builds the code, he makes many choices, some of which make the code much easier or harder to understand. Each refactoring is aimed at correcting a common pattern that makes the code harder to read. Code that is easier to understand is easier to maintain, and code that is harder to understand is harder to maintain.

In practice, maintenance tasks on spaghetti code are extraordinarily difficult and time-consuming. Refactoring that code can make each of these maintenance tasks much easier. Well-designed code is much easier to work on than poorly designed code, and refactoring improves the design of the software while it is being built. Any programmer who has to maintain spaghetti code should make extensive use of refactoring. It usually takes about as much time to refactor a block of spaghetti code as it does to simply try to trace through it. In fact, many programmers have found that it's much faster to use refactoring to permanently detangle messy code than it is to try to just fix the one problem that popped up at the time

In addition to saving time on programming, refactoring can also help a programmer find bugs more quickly. Poorly designed code tends to have more defects, and tracking these defects down is an unpleasant task. If the code is easier to read and follow, it is easier to find those bugs. And since much of the duplicated code has been eliminated, most bugs only have to be fixed once. Of course, the clearer the code is, the less likely it is that defects get injected in the first place.

There is no hard-and-fast rule about when to refactor. Many programmers find that it's effective to alternate between adding new behavior and refactoring the new code that was just added. Any time a reasonably large chunk of new code has been added, the programmer should take the time to go through it and find any possible refactorings. The same goes for bug fixes—often, a bug is easier to fix after the code that it's in has been refactored.

# Unit Testing

Before a build is delivered, the person or program building the software should execute unit tests to verify that each unit functions properly. All code is made up of a set of objects, functions, modules, or other non-trivial units. Each unit is built to perform a certain function. The purpose of unit testing is to create a set of tests for each unit to verify that it performs its function correctly. Each unit test should be automated: it should perform a test without any input or intervention, and should result in a pass (meaning that the test generated the expected results), failure (the results of the test differed from what were expected), or error (meaning the code reached an error condition before the test could pass or fail). Many people require that unit tests have no dependencies on external systems (networks, databases, shared folders, etc.).

Automated unit testing is a stepping stone to test-driven development . Test-driven development is a programming technique in which a programmer writes the unit tests before he writes the unit that they verify. By writing the tests first, the programmer ensures that he fully understands the requirements. It also guarantees that the tests will be in place, so that they aren't left until after all of the other programming activities are completed (and then possibly dropped, due to schedule pressure).

The main activity in unit testing is creating test cases that verify the software. A test case is a piece of code that verifies one particular behavior of the software. Each test should be able to run without any user input; its only output is whether it passed, failed, or halted due to an error. The test cases for a software project are generally grouped together into suites, where there may be a number of suites that verify the entire software. It's often useful to design the suites so that each one verifies specific units or features; this makes the test cases easier to maintain.

The most common (and effective) way for programmers to do unit testing is to use a framework, a piece of software that automatically runs the tests and reports the results. A framework typically allows a programmer to write a set of test cases for each unit. Most frameworks provide an automated system for executing a suite of unit tests and reporting the results. This allows a full battery of unit tests to be executed automatically at any time, with little or no effort. Unit testing frameworks are available for most modern programming languages.

The framework usually provides some sort of object model, API, or other language interface that provides test cases with functionality for reporting whether the test passed or failed. Most frameworks allow the programmer to indicate which tests are associated with various units, and to group the test cases into suites. [Table 7-9](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch07s04.html#appliedprojectmgmt-CHP-7-TABLE-9) shows some of the test frameworks available for various languages. (This list is by no means exhaustive—there are many other frameworks available for these and other languages.)

*Table 7-9. Test frameworks available for languages*

| **Language** | **Framework name (URL)** |
| --- | --- |
| Java | JUnit ([http://www.junit.org](http://www.junit.org/)) |
| Visual Studio .NET | NUnit ([http://www.nunit.org](http://www.nunit.org/)) |
| C | CUnit ([http://cunit.sourceforge.net](http://cunit.sourceforge.net/)) |
| C++ | CppUnit ([http://cppunit.sourceforge.net](http://cppunit.sourceforge.net/)) |
| SmallTalk | SUnit ([http://sunit.sourceforge.net](http://sunit.sourceforge.net/)) |
| Perl | Test (<http://search.cpan.org/~sburke/Test>) |
| Python | PyUnit ([http://pyunit.sourceforge.net](http://pyunit.sourceforge.net/)) |
| Borland Delphi | DUnit ([http://dunit.sourceforge.net](http://dunit.sourceforge.net/)) |

## Test All of the Code, Test All of the Possibilities

The name "unit test" comes from the fact that each individual unit of code is tested separately. In object-oriented languages like Java, C#, and SmallTalk, the units are objects. In imperative languages like C, the units will correspond to functions or modules; in functional languages like Lisp and SML, the units will generally be functions. (Some languages, like Visual Basic and Perl, can be either imperative or object-oriented.)

It takes multiple tests to verify a single unit. The framework will have a way to build suites of test cases and indicate that they correspond to a specific unit.

A good test verifies many aspects of the software, including (but not limited to) these attributes:

* The unit correctly performs its intended functions.
* The unit functions properly at its boundary conditions (like null or zero values).
* The unit is robust (it handles unexpected values and error conditions gracefully).

Unit tests must be able to run within a developer's test environment. Real-time or production resources like databases, data feeds, input files, and user input are not necessarily available to the test. To get around this limitation, a programmer can use a mock object— an object that simulates a resource that is unavailable at the time of the test. (It is beyond the scope of this book to describe how to implement mock objects.)

## JUnit

JUnit is the unit testing framework for Java. It was created by Erich Gamma and Kent Beck, based on Beck's work with SmallTalk. JUnit has been very influential in the world of unit testing; many unit test frameworks are ported from, or based on, JUnit. The test case examples below are JUnit test cases, which are part of an automated suite of unit tests for the FeeCalculation() function above. The unit tests allowed the programmers doing a code review of this function to successfully refactor it without injecting defects. After each refactoring, they executed the unit tests. If any of them failed, the programmers tracked down the problem and fixed it.

The tests use a few additional commands defined by JUnit to tell the framework whether the unit test passes or fails:

assertEquals([String message], expected, actual [, tolerance])

Causes the unit test to fail if expected is not equal to actual. If tolerance is specified, the equality for floating point numbers is calculated to that tolerance.

assertSame([String message], expected, actual)

Causes the unit test to fail if expected does not refer to the same object as actual .

assertTrue([String message], boolean condition)

Causes the unit test to fail if the Boolean condition evaluates to false.

assertFalse([String message], boolean condition)

Causes the unit test to fail if the Boolean condition evaluates to true.

assertNull([String message], java.lang.Object object)

Causes the unit test to fail if object is not null .

assertNotNull([String message], java.lang.Object object)

Causes the unit test to fail if object is null .

fail([String message])

Causes the unit test to fail immediately.

Each assertion can optionally be given a message. In that case, if the test fails, the message is displayed in the test report generated by the framework. In JUnit, a test that completes without failing is considered to have passed.

Every test in JUnit must be able to be run independently of every other test, and the tests should be able to be run in any order. The individual tests are grouped together into a test case. Each test is a method in a test case object, which inherits from junit.framework.TestCase. (The above assert commands are inherited from this class.) Each test case can optionally have a setUp() function, which sets up any objects or values required for the tests, and a tearDown() function, which restores the environment to its condition before the test was run.

## Unit Testing Example

The examples in this section are the individual test methods from a test case object called testFeeCalculation. There are many tests that would exercise the fee calculation function shown in the Refactoring section above. This example shows six of them. All of them require an instance of the FeeCalculation class, which is set up using this setUp() function:

public FeeCalculation feeCalculation;

public void setUp() {

feeCalculation = new FeeCalculation();

}

The first test simply verifies that the function has performed its calculation and has generated the right result by comparing the output to a known value, which was calculated by hand using a calculator:

public void testTypicalResults() {

Account accounts[] = new Account[3];

accounts[0] = new Account();

accounts[0].principal = 35;

accounts[0].rate = (float) .04;

accounts[0].daysActive = 365;

accounts[0].accountType = Account.PREMIUM;

accounts[1] = new Account();

accounts[1].principal = 100;

accounts[1].rate = (float) .035;

accounts[1].daysActive = 100;

accounts[1].accountType = Account.BUDGET;

accounts[2] = new Account();

accounts[2].principal = 50;

accounts[2].rate = (float) .04;

accounts[2].daysActive = 600;

accounts[2].accountType = Account.PREMIUM\_PLUS;

float result = feeCalculation.calculateFee(accounts);

assertEquals(result, (float) 0.060289, (float) 0.00001);

}

This test passes. The call to feeCalculation() with those three accounts returns a value of 0.060289383, which matches the value passed to assertEquals() within the specified tolerance of .000001. The assertion does not cause a failure, and the test case completes.

It's important to test unexpected input. The programmer may not have expected feeCalculation() to receive a set of accounts that contained no premium accounts. So the second test checks for a set of non-premium accounts:

public void testNonPremiumAccounts() {

Account accounts[] = new Account[2];

accounts[0] = new Account();

accounts[0].principal = 12;

accounts[0].rate = (float) .025;

accounts[0].daysActive = 100;

accounts[0].accountType = Account.BUDGET;

accounts[1] = new Account();

accounts[1].principal = 50;

accounts[1].rate = (float) .0265;

accounts[1].daysActive = 150;

accounts[1].accountType = Account.STANDARD;

float result = feeCalculation.calculateFee(accounts);

assertEquals(result, 0, 0.0001);

}

The expected result for this test is 0, and it passes.

It's not enough to just test for expected results. A good unit test suite will include tests for boundary conditions , or inputs at the edge of the range of acceptable values. There are many kinds of boundary conditions, including:

* Zero values, null values, or other kinds of empty or missing values
* Very large or very small numbers that don't conform to expectations (like a rate of 10000%, or an account that has been active for a million years)
* Arrays and lists that contain duplicates or are sorted in unexpected ways
* Events that happen out of order, like accessing a database before it's opened
* Badly formatted data (like an invalid XML file)

A few tests will verify that these boundary conditions are handled as expected. This unit test verifies that calculateFee() can handle an account with a zero interest rate:

public void testZeroRate() {

Account accounts[] = new Account[1];

accounts[0] = new Account();

accounts[0].principal = 1000;

accounts[0].rate = (float) 0;

accounts[0].daysActive = 100;

accounts[0].accountType = Account.PREMIUM;

float result = feeCalculation.calculateFee(accounts);

assertEquals(result, 0, 0.00001);

}

This test passes in an account with a negative principal (a calculator was used to come up with the expected result by hand):

public void testNegativePrincipal() {

Account accounts[] = new Account[1];

accounts[0] = new Account();

accounts[0].principal = -10000;

accounts[0].rate = (float) 0.263;

accounts[0].daysActive = 100;

accounts[0].accountType = Account.PREMIUM;

float result = feeCalculation.calculateFee(accounts);

assertEquals(result, -9.33265, 0.0001);

}

In this case, the programmer expects the correct mathematical result to be returned, even though it may not make business sense in this context. Another programmer maintaining the code can see this expectation simply by reading through this unit test.

The next test verifies that the software can handle a duplicate reference. feeCalculation() takes an array of objects. Even if one of those objects is a duplicate reference of another one in the array, the result should still match the one calculated by hand:

public void testDuplicateReference() {

Account accounts[] = new Account[3];

accounts[0] = new Account();

accounts[0].principal = 35;

accounts[0].rate = (float) .04;

accounts[0].daysActive = 365;

accounts[0].accountType = Account.PREMIUM;

accounts[1] = accounts[0];

accounts[2] = new Account();

accounts[2].principal = 50;

accounts[2].rate = (float) .04;

accounts[2].daysActive = 600;

accounts[2].accountType = Account.PREMIUM\_PLUS;

float result = feeCalculation.calculateFee(accounts);

assertEquals(result, 0.0781316, 0.000001);

}

It's also possible to create tests that are expected to fail. The programmer expects calculateFee() to choke on one particular boundary condition—being passed null instead of an array:

public void testNullInput() {

Account accounts[] = null;

float result = feeCalculation.calculateFee(accounts);

assertTrue(true);

}

The assertion assertTrue(true) will never fail. It's included for the benefit of any programmer reading this unit test. It shows that the test is expected to get to this line. Unfortunately, calculateFee throws a NullPointerException error.

In this case, that's exactly the behavior that the programmer expects. The unit test can be altered to show that it expects the call to calculateFee() to fail:

public void testNullInput() {

Account accounts[] = null;

try {

float result = feeCalculation.calculateFee(accounts);

fail();

} catch (NullPointerException e) {

assertTrue(true);

}

}

The fail() assertion is placed after calculateFee() to verify that it throws an exception and never executes the next statement. The assertTrue(true) assertion is then used to show that the call is expected to throw a specific error, and the test expects to catch it.

These test methods by no means represent an exhaustive test case for the FeeCalculation class. But even this limited set of tests is enough, for instance, to ensure that a refactoring has not broken the behavior of the class. It would not take a programmer much longer to come up with a more exhaustive test case for this example.

## Test-Driven Development

Test-driven development means that the unit tests are created before the code is built. Before a programmer begins to build a new object, she must first create the test case that verifies that object. By the time the test case is finished, she has defined all of the expected inputs, outputs, boundary cases, and error conditions for the object, and she has a test case that verifies that it works. As she builds each part of the object, she can run its unit tests to verify the code that has just been built. Many defects found by testers or users have to do with unexpected inputs or error conditions; since the programmer has already planned out all of the ways that the object might fail, she will catch many more of these the first time she builds the software.

Test-driven development also helps programmers understand the requirements better. It's possible—and often tempting—to begin coding with only a partial understanding of what it is that the code is supposed to do. It is not uncommon for a programmer to "go off half-cocked" and begin coding before really taking the time to understand the behavior of the code. This is understandable; it's more fun to write code than it is to sit and pore through requirements documents. It's even more tempting when there are no requirements documents. If the programmer has to build the software simply based on notes from some conversations and a vague understanding of the scope, it's much more fun to just jump in and start coding! Taking the time to write the unit tests really firms up the requirements in the programmer's mind, and often helps her to see exactly where she is missing information.

There are several important benefits of test-driven development . The most obvious one is that it guarantees that unit tests are always written. When a team is under pressure to release, it's very tempting to release the code that seems to work. And when the unit tests are the last thing that the team has to do, a senior manager facing a deadline will often decide to release the build as is. Without the unit tests, the code will have more defects. These defects will have to be fixed later, often after the programmers have moved on to other projects and this code is no longer fresh in their minds.

Another important benefit of test-driven development is that the unit tests have a very positive influence on the design of the code and the object model. Many design problems stem from the fact that when an object is built, the programmer makes a decision about the interface that later turns out to make that interface difficult to use. By the time she writes code that uses that interface, the object is already written, and it may be difficult to rewrite it to accommodate the way it really needs to be used. The unit test forces her to start by writing code that uses the object; many poor interface decisions immediately become apparent, before the code for the object is written.

A complete suite of unit tests also makes it much easier to refactor the software. The unit test suite can be run after each refactoring, helping the programmers to immediately identify any defects that they might have accidentally injected. Running unit tests after each refactoring removes most of the risk and ensures that the refactoring really does not alter the behavior of the software.

Unit testing is an efficient way to build better software. Test-driven development often yields code that has fewer defects than standard development, and many programmers who do test-driven development find that they are able to produce that code more quickly than they had in the past.

## Everyone Is Responsible for Quality

In some organizations, there seems to be a growing tension between the programmers and testers. The testers will find an increasing number of defects, which they feel should have been caught before the build was delivered to them. The programmers, on the other hand, start to feel that they aren't responsible for testing of any kind—not even unit tests—because they feel the testers should catch every possible problem. Project managers who start to sense this tension often feel powerless to do anything about it. In situations like this, automated unit tests can help.

Many programmers are confused about exactly what it is that software testers do. All they know is that they deliver a build to the QA team. The QA people run the program and find bugs, which the programmers fix. It is often hard for them to figure out where unit testing ends and functional testing begins. A good project manager will keep an eye out for this confusion, and help to clarify it by making sure that the programmers understand what kinds of testing are expected of them.

There are different kinds of testing that serve different purposes. The purpose of unit tests is to verify that the software works exactly as the programmer intended. Software testers, on the other hand, are responsible for verifying that the software meets its requirements (in the SRS) and the needs of the users and stakeholders (in the Vision and Scope Document). Many defects arise when a programmer delivers software that worked as he intended, but did not meet the needs of the users. It's the software tester's job to catch these problems (see [Chapter 8](https://learning.oreilly.com/library/view/applied-software-project/0596009488/ch08.html)). This is why both the programmers and testers must test the software—they are looking for different problems. If this distinction is clear to the programmers, they should understand why unit testing is their responsibility.

By adopting unit tests and test-driven development, the programmers can develop a very clear picture of exactly what their testing responsibilities are. Before the programmers deliver their code to QA, the code should pass every unit test. This does not necessarily mean that the software does what it's supposed to do, but it does mean that it works well enough that a tester can determine whether it does its intended job.

It should make intuitive sense that, since the QA team runs the software as if they were users, they do not have access to the individual units (objects, functions, classes, database queries, modules, etc.) that the programmers create. If these units are broken, it is often possible for those defects to be masked, either in the user interface or elsewhere in the software. Even if those units seem to function properly, there may be defects that will only be found when those units are used together in complex ways. This is what QA engineers do—they simulate complex actions that could be performed by the users in order to find these defects. From this perspective, it is not hard for a programmer to see that only the programmers can test those units, and that those units must be working before the testers can do their jobs. Implementing automated unit tests can ensure that those units work, letting the QA team concentrate on more complex behavior. By helping the programmers understand the line between unit testing and functional testing, and take on the responsibility for unit testing, the project manager can help reduce the tension on the project.

## Unit Testing Saves Programming Time and Effort

Some project managers find that programmers are resistant to unit tests. Sometimes the programmers resent the assumption that their code isn't perfect, even when there is a history of defects that had to be fixed on previous projects. Other times, they assume that the QA team's job is simply to find the programmers' bugs. But mostly, they don't like spending time writing and running unit tests that won't be delivered in the build. It feels like a waste of time, without adding much value. But in fact, the opposite is true. Most programmers who adopt unit tests find that it actually reduces the total time it takes to build software. It may take time to write the tests up front, but it costs more time to go back later and fix all of the bugs that the unit tests would have caught.

One way that unit tests help the programmers deliver better code is by improving the object interfaces or function definitions that the programmers use. Many experienced programmers will recognize the feeling of regret that comes when they start using an object that they built to encapsulate some functionality, only to realize later that they should have designed the interface differently. Sometimes there is functionality that the object needs to provide that the programmer didn't think of; at other times, it may be that the methods are awkward to work with, and could have been laid out better. But the code for the object is already built, and it's too late to fix it. She will just have to work around the awkwardness, even though building the object right from the beginning would have saved her some time and effort. Or she may be frustrated enough to go back and rebuild the object entirely, which costs even more time. This is a common trap that plagues object-oriented design—there's no way to know how easy it is to use an object until code is built that uses it, but there's no way to build that code until the object is done.

In practice, test-driven development is an effective way to avoid that trap. The programmer can build a series of simple unit tests before the object is built. If the object is needed, she can build a mock object to simulate it. By the time she is ready to define the interface, she has worked out many of the details and has discovered and avoided the potential problems with the interface before the object is built.

Programmers also find that effort is unnecessarily wasted in situations when one person has to use an object that was designed by someone else. Often a programmer finds that the object's interface is not clear. There could be ambiguous function names or variables, or it could be unclear how to use objects that are returned by certain functions.

This is another case when unit tests can be very useful. When a programmer consults documentation for an object or an API manual, the first thing she usually looks for is an example of the functionality that she is trying to implement. The unit tests serve the same purpose. When an object comes bundled with a series of unit tests, the programmer can consult them to see how the object was intended to be used. Not only do they provide her with a substantial amount of example code, but they also show her all of the behavior that the object is meant to exhibit, including the errors that it is expected to handle.

# Use Automation

There are many tasks over the course of a software project that can be automated. Unit tests are a good example of automation —before programmers started using automated unit tests, they had to manually verify each function and user interface element before delivering a build. By automating unit tests, the programmers were able to make them much less time-consuming tasks and, as a result, many more programmers take the time to build unit tests.

But unit tests are not the only manual programming task that can be automated. Automation can ensure that the software is built the same way each time, that the team sees every change made to the software, and that the software is tested and reviewed in the same way every day so that no defects slip through or are introduced through human error.

When projects become complex and require many steps to build, it's easy for programmers to forget a step. A programmer may build a version of the software that is missing a library, or is compiled with incorrect options. What's more, some programming teams have discovered that, as their build process becomes more and more complex, they have to dedicate more of a senior team member's time to generating new builds on a regular basis, which can cause delays in the project. Eventually, it becomes difficult for the team to even generate a reproducible build. When the build is delivered to users, there are often defects that can be traced back to missing libraries, or to required files that should have been included.

There are many automated build tools that address all of the problems caused by an unpredictable and difficult-to-reproduce build process. Popular ones include:

* Make (which ships with most Unix-like operating systems and some IDEs)
* GNU Make (<http://www.gnu.org/software/make>)
* Apache Ant ([http://ant.apache.org](http://ant.apache.org/))
* Jam (<http://www.perforce.com/jam/jam.html>)

Each of these tools automates a set of steps required to build the project. There are ways to ensure that dependencies are honored, so that a piece of code is only built after the tool first builds every library that it depends on. With an automated build tool, a programmer can cut a new build at any time and without extra input or interaction. The team can be sure that the software is built the same way each time and includes all of the necessary files and dependencies.

One of the most important ways a software project can be automated is with a development monitoring system . Two of the most popular ones are CruiseControl ([http://cruisecontrol.sourceforge.net](http://cruisecontrol.sourceforge.net/)) and TinderBox (<http://www.mozilla.org/projects/tinderbox>), both of which are free and open source. These tools allow a project team to construct a set of tasks to be run automatically on a schedule. These tasks include:

* Retrieving the latest build from the version control system, building it, copying it to a folder, and reporting any build warnings or errors
* Running automated unit tests, generating a test report, and reporting critical failures
* Running automated code review tools and reporting any warnings or rule violations
* Listing any changes that have been committed and by whom, including links to code listings for the changes
* Emailing results as text or visual reports (or sent via SMS text messages or some other communications system)

These tasks can be configured to run on any chosen schedule. Many teams will schedule daily builds. Some will require builds on an hourly basis (or another schedule), while others will only require automated builds to kick off when changes are checked in. The important aspect of this is that the team can depend on these tasks being run the same way each time. They are kept aware of every change made to the code, and they are always aware of the health of the current build. If code is checked in that causes the build to break or important unit tests to fail, the team will know about it immediately and will be able to fix the problem early on, before any other changes are made.

It is especially useful to combine the reports from a development monitoring system with code reviews. This is an especially common practice on successful open source projects, where every change or patch applied to the code is emailed to the developers and reviewed before it is incorporated into a production release.