The Version Control Process

Source-code version control is a set of working rules for code sharing that lets developers modify files in an exclusive way. As such, it is one of the most important, yet least understood, areas of software development.

By Aspi Havewala, Dr. Dobb's Journal
May 01, 1999
URL:http://www.ddj.com/184410941

Aspi manages projects that require extensive coordination in software, hardware, and mechanical development. His area of interest is in developing processes and methodologies for team-based project management. You can contact him at ahavewala@hotmail.com.

Even though source-code version control is a critical part of project management, it remains one of the most neglected aspects of the development process. Poorly maintained version control can itself create bugs that jeopardize project schedules and software quality. The good news is that most of these problems can be avoided by properly using source-code version control software.

There are a number of commercial and free source-code version-control packages available, including:

- Intersolv PVCS Version Manager (http:// www.microfocus.com/products/pvcs.htm).
- Microsoft Visual SourceSafe (http://msdn .microsoft.com/ssafe/).
- Burton Systems TLIB (http://www .burtonsys.com/).
- Source Code Control System (available on UNIX systems).
- Rational ClearCase (http://www.rational .com/products/ccmbu/).
- Concurrent Version System (CVS) (http:// www.cyclic.com/).
- Project Revision Control System (PRCS) (http://xcf.berkeley.edu/~jmacd/prcs.html).

For a more complete list of commercially available packages, see http://www.loria .fr/~molli/cm/cm-FAQ/cm-tools-7.html. A list of freely available version-control alternatives (primarily for UNIX) can also be found at http://www.loria.fr/~molli/ cm/cm-FAQ/cm-tools-6.html.

When it comes down to it, however, which version-control software package you use is secondary. What is really important is how you use it. Consequently, in this article, I'll describe potential problems and investigate practices that will help avoid problems, no matter which source-code version-control package you use.

The Need for Version Control

One of the central activities in any project is the constant addition, deletion, and modification of source code. In a simplistic scenario, individual developers work on a given set of files and never have a reason to modify files outside a given set. In today's world of substantial code reuse brought about by object-oriented programming, however, this simplistic scenario is unrealistic. A more common scenario involves several developers simultaneously modifying shared source code. If a developer is making changes to a particular file, other developers shouldn't be making changes to it. Source-code version control, therefore, is a set of working rules for code sharing that lets developers modify files in an exclusive way.

In addition to coordinating access, whenever a developer makes modifications to a file, version control maintains a separate version of each file in a database. Each version can then be referenced individually if desired. Thus, it maintains a journal of changes made to each file in the source code. At any given time, a set of files may have several different versions in the database (depending on how many times the files were modified). Version control lets you specify a label for a set of files, which marks the current version of each file at a given point in time. Such a label can then be used to retrieve the set of files that were current at the time the label was assigned.

How Version Control is Used

Version control can be realized via one of many commercially available software packages. Version control maintains copies of all the files comprising the source code of a given project in a version-control database. This database is usually readable only by the version-control software front end. To extract a file from the database, you perform a Get or Refresh operation to extract the latest version of a file from the database and copy it onto your hard disk. Where the file is copied depends on the working directory setup by users. Working directories

can be associated with individual files or an entire project.

When you are ready to modify a file, you lock it. This marks the file "in-use" in the version-control database and lets you modify it in an exclusive way. When you finalize changes, you "check-in" the file. This causes the file to be copied back into the version-control database. The lock on the file is removed, thus making it available to other developers who may want to modify it. Some version-control systems combine the Get/Refresh and Lock operations into one operation called "check-out." Figure 1 illustrates the state transitions for a file under version control.

In addition to developers, administrators of version-control systems perform special duties to ensure version control functions smoothly and the integrity of the database remains intact. Administrators have rights to perform maintenance operations off-limits to developers. Figure 2 is a use-case diagram that shows the functionality exported by version control used by different types of users. The diagram illustrates the partitioning of functionality I recommend here. Your version-control software may not necessarily enforce such a partitioning of use cases.

Designing a Directory Layout

One of the first tasks you should assign when starting a new version-control database is designing a directory layout. The question behind this is simple but important: How will your modules be organized in their source form? Designing a directory structure for your modules is important for several reasons. They force you to think in terms of the implementation of your design. I refer to such tasks as "designing your implementation." This sounds like a contradiction, but it makes sense. If you are writing a device driver, for example, you can factor this into several implementation issues. Will your driver be in assembly, C, or C++? If it needs to operate on more than one platform, how will the platform-dependent versus the platform-independent code be structured in terms of files? How will different versions of your driver be built? Will they come from the same source code or will new directories be created for feature revisions of your driver? What happens when the hardware itself is revised? Will you use conditional compiling to support differences in your hardware? Or will you create a new project in your version-control database? Is some source code shared with another driver or is an API shared with an application that talks to your driver? Where will this common source code be placed so that it is accessible to all the modules that use it?

Such questions let you map out a strategy for implementing your design. If these questions are not addressed at the beginning of the implementation cycle and you leave each individual team members to figure things out individually, it's like starting a small fire in a room full of explosives. Before you know it, one explosion will lead to another and the entire room will be such a mess that you'll have to start all over again.

The answers to these questions are also critical in helping you design a directory layout for your source code. Once you have a directory layout designed and documented, I suggest creating a project structure in your version-control database that maps to your directory layout. In other words, each directory on your hard disk becomes a project (or subproject) in your version-control database. This simple one-to-one mapping between directories and version-control projects lets you structure source so that it is optimal for expanding, maintaining, and building. It also creates a paradigm for version control (based on directories) that your team will find easy to understand and work with.

Finding a Home for Shared Code

One of the key aspects of designing a directory layout is visualizing all the modules in your project and understanding their position in the software hierarchy. You should also spend time figuring out their relationship to other modules. How the source code changes in the future is something you can't predict, but even a partial prediction will help you structure directories. Once you identify bare bones relationships between your modules, you need to designate directories that hold source code shared between several modules. Header files are typical candidates for this type of code sharing, but there may be entire modules whose source needs to be shared, such as a debug library or a utility class.

Figure 3 is a hypothetical directory structure starting from a directory called "DevRoot," which is assumed to be at level 0. The level of each directory is enclosed in parenthesis next to its name. The directory named "Com" refers to "Common." The root directory is always designated to be at level 0. I assign the level *n* to a directory that is *n* directories below the root. A directory at level 2 is said to be lower than a directory at level 1. You always add shared source code under a directory called Common. Next, propose some rules regarding the directory placement of modules that need access to this shared code. Modules can share code in a Common directory at a level no lower than *n*, given that the modules themselves are in a directory at level *n*. That is, no module at level *n* can include a header file or source file, nor link to a library that exists in a directory at a level *n*+1. To enjoy the benefits of this rule, you combine it with another one. The Common directory that a module may share code from must be a sibling directory of a direct ancestor. In other words, when looking for code shared by a module, you can only traverse upwards from that directory in your search for directories called Common.

The higher the level of a Common directory, the greater its potential for affecting other modules. In Figure 3, the source in the directories under Com (3) can be shared by the modules Cm1 (3), Cm1 (4), and Cm2 (4). However, module Cm1 (2), Cm2 (2), and Cm3 (2) cannot access the code in this directory. However, all modules in this directory hierarchy may access code in Inc (2) and DebugLib (2).

The advantage of these rules is that relationships between modules that share code are easy to maintain. Why all this interest in shared code? By its nature, shared code is of special interest to everyone in the project. It represents an area of productivity you can really leverage, but also harbors a high amount of risk when modified.

By simply browsing the directory structure, you can determine whether code is shared (by looking for the directory Common) and which modules are likely to share it (by looking for modules at the same level or below under sibling directories of Common). One quick and

tempting way to follow this rule would be to always create a single, universal Common directory at the top-most level and place all shared code there. However, this approach has several disadvantages that can create ongoing maintenance problems. For large projects in which several modules at different levels may share code, this universal Common directory may become populated with a large number of files. It is impossible to track which modules share what code in this universal Common. By providing such a global repository for shared code, you also offer developers a simplistic way to share code. But isn't making code easy to share a good thing? Not necessarily. Sharing code between modules is a decision that needs to be made with careful thought.

Say you are managing the source code for a project. You follow the rules and share some common code at directory level 20. At this level, you have three modules and so you know that at least these modules could include the shared code. Furthermore, say a module at level 15 needs to use this shared code. This violates the rule that states that the shared code must exist at a directory level equal to or higher than the one in which the module exists. The solution would be to promote the shared code to a sibling directory at level 15. You would now have a Common directory at level 15 for shared code. You also know that all modules at level 15 in sibling directories may share this code. Such a promotion should set some alarm bells off in your mind. When making such decisions during feature development, or even during bug fixing, you can ascertain the level of risk by looking at the level of the Common directory and counting the number of potential modules that could be affected.

You should have a good idea of the benefits of the twin rules by now. Following them lets you organize shared code in a tight, logical way. In contrast, consider shared code that is sprinkled across your directory hierarchy without any uniformity of location. A module in a top-level directory could be sharing source code from a directory buried somewhere in your directory tree. Sounds like something you don't want to deal with, right?

Adding Files to Version Control: The Great Debate

I've almost never worked with a team where the issue of what type of files to add to the version-control database hasn't been debated. Obviously, adding a source file is never an issue. Things heat up when you discuss files like documents, schematics, object modules, and executable files. Documentation should potentially go into a separate Documents database. If you don't have one, create one. There are plenty of document-management packages available. Most word processors or document producing software also let you implement some form of versioning. Schematics are like hardware source code, so an argument can be made that those need to go into version control.

The temptation of adding objects, executables, and temporary files comes from the fact that you can take a full build of your source code, add all the executables and intermediate files to version control, and have it serve as a checkpoint. This is misuse of a version-control system. There are too many intermediate and output files produced by compilers and other tools these days, and making sure all of them have been added and updated from one checkpoint build to another is a logistical nightmare. Second, it grows your version-control database significantly. Remember that your database must be able to stand up to the rigors of constant check-ins and check-outs. Don't stress the version-control database any more than you have to.

How do you maintain those build checkpoints? By setting up a reproducible automated build. The basic idea is that you should be able to check-out the entire source at a given checkpoint and produce the same build made from the same source at an earlier date.

The rule is this: If there are any files generated from the build process, don't add them to version control. This includes translated files such as headers for different languages or platforms, which are generated from a master header file. If you keep the version-control database lean and clean, maintenance requires significantly less effort, and developers will thank you when the refresh doesn't force them to take a coffee break.

Where do Tools Go? Anywhere but Here!

One interesting debate I participated in came from a suggestion that we should add all our tools to version control. This has the benefit of not having to install all the tools from scratch when someone needs to start working on the code. You simply do a refresh and get everything -- the source code and all the tools you need to edit and build code. Although interesting, this approach is flawed in several respects.

For one, compilers and their IDEs (think Visual C++ 6.0 here) are becoming increasingly complex. Which of the several hundred files will you add to version control? How will you add the files needing to go into a specific installation directory and how will you add those needing to go into your operating-system directory? When you install a modern compiler and its IDE, the installer may create registry entries. Who will make these entries when you refresh the compiler from the version-control database?

Say a new version of your compiler arrives at your doorstep and, after running some compatibility tests and agreeing that everyone on your team should switch to it, you face the task of adding this new version to the version-control database. Any volunteers for this task?

Finally, depending on the tool vendor and your license agreement, adding the tool to the version-control database might violate software copyright. My general guideline is: If you need to track multiple versions of a file, consider adding it to the database. If you don't (like the files in a tool), forget about it.

So what's the best way to manage shared tools? You could create your own software library that is maintained by someone, typically the person in charge of automated builds -- the "build captain." This person verifies the tools, environment, and build process, and produces a concise document that tells newbies how to set up their development environment so that they can start building the source. Developers setting up their environments could then get the tools from the build captain and install them according to the installation document. If

maintaining a software library sounds too tedious, set up a CD-ROM changer that is shared by all developers over the LAN (make sure your license agreement with the tool manufacturer covers this type of usage).

Monitoring and Refining Usage Patterns

A good version-control administrator constantly monitors the usage patterns of developers using the system. This lets you nab some practices that are not healthy for your project. Often, the most common error made by developers is forgetting to refresh and lock a file before they start working on it. When it comes time to check-in the file, the developer (I'll call him X) realizes that the file is not locked in his name. Two things can now happen and they both depend on the features of your version-control software.

In one case, Developer X immediately locks the file in his name and checks in his version. This violates the state change diagram in Figure 1. According to this operation, the lock must be performed before the modifications to the file are made. This is enforced because in the time Developer X was modifying that file, Developer Y could have made changes to it. Developer Y checks in his changes, but Developer X never sees them because he had done a refresh before Developer Y checked in his changes. Now when Developer X checks in his version of the file, Developer Y's changes are lost. Needless to say, this causes heartburn when the feature or bug fix added by Developer Y stops working and he has to spend a day or two finding out that Developer X (and his sloppiness) are responsible.

In another case, your version-control software may play "good Samaritan" and refresh the file automatically when Developer X attempts to check it out. Hopefully it warns the developer that this is about to happen because if it doesn't (or if Developer X chooses to ignore the warning), he will lose all those changes he was working on.

The administrator needs to establish a pattern of usage whenever such things start happening (hopefully he will do it before this happens). Make team members lock any file that they plan to make changes to. Make sure they unlock it or check back their changes as soon as they are done.

Some developers like to run experiments that require changes to files that they don't necessarily want to lock -- a scenario that can turn ugly. If a developer makes changes to a series of files that work but loses track of those changes, he will have to check-in all those files without really knowing exactly what it is he has added. He may even forget to check-in a file that had changes made to it (if he didn't check it out, there is nothing to remind him). Typically, this will result in a bug fix or feature addition that works fine on the developer's workstation but cannot be reproduced in a build made on another machine. On the other hand, the experiment may fail and the user may have locked the files for naught. What's a poor developer to do?

My recommendation is to lock all the files you need when you make your changes on an experimental basis. Mark all of your changes with a unique comment. When it comes time to check-in your changes, search through all your source files, identify the changes, and decide if they should be kept. Remove comments if you don't want them in there. If you decide not to check-in changes, simply revoke your lock on those files. Refresh when you do this so that you are not carrying around your experimental changes in your build.

If you don't feel comfortable locking all the files (maybe someone else has them locked and your experiment will be a quick one), make detailed notes on where and how to add changes. When you are done and decide to check-in changes, start from scratch. Refresh your source in such a way that you lose all changes, lock the relevant files, then add in changes from your notes.

Broken Builds

Another common error to watch out for involves checking in multiple files with changes. Say a developer adds a new constant in several of his source files. He adds the definition for this constant in a header file in a shared common directory. Because the definition is added just once, it is easy to forget about it. When it comes time to check-in the files, the developer might check-in all the source files but neglect to check-in the header file. This will result in a compile-time error and a broken build. Fortunately, the generated error is easy to track down. Track down the offending file and check to see who was the last person to make changes to that file. This person will more often than not be the offending party.

Broken builds should be taken very seriously. More than bugs in the code, a broken build brings development for the team to a halt -- especially if developers are in the most excellent habit of refreshing their files frequently. Teams have different ways of punishing offenders when it comes to breaking the build. We once had a donut rule in effect. Whoever broke the build would have to bring in donuts the next day. To preserve the general health of our team, we relaxed the rule so that one could substitute bagels instead. The message was clear, "Donut break the build!"

Contention for Files

Here's a common scenario: A large number of developers are working on the source code and have locks on a lot of files. At some point, a developer needs a file checked out to someone else. Some developers avoid the confrontation by making local changes to a file and waiting for the original developer to unlock the file at a later date. This saves time only in the short run. The best thing to do is encourage developers to let others know you want the file that is locked. Two developers who contend for a file should be able to work out a solution where both have access to the file when they need it. If the debate is heading nowhere, step in and resolve it as an administrator. This sort of communication within your team is helpful. It encourages developers to foster relations with other team members and potentially lets developers discuss planned changes to code they are working on. If you see a situation where developers are constantly in contention for a

particular file, you should take a closer look at that file. It is probably a good candidate for splitting it up into a number of smaller files. Provided you divvy it up according to areas of functionality, you will significantly alleviate the contention problem.

Fumigating the Hidden BugLord

There is a nasty "BugLord" lurking in your source code that periodically creates bugs and makes life miserable. Say developers are working on some critical features. If your project is anywhere near normal, these features will interact with each other. Now imagine a developer furiously working on a feature that has not been checked in for about a month. In other words, he is working with an older version of the source code and no other developer on the team has worked with his changes. Do the words "integration problems" come to mind?

This is the hidden problem that most project managers fail to account for and some administrators never recognize. You can effectively use version control to address this problem. Developers need to check-in working changes as soon as they are done. Holding on to something that is in good working order is just not productive for the rest of the team. Often developers and project managers shy away from checking-in changes for fear of "destabilizing" the build. This works in the short run, but the time you will spend integrating at a later stage will more than make up for any short-term savings. To alleviate destabilization, discuss when a major check-in is expected of team members and insist they carry it out as soon as they are ready -- even if there is an immediate penalty to pay in terms of integration time. Integration carried out in smaller steps is beneficial because it helps you track the status of the project more accurately. With this approach, surprises don't really have a chance to become surprises if detected early on. Over a period of time, team members will become better (and consequently more aggressive) at integrating smaller pieces. As a result, a project lead will become better at predicting the general health of the source code, resulting in more accurate schedule predictions.

The Integrate and Test Phase

You may find it helpful to schedule at least two major "integrate and test" phases in the life of your project. Depending on how much code you plan to produce, and the overall length of the project, you may assign sufficient time to this task to account for the entire team working on nothing but integrating all changes and producing a working build. Once again, you'll be in a position to track a project much more accurately this way.

Enforcing a rule at this point is helpful: No one should check-in changes unless they have tested the build to see that it compiles and works reasonably well. Granted "works reasonably well" is a highly subjective term, but you should be able to decide with your team what this term means. Maybe you have a regression test suite that you'd like developers to run before checking-in their changes. Alternately, you may want to create a manual test script to be executed by developers to verify that the build is in working order. If team members have been refreshing source code often enough, they will have relatively fewer problems during this unit test phase. In any case, they need to make sure they refresh the entire source code (other than the files locked by them) at least once before creating the build for unit testing.

When and Where to Branch Code

Every project linked to an ongoing product release must square up to the prickly topic of branching code. Say you are working on Version 1.0 of a product that is now in beta. Bug fixes need to be made but you are winding down on the release. Meanwhile, a team assembled to work on Version 2.0 needs to forge ahead. Although they need to use your code base, you don't want their features to be added to the version you are working on. Yet you want to make sure the 2.0 team gets all the bug fixes you plan to make for Version 1.0.

A technique most teams employ in such situations is to create a branch in the source code. By branching the code, you are creating two version streams. One ongoing stream, which I'll call the "current stream," is used to make bug fixes for the current version. The other stream, the "new stream," is handed over to the team working on the next version for feature additions.

When bug fixes are completed and the current version is released, developers migrate to the new stream. An alternate approach to this is to have developers working on the current stream migrate their changes to the new stream simultaneously. This might take away precious time from the current development effort. If you have to migrate the changes from the current stream to the new stream at a later date, make sure these changes are well documented. If possible, assign the same developer to make changes to both streams. This developer is more likely to be able to read the documentation and recall the steps required to migrate the changes to the new stream.

In any case, the decision to branch must be made after a careful evaluation of the status of your current project, the estimated schedule, and the resources and timeline for the next version of the product. Only administrators should have the authority to branch source code. When the merge to the new stream is complete, administrators can choose to delete the old stream. Alternately, you can keep it around for reference but should archive and delete it as soon as the next iteration of the product reaches stability. This will help you keep your database trim.

As an offshoot of this rule, only administrators should have the rights to permanently delete a file from version control. Giving developers this kind of permission can result in disaster.

Labels: Adding Checkpoints to your Source Code

Most major version-control software packages let you create labels that span across the entire database and identify the current version of a file when the label was assigned. You can then retrieve the current version of each file when the label was assigned by retrieving files

associated with the label. Sounds simple enough. Assigning and cataloguing labels is important. If used injudiciously, labels can be difficult to decipher and lose their benefit quickly.

Devise a standard format for creating labels that will be easy to read and decipher. If a label affects a major part of the project, assign it across all files in your project. The administrator is the only person who should be assigning such global labels. If multiple projects are assigning a label to the same files, you should attach a prefix to each label that identifies the project for which the checkpoint is being created. An example format for such a label would be PROJECTNAME_VERSION_PURPOSE.

What about private labels -- those that developers want to assign privately to a their portion of the code as a personal checkpoint? This can be a valuable tool when it comes to marking critical stages in the development of a module. Developers can usually assign a label provided it is meaningfully worded. This, unfortunately, is at the discretion of developers. However, you can insist on a couple of rules. A private label cannot be assigned to all files in the project. Only administrators can do that. Second, a private label must be prefixed with the developer's username or initials. The first rule will prevent global labels from growing to an unmanageable volume. The second rule gives you a rudimentary journal capability. If you ever have a question about a particular label, or if it sounds cryptic to you and you want developers to change it, you have an easy way to figure out whom to talk to.

Version-Control Inspections

The best way to ensure that everyone is following essential rules and guidelines is to make frequent version-control inspections. These would be similar to code inspections, but you will spend most of your time looking at how the code is organized rather than how it is written. More specifically, where are new directories being added in the hierarchy, how is code being shared, and are files being named meaningfully? Administrators have the best bird's-eye view of the project and its source code. Watch out for generic file names. What, for example, does a file called "status.c" contain? Make sure developers use filenames that are specific enough so that you don't need to open up the file and look at its contents to figure out what is in it. Rename files like "status.c" to "FileStatus.c" if it clarifies the contents of the file.

Although code inspections are separate exercises with different goals, they can be helpful in picking out potential code organization problems. You'll almost always come across some constants that should have gone in a header file but were added to a source file instead. Convince the code-inspection team that such problems should draw their attention and be recommended for correction.

Conclusion

Version control is an often-neglected activity in team-based software development. Its correct and smooth functioning ensures that projects won't have major hiccups. It is difficult to document the impact of problems that come from incorrect or sloppy use of the version-control system. But, by allocating special attention to this aspect of development, you can avoid potentially hazardous problems that affect the final outcome of your project. A good way to start is by designing the implementation of your source code at a coarse level, then lay down rules and guidelines regarding how to use version control. The best rules and guidelines are those that evolve from within a team, so remember to bring everyone together and come up with a set of rules by consensus. If you get buy-in from the team, you'll have enthusiastic developers working with version control and leveraging it to its fullest extent.

DDJ

Copyright © 1999, Dr. Dobb's Journal

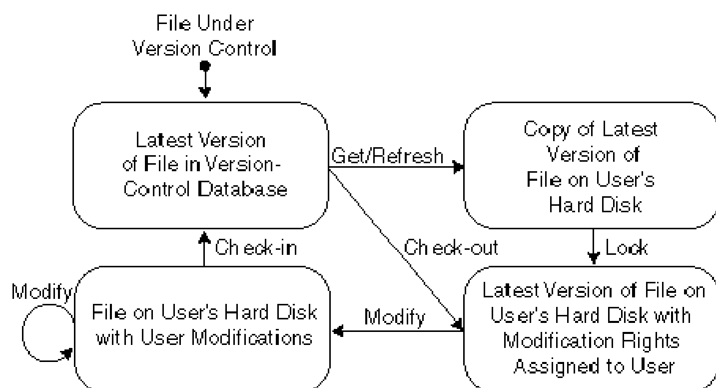

Figure 1: State diagram for file under version control.

Copyright © 1999, Dr. Dobb's Journal

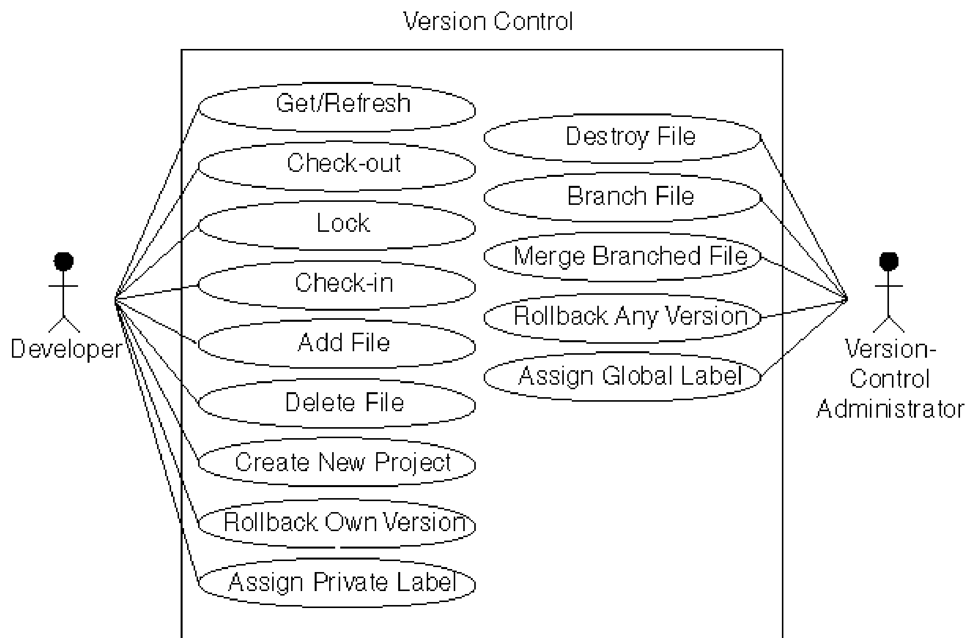

Figure 2: Use-case diagram for version control.

Copyright © 1999, Dr. Dobb's Journal

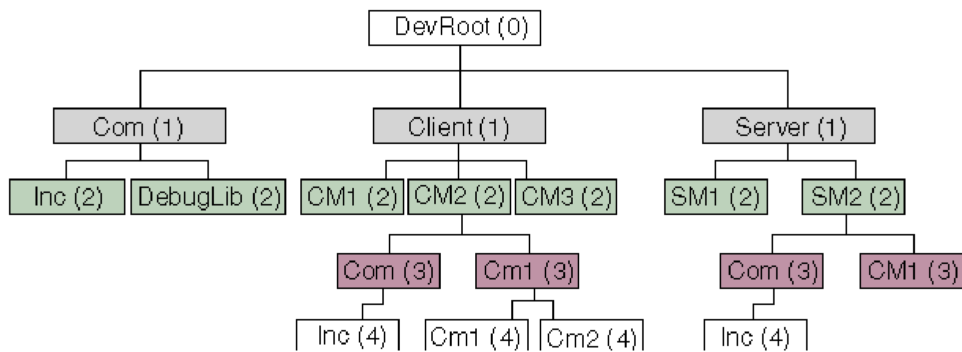

Figure 3: Directory levels and sharing.

Copyright © 1999, Dr. Dobb's Journal

&RS\ULJKW &03 0HGLD //&