Lunds tekniska högskola
Datavetenskap, Nov 21, 2014[1]
EDA260 Programvaruutveckling i grupp—projekt
Labb 2 (CVS): Bakgrundsmaterial

Introduction to Version Control with CVS

Abstract

CVS (Concurrent Versions System) is a widely used version control tool for tracking modifications made to project source code files. Developers can work on the same files and coordinate their contributions through a common repository. This paper presents an overview of CVS, and presents two case studies, intended to illustrate the most important CVS features used when programming alone and in teams.

1 Introduction

When working in software development projects, there is a need for tool support to coordinate the changes done by different developers. There are several problems that need to be solved: Keep track of the different versions of the different files, allow the developers to work in parallel, keep track of the latest stable version of the system, and integrate (merge) changes done by different developers.

To solve these problems manually, without special tools, quickly becomes a nightmare. For example, one manual method is to let the developers email each other new versions of files. This might work reasonably if you have a single file and take turns in editing it. But if you want to work in parallel, and start working with many files, these manual methods become much too cumbersome and error-prone to use.

To solve these problems, *version control systems* are used. The Concurrent Versions System (CVS) is a classical version control system, and has been used for many open-source projects. A wide range of software have been developed with support by CVS. Examples include `emacs`, the well-known programmable text editor, `gcc`, the GNU Compiler Collection, `apache`, the most widely used web server in the world, and in fact CVS itself. All these software products are continuously developed by hundreds of developers around the globe; teams that would hardly be able to handle their file exchange using email.

Recently, many open-source projects have switched from using CVS to newer tools like Subversion and Git. Subversion is very similar to CVS, with commands like *checkout*, *update* and *commit* (described below). They are both based on a *centralized* version control model, where developers collaborate via a central repository. Git is a *distributed* version control system, which is more complex than CVS, but also more powerful. In using Git, a central repository is often set up, just like in CVS or Subversion. But with Git it is also possible for developers to use version control locally, and to collaborate directly, peer-to-peer, without a central repository. Other examples of popular distributed version control systems are Bazaar and Mercurial. The Eclipse Java Development Tools currently has built-in support for CVS and Git, but other version control systems, e.g., Subversion, are supported by plugins.

CVS is implemented in C and binaries exist for most operating systems, including different flavors of Unix and MS Windows. CVS is primarily intended for handling source code files or other text files. But it can also be used for managing binary files, although such use is not described in this paper.

[1]Original version from 2003 by Christian Andersson. Revised by Görel Hedin

2 CVS overview

2.1 Important concepts

Consider a team of developers working together to build some software. Using CVS, the source code is stored in a **repository**, residing on a server machine. Whenever a developer wants to change some source code, she never changes the repository directly. Instead, she performs a *check out* operation which creates a directory on her own client machine, holding local *copies* of the source files. The directory with the copies is usually referred to as a *workspace*[2]. The developer can now change the files in the workspace, for example to implement a new feature. The files can then be copied back to the repository through an operation called a *commit*. The developer can also do an operation called *update* to update the workspace by copying the newest versions of the files from the repository. This is useful if other developers have done commits meanwhile. When doing an update or commit, there may be conflicts between the different versions of the files in the workspace and the repository, requiring different versions of the same file to be merged. This will be discussed in Section 4.

The repository actually contains *all* versions of the source code that have ever been committed. This allows old versions of files to be retrieved from the repository.

A repository consists of a set of **modules**, each containing a number of files. When checking out a workspace, it is for one of those modules, not for the whole repository. Typically, one module will correspond to one "IDE project", for example a program or a library. So for a simple product, it may be sufficient with one module in the repository. But for a more complex product, consisting of several independent libraries, layers, components, or programs, it may be useful to store these parts as separate modules.

The files in the workspace directory are normal source code files that the developer can edit, compile, test, etc. with normal tools. Here is a summary of the operations used to for check out, update and commit:

checkout for creating a new workspace

update for updating the workspace, i.e. copying newer files from the repository to the workspace directory, possibly merging with the existing files in the workspace

commit for committing changes to the repository, i.e. copying changed files in the workspace to the repository

2.2 File structure

Before going into examples of CVS use, it is good to know a few things about how information is represented, both in the repository and in a workspace. Suppose you have a Java project `MyProject` with two subdirectories: `src` for source files and `doc` for documentation. Inside `src` there is a main program `Product.java`, and a package `pack` with java files inside. Using CVS, you would store `MyProject` as a module in a CVS repository, say `MyRepository`. When you check out `MyProject` you would get a workspace directory with the following file and directory structure:

```
MyProject/
  CVS/
  src/
    CVS/
    Product.java
    pack/
      CVS/
      file1.java
```

[2]Other common terms for workspace include *working directory*, *sandbox*, or *working copy*

```
    ...
  doc/
    CVS/
    MyDocumentation.tex
    ...
```

The only difference from an ordinary (non-CVS) project, is that in each directory, there is a `CVS` subdirectory containing information that CVS uses. The information in the `CVS` subdirectories keeps track of what module and repository the files have been checked out from, and what versions the latest updates are from.

In the repository, the file and directory structure is similar, but not exactly the same. The repository directory contains a `CVSROOT` directory which stores information about the modules, and about what users may access the repository. Each file is represented as a *v-file*, for example, `Product.java,v`. A v-file contains *all* the versions that has ever been committed of that file.

```
MyRepository/
  CVSROOT/
  MyProject/
    src/
      Product.java,v
      pack/
        file1.java,v
        ...
    doc/
      MyDocumentation.tex,v
      ...
```

Normally, you will not look directly at these repository files. But it is good to know about this structure in order to understand the CVS commands better.

2.3 Important work habits

Normally, a complete module is checked out into a workspace, giving the developer access to all the files for that module. The developer can then edit, build, and test within her workspace, in isolation from other developers, and commit changes to the repository now and then. At the same time, other developers may work on the same software, in their own workspaces, and commit other changes now and then. To prevent mistakes, CVS will refuse to commit files that do not build on the latest version in the repository, However, this is not enough to **keep the repository clean**: the latest version in the repository should be consistent working software. It should compile without errors and all existing tests should run without errors.

To keep the repository clean, the developer should ensure two things before a commit: 1) that the software has been integrated with the latest version in the repository (by doing an update), and 2) that the software in the workspace compiles and tests. Therefore, right before committing, the developer should do an update, compile and test. Otherwise, the committed software might work in the workspace, but not together with newer changes committed by other developers. It does not help that CVS refuses to commit files that need update. If another developer has committed files that the first developer hasn't touched, the result could still be that the repository is not clean. (Exercise for the interested reader: try to think about an example of how this could happen.)

It is advisable to do **update often**, while working in the workspace. This keeps each integration step small and easy to manage. If you wait and do update more seldom, the update might bring large changes that are more difficult to integrate with your code. This way of working, updating often and always before commit, is sometimes called **update-update-update-commit**.

It is also advisable to **commit often**, as soon as you have a new piece of functionality that works. This will help both yourself and your teammates. It will help you, because it will force your teammates to integrate with your code, rather than you having to integrate with their code. It will help your teammates because each of their integration steps will be smaller (if they do update often).

To keep the repository clean, you need **strict atomic commits**. Atomic commits[3] means that a commit will result in either *all* locally changed files being committed, or *none* of them being committed (because some of them have conflicts with the latest version in the repository). With *strict* atomic commits, the commit will only take place *if there has been no other commits to the repository since your latest update*. Strict atomic commits ensures that a commit cannot result in a repository whose latest version differs from your local version. So, if you have ensured that your system is clean locally (compiles and tests correctly), then strict atomic commits will ensure that the repository will also be clean.

Unfortunately, CVS does not support strict atomic commits when run in its normal configuration mode (client-server). Therefore, you should *simulate* strict atomic commits by doing an update right before your commit, and commit only in case no files were actually updated. But even if you are very fast in doing the update-commit in your workspace, there is a small risk in CVS that another developer commits between your update and your commit, resulting in an unclean repository. For a small team, this is not very likely to happen, and therefore not much of a problem in practice. (Exercise for the interested reader: If you wanted to eliminate this risk completely, how would you do it?)

3 Case study: Single User

The simplest scenario on using CVS is where we have only one developer. Let's say her name is Alice[4]. For the sake of this example, Alice has just started up a very simple project on creating a Mandelbrot set visualization tool called MandelViz, which is to be implemented in Java by Alice *herself*. Alice's boss has done her the kindness of designing a class for complex calculations and also writing a program for testing implementations of this class.

In Alice's company (Herring, Inc.) they have a computer `salmon.herring.org`, where Alice has an account with the user name `alice`. This computer runs a CVS repository in a directory `/REPO`. The preparations made by the boss have resulted in a module, `mandelviz`, containing two Java source code files, `Complex.java` and `TestComplex.java`.

3.1 The Checkout Procedure

In order for Alice to get to work with the implementation of the MandelViz tool, she needs to get a copy of the `mandelviz` module in a local workspace directory of hers. This copy is made once and for all through a CVS procedure called checkout. First, she creates some suitable directory on her own workstation where she wants to do the checkout, say `salmonproject`, and moves to that directory. (Alice's writing slanted)

```
$ mkdir salmonproject
$ cd salmonproject
```

To communicate with the repository and perform the checkout, Alice needs to log in to `salmon` using CVS. This is done by issuing the following command:

```
$ cvs -d :pserver:alice@salmon.herring.org:/REPO login
```

[3] also sometimes called long transactions

[4] As a matter of fact, this Alice seems always to show up as a single developer in documents like this. And when she finally gets a development team mate, often a bit later on in the description, his name is always Bob. However, even more confusing, in our projects Alice and Bob are always four persons working in pairs, and each of them having names like `dat08xyz`, where `xyz` are arbitrary letters and digits.

CVS asks for Alice's password, where she answers `roach` (which is the correct password, unfortunately). But we have quite a few things to explain here — what are those `pserver` knick-knacks, for example?

The answer is, of course, that Alice has to tell CVS not only that she wants to log in, but also *where* she wants to log in, that is, into what repository. All repositories are identified with a so-called CVSROOT string, which has the following general format:

```
:<protocol>:<username>@<hostname>:<directory>
```

The protocol Alice is using when communicating with the repository (and which we are going to use) is a special CVS stream protocol called `pserver`. So, that explains the first part. The rest of the CVSROOT string shouldn't be that difficult to decode. The `-d` option simply tells CVS that a CVSROOT string is next to come on the command-line.

Apart from being logged in to the repository, Alice now has a new file in her home directory, called `.cvspass`, which contains a line for the `salmon` repository, along with her user name and her password, the latter in an encrypted form. The existence of this file makes it possible for Alice from now on to communicate freely with the repository, without logging in each and every time.

Now comes the interesting part — Alice is going to do the real checkout! But first, she double-checks that she is currently in the directory where she wants the workspace:

```
$ pwd
/home/alice/salmonproject
$ cvs -d :pserver:alice@salmon.herring.org:/REPO checkout mandelviz
U mandelviz/Complex.java
U mandelviz/TestComplex.java
```

This was the last time she had to issue that darned CVSROOT string.

What happened? CVS created a new directory called `mandelviz` (the workspace) and inside of it, the two files `Complex.java` and `TestComplex.java` are present. But also another thing:

```
$ cd mandelviz
$ ls
CVS/         Complex.java         TestComplex.java
```

Alice, who has read a lot of books about CVS, knows a few things about that `CVS` directory. The fact is, that it is created in each directory that is checked out from a repository and it contains system files pointing out the address of the repository and some important information about local changes to the files of the workspace. The files in it are controlled completely by CVS itself and they are important for its functionality. Hence, she doesn't tamper with them.

Important for Alice's concern is that the toughest part of her CVS use is now done. This checkout procedure is only made *once*, and since it is somewhat tricky issuing those heavy commands (with CVSROOT strings and so on), Alice is quite satisfied being finished with it.

From a systematic point of view, the procedure of checking out the module has had the very important consequence of providing Alice with a copy of the source code to which she is free to make any changes that she likes to, hopefully eventually leading to a working program. The fact that CVS stores system information in all checked-out modules makes it possible to have several modules checked out at the same time, even from different repositories. Each of them are tightly associated with their repositories, respectively. One should, however, be *very* careful not to check out modules inside of each other.

3.2 Getting Information

Before getting started with the real hacking, Alice is quite interested in getting some information on what she has checked out. Therefore, standing in the `mandelviz` directory, she issues the following command:

```
$ cvs status Complex.java
===================================================================
```

```
File: Complex.java      Status: Up-to-date

   Working revision:    1.1
   Repository revision: 1.1     /REPO/mandelviz/Complex.java,v
   Sticky Tag:          (none)
   Sticky Date:         (none)
   Sticky Options:      (none)
```

The information provided talks about *revisions*, which is the CVS name for file versions. The *working revision*, that is, the revision of the file in Alice's workspace, is 1.1, which is actually the first revision in CVS, not the second. Revision 1.0 doesn't exist.

The information also says that the latest revision available from the repository, that is, the *repository revision* is also 1.1. That is, Alice's revision of the file is *up-to-date*, which is consequently the current status of the file in the workspace. The other information is not important for the moment.

Also checking the status of the `TestComplex.java` file gives the following information:

```
$ cvs status TestComplex.java
===================================================================
File: TestComplex.java  Status: Up-to-date

   Working revision:    1.4
   Repository revision: 1.4     /REPO/mandelviz/TestComplex.java,v
   Sticky Tag:          (none)
   Sticky Date:         (none)
   Sticky Options:      (none)
```

Also this file is apparently up-to-date. However, `TestComplex.java` has revision number 1.4, that is, it has reached its fourth version. The boss has obviously already made a few changes to his original version of that file. When Alice uses `less` to inspect the two source codes, she is not surprised. The `Complex.java` file is really simple, just an empty interface skeleton. From the `TestComplex.java` file, however, she doesn't understand anything. There is no question of the relative difficulty for the boss in implementing such a mess. Obviously, he has had to make a few fixes before it actually worked[5].

The only important information provided from the `cvs status` command is actually in most cases that single line telling about the file name and its status. This means that Alice can get all important information about her workspace in a more compact form by being just a little bit more intelligent when issuing the command:

```
$ cvs status | grep Status
File: Complex.java      Status: Up-to-date
File: TestComplex.java  Status: Up-to-date
```

We see that, if leaving out the file name argument of `cvs status`, the command will respond with information about all files in the current directory.

3.3 The Modify Procedure

Now, Alice starts her implementation work. She edits the file `Complex.java` using Emacs, which happens to be her favourite text editor[6]. In order not to waste too much time on fixing difficult programming

[5] Basically, the complexity is a consequence of the fact that the boss didn't know of a tool called JUnit, or at least he didn't use it. Luckily, JUnit is studied later in our course, in order for us to be able to write much simpler test programs.

[6] Alice is a bit old-fashioned, in that she uses a separate text editor and compiler, rather than a modern integrated development environment like Eclipse. In fact, Alice looks forward to start using Eclipse later on with all its integrated support for things like CVS. And this will be very easy since she has worked with CVS at the command line level, which gives a good understanding of how it works.

errors, she takes great care to code in such a way that she can use the compiler extensively and often. That is, between every two consecutive compiling attempts, she makes only *the smallest possible* change to the code that leads towards the solution. Thanks to this intelligent coding approach, Alice never has any problems finding those small errors she still puts into the code. (Who doesn't fail to match parentheses or forget to write one of those semi-colons now and then?) Furthermore, by often executing the `TestComplex` program, she gets frequent feedback on her progress.

After a whole day of work, Alice has managed to complete the implementations of all the constructors, the different `get`-methods and the `add` procedure of `Complex.java`. It is soon time to call it a day and leave for the evening, but first Alice wants to secure her work. She now checks the status of the workspace again, finding that the output looks reasonable, that is, that CVS has detected her changes:

```
$ cvs status | grep Status
File: Complex.java      Status: Locally Modified
File: TestComplex.java  Status: Up-to-date
```

3.4 Committing Changes

First, Alice makes a final check that everything she has written is reasonably correct, that is, it is compilable without errors or warnings and that the `TestComplex` program runs successfully for those parts that she has implemented. In other words, she runs the Java compiler on her files:

```
$ javac *.java
```

until it returns without error messages, and then she runs the test program:

```
$ java TestComplex
```

that tests her implemented code.

If the test program runs successfully, Alice feels satisfied and she is ready to create a new revision. Therefore she starts with carefully shutting down all editors. Standing in the `mandelviz` directory, she issues the following command (This is quite critical!):

```
$ cvs commit
```

Now several things happen. First, CVS restarts Emacs with the following content in the editor window:

```
CVS: ----------------------------------------------------------------------
CVS: Enter Log.  Lines beginning with 'CVS:' are removed automatically
CVS:
CVS: Committing in .
CVS:
CVS: Modified Files:
CVS:    Complex.java
CVS: ----------------------------------------------------------------------
```

As is told in the text, all lines are comments. What Alice does is adding a log to the file, possibly ending up with something like this:

```
Added partial implementation of complex numbers:
    * creation of complex numbers
    * addition of complex numbers
CVS: ----------------------------------------------------------------------
CVS: Enter Log.  Lines beginning with 'CVS:' are removed automatically
CVS:
CVS: Committing in .
CVS:
```

```
CVS: Modified Files:
CVS:    Complex.java
CVS: ----------------------------------------------------------------------
```

Alice knows that it is crucial writing a good log message here, explaining the purpose of the change she has made to her workspace, in order for her to have a good possibility of later tracing back through the development path. This log text is kind of a diary, which Alice and others can consult later when analyzing the work done. The log text is saved by CVS, together with the files she commits into the repository.

After having saved the text and quit from Emacs (`<CTRL>-x <CTRL>-s <CTRL>-x <CTRL>-c`)[7] the commit procedure continues implicitly:

```
Checking in Complex.java;
/REPO/mandelviz/Complex.java  <--  Complex.java
new revision: 1.2; previous revision: 1.1
done
$
```

What has happened now is that Alice's changes have been added to the repository, creating a new revision 1.2 of `Complex.java`. CVS automatically leaves `TestComplex.java` out of account, since no modifications have been made to it.

Now checking the status of the workspace gives the following:

```
$ cvs status | grep Status
File: Complex.java      Status: Up-to-date
File: TestComplex.java  Status: Up-to-date
```

It should be noted that this commit has *not* destroyed the previous revision of `Complex.java`. It is still left in the repository, and it is still accessible, although the process of retrieving it is not described in this document. The reader is referred to the resources listed in Section 5.7.

During the following days at work, Alice repeats steps 3.2–3.4, until the implementation is finished. That is, at every working pass, she starts with getting updated on the current workspace status, then she implements and tests new functionality, and finally she makes a commit. If it wasn't for that `arg` method, Alice would have finished her implementation task all by herself. However, she runs into deep trouble on this method, which turns out to be terribly difficult to implement correctly. Help seems to be required — a fact that we shall return to in section 4.

3.5 Retrieving Historic Information

Staying for the moment in Alice's workspace, we will here just make a few additional remarks. First, these commit log messages can be inspected at any time from the workspace by issuing the following command:

```
$ cvs log
```

The output from this command, which is not reproduced here because of its verbosity, will specify for each revision of each file the commit log for that revision. That is, all commit logs for, e.g., `Complex.java`, including the initial revision 1.1 made by the boss and all the subsequent revisions 1.2, 1.3, 1.4,..., will be output to the terminal. Even for very small repositories, the output fills several screens, so it is usually convenient to pipe it through `more` or `less`, more or less.

Another command, which is actually interesting only on rare occasions, is

```
$ cvs history
```

The output consists of a number of lines, each specifying an operation made between the repository and some workspace. In our case, for example, the initial commits made by the boss, Alice's initial checkout, and her subsequent commits will be specified.

[7] Who said Emacs doesn't have a simple and intuitive user interface?

3.6 Summary

Up to this point, we have described all important parts of CVS that deal with single-user systems. Quite a few commands have been introduced, the most important of which are

cvs login Authenticate a user against a repository and save user and password information into a home directory file `.cvspass` for future access by CVS. (Made just once for each repository.)

cvs checkout Create a local workspace version of the newest revisions of the files of one repository module. (Made just once for each repository module.)

cvs status Check the files of a workspace for their current relation to the corresponding files of the repository. So far, we have seen that the files can be

- `Up-to-date`, meaning that they are unchanged in the workspace and no newer revision exists in the repository, or
- `Locally modified`, meaning that the current local workspace revision is the same as the latest revision of the corresponding file in the repository, but the file in the workspace has been subject to some editing by the workspace owner.

(The `cvs status` command can be issued at any time inside a workspace directory and it does neither modify the workspace nor the repository.)

cvs commit Bring the repository up-to-date with the local workspace revisions of files. Creates new revisions of the files changed since the last commit, but keeps all the old revisions. (May be issued at any time when the workspace files meet some criterion of stability, defined by the development process, for example, the state of being compilable and tested. Committing is a critical operation since it alters the central repository.)

cvs log Inspect the commit log messages of a series of revisions of the files currently in the workspace.

Although it seems practical to have a possibility of saving several versions of a source code project, the CVS usage is, so far, quite simple and not that impressive. But it gets really interesting when we allow multiple developers to concurrently work with the same code repository. This will, however, imply a need for refinement of our policy.

4 Case study: Multiple Users

One morning when Alice comes to her office and starts her day in the usual way by issuing a status request, she is hit by a new frightening experience:

```
$ cvs status | grep Status
File: Complex.java      Status: Needs Patch
File: TestComplex.java  Status: Up-to-date
```

Alice is convinced that both files were up-to-date when she left work last night, after having committed her afternoon changes into `Complex.java` revision 1.26. So, what has happened?

The answer is that Alice's boss, who has actually heard of Alice's troubles with the `arg` implementation, yesterday assigned another developer for the MandelViz project, who is supposed to give her a hand with the remaining parts of the `Complex` code. The new developer's name is Bob!

Bob, who is often called Night-owl by his work mates, started up his work on the project a couple of hours after Alice had left her office (and he had woke up from his beauty sleep). Just as Alice did a few weeks ago, Bob started up by checking out a fresh version of the `mandelviz` module as a local workspace (see Section 3.1). By requesting status and log messages and inspecting the code, Bob got some information of the current state of the `Complex` implementation and the changes made so far by

Alice. Since Alice's commit log messages turned out to be very informative, Bob easily concluded from them that `arg` was the big issue for her and decided to give it a try.

A couple of hours later Bob seemed to have solved the problem. And he was so exhausted that he had to go home for some sleep. (It is tiring sleeping during daytime.) Before leaving, however, he committed his changes, telling via the log message that the `arg` problem was solved and creating a new repository revision 1.27 of `Complex.java`.

When Alice starts working the day after, the situation is that she has an unaltered revision 1.26 in her workspace, but the repository contains a newer revision 1.27. (Alice concludes from the commit log that Bob is responsible for the new repository revision, and that he has fixed the `arg` method.)

4.1 The Update Procedure I (Non-merge Case)

What Alice has to do now is to somehow update her workspace to the more recent revision. Not surprising, this is made by issuing the following command:

```
$ cvs update
U Complex.java
```

Since Alice doesn't have any uncommitted changes in her workspace, CVS simply replaces her revision 1.26 of `Complex.java` with the newer revision 1.27, which was committed by Bob the night before. Alice's update has implied that she and Bob are now "synchronized" with respect to their workspaces.

In this case, with Alice and Bob working one at a time during different parts of the day and both of them carefully committing their respective changes before the other starts to work, the update procedure is particularly simple. As we shall see, there are more difficult cases.

4.2 The Update Procedure II (Simple Merge Case)

Alice starts her implementation day by giving `mul` a try. It turns out that this method isn't that easily implemented either, so she needs a couple of hours getting it right. While she is working, Bob comes back to his office. Without knowing that Alice is currently working with the code, he starts implementing `toString`. Hence, two people are now concurrently working with different parts of the same code in their respective workspaces without knowing about each other. Both of them have up-to-date or locally modified status on all their files, since none of them have committed any changes yet.

Let's say Alice finishes first and makes her commit. As usual, she writes a very good log message about the new `mul` implementation, and `Complex.java` revision 1.28 is created in the repository and in her workspace. Half an hour later, when Bob checks his workspace status, he gets the following result:

```
$ cvs status | grep Status
File: Complex.java      Status: Needs Merge
File: TestComplex.java  Status: Up-to-date
```

A log message request of `Complex.java` tells Bob that Alice has committed a new version of the file. Bob, who is almost finished with his `toString` implementation really doesn't want to lose the changes he has made. Luckily, CVS can help him do a merge of his and Alice's respective changes, so he saves his `Complex.java` file, quits from the editor and types as follows:

```
$ cvs update
RCS file: /REPO/mandelviz/Complex.java
retrieving revision 1.27
retrieving revision 1.28
Merging differences between 1.27 and 1.28 into Complex.java
M Complex.java
```

The `M` at the beginning of the output line means that CVS has now made a *merge* of the repository revision 1.28 and Bob's changed revision 1.27. The automatic merge was possible thanks to the fact that Alice and Bob had been editing *different parts* of the file.

Now requesting status information will characterize Bob's file as "locally modified", which is true as his changes are still not committed. However, through the update command CVS has incorporated Alice's implementation of `mul` into Bob's modified file, and his file is now considered being a changed revision 1.28 instead of 1.27. When he is finished with the `toString` implementation, he can commit the files, creating `Complex.java` revision 1.29.

What we have seen here is how CVS can be of help managing changes made to a file by two different developers working concurrently. Of course, this procedure extends to more than two developers in a natural way, but we should not forget the important restriction: The developers have to make *independent* changes, that is, changes at different locations in the code. Otherwise, CVS will not be able to do the automatic merge.

It should also be noted that an automatic merge made by CVS doesn't necessarily result in a file that is correct with respect to compilability or testability. The merge is purely textual and it doesn't take any notice of Java syntax or semantics. The result should therefore always be carefully checked by the user before it is committed. At every merge operation, CVS starts by making a copy of Bob's original file:

```
$ ls -a
CVS/        .#Complex.java.1.27     Complex.java     TestComplex.java
```

Here, the file `.#Complex.java.1.27` is an exact copy of Bob's version of the file before the merge. If it turns out that CVS has destroyed `Complex.java` completely by automatically merging in Alice's changes, a last backup solution might be to throw the merged version away in favour of Bob's solution, which of course also would destroy Alice's implementation of `mul`.

4.3 The Update Procedure III (Less Simple Merge Case)

But what if Bob had started implementing `mul` instead — just as Alice? In that case, the update procedure from above wouldn't have passed through in such an easy way. Let's say Alice has committed the following code into revision 1.28:

```
public Complex mul(Complex c) {
    double newRe = re * c.re - im * c.im;
    im = re * c.im + im * c.re;
    re = newre;
    return this;
}
```

Bob's implementation in his checked-out revision 1.27 looks like this:

```
public Complex mul(Complex c) {
    double oldRe = re;
    re = re * c.re - im * c.im;
    im = oldRe * c.im + im * c.re;
    return this;
}
```

To merge the two solutions in this case would mean to *choose* one of the two implementations, preferably the best of them, and to throw the other one away. Alice's solution is obviously the best one here, since it is correct and Bob's corresponding implementation is wrong, but CVS cannot possibly know anything about that. So, what happens?

```
$ cvs status | grep Status
File: Complex.java      Status: Needs Merge
File: TestComplex.java  Status: Up-to-date
```

```
$ cvs update
RCS file: /REPO/mandelviz/Complex.java
retrieving revision 1.27
retrieving revision 1.28
Merging differences between 1.27 and 1.28 into Complex.java
rcsmerge: warning: conflicts during merge
cvs update: conflicts found in Complex.java
C Complex.java
```

CVS didn't manage to merge the two changes because they were conflicting. The merge is left to be made by Bob manually. Since a merge has been tried, CVS has stored Bob's original version into a new file:

```
$ ls -a
CVS/       .#Complex.java.1.27     Complex.java     TestComplex.java
```

This time, however, the file `Complex.java` will contain *both* Alice's committed revision 1.28 and Bob's modified revision 1.27. It looks like this:

```
    public Complex mul(Complex c) {
<<<<<<< Complex.java
        double oldRe = re;
        re = re * c.re - im * c.im;
        im = oldRe * c.im + im * c.re;
=======
        double newRe = re * c.re - im * c.im;
        im = re * c.im + im * c.re;
        re = newre;
>>>>>>> 1.28
        return this;
    }
```

As we can see above, CVS has written Bob's version first, beginning with a special <-tag, which is certainly not Java-code. After the =-tag, Alice's committed revision is written, ending with a >-tag, also telling Bob about the revision number of her file. Now, Bob has to use his editor to keep one of the implementations and to remove the other and the CVS tags. In this case, the result will be equal to Alice's committed revision 1.28, which Bob reluctantly admits is the better one. After having saved the file and quit the editor, Bob can commit his file, creating revision 1.29.

4.4 Summary

In this section, we have dealt with the problem of synchronizing multiple developers in a single project. It is important to remember that all merging is made in user workspaces; no operation is allowed to jeopardize the integrity of the repository.

We have seen two more possibilities of workspace file status:

- `Needs Patch` means that the local file revision is unmodified, but a new revision has been committed to the repository by some other developer.

  An update operation is always simple when a file is in this state since it can simply be exchanged with the newer revision without losing any information.

- `Needs Merge` means that the local file is modified and that a newer revision has been committed to the repository. It can be interpreted like `Locally Modified + Needs Patch`.

  An update request will have one of the following three effects:

1. CVS makes the merge automatically with a correct result; or
2. CVS makes the merge automatically with a faulty result; or
3. CVS fails to merge because the different changes are conflicting.

The second of these possibilities implies that it is essential to always manually check a merge, even when it seems to have been successfully and automatically fulfilled by CVS itself.

When some file of a workspace reaches a state where it needs to be patched or merged, it is generally recommended to do an update as soon as possible, in order to minimize the possible trouble that might come up regarding merge conflicts. No file will ever leave any of these states without the user doing an update of his workspace, and the possibility of trouble implied by merge conflicts will not decline by waiting.

The most difficult of these cases is of course when a conflict needs to be manually resolved. Empirical studies show, however, that this case is fairly uncommon. Furthermore, when merge conflicts do occur, the manual merge is often fairly easy.

Good communication within the team is very important to ensure that developers do not work on the same part of some source code without reason. With better communication between Bob and Alice, they should never had both tried to implement the `mul` method. But sometimes it can be motivated to work in parallel on the same piece of code. For example, to make progress on two different problems that both happen to affect the same method. Good communication within the team can then help simplifying the unevitable upcoming file merges.

5 Additional CVS Functions

We need a few more functions of CVS, in order to use it on a daily basis. For example, you may need to add, remove or rename files or directories, or add new modules to the repository.

5.1 Adding Files to a Module

Let's look into what happens when Charlie also gets involved with the development of MandelViz. Charlie, who is a fan of higher mathematics (Alice actually thinks he is a real nerd), has found an *extremely* elegant new way of describing the Mandelbrot set in terms of *quaternions* instead of ordinary complex numbers[8]. Since Charlie's new model is so complicated that neither Alice nor Bob understands even half of it, we leave out the details of it from this document.

A natural way for Charlie to use his new ideas is to start by implementing a new class `Quaternion` which "extends Complex", something that he can obviously do in five minutes in his sleep. Since the result is quite good, Charlie feels that he would like to "publish" his new class in the repository, along with the other files. That is, he wants to put it under version control. Just issuing a commit command from his workspace will not suffice, since CVS doesn't know about his new file. So, he types in the following command, standing in his `mandelviz` workspace directory:

```
$ cvs add Quaternion.java
cvs add: use 'cvs commit' to add this file permanently
```

What Charlie has now done is telling CVS about the existence of his new file, and the next time he issues a commit, it will be copied into the repository module and be given the revision number 1.1.

```
$ cvs status | grep Status
File: Complex.java      Status: Up-to-date
```

[8] A quaternion is an example of a so-called hypercomplex number with three imaginary units, i, j, and k, instead of simply that boring i. These units are defined in such a way that $i^2 = j^2 = k^2 = ijk = -1$ and furthermore, due to Hamilton, $ij = -ji = k$, $jk = -kj = i$, and $ki = -ik = j$. The set of quaternions, denoted by $\mathbf{H}$, is a division algebra under addition and multiplication.

```
File: TestComplex.java  Status: Up-to-date
File: Quaternion.java   Status: Locally Added
$ cvs commit
Checking in Quaternion.java;
/REPO/mandelviz/Quaternion.java,v  <--  Quaternion.java
initial revision: 1.1
done
```

Any user who issues an update request after Charlie's commit will receive a copy of the new file.

5.2 Removing Files from a Module

In order to remove a file from a module, CVS supports a command similar to the `add` command above. Probably, Alice will soon issue the following commands in her workspace:

```
$ rm Quaternion.java
$ cvs remove Quaternion.java
cvs remove: use 'cvs commit'  to remove this file permanently
```

After a commit, the `Quaternion.java` file will be removed from the repository in all subsequent versions. Users updating their workspaces after this commit will get their copy of the file removed.

But what if someone has a locally modified version of the file? (In this case it must be Charlie, since none of the other developers would even touch his file with a pair of tongs.) CVS (correctly) considers this being a conflict.

Charlie checks his workspace status:

```
$ cvs status Quaternion.java | grep Status
File: Quaternion.java   Status: Unresolved Conflict
```

So can he solve this by updating? (Think about this before reading on...)

```
$ cvs update
cvs update: conflict: Quaternion.java is modified but no longer
                      in the repository
C Quaternion.java
```

Answer: No! The only thing Charlie can do about this is to remove his file (possibly securing it in a safe place outside his workspace) and then issue another update request:

```
$ cvs update
cvs update: warning: Quaternion.java is not (any longer) pertinent
```

which it perhaps isn't. Perhaps he should talk to Alice....

5.3 A Note about Subdirectories

Most software projects do not reside in one single root directory, but are distributed over a tree of subdirectories.[9] CVS provides version control only for files, not for directories. So how do we extend a module to include subdirectories?

[9]The author, however, knows one person who keeps all his files in his home directory root. At the time of this writing, the number of files were 3.419, none of them were directory files. Obviously we all have different ways of bringing order into our lives!

Suppose Dave is joining the MandelViz project with the task of producing a user interface. As we all know, those Swing-GUIs tend to increase the number of files in a project dramatically. So Dave wants to put all these user interface files in a subdirectory package GUI, which he creates in his `mandelviz` workspace:

```
$ mkdir GUI
```

Now, he populates the directory with lots of boring user interface code. There is only one problem: He can not `add` the files, because the GUI directory is not a CVS directory. That is, it has no `CVS` subdirectory, which all workspace directories should have.

The solution is quite strange: He has to turn the GUI directory into a CVS directory by *adding* it!

```
$ pwd
/home/dave/salmonproject/mandelviz/GUI
$ cd ..
$ cvs add GUI
Directory /home/dave/salmonproject/mandelviz/GUI added to the repository
```

Note that this operation is not the same as adding files! For example, Dave doesn't need to confirm the operation by committing. When doing `cvs add` on a directory, the only effect is that a `CVS` subdirectory is created in that directory. Now Dave can add his GUI-files:

```
$ cd GUI
$ cvs add *.java
cvs add: use 'cvs commit' to add these files permanently
```

When you do the `update` and `commit` commands, it is very important to make sure that you do this at the module root level. CVS will automatically traverse all subdirectories recursively. You should never do `update` or `commit` down in a subdirectory. The reason for this is that if individual subdirectories (or individual files) are updated or committed, it is easy to end up with an inconsistent workspace or repository. For example, if Dave does update and then commit on the new GUI only, one of the other developers might have committed changes to the main functionality that are inconsistent with the new version of the GUI.

Finally, it should be noted that even if files and directories are removed in a workspace, they will not be removed in the repository. This is because the repository contains all versions of all files. So if you browse the repository, don't be surprised to see a lot of old files and directories there.

5.4 Renaming/Moving Files

There is no specific support in CVS for renaming or moving files or directories. Instead of renaming, what you need to do is to remove the file and add it under the new name. CVS will view this as two separate unrelated operations. This means that the new file will be committed with the revision number 1.1, and all parts of its history will be "forgotten". Of course, if you remember the old name, you can find the earlier revision history for that file.

The lack of support for rename is a deficiency of CVS—the frequent need for doing such operations is quite obvious.

5.5 IDEs, modules, and repositories

Adding and removing files is obviously a bit cumbersome in CVS. In many IDEs, e.g. Eclipse, there is built-in support for CVS. If you rename, add or remove files in your workspace, Eclipse will automatically do the `cvs add` and `cvs remove` commands for you when you commit. This simplifies working with CVS a lot. You normally only have to change your files as usual, and then do update and commit to synchronize with the repository. Eclipse also has interactive support for merging files, and for looking at differences between file versions.

The Eclipse CVS plugin performs atomic commits. That is, if you try to commit, and any of the changed files needs patching or merging, none of the files will be committed. Note, however, that it does not perform *strict* atomic commits. You therefore still need to do update right before commit: other files (that you have not changed) could have been committed by other developers after your latest update.

Eclipse lets you easily put an existing non-CVS project under CVS version control, i.e., to add it as a new module in an existing repository. This is called "share project" in Eclipse. If you are not using an IDE, you can do this manually, using the cvs command `import`.

Eclipse also gives you the opportunity to browse the contents of repositories. To create a new repository, however, is done directly at the CVS server machine, and you would need to contact the CVS administrator to get help with that.

5.6 Additional features

We have discussed many of the functions of CVS in this paper, but there are lots of features left out, for example:

- *release tagging*, for marking a set of different revisions of files that together constitute a source code release version;
- *historic file revision restore*, for retrieving old revisions of individual files;
- *repository branching*, for creating several development lines in one single project — instead of having one trunk of evolving files in the repository, we may have several; or
- *repository export*, for creating checked out source code releases, suited for shipment to a customer.

5.7 Learn more

Those who want to learn more might appreciate the following resources:

- `man cvs`

  As all Unix manual pages, this one is intended as a quick reference and is not very well suited for learning how to use the basic functions of CVS.

- Per Cederqvist et al: *Version Management with CVS*

  This is the official reference manual of CVS, often simply called "*The Cederqvist*". You can find it in wiki and HTML form at, e.g., `http://ximbiot.com/cvs/manual/`

- `http://savannah.nongnu.org/projects/cvs`

  The official home page for CVS.

A CVS Best Practices in Brief

- Only checkout a module once, then do updates of your workspace.
- Do not ever checkout modules inside of each other.
- Stay in sync with the repository and keep up your awareness of other people's changes by updating often.
- Share your changes by committing often.
- Always commit from the module root directory.
- Keep the repository clean: only commit working versions of your software.
- Simulate strict atomic commits by always doing update before commit.
- Write short informative commit logs.
- Do not work outside the workspace.
- Generally, do not commit binary files into the repository.
- Try to keep up correctness and integrity of your own workspace.
- Always check merges made by CVS.
- Solve conflicts immediately.
- Maintain a consistent coding style in order to avoid unnecessary merge conflicts.
- Communicate with the other developers of the team.