MEMORIAL UNIVERSITY
Faculty of Engineering and Applied Science

Lab Handout
ENGI3891

16,23 Sep 2015

Lab 0: Version control

1 Purpose and outcomes

This lab will give you hands-on experience with an essential development tool: a version control system (VCS). Serious software development is always done with at least one VCS, and so is the **submission of assignments** in this course. After completing this lab, you should:

1. understand client/server VCS architecture,
2. have skills required to submit assignments for review and
3. be able to manage file version merging.

2 Preparation

You need a copy of the Subversion version control system installed on your computer — ECS will have installed it on the lab computers, but it may be illustrative if you and your lab partner work on separate computers. Mac OS X comes with Subversion already installed; Windows users may wish to install command-line tools from `http://www.visualsvn.com/downloads` and/or more featureful GUI tools from `http://tortoisesvn.net`. I highly recommend installing Subversion on your own computer now: **you will need it to submit assignments**!

You will also some details from ECS: credentials (a username and password) and the URL (Uniform Resource Locator) for the ENGI 3891 Subversion repository, e.g., `svn+ssh://tera.engr.mun.ca/~anderson/example`).

3 Theory

Software development is a collaborative activity. Even solo projects are effectively collaborations between the *present* and *future* me ("Where was I? Why did I do it this way?") or between a *developer* and a *customer*[1]. The vast majority of software projects, however, are created by teams.

3.1 The Problem(s)

Consider a development team with three collaborators, Alice, Bob and Charlie, and no VCS. Whenever Alice makes a change in their jointly-developed software, she emails the latest version to Bob and Charlie, who can then copy Alice's version onto their computers. This (partially) solves the *versioning* problem: if a customer reports a problem with a particular version of the software, any of the developers can find that version in their email archive and try to reproduce and fix the problem. However, this only works as long as one person changes the software at a time, e-mailing out "the new version" before anybody else can make other changes. What happens if Bob and Charlie are working at the same time, making changes to the same source code that Alice is working on? Perhaps Alice changed the calculation of certain values to make it more efficient while Bob changed the user interface and Charlie modified the network code. An e-mail–based system of source code management may partially solve the versioning problem but it does not solve the *consistency* problem: ensuring that everyone agrees what "the current version" means, even when different people are working on the code at the same time.

[1] In this course, you may think of the TAs and myself as your customers!

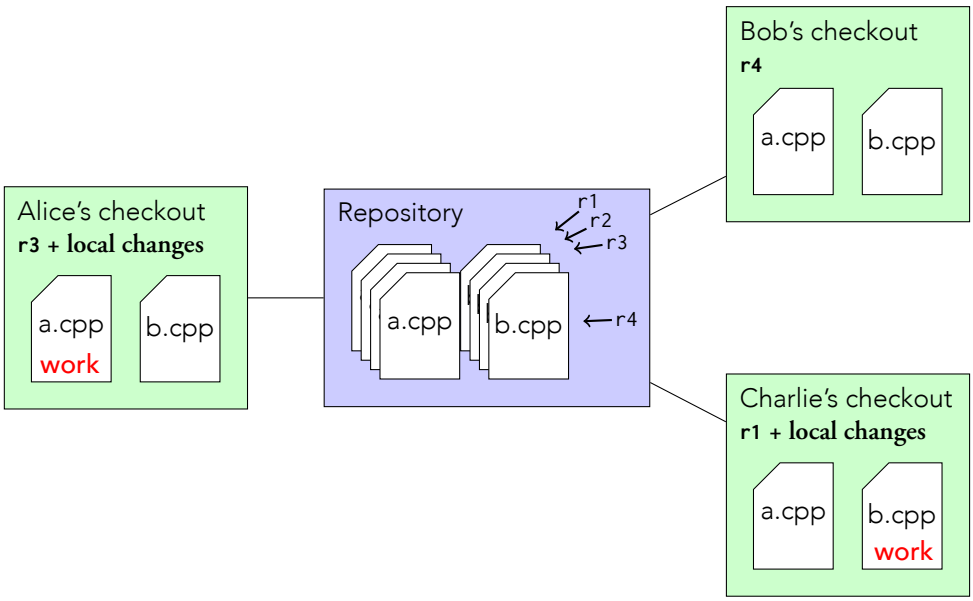

Figure 1: Each user interacts independently with the repository, which stores canonical versions of source files. User *checkouts* hold different versions of files, with user changes that have not yet been *committed* to the repository.

3.2 The Solution

One solution to the consistency problem is to have a single authoritative server that holds the *canonical* version of the source code and enforces a sequential ordering of changes to it. This server is called a *repository*, and each developer communicates with it as shown in figure 1 rather than sending changes directly to each other.

In this model, the central repository stores a complete history of all versions — *revisions* — of the source code. Before starting work, Bob polls the repository for the latest version of the software and updates his *local checkout* to that version. His version control software asks the respository, "I have `r3`, what changes do I need to apply in order to get `r4`?" The repository will reply with the set of changes that have been made to the source files between `r3` and `r4`, expressed as a *diff* (described in section 3.2.1).

3.2.1 Diffs

The differences between two text files can be expressed in a format called *unified diff*. If Alice adds a new function to a file, the extra lines can be expressed with plus signs as in listing 1. Removing lines is expressed with minus signs, and changing lines is expressed with a combination of the two, as shown in listing 2 on the next page. When Alice, Bob and Charlie communicate with the repository, sending and receiving changes, you can think of them sending these *diffs* back and forth.

Listing 1: Adding lines to a file, expressed in the unified diff format.

```
+#include <logarithms.h>
 #include <iostream>
 using namespace std;

 int main()
 {
         cout << "Hello, world!" << endl;
+        cout << "log2(42) is " << log2(42) << endl;

         return 0;
 }
```

Listing 2: Changing a line by removing the old version and adding a new version.

```
 int main()
 {
-        cout << "Hello, world!" << endl;
+        cout << "Hello, everyone!" << endl;

         return 0;
 }
```

3.2.2 Merging

In figure 1 on the facing page, what happens when Charlie wants to commit his changes to the repository? Charlie started with revision `r1` and then modified `b.cpp`, but the repository is now on revision `r4`. Before Charlie commits his work to the repository, he must first update his checkout to the most recent revision (`r4`) and then *merge* his changes into the `r4` version of `b.cpp`. For instance, if the difference in `b.cpp` between `r1` and `r4` is:

```
 int main()
 {
         cout << "Hello, world!" << endl;

         do_some_setup();
+        do_some_real_work();

         return 0;
 }
```

and Charlie's change to `b.cpp` can be expressed as:

```
 int main()
 {
-        cout << "Hello, world!" << endl;
+        cout << "Hello, everyone!" << endl;

         do_some_setup();
```

then Charlie's revision-control software can automatically *merge* the two changes together because they do not interfere with each other: one changes an existing line while the other adds a new line. After merging, Charlie's local changes will be based on `r4`:

```
 int main()
 {
-        cout << "Hello, world!" << endl;
+        cout << "Hello, everyone!" << endl;

         do_some_setup();
         do_some_real_work();
```

Charlie can then commit these changes as `r5`.

3.2.3 Conflicts

What if, however, Charlie and Bob both changed the same part of the same source file? Suppose the differences in `b.cpp` between `r1` and `r4` (changes that Bob committed while Charlie was doing his own work) were:

```
 int main()
 {
-        cout << "Hello, world!" << endl;
+        cout << "Let's do some setup." << endl;
-
         do_some_setup();
+
+        cout << "Let's do some real work!" << endl;
+        do_some_real_work();

         return 0;
 }
```

In this case, Charlie wants to change the text on line 3 from "Hello, world!" to "Hello, everybody!" while someone else (Alice or Bob) has already changed it from "Hello, world!" to "Let's do some setup." What can Charlie's revision control client do to merge these changes?

Unfortunately, the answer is "nothing". Charlie's client software only understands *that* a file has changed, not *why* it changed. Why did Bob change `b.cpp` in one way while Charlie changed it in another way? Whose change is more important? Whose change should we keep, and whose should we discard? These technical, or even social, questions must be answered by the programmer, not the programmer's tools. When Charlie updates his local checkout, his revision control software will merge what it can, but it will insert *conflict markers* where it cannot automatically merge the conflicting versions. These markers remind Charlie to *resolve* the conflict manually:

```
int main()
{
<<<<<<<
        cout << "Hello, everyone!" << endl;
=======
        cout << "Let's do some setup." << endl;
>>>>>>>

        do_some_setup();

        cout << "Let's do some real work!" << endl;
        do_some_real_work();

        return 0;
}
```

Generally, Charlie has three options for resolving the conflict: choose his changes, choose Bob's changes or find a way to combine the two. In this case, Charlie could easily combine the two output strings, but sometimes there will be more complicated conflicts that require negotiation between Bob and Charlie or another look at the software's specification, customer use cases, etc.

Once Charlie resolves the conflict, the merge is complete and he can commit to the repository as before.

4 Procedure

As a pair of lab partners, follow the instructions given below. You may work on the same computer or separate computers, but follow the instructions together, in order: you both need to observe each other's work. **Whenever you encounter the text <<output>> in the instructions,** make a note of the program output you see on the console. These outputs should go in your lab report, along with answers to any questions below.

Partner A & B

Open a command prompt:

Mac OS X: ⌘ + Space ▷ `Terminal` ⏎

Windows: Start ⟩ Run ▷ `cmd` ⏎

BSD/Linux: You're probably familiar with how to access the command line.

Partner A & B

Create a directory for your lab work.[2]

```
H:\> mkdir engi3891
H:\> cd engi3891
H:\engi3891>
```

Partner A

Check out your assigned Subversion repository.

```
H:\engi3891> svn checkout --username=${username} https://${repo}/lab0/${group} lab0
<<output>>
H:\engi3891> cd lab0
H:\lab0>
```

Partner A

Save the following C++ program as `hello.cpp` in the `lab0` directory:

```
1 #include <iostream>
2
3 int main(int argc, char *argv[])
4 {
5         std::cout << "Hello, world!\n";
6
7         return 0;
8 }
```

Partner A

Add `hello.cpp` to the checkout and commit it to the repository:

```
H:\lab0> svn add hello.cpp
<<output>>
H:\lab0> svn commit -m "Added hello.cpp." hello.cpp
<<output>>
```

[2] The command-line interaction in this lab handout uses a prompt (`[hostname:~]$`) like one you'd find on Mac OS X or any open-source software distribution. Windows users will see a prompt more like `C:\My Documents\>`. For this lab, the difference doesn't matter.

Partner B
Check out the repository.

```
H:\engi3891> svn checkout --username=${username} https://${repo}/lab0/${group} lab0
<<output>>
H:\lab0> cd lab0
H:\lab0>
```

Partner B
Modify `hello.cpp` to print out your name instead of "Hello, world!", then commit your changes:

```
H:\lab0> svn status
<<output>>
H:\lab0> svn diff
<<output>>
H:\lab0> svn commit -m "Personalize the welcome message." hello.cpp
<<output>>
H:\lab0> svn status
<<output>>
```

Partner A
Observe differences (if any) to local checkout.
Question: *explain what you observe.*

```
H:\lab0> svn status
<<output>>
H:\lab0> svn diff hello.cpp
<<output>>
```

Partner A
Make the following change to `hello.cpp`:

```
 #include <iostream>

+// This is a harmless comment.
 int main(int argc, char *argv[])
 {
        std::cout << "Hello, world!\n";
```

Partner A
Update to `r2`.
Question: *why does this cause a merge rather than a conflict?*

```
H:\lab0> svn update
<<output>>
H:\lab0> svn status
<<output>>
H:\lab0> svn diff
<<output>>
```

Partner A
Make further changes to `hello.cpp`:

```
1  #include <iostream>
2
3  // This is a harmless comment.
4  int main(int argc, char *argv[])
5  {
6  -       std::cout << "My name is Partner B.\n";
7  +       std::cout << "I'd like this message to say Partner A.\n";
8
9          return 0;
10 }
```

Partner A
Commit changes.

```
H:\lab0> svn diff
<<output>>
H:\lab0> svn commit -m "Made welcome message use my name." hello.cpp
<<output>>
```

Partner B
Make the following changes to `hello.cpp`:

```
1  #include <iostream>
2
3  int main(int argc, char *argv[])
4  {
5  -       std::cout << "My name is Partner B.\n";
6  +       std::cout << "I really like the name Partner B.\n";
7
8          return 0;
9  }
```

Partner B
Attempt to update to `r3`.
Question: *why isn't the merge successful?*

```
H:\lab0> svn update
<<output>>
Select: df
<<output>>
```

Partner B
Choose 'p' ("postpone").

```
Select: (p) postpone, (df) show diff, (e) edit file, (m) merge,
        (mc) my side of conflict, (tc) their side of conflict,
        (s) show all options: p
```

```
<<output>>
H:\lab0> svn status
<<output>>
H:\lab0> svn diff
<<output>>
```

Partner B

Fix the conflict (see section 3.2.3 on page 4), then commit:

```
H:\lab0> svn commit -m "Resolve conflict over welcome message." hello.cpp
<<output>>
H:\lab0> svn resolved hello.cpp
<<output>>
H:\lab0> svn commit -m "Resolve conflict over welcome message." hello.cpp
<<output>>
```

Partner A

Update to the latest revision.

```
H:\lab0> svn update
<<output>>
```

Partner A

Look at the revision history:

```
H:\lab0> svn log
<<output>>
```

Partner A

Update to a time between `r2` and `r3`.

```
H:\lab0> svn update -r "{2014-09-10 10:06}"  # use time from your log
<<output>>
```

5 Endnote: assignments

The method of revision control that you have just learned is how we will handle assignment submission in this course. To grade your work, I will run `svn update -r 'deadline'`; if you commit your assignments ten seconds after the precise deadline, my `svn update` will not see your work. I therefore recommend that you **commit early and often**: as soon as your assignment compiles, **commit it**. After you've implemented some functionality, **commit**. Any time you add functionality to your code, **commit**.

Your goal should be to create a series of revisions in the repository, each implementing more functionality than the last. This incremental progress towards a defined goal is a very simple example of a **software engineering methodology**. More complex methodologies will become very important as you progress through your career.