Lunds tekniska högskola
Datavetenskap, Nov 2013
EDA260 Programvaruutveckling i grupp—projekt
Labb 2 (CVS): Labbhandledning

Lab Exercise
CVS: A centralized version control system

Goal

This lab is intended to demonstrate basic usage of a centralized version control system, namely CVS. You are supposed to work in pairs on one computer, simulating a small software team developing a simple program component under version control. The lab also demonstrates the XP practices of Collective Ownership and Continuous Integration.

Note!

- During the lab, keep notes of what you do.
- You may use a text file instead of paper for your notes, if you think that is easier.
- Write down the commands that you use and brief descriptions of the results.
- Write down brief answers to all questions appearing in this lab text.
- Be prepared to discuss your notes with the lab coach.

1 Working from the command line

Later on in the project, you will use version control tools in Eclipse. In this lab exercise, however, you will work with the version control tools directly from the command line, and edit files using an ordinary text editor. This will give you a better insight into how the tools work, than if you had only used them via Eclipse. If you are unfamiliar with unix concepts and commands, you can find information in "Introduktion till LTH:s Unixdatorer" (`http://www.ddg.lth.se/perf/unix/unix-x.pdf`), and you can also google different commands.

2 Getting started with CVS

Before you start using CVS you should add a configuration file and a password file in your home directory. You should also create a suitable directory where you can check out modules from CVS. Here is how you do these things:

Configuration file The configuration file should have the name `.cvsrc` and be placed in your home directory. It defines suitable default behavior for the different CVS commands. At the course web site, there is a prepared configuration file that you should copy to your home directory. Feel free to look at the file using an editor, but if you change it you are completely on your own. A few of you, however, might want to change the default editor used for editing commit logs. (Our default choice is to use NANO, but you could change this to, e.g., EMACS or VI if you like.)

Password file Create a CVS password file, unreadable for everyone but you:

```
$ touch ~/.cvspass
$ chmod 600 ~/.cvspass
```

Directory Create a suitable directory where you can work with CVS in this lab, e.g., `pvglab2`. Enter the new directory! Now you can start working with CVS.

3 Single Developer

In this lab exercise, you will use a module `bstmapNN` that you can both check out and commit changes to. The `NN` in the module name is a two-digit number which will be given to you and your partner by the lab coach. In this part of the lab, the two of you are supposed to work as an ordinary XP developer pair, together implementing a piece of software. We will start by exploring CVS as a single developer (or rather, a single *pair* of developers, since you are pair programming).

The software is a class `BSTMap` that implements a map using a binary search tree. It is intended to become an implementation of the standard Java class `AbstractMap`. Most of the methods in BSTMap are, so far, not fully implemented. This is indicated by letting them throw a specialized run-time exception.

3.1 Checkout

You can find the lab module `bstmapNN` in the following repository

```
:pserver:username@cvs.cs.lth.se:/local/cvs/eda260/cvslab
```

where *`username`* is your CVS pserver username. You should have gotten an email with this information earlier. If not, ask your lab coach.

Login and checkout Login to the repository and check out the appropriate module.

```
cvs -d :pserver:username@cvs.cs.lth.se:/local/cvs/eda260/cvslab login
cvs -d :pserver:username@cvs.cs.lth.se:/local/cvs/eda260/cvslab checkout bstmapNN
```

Look at status info Are the files up to date?

```
cvs status
```

Look at revision info What is the current revision number of the `BSTMap` class? Who has committed this revision?

```
cvs log
```

Compile Check that all source code is compilable without errors: `javac *.java`

Look at the code Take a brief look at the code. *Note!* You will just do a few small changes to this code during the lab exercise in order to experiment with CVS. You don't need to understand all the code in detail.

3.2 Modify

Now, we assume that the two of you are developer Alice. The scenario is that you will implement a few of the methods, but, for this exercise, it is fine if you fake the implementations. For example, "implement" a method by letting it return some constant like zero, 42, or null. Or change it by adding a comment. However, make sure that after any change, the code still compiles without errors.

Your first task is to make an implementation of `BSTMap.size()`.

Implement Implement `size` and then check that the status of `BSTMap.java` has changed to `Locally Modified`.

Try diff Try the following command:

```
$ cvs diff BSTMap.java
```

What information is provided? Try to explain in what way the output text could be used.

3.3 Commit

Before committing, we should make sure that we are committing a clean version of the code into the repository:

- The code *must not* contain programming errors, i.e., it has to be compilable without error messages or warnings.
- The code *must* have been tested, so that we know that it behaves as intended.

Both these requirements should be quite natural. By keeping the repository clean, any developer can check out a fresh, working, compilable version at any time.

In this lab we are just simulating development, and we will therefore focus only on the compilability requirement. Testing will be introduced later in the course. The compilability requirement is, however, very important:

From now on, in this lab session and in the upcoming project task, each and every line of code that is committed into the repository, and thereby integrated with common team resources, has to be compilable without errors or warnings.

Commit When you have implemented the `size` method, and verified that it is compilable, try to do a commit.

```
cvs commit -m "Implemented the size method"
```

If you leave out the -m option and the message, an editor will open where you should write the commit log message. After saving and quitting the editor, the commit proceeds. Use concise messages in any case.

Check status After the commit, check the workspace status and the commit logs; verify that they are correct. What is now the revision number of `BSTMap.java`? What is its status? What is returned from the diff command?

3.4 Repeat and reflect

Establish and improve your skills by choosing another method of `BSTMap` and repeat Sections 3.2–3.3. **Make sure you understand the process of checking file status, doing diff, and committing changes.**

- What is the exact meaning of `Up-to-date` and `Locally Modified` in the context of the CVS status command?
- What information is relevant to put in the commit log messages?
- Write a list of the CVS commands you have seen so far and give a short description of their purposes. Indicate which commands that modify the repository or the workspace, respectively. Which commands change neither of these?

4 Multiple Developers

In this section, we are going to simulate multiple developers by the introduction of a new developer in the team. His name is Bob! The role of Bob should be played by the one of you who is not already logged in at the workstation. Now, we want Bob to login to the LTH student system. This can be done in the following different ways:

separate workstation If you are lucky, there might be a free workstation beside the one you are already using. In that case, Bob can login on that workstation.

your own laptop If one of you have your own personal laptop with you, you can start up a terminal window on that laptop, and let Bob do a remote login to the LTH student system:

```
$ ssh -X bob@login.student.lth.se
```

same workstation A third alternative is that you use the same workstation that you are already using, and let Bob login in a new terminal window:

```
$ ssh -X bob@login.student.lth.se
```

Note that you are doing a Unix login now, so replace `bob` by your student LTH Unix username, and use your usual Unix password. The -X option will make windows created from that new shell to appear on the same screen.

Write into the tabular below which user account that corresponds to Alice and Bob, respectively:

	LTH student account
Alice:	
Bob:	

From now on we are going to use a box in the margin with either the name Alice or the name Bob to indicate who is currently issuing commands. In case you are using the same workstation for Alice and Bob, keep close track of which terminal window belongs to whom to not get confused.

4.1 Configuration

Bob's Unix account needs to be configured for CVS use. Redo the steps of Section 2 on Bob's account as well. Hint: you can easily copy the .cvsrc file from Alice to Bob by going to Bob's home directory and doing: *Bob*

```
$ cp ~alice/.cvsrc .
```

4.2 Checkout

Login to the repository server, now using the CVS username of Bob, and check out the same `bstmapNN` module into his workspace. Check by using the status command that he receives the latest version of `BSTMap.java`. Also inspect the commit log, to see the messages from Alice's earlier commits.

4.3 Modify

Now, Bob uses his editor to implement a constructor of `BSTMap.java`. (Again, do not focus on difficult programming issues. Simply changing the throw statement into a comment saying "`Constructor now implemented`" or "`Bob was here, 2008`", is a perfect implementation here. However, making a *compilable* implementation is *absolutely essential*, as it will be committed.) Check that Bob's file is now `Locally Modified`, compile it and quit from his editor. Do *not* commit the changes yet.

Alice

At the same time (well, almost), Alice is implementing one of the methods from the end of the class definition. Choose any method you like and implement it. Do *not*, however, implement the same constructor as Bob did. What is the status of Alice's file after her editing? Check compilability and quit from the editor.

4.4 Merge without textual conflicts

When Alice's implementation is done, commit her changes. Remember to specify a good log message, accurately specifying what has been done by Alice. Check the status of Alice's file.

Bob

Now, try to commit Bob's changes.

- What happens, i.e., what instruction is given by CVS? Why? What is the status of Bob's file?
- What do you think will happen if you follow the instruction given?
- Try it! What happened? Check the result using the editor. Then compile!
- Why is it important to check the result before the compilation attempt and the commit?
- How did Bob's action affect Alice's workspace? What is the status of Bob's file now?

Try again committing the file. What happened this time? Why? How did this affect Alice's workspace? What is the status of Alice's file? Solve the problem!

4.5 Merge with textual conflicts

Now, repeat Section 4.3, but try both to implement the same method in different ways. Let Bob as well as Alice make their changes before anyone of them commit.

- Let Bob commit first. What problems occur? *Bob*
- What is now the status of Alice's file? What happens if she updates? Why? How about the file status after the update operation? *Alice*
- Fix the file using the editor. What is the status of Alice's file now?
- Try to commit Alice's changes! What happens? Why?

4.6 Repeat and Reflect

Repeat Sections 4.3–4.5 again in order to strengthen your skills.

- What is the exact meaning of `Needs Patch` in the context of the CVS status command?
- What other file status descriptions have you seen? What do they mean exactly?
- Complete your list of CVS commands and describe how they affect the workspace and/or the repository.

5 Committing different files

In the merge scenarios above, Alice and Bob have been editing the same file. Now we will look at what problems can occur when they edit different files.

5.1 Alice adds a method

First, Alice adds a new method `doit()` to class `BSTMap.java`: *Alice*

- Update, to make sure you are working with the latest files.
- Edit `BSTMap.java` to add the new `doit()` method.
- Make sure everything compiles.
- Commit.

5.2 Bob performs a change

Bob is now ready to continue with his work: *Bob*

- Do update to make sure you are working with the latest version of the software
- Compile, to be sure you have a consistent version of the source code.
- Look at the commit logs to see what the recent changes are.

Now, Bob takes a look at the new `doit` method. He really likes it, but decides to make some enhancements. He adds a comment inside, and he also changes its name to the (in his opinion) more informative name `doMap`.

- Do the changes to the file.
- Make sure everything compiles.
- Commit.

5.3 Alice adds a new file

Alice now wants to add a program that tests `doit` by calling it from a main method. But she is not aware of the changes that Bob has done. (It's a bit strange that Bob didn't tell Alice he did that rename. He really should have, but maybe he just hasn't gotten around to it yet.) *Alice*

- Create a new source code file, `Program.java` which has a main method that creates a `BSTMap` object and calls `doit()`.
- Put the new file under version control, using the cvs `add` command.

  ```
  cvs add Program.java
  ```

- Check that everything compiles.
- Commit. Does the commit succeed? Should it?

5.4 Bob continues to edit

Bob is now going to continue with his changes. But cautious as he is, he again does an update to make sure he is working with the latest version of the software. And he compiles to make sure everything is in a consistent state. *Bob*

- Update.
- Compile all files.

Oops! What has happened here? What went wrong? To keep the repository clean, it is not sufficient to make sure everything compiles and works before committing. We also need to make sure that the workspace we are committing is based on the *latest* version of *all* files in the repository. Otherwise, we need to update and compile and test again. Clearly, when Alice committed, the repository ended up in an unclean state. What should she have done to avoid this?

There are some important habits that you should have when working with version-controlled software:

Update often Do `update` often. Why?

Commit often Do `commit` often. Why?

Update before commit Right before you are about to `commit`, be sure to make an `update`. Why?

Commit working software Before you commit, make sure your software compiles and works correctly. Why?

6 Lab Reflection

When you feel that you master CVS updating, committing, and merging in all the different cases shown above, call for one of the lab coaches and discuss with her/him what you have learned. Be prepared to show legibly written answers to the lab questions and possibly to answer a few more questions coming up in the discussion.