

1 Introduction

In this lab, you will look at build management and source management tools, particularly Ant and Subversion (svn). These tools (or tools like them) will be vital to the success of your final project.

1.1 Ant

READ THIS IF YOU ARE BORED Ant is a Java-based build tool. Initially, it was part of the Tomcat code base, when it was donated to the Apache Software Foundation. It was created by James Duncan Davidson, who is also the original author of Tomcat. Basically it was just for building Tomcat.

Soon thereafter, several open source Java projects realized that Ant could solve the problems they had with Makefiles. Starting with the projects hosted at Jakarta and the old Java Apache project, Ant spread like a virus and is now the build tool of choice for a lot of projects.

However, it wasn't until January of 2000 that Ant became its own independant project and became Apache Ant.

Ant originally was an acronym for "Another Neat Tool", but other explanations have been given to it name such as "ants do an extremely good job at building things", or "ants are very small and can carry a weight dozens of times their own" - describing what Ant is intended to be.

In short Ant is a very useful tool. As you will discover Ant can be used for more than just compiling and running java code. **OK NOW YOU HAVE TO READ EVERYTHING**

1.2 Subversion

Subversion is source control system, and we recommend it highly¹.

It was created as an alternative to CVS and they are rather aggressive in their efforts to replace it. CollabNet keeps a few developers on salary, but it is Free Software on an Apache/BSD license.

2 Getting started with Ant

Ant files have the following basic layout:

```
<project .. >
  <property ... />
  <target ... >
  </target>
</project>
```

When you run `ant cmd`, `ant` will execute the commands corresponding to the target with the name `cmd` in your build file. Think of properties as variables that you can use throughout your build file.

Before you begin with the assignment you should look over the Ant Manual which can be found at:

<http://ant.apache.org/manual/index.html>

Specifically under the Using Ant section. You'll find a tutorial on how to make a simple build file. You may also want to look up the `exec` task under Ant Tasks → Core Tasks. As you will need to understand how it works to finish this lab.

¹Subversion supports the entire command set of CVS, a different versioning system, as well as directory moves, easy branching semantics, and a cool name that is not a pharmacy.

3 Assignment

Make a directory called lab06 and copy the latex file from `/course/cs032/pub/labs/05-antsvn/` to it. Now make a build.xml file in the same directory. Set up your build file and write a target called dvi which converts the latex file to a dvi. You can do this normally by running `latex <file>.tex`. This will create a dvi file which you can view with `xdvi`. Make sure your file names are properties and not hard coded into the target. If you want an additional challenge, you can try to make the file name a command line argument by using `$args` and calling `'ant -Dargs="value" target'`.

CHECKPOINT 1

Now add another target called pdf which converts the dvi to a pdf. You can use the `dvipdf` script. Now when you run it you should be able to view your file using `kpdf`.

CHECKPOINT 2

Now make a target called view which opens the file in `kpdf`. After you have done this you might have noticed an issue with the last two targets that you made. If you ran pdf without running dvi it wouldn't work. Instead of making the user have to constantly run dvi then pdf then view you can set up dependancies. You can set up dependancies by listing the targets that need to run before the current target is run. Example:

```
<target name="run" depends="clean, compile">
    ...
</target>
```

When you run this target clean will go first, then compile and then run. For **CHECKPOINT 3**, make pdf depend on dvi and view depend on pdf.

Finally, you usually want to write a clean target which gets rid of any messy files you don't need any more. You'll notice that latex leaves a few files lying around after it compiles. Write a clean target which gets rid of all the extra files (including the dvi file). We suggest using the `delete` task.

CHECKPOINT 4

4 Getting started with Subversion

Subversion has an excellent manual available at <http://svnbook.red-bean.com>. You should look over it, particularly the first few sections if you're not sure what the point is at all. We are using version 1.1.1.

We will not cover the more advanced topics of subversion, like branching and tagging. You shouldn't need these for your final projects.

Subversion is legit for one person, if only to simply keep track of revision histories. Subversion is super clutch for more than one person – to coordinate merging of files, etc. Therefore, **you will need to assemble into groups and work together on a single repository**.

Note that `$` represents your prompt. It probably says something more like `cs1ab7g /u/mcfeldma %`.

5 Setting up your repository

Choose a group member to own your repository. Only they will execute these commands. Seriously. Keep in mind that an empty repository will be about 1MB.

You need only create your repository in a place accessible to your group. On your final project, you will be given space in `/pro`, but for now you should create it in your directory. We recommend something hidden and inoffensive, such as `/.svn`. Create the repository:

```
$ umask 002                                # make it group writable
$ svnadmin create --fs-type fsfs /u/yourloginhere/.svn
$
```

Congratulations – you’re ready to control versions.

Now you need to get something to control, for what does power seek but more? Every subversion repository begins by importing something. You’ll be importing the slickest game around, GemQuest. Watch for it in stores next year. Copy the code:

```
$ cd                # go home
$ mkdir repos      # the repository base directory
$ cd repos
$ mkdir gemquest   # somewhere to store the code
$ cd gemquest
$ tar xzf /course/cs032/pub/labs/05_antsvn/gemquest.tgz
$ cd                # go home again
```

And then import into your repository:

```
$ svn import repos /u/yourloginhere/.svn -m "Initial import."
Adding      repos/gemquest
Adding      repos/gemquest/gemquest
Adding      repos/gemquest/gemquest/GemQuestMapGUIImpl.java
Adding      repos/gemquest/gemquest/App.java
Adding      repos/gemquest/gemquest/GemQuestFactory.java
Adding      repos/gemquest/gemquest/GemQuestParser.java
Adding      repos/gemquest/gemquest/GemQuest.java
Adding      repos/gemquest/gemquest/GemQuestMapImpl.java
Adding      repos/gemquest/build.xml
```

Committed revision 1.

```
$
```

The `-m ‘‘Initial import.’’` line is a comment, and it’s necessary for every repository changing event. If you don’t specify it on the command line, your favorite editor (or whatever is in the `$EDITOR` variable) will pop-up and allow you to interactively enter your comment. While it is possible to enter a commentless commit, there’s never a good reason to do it.

You may now reset your `umask` (the default is 022).

CHECKPOINT 5

6 Checking out and committing

Now the whole group gets in on the action. You’ll need to first create a *wrapper script* for subversion, so that your `umask` will always be properly set. That’s easy enough; create an executable (`chmod 755` or `chmod +x`) file named `svn` in your home directory:

```
#!/bin/bash

umask 002
/usr/bin/svn "$@"
```

Each user can now get their own working copy of the repository, a *checkout*. Changes made to the working copy won’t be visible until you *commit* your changes.

```
$ ~/svn co /u/loginofrepositoryowner/.svn/gemquest # co is short for checkout
A gemquest/gemquest
```

```

A gemquest/gemquest/GemQuestMapGUIImpl.java
A gemquest/gemquest/App.java
A gemquest/gemquest/GemQuestFactory.java
A gemquest/gemquest/GemQuestParser.java
A gemquest/gemquest/GemQuestMapImpl.java
A gemquest/gemquest/GemQuest.java
A gemquest/build.xml
Checked out revision 1.
$

```

Now your code is checked out. Let's play around a bit.

Have a group member create a text file; name it whatever you want. They will then add it to the repository.

```

$ cd gemquest          # go to your working copy
$ xemacs -nw test      # vim if your cool like that
$ ~/svn add test
A      test
$

```

Note that you didn't have to specify the repository; that's because you're in a working copy:

```

$ ls -a
./ ../ .svn/ build.xml gemquest/ test
$

```

The directory `.svn` contains various data that direct subversion.

Returning to the example, the A next to test indicated that the file is marked to be added. You can view a list of all of the changes made to your local copy like so:

```

$ ~/svn status      # try svn diff for a good time!
A      test
$

```

Since this change is valid, we can commit it to the repository. **Never, ever commit unless your code works!** Nothing is more irritating than having people submit code as working that is not. You can, if you like, go so far as setting up a script to reject commits that fail unit tests (or compilation, or some other standard). Let's commit the change:

```

$ ~/svn ci test -m "The handout told me to do this, and I follow blindly."
Adding      test
Transmitting file data .
Committed revision 2.
$

```

Now everyone else can update their repository to view the file, like so:

```

$ ~/svn update
A test
Updated to revision 2.
$

```

Now everyone is synced up. Don't you feel closer to one another?

CHECKPOINT 6

7 Questin' for Gems

There are typos in each of the GemQuest files. Your dungeon questing cannot begin until you repair these errors. Assign a group member to each file, have them fix it, and then commit their changes. Do this until it runs, at which time the lab is over and you can quest to your heart's content.

To compile/run GemQuest, put `/course/cs032/lib/gemquest.jar` on your classpath. The associated buildfile does this automatically.

CHECKPOINT 7