Programming with Haiku

Lesson 4

Written by DarkWyrm

Source Control: What is It?

In my early days as a developer on the Haiku project I had troubles on occasion because I had to use a source control manager (SCM). I didn't understand it and I didn't want to take the time to learn about it from some tutorial online. I wanted to be able to write code with as few hurdles as possible. How I wish that I'd understood source control then.

Source control, also known as **revision control** or **version control**, is a tool or set of tools which facilitates development on a single codebase by many people at once. This is done by recording the changes that are made by each person and ensuring that one person's changes cannot be applied at the same time as another's. Most also provide for working on a separate copy of the main sources, called a **branch**.

Using an SCM forces your workflow to have some structure, which is actually a good thing for those who have a hard time getting organized, provided that they are willing to work with it. Day-to-day coding involves checking out others' updates and checking in your own. On occasion, a change must be undone, called **reverting** a change. Sometimes a feature is large enough that it necessitates working over the course of several check-ins, called **commits**. In these cases, a branch is created so that the development of the feature benefits from source control without disturbing others' work. When the feature is completed, it is merged back into the main part of the tree.

Source Control: Why Use It?

The benefits of using an SCM as part of your workflow far outweigh any drawbacks almost 100% of the time. If you are part of a project with more than one person, the automation of change tracking frees up time to spend on other tasks. It also removes the human factor of applying changes. While it might be possible for two or three people to work together by e-mailing files to each other, the potential for mistakes is great and it cannot be easily sustained for an extended period of time. The ability to undo changes with a command or two is a major time-saver.

A few drawbacks accompany using source control. Setting up a project in source control can take some time. Changing from one SCM to another is not always easy. Choosing a source control manager isn't easy – there are many available and the advantages of each are not always clear. Setting up your own SCM server, such as for a business or for private hosting of a closed source project, has challenges of its own as well. These headaches are considered minor by most veteran coders because often they have had at least one instance where source control got them out of a jam.

Source Control: Which One?

On other platforms, such as Linux or OS X, there are a plethora of different SCMs from which to choose. As of this writing, there are four available for Haiku, described below.

Concurrent Versioning System (CVS)

CVS is one of the oldest SCMs still available and in use. It was originally developed as an improvement over the set of tools known as Revision Control System (RCS). It uses a client-server architecture and is still widely used. CVS is considered by some to be not worth using. When compared to other choices, it tends to come up short on features or has limitations that

others do not have, such as not being able to rename or move files. It is not recommended for inexperienced developers.

Subversion (SVN)

Subversion was started with the intention of fixing the problems present in CVS, and is also a widely-popular, well-liked source control tool. Like CVS, Subversion uses a client-server architecture. It supports moving and renaming files, branches are relatively fast, and commits are truly atomic operations. It does have problems, as well, though. One common criticism is that while branching is simple, merging a branch back into the main one does not always work well and can be tedious and/or time-consuming. Overall, it is a good tool and, as of this writing, is the tool used by the Haiku project.

Git

Git was originally developed by Linus Torvalds for work on the Linux kernel. Unlike Subversion and CVS, it uses a distributed model where each developer has a full copy of the repository. It was written to be whatever CVS isn't – to protect against corruption, to be very fast, and to work much like the proprietary SCM BitKeeper. Git has quickly developed a passionate following. One of its main drawbacks is that while there are two implementations for Windows, neither is an ideal solution. While very, very fast, it can also be somewhat confusing while getting accustomed to using it. In many ways, it has a design perspective similar to that of Linux as an operating system.

Mercurial (hg)

Mercurial is a brother-in-arms to Git, having been started at about the same time and for the same basic reason: the company behind BitKeeper, BitMover, withdrew their version of BitKeeper which was free to open source projects not developing a competing SCM. It is written mostly in Python and is available on most major platforms and Haiku. Like Git, it has a loyal following and there is a never-ending controversy between Mercurial users and Git users about which is better, similar to that of two well-known colas. The design perspective for Mercurial could be likened to that of OS X or Haiku – simple, but not oversimplified. It is a recommended starting point for a developer to learn how to use source control and will be the one referenced in the future. Just like a preferred IDE, if you have a different preference, that's just fine.

First Steps Using Source Control

Although Paladin provides a convenient interface to using source control, it does not go in-depth into some of the finer points of using any of those which it supports. For the purposes of this lesson, we will just cover the basics of using Mercurial from the command line.

The first task on the order of business is to tell Mercurial who you are. Create a file in your home folder called `.hgrc`, open it in a text editor, and place the following contents in it:

```
[ui]
username = My Name <myemail@foo.com>
```

Of course, you will want to substitute your own name and e-mail address. Also, unless you prefer the `nano` editor included in Haiku, you'll want to set the text editor used by Mercurial when you describe your changes when you check them in. My preference is to use Pe, an

excellent text editor written specifically for Haiku. To use Pe as your text editor, create a file in your home folder called `.profile` if it doesn't already exist and add this line to it:

```
export EDITOR=lpe
```

If you have a different editor you'd prefer to use, substitute the name of your preferred editor for lpe. If the executable is not in `/boot/system/bin`, `/boot/common/bin`, or `/boot/home/config/bin`, you'll need to use the full path to the executable. Your `.profile` file is a Bash script executed at the beginning of each Terminal session, so you can place other customizations here, too, if you like.

Now that the initial setup is done, let's make a repository. Mercurial can be used to add source control to a new project or an existing one. Open a Terminal window, go to an empty folder, and enter this command:

```
$ hg init
```

This uses the current directory as the top folder in the repository. All subfolders are considered part of it, although empty directories will be ignored. You won't see Mercurial print anything after typing the command, but if you issue an `ls -a` command, you'll see that a `.hg` directory was created – this folder is where Mercurial keeps all of its information for the entire repository. While you're at it, create or copy over a couple of test text files in the directory that we'll use in a moment.

Now we will add your test files to the repository. Mercurial and other SCMs will track only those files which you have told it to track, although they will tell you if they see files they don't recognize. Adding files to Mercurial's list of files to manage is simply another command:

```
$ hg add
adding ObjectArray.h
adding foo.cpp
```

All files in the current folder and all subfolders will be added to the repository. Of course, you can specify a file or list of files after the word `add`, instead. Adding files to the repository doesn't change anything unless you check in the changes.

Before we make our first commit, it's often good to make sure you know ahead of time what changes have been made. This can be done in one of two ways. The first way is to use `hg status`. This will print out a list of file which are not up-to-date, which can be added, removed, modified, or unrecognized files. The other way is `hg diff`, which shows the actual changes made to each file. Here is what `hg status` looks like:

```
$ hg status
A ObjectArray.h
A foo.cpp
```

Two files have been added since the last commit, or in our case, since the creation of the repository. Other codes for files can be found in the following table.

Status Code	Meaning
M	Modified
A	Added
R	Removed
C	Clean
!	Missing – file does not exist, but is still being tracked
?	Not tracked
I	Ignored

If you are not already familiar with the `diff` command used in the Terminal, give the `hg diff` command a try to see how it looks, but be prepared for a lot of text to be printed. It is often best to view a diff using a pager in the Terminal such as `hg diff | less` or doing it from Paladin where you get a scrollable text window.

Before we do our commit, let's examine what we would need to do if something happened that we didn't want to commit to the repository. Let's say that you added a bunch of `printf()` calls to a file for some debugging, but you don't want them going into the tree. Although you could go back and remove them all manually, if these `printf()` calls were the only changes you made to the file, you could `revert` it.

Reverting a file undoes all changes made to it since the last commit. If you've accidentally added a file that you don't want tracked, it will un-add it. If you accidentally delete a tracked file, Mercurial will replace it with a new copy. Modified files will go back to an unmodified state. In short, it fixes any mistakes you've made, and when you revert a modified file, Mercurial creates a .orig file which contains the changes you made in case you want them after all.

A revert can be done in several ways. You can revert just one file or the entire tree. The revert can go back to a specific revision. It can also be done without backing up the changes made. Here are some of the options for `hg revert`:

Command	Action
`hg revert --all`	Reverts all files in the tree
`hg revert myFile.cpp`	Reverts myFile.cpp. Changes will be backed up to myFile.cpp.orig
`hg revert --no-backup myFile.cpp`	Reverts myFile.cpp, but makes no backup
`hg revert -r d8787f07dd69 --all-files --no-backup`	Reverts the entire source tree back to the specified revision (changeset), making no backups to changes

Considering the myriad ways that a developer can make mistakes, knowing the different ways that `revert` works can save you time, effort, and stress.

Let's move on and check in our changes. Enter this command to initiate a commit:

```
$ hg commit
```

After entering this command, Mercurial will open an editor to allow you to add a message to describe the commit. If you didn't specify an editor in your `.profile`, this will be the console text editor `nano`. Use the message "Initial check-in" or something similar, save the file in the editor, and close it. Mercurial may or may not print anything. Rest assured that even if it doesn't, your first commit was successful. You can confirm this, though, with the `hg log` command, which can be done for the entire repository or just certain files.

```
$ hg log
changeset:   0:0dbb51f0e1fa
tag:         tip
user:        DarkWyrm <darkwyrm@gmail.com>
date:        Sun Aug 15 21:30:56 2010 -0400
summary:     Initial commit
```

Working with Others Using Mercurial

If you're just working on your own project and have no intention of working with others on it, these commands are all you'll need. However, having a project on an open source hosting site such as BitBucket or Sourceforge involves a few others, as well.

Let's pretend for a moment that you have applied for hosting a project called MyProject at a site called MyMercurial. After submitting the application, you are approved and the site has created your repository. Now what?

First, you will need to make a local copy of the repository hosted by MyMercurial. You will need to get the URL of the repository from the hosting site which is specific to the hosting site and your project. For our example, the repository URL is http://mymercurial.foo/hg/myproject. To make our local repository, we will use the `hg clone` command. Mercurial will probably print something similar to this:

```
$ hg clone http://mymercurial.foo/hg/myproject
destination directory: myproject
requesting all changes
adding changesets
adding manifests
adding file changes
added 0 changesets with 0 changes to 0 files
updating to branch default
0 files updated, 0 files merged, 0 files removed, 0 files unresolved
```

You will now have a subfolder of the current one called `myproject`. It doesn't have any files in it, but it will be easy for us to send changes to the remote one. Most of the rest of the work is exactly the same as what we did a moment ago: copying files for the project into the folder, using the `hg add` command to add files to it, and `hg commit` to check them in.

When working with a project hosted online, there is one extra step in the workflow: pushing changes. Commits only apply to the repository sitting on your hard drive, unlike centralized source control tools like CVS and Subversion. Getting your changes to the online repository is done with `hg push`. The results look similar to this:

```
$ hg push
```

```
pushing to /boot/home/testrepo
searching for changes
adding changesets
adding manifests
adding file changes
added 1 changesets with 1 changes to 1 files
```

Getting changes from the online repository is similar to publishing them to it. This is done with the command `hg pull`. This grabs the changes from the online repository and downloads them. It does not, however, automatically merge them into your sources unless you add the `-u` switch. Omitting the switch means that once the `pull` is complete, executing an `hg merge` and `hg commit` is needed afterward.

Wrapping it All Up: Source Control in a Nutshell

Working with source control can seem complicated at first, but concepts carry over from one SCM to another, and the basics don't involve much in most situations. Most of the time, you'll follow a workflow something like this:

1. Write and/or modify code.
2. Commit your changes locally.
3. Repeat steps 1 and 2 as many times as desired. When you're ready to update the online repository, continue to step 4.
4. Pull and merge remote changes.
5. Push your modifications to the remote repository.

When you look at the workflow this way, it doesn't seem very complicated and for a very good reason: it isn't complicated. More advanced source control use, such as using branches, is beyond the scope of this lesson but not much more complicated than the above series of steps.

If source control is so simple, why doesn't everyone use it? In most cases, it is because of ignorance, laziness, or both. Integrating source control into your development workflow will make your work easier and potentially save you from major problems.