HOPPERS HACK NIGHT

2 February 2012

Maxim Cramer

Getting started

The command line:

You think it's absolutely terrifying and I agree. We've all been there at some point. However, it is by far one of the best skills to pick up during your time as a CompSci student and here's why:

There will be times where you'll be without a Graphical User Interface (GUI), a.k.a. a screen. For instance, when you'll be connecting into the DICE networks from home (through ssh), most of your work will probably be through the command line. You'll submit your coursework through the command line. You'll write, compile and execute code through the command line. The list is endless.

So what do you need to know? For now, just the basics: moving through the file system and creating a few things. Everything else will come with time and practice.

To start:

Unix filesystems (which is what DICE has) are a bunch of folders in folders. When you start up the terminal, you'll usually find yourself in your *home folder*. This is the first *directory* (folder) that is completely yours, contained in the users folder.
Here are a few commands you could try:

~ ls	Will **l**i**s**t the files and folders in the directory you're in right now.
~ cd Documents	Will **c**hange **d**irectory into another folder, e.g. Documents.
~ cd ..	Change directory up one folder (so back to your home folder again)
~ mkdir test	**M**a**k**es a **dir**ectory called test
~ rm -rf test	**R**e**m**oves test. The - means extra options: **r**ecursively and **f**orced, i.e. everything contained in the folder as well.

Arguments you could pass:

~ ls .	. means the current directory. ls already does this.
~ ls ..	.. means the parent directory
~ cd -	- means the directory you were in before
~ rm *	* means everything. Like, everything. It's the wildcard operator. E.g.:
~ rm *.txt	will remove all the text files in this folder, regardless of their names

Some more operations:

~ touch Documents/test.txt	Will create an empty file called test.txt
~ mv Documents/test.txt .	Will **m**o**v**e the file test.txt to the current folder (.) Check if it's there and gone from Documents with ls!
~ mv test.txt test_renamed.txt	mv also renames files! Check again with ls.
~ cp Documents/test.txt .	Make a new test.txt file and **c**o**p**y it to the current directory Check again with ls!

This would be all the command you'd need to move around files and folders as well as add, copy, rename and delete them!

Remember: you can only do these with files you've got *write* permissions to, i.e., *your* files!

There are many many resources online, some of my favourite being the lifehacker ones:

http://lifehacker.com/5633909/who-needs-a-mouse-learn-to-use-the-command-line-for-almost-anything

http://lifehacker.com/5743814/become-a-command-line-ninja-with-these-time+saving-shortcuts

Version control:

For those that have already attended some lectures on software engineering or read online guides to sensible project management, I'm sure you've heard this before. For those of you that are haven't, here's an overview:

A version control system (VCS) is a system that will keep track of a set of files and their history. Say you're working on an essay - every time you finish a section you could save a version (also known as a *commit*). The VCS will then log this in its history and it will be a snapshot of the file that you can then *revert* back to.

Why is this useful? I can guarantee that every has been or will be in the situation where you've got code that essentially works. You then want to add one last little insignificant thing. And suddenly - explosions. With the look of utter shock and frustration on your face you shout: *"Y U NO WORK NO MORE?"* As it appears that you're code is now broken. What did you change? How do you go back? This can be simple if you've used version control.

You could set up this up locally (on your computer) or online, also known as distributed VCS. This means that you can keep all these versions you commit on 'the cloud'. This is extra beneficial, because it means that, not only when your code stops working, but your *computer* stops working, you can download your files and versions and keep calm and carry on!

So in theory this all sounds lovely, but how do you actually use it?

There are different types that you could use, such as CSV, SVN, mercurial and Git. The latter two are currently the more popular ones. These are different methods. They all have their own commands and ways of keeping track of your versions.
Then there are also different websites that do distributed version control of a certain type such as github (git) and bitbucket (git and mercurial).

Example
The one I'm personally most familiar with is Git, so I'll use that here.

Say you're working on INF2B and you've got to write a python music recommender system for your first assignment. You go to your `University/courses/INF2B/assignments` folder and probably make a new directory called `cwk1` or `music_recommender`, or something along those lines. You then create a python file and you start playing around. I, however, encourage you to include "initialise your VCS repository (repo)" to that list of tasks. This is how:

In the command line, navigate to the folder that will contain your source files

~ git init	This initialises git in your current folder
~ git add *filename*	Add the file to the list of files to be "version-ed"
~ git commit -m "message"	Commits a set of versions. Adding a message (-m) is useful, so you know what changed at this point in time

Rinse and repeat.

Advanced

Branches:
Things can get more complicated when you add branches which are basically, branches. This means that whenever you have a new task you're working on, you could make a new branch, e.g. "task_one". A branch will copy the state that's on the main branch (always called master) or the branch you were working on, and will then keep track of a new set of history just on that branch. At any time, you can then move branches that (can) contain a complete different state of your code.

It's a standard to have "deployable code" on the master branch, i.e. code that fully works. Whenever you would want to add a feature or fix a bug, you would make a new branch for said issue.

If this all sounds too complicated - don't stress! All in good time. There is plenty of information online, so whenever you feel the need to incorporate branches in your workflow, it'll be relatively easy to pick up on.

.gitignore:
A file telling git to ignore certain files, such as .DS_Store or java .class files. Very handy!

Distributed VCS:

If you're using a distributed VCS, you'll want to add additional steps to your workflow.

When you make a repository online, it will provide you with an url you can clone from

~ git clone url	Will download the git repo
~ git pull	Will download the latest version that is online
~ git push -u origin master *or other branchname*	Will push your commit to your online repository Very useful for when working with others, as they can then pull your code and get the latest version.

Good online websites to host your projects:

github.com	Very popular, hosts a lot of open source code that is good to follow Limited private repo's without a paid plan Very good to put work online to show Very social
bitbucket.org	Unlimited free private repo's (handy for university work!) Up to 5 collaborators (good for group projects)

Some terminology:

Repository: like a folder, but for version control. It contains your files and all the VCS data.
Commit: saving a version
Pull: fetching the latest state of the files from an online repository
Push: saving a commit to an online repository (do this after committing locally)
Branch: copy of previous work that is a safe playground to implement something new
Clone: initialise a local VCS repository that already lives online

Other handy commands:

~ git status	See which files have changed and which you've added
~ git log	History of all the commits done by whom
~ git diff	Shows the differences in files changed since your last commit

Simplification disclaimer:
I know things are a lot more technical and that I may have simplified matters too much in this document. This is not because I don't know how it works or that I think people reading this won't be able to understand otherwise. It's purely how I would have preferred to learn, because back when I was new to all this, I had a hard time following certain terminology!