# CPSC 310 : Assignment 2 – Git Introduction

## *Introduction*

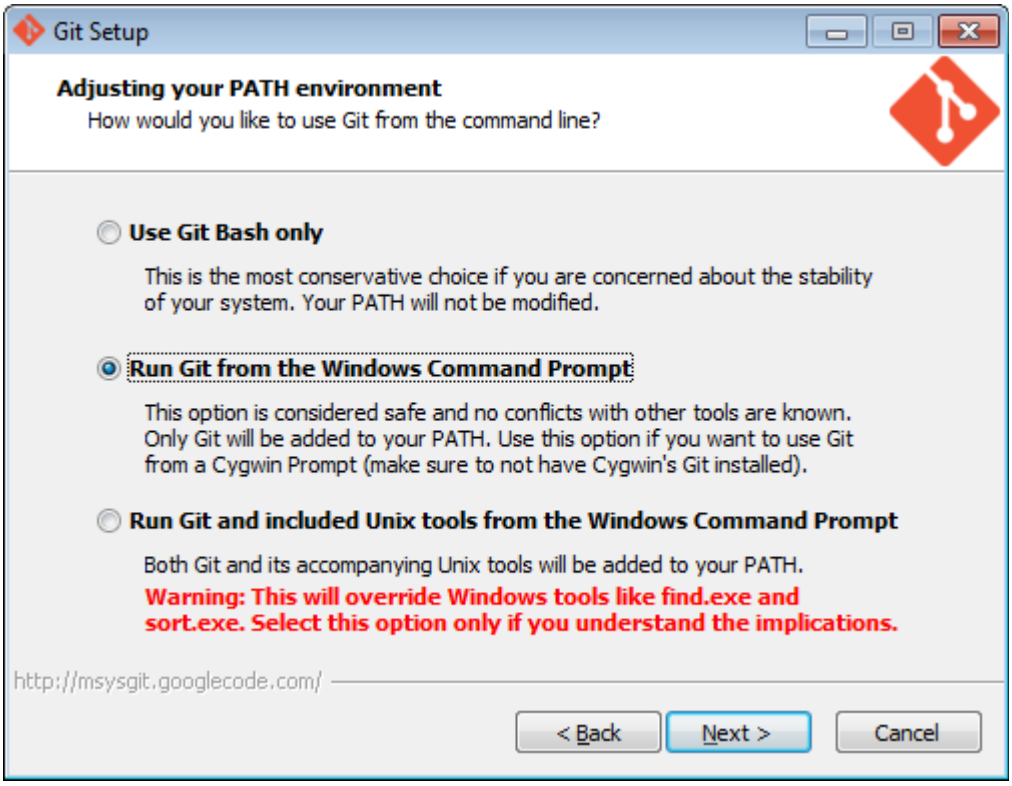
This lab is designed to help you become familiar with the Git Source Control system that you will be using to manage the source code of your project.

## Before the Lab

1) You should do Level 1,2,3 from the Introduction Sequence part from this online tutorial: http://pcottle.github.io/learnGitBranching/

2)You should read chapters 2 and 3 of Pro Git, available here: http://git-scm.com/book. It is not expected that you have memorized the details of the book but you will need to refer to it and to Google during the lab. You can skip section 3.6 – Rebasing.

3) If you are using your own laptop you should ensure that the command line version of Git is installed on your computer and that you know how to access it via the command line / terminal interface. If you are installing on Windows, you will be asked how you would like to run bash, the best option to select is "*Run Git from the Windows Command Prompt*". This will allow you to use git from a standard windows command prompt. The first option will require you to a different command line program bash (which is the default on Linux and Mac OS X).




4) For the project you should also make sure you have Eclipse and EGit installed, although you should not use EGit for Part 2.

**Note:** The purpose with this lab is to get you familiar with the concepts and abstractions of Git, so that if EGit gets stuck you can still fix your project manually. Additionally Git has a much more resources for support on-line and familiarity with it will make it easier to resolve problems with EGit. As such for Part 2 of this lab you may **NOT** use a front-end like EGit for this assignment and must use the command line version.

## *Part 1 – Git Hub Account*

**Step 1)**

Obtain a GitHub account by browsing to https://github.com/ Note that GitHub is hosted in the United States and you therefore may want to use an anonymous username and email address when you register.

**Step 2)**

Set an avatar for your account by clicking your account name in the upper left-hand corner, and then clicking on the giant picture. It doesn't matter what the avatar actually is, it will simply make reading the commit history much easier when working on your project. Once you have it setup, it may take upwards of 15 minutes for it to take effect, but if you clear the cache in your browser it may be instant.

## *Part 2 – Reconciling Projects Using the Command Line*

Inevitably during your course project you are going to have to merge other people's code into yours. There are various strategies, all of which mainly involve good group communication that can minimize the pain involved, but you will need to be familiar with this process no matter what. In this part of the assignment you will check out a project that has basic functionality in it's `master` branch, and the required functionality in the `ui` and `async` branch.

**Note**: Except for a single line in Step 5, you do not need to write any new code for this lab to complete this lab.

**Warning:** If you switch branches or modify files on the command line, you will need to right click on your project in Eclipse and hit Refresh for the changes to become apparent.

### Tips

**1)** None of the conflicts in this section are particularly onerous to solve if you spend time to understand what caused them and all the different ways to resolve them.

**2)** You do not need to create any commits during the lab. Occassionally if you switch branches, git may ask you to make a commit because of local changes, for example:

```
error: Your local changes to the following files would be overwritten by checkout:
        src/ca/ubc/cpsc310/gitlab/client/GitLab.java
Please, commit your changes or stash them before you can switch branches.
Aborting
```

You can revert all the local changes you have made easily with the command:

```
git checkout -- .
```

**3)** You can avoid having to type in a manual message using vim by using **git commit -m "My commit message"**

**4)** For now, you will access remote branches via their full name for example `origin/async` or `origin/ui`. When you check out a branch newer versions of git will automatically set up remote tracking for you so you can refer to the branch as `async` or `ui`. For more information see Section 3.5 of Pro Git (http://git-scm.com/book/en/Git-Branching-Remote-Branches).

**5)** Some questions or tasks may be easier to do by browsing the repository on GitHub than on the command line. You can view the repository and all the files here:

https://github.com/SJrX/UBCCPSCGitLab

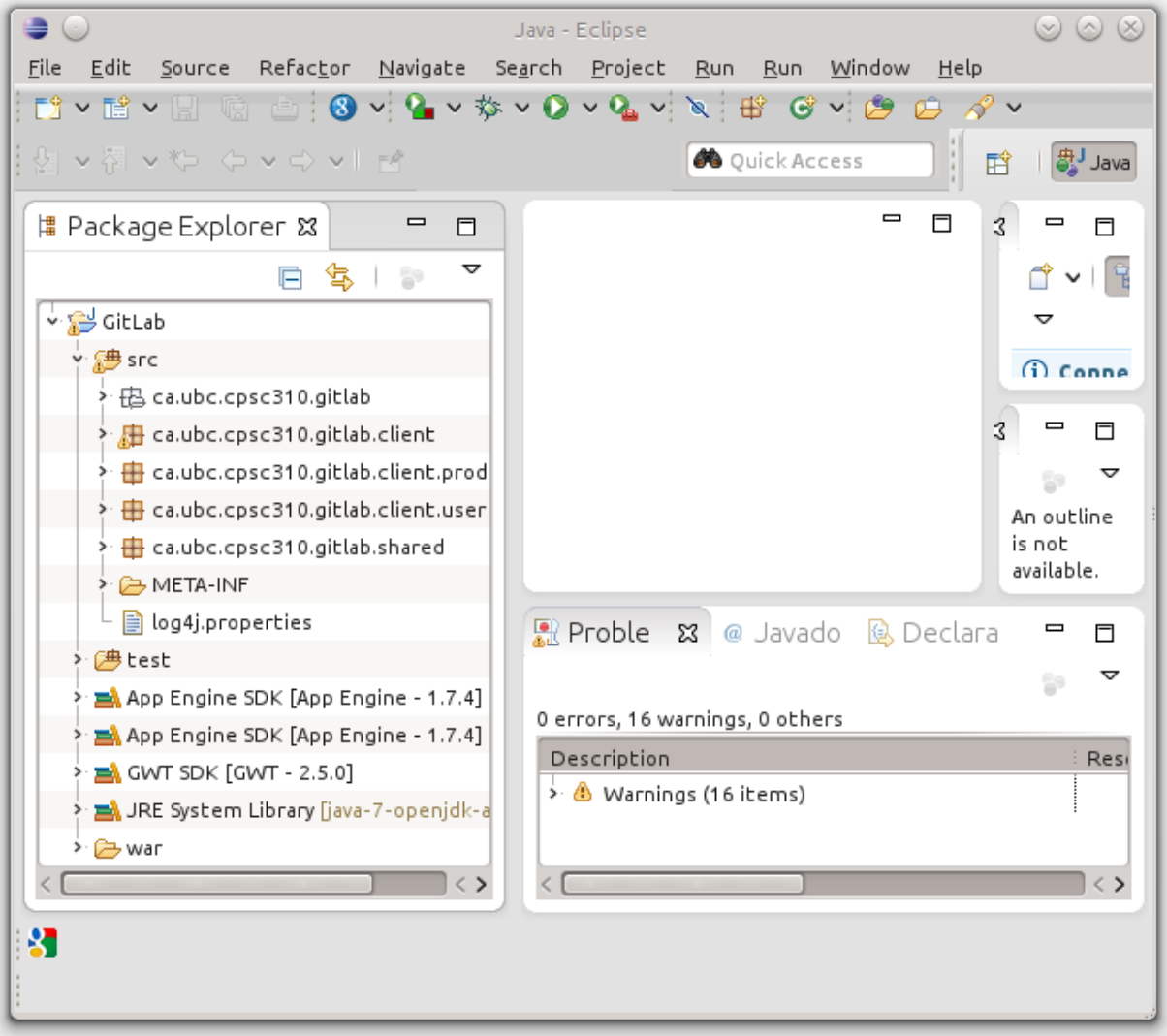
### Step 1) Clone the Repository

Clone the following repository somewhere on your disk:

```
git://github.com/SJrX/UBCCPSCGitLab.git
```

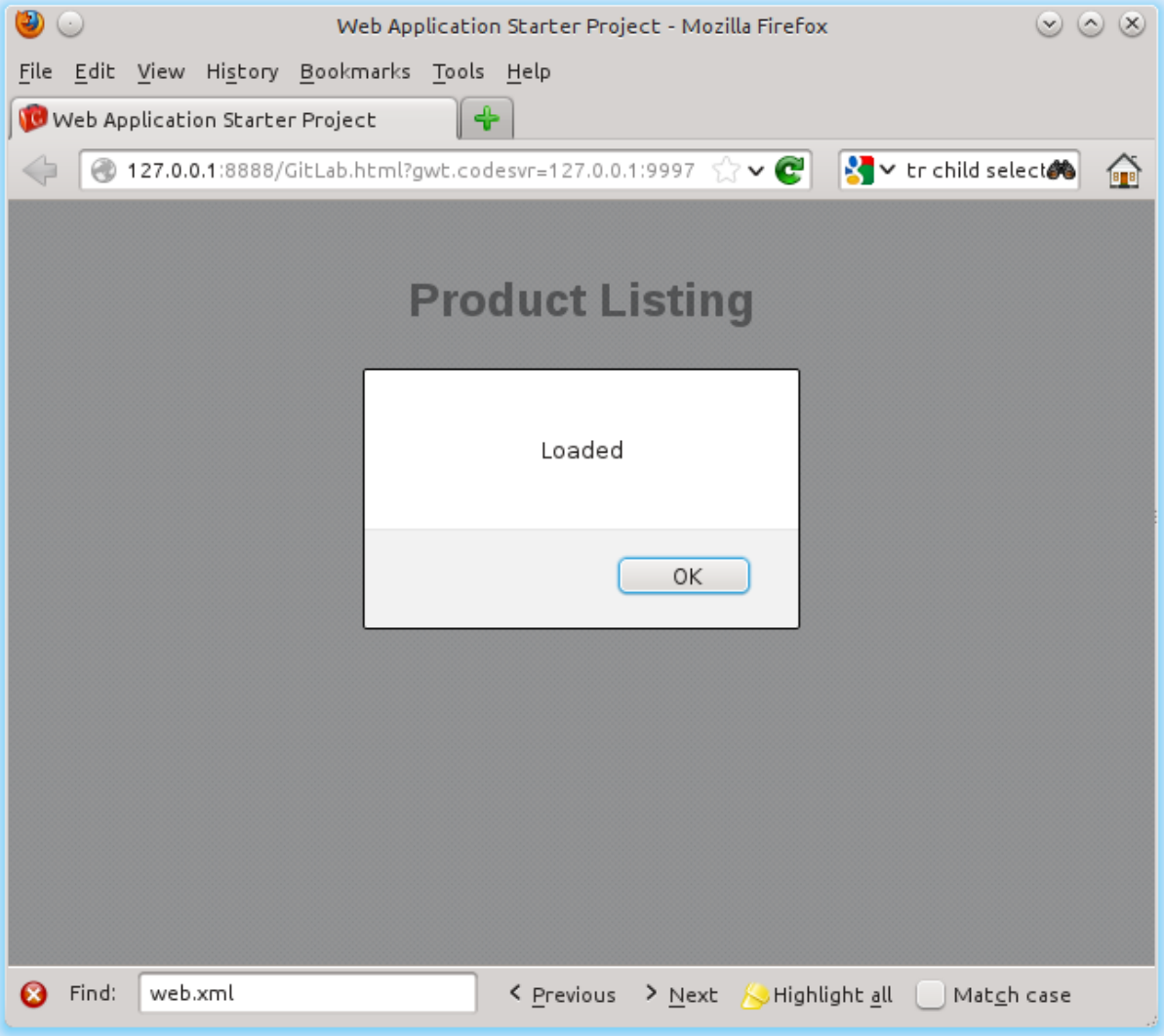
### Step 2) Import Into Eclipse

Import your project into Eclipse, by clicking File → Import → General → Existing Projects into Workspace. The root directory should be the UBCCPSCGitLab. When it's imported you should see a screen similar to:




Make sure the project runs when you import it into Eclipse. You can do this by right clicking on the GitLab project folder → Run As → Web Application.

When you run the application you should see the following:




### Step 3) The ui Branch

The version of the application on the `master` branch currently is very bare bones but additional functionality has been implemented in two separate branches, the `ui` branch and the `async` branch. In this step we will integrate the `ui` branch into the `master` branch.

If you run the version of the project in the `ui` branch, it will do nothing currently. You can test the `ui` branches' implementation by adding the following to the `onModuleLoad()` method if you'd like but this part is not strictly necessary.

```
/**
 * This is the entry point method.
 */
public void onModuleLoad() {
      List<IUser> users = new ArrayList<IUser>();
      User user = new User();
```

```
        user.setName("Alex Anderson");
        user.setLanguage("FR");
        users.add(user);
        displayUsers(users);
    }
```

**Question 3.1)**

What is the last commit on the `ui` branch (first few characters of hash is fine)?

**Question 3.2)**

Before you can merge a branch you must perform a checkout on that branch. Note that `git merge branch-name` means merge the branch branch-name with the **currently checked out branch.**

When merging with other people's code you should always check exactly what will end up being merged, in case something horribly breaks your project. Which .java files changed between the ui and master branch?

Now, Merge the `ui` branch into the current branch `master`, resolving any conflicts as needed. You can check if the merge was successful by ensuring that there is a `displayUsers(List<IUser>)` method in `ca.ubc.cpsc310.gitlab.client.GitLab` class.

**Question 3.3)**

After the merge, what if anything conflicted? What do you think caused the conflicts? What could have prevented them?

**Step 4) The async branch**

At this point you have successfully merged the `ui` branch into the master branch, and consequently have a functioning User Interface. In this step we will merge the `async` branch into the `master` branch, incorporating the service implementation that the `async` branch contains with functionality in the `master` branch (which now includes a working User Interface).

**Note:** If you run the code on the `async` branch, it will unfortunately throw a `NullPointerException` on the server, and the client will see a `StatusCodeException` (we will fix this in Step 5).

**Question 4.1)**

What was the last commit on the `async` branch (first few characters of hash is fine)?

**Question 4.2)**

As before, we should check exactly what the differences are in the `async` branch with respect to your current master branch, prior to the merge to ensure that it will do what we expect. Which .java

files changed between the async and master branch?

Merge the `async` branch into the current branch (`master`), resolving any conflicts as needed.

**Question 4.3)**

After the merge, what if anything conflicted? What do you think caused the conflicts? What could have prevented them?

**Step 5) Restoring a file from a previous commit.**

At this point both branches have been merged in order to complete the integration you need to change the code to have the onSuccess() method call displayUsers:

```
@Override
public void onSuccess(List<IUser> result) {
      displayUsers(result);
}});
```

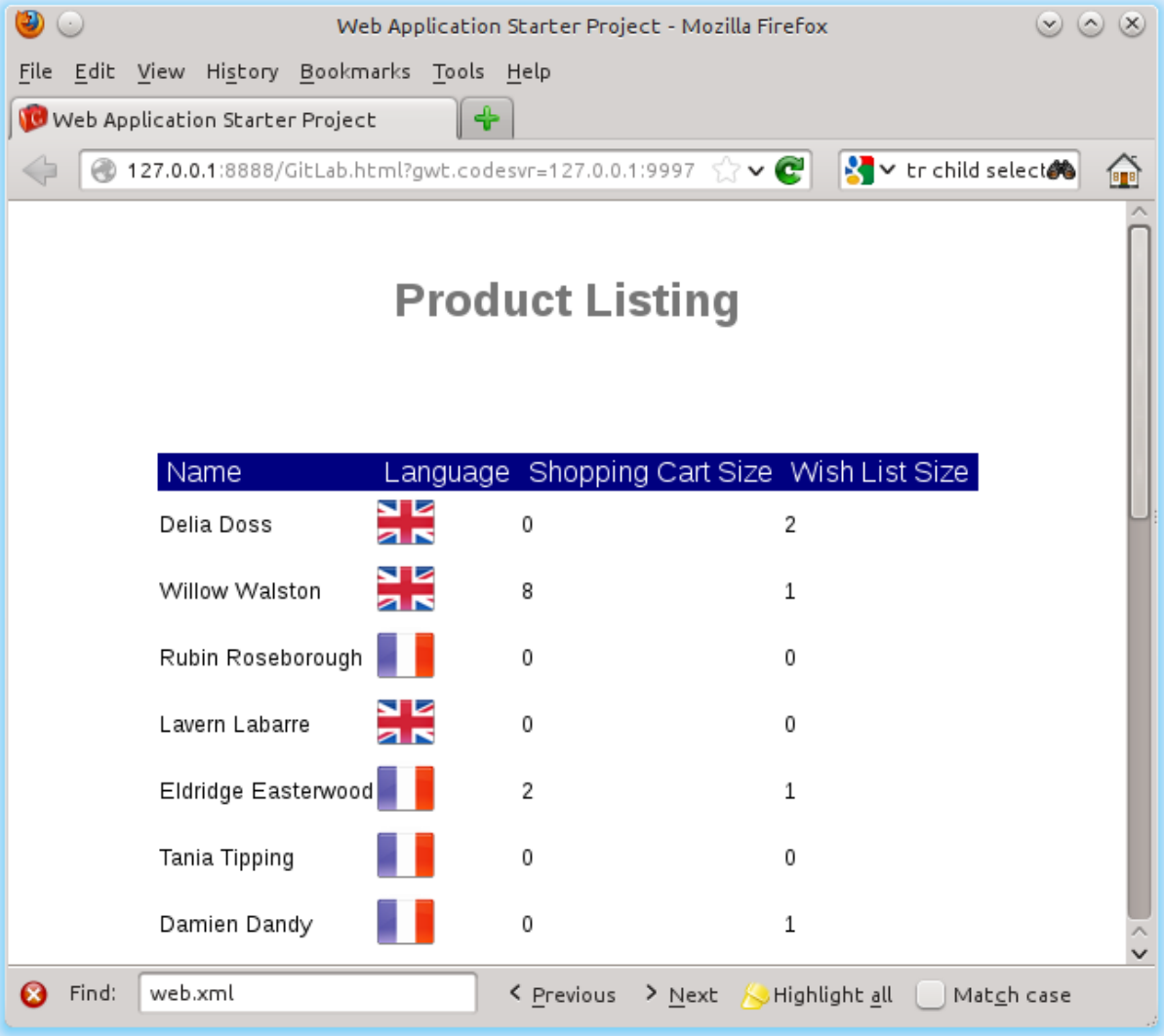
Unfortunately if you try and run the project you will get a `NullPointerException` on the console and a `StatusCodeException` in the browser. This is because a file `src/userlist.txt` was deleted by accident some time ago. Find this file in the commit history, and restore it.

**Question 5.1)**

What was the commit hash that deleted the file (first few characters of hash is fine)?

**Step 6) Run the program to test**

Try running the program. If you did everything correctly you should see the following:




If when you run you get the following error:

```
com.google.gwt.user.client.rpc.SerializationException: Type
'ca.ubc.cpsc310.gitlab.client.products.ProductItem' was not included in the set of
types which can be serialized by this SerializationPolicy or its Class object could
not be loaded. For security purposes, this type will not be serialized.: instance =
...
```

Then you made a mistake while merging the `async` and `ui` branches. In this case this error indicates that communication between the client and server is now broken, and it was working in the `async` branch previously. You should re-examine how you resolved the merge, and perhaps try resolving it a different way. There is no need to undo your previous merge, you can simply modify the files in the `master` branch to get this working.

## *Part 3 – EGit*

A proper understanding of the Git workflow is important for working with it, but the built in Eclipse user interfaces is much easier to use. The recommended User Interface for Git, is EGit. If you are on a lab machine, you should already have it installed. Otherwise you can install it by opening Eclipse, and clicking Help → Install New Software → – All Available Sites – (Under Work with:) → Collaboration. Select the Eclipse EGit and Eclipse Egit Mylyn GitHub Feature, and follow the instructions.

**Note:** If you don't see anything come up in the list when you select – All Available Sites –, click add and put the following update site in:

`http://download.eclipse.org/releases/kepler`

**Tip:** When working in EGit, you can actually simultaneously use git on the command-line to make fixes as necessary. Make sure that after you use Git on the command line you right click on the project and hit Refresh.

The main Egit User Guide is available here: (http://wiki.eclipse.org/EGit/User_Guide).

You should now re-do he previous steps in Part 2, but this time with EGit. You don't have to following the exact same sequence of steps, since you know most of the answers, but you should do the following:

1. Re-clone the project. (http://wiki.eclipse.org/EGit/User_Guide#Working_with_remote_Repositories)
2. Merge the ui branch into the master branch & resolve any conflicts (http://wiki.eclipse.org/EGit/User_Guide#Merging_a_branch_or_a_tag_into_the_current_branch) & (http://wiki.eclipse.org/EGit/User_Guide#Resolving_a_merge_conflict)
3. Merge the async branch into the master branch & resolve any conflicts (http://wiki.eclipse.org/EGit/User_Guide#Merging_a_branch_or_a_tag_into_the_current_branch) & (http://wiki.eclipse.org/EGit/User_Guide#Resolving_a_merge_conflict)
4. Restore the previous version of the file. (http://wiki.eclipse.org/EGit/User_Guide#Inspect_History)

You should also know how to:

5. Find the differences between your branch and another branch

(http://wiki.eclipse.org/EGit/User_Guide#Comparing_with_Branches_.28Synchronize.29)
(http://wiki.eclipse.org/EGit/User_Guide#Inspect_History)

6. Find commit hashes (this is simply the ID in Eclipse)

## *Part 4 – Pushing and Pulling to GitHub*

Now that you have some experience with resolving conflicts, we will look at generating them. In groups of two, you will need to complete each of the following (You can use Git or EGit):

1) Create a new public repository on GitHub.

2) Push an initial project (for instance the StockWatcher project from the previous week or the Product Listing project you worked on this week) to GitHub.

3) The person who created the repository should create a new commit, and push that version to the server. (http://wiki.eclipse.org/EGit/User_Guide#Pushing_to_upstream)

4) The other person should clone the repository, make a change, commit and push it back to GitHub.

5) Each of you should take a turn generating a merge conflict that the other person has to deal with.

## *Git Command Reference*

The following commands may be helpful:

```
git help
git commit [-m “commit message”]
git checkout <branch | commit-hash>
git clone <url>
git show <branch | commit-hash>:<file>
git push
git merge
git log [--stat]
git diff [-R] [--stat] <branch/commit-hash>
git pull
git status
```

If you have a branch/tag/commit-hash that points to a specific commit ~1 is the previous commit, ~2

two commits back, etc... For instance if commit `0b248ab`... is the last commit on the branch `foo`, and `b821902`... is the second last commit. Then the following are all synonymous.

```
git diff foo~1
git diff 0b248ab~1
git diff b821902
```

## *Deliverables*

1) The answers to the questions from Part 2

2) A link to the repository in Part 4. The repository should clearly show merge conflicts that have been resolved.

## *Further Reading*

1) The following article provides an example of a typical work flow for git branches that is used in industry:

http://nvie.com/posts/a-successful-git-branching-model/

2) A nice cheat sheet for Git is available here:

http://rogerdudler.github.io/git-guide/files/git_cheat_sheet.pdf