CSC2620 Object Oriented Design – Spring 2024
Unit 0 – Version Control

For Next Class

- Re-read and familiarize yourself with the syllabus.
- **Important:** Watch the second git video on your own.
- Download the textbooks locally.

0.1 Overview

- A tool for tracking versions of software is called a *source code manager* (SCM).
- There are several types of SCMs:
  - Git
  - Mercurial
  - SVN
  - SCCS
  - CVS
  - etc.
- Every SCM supports a set of core features:
  - A method for creating a *repository*
  - A method for adding files to the repository.
  - A method for removing files from the repository.
  - A method for making changes to files in the repository.
  - A method for tracking and reporting history of files in the repository.
  - A method for branching.
  - A method for updating a local copy of the the repository.
- What should you track in a repository?
  - Files that change and need to be tracked with history.
  - Any dependency your code has that is needed for compilation.
  - Art assets.
- What should you *not* track?
  - Automatically generated files.
  - PDFs that are housed elsewhere
  - Design documents
    * These are normally housed using some other version control software (e.g., Google Docs).
- Git
  - As mentioned previously, we will use Git as our SCM.
  - It is a distributed SCM that is popular in many industry settings.
  - It allows for many different workflows that are common.

0.2 Git Basic Operations

- The best place to find everything you want to know about Git is at the Git project home page.
  - `https://git-scm.com/`
- The first task in any project is to create a new repository. This is easy in Git.
  - Any directory can be turned into a repository by using the `init` command to `git`.
  - You will be able to do this with a mouse click, or two, in your IDE.
  - If you are command line warrior, the command is `git init .` in the directory you want to make a repository.
- Things to do after you create your repository:
  1. Clone it.
  2. Add some files.
  3. Set up a `.gitignore` file – you'll thank me later.
- The first step we said was to clone your repository.
  - This is promoting your original repository to be an official copy of sorts.
    - The *golden bits*.
  - A clone is nothing more than a copy of the repository at a given point in time.
    - The clone lives on your local machine while the repository may not.
    - In Git all of the history for every file in the repository is also copied down into your clone.
    - Any changes you make to the clone, are only in that clone.
- To add files to the repository you will use the `add` command.
  - Again, this can be done using your IDE with a few mouse clicks.
  - From the command line you can use `git add .` to add all *untracked* to *clone repositories* tracking.
- The `.gitignore` file is very important.
  - It tells Git what files in the clone that it should ignore.
  - Generally this is automatically generated files and output files.
    - For example the `build` directory generated by an IDE.
  - The format of the file is very simple.
    - Each line is a relative path or file name to ignore.
  - For example, the line `build/` will ignore all `build` directories.
    - Hint: You will want to do this, especially if developers are working with different development platforms.
  - It also supports pattern matching, similar to regular expressions, see `https://git-scm.com/docs/gitignore`.
- Once you are satisfied with your changes, you should stage them for submission to a repository.
  - In Git files that have been modified are said to be *staged.*
    - You can see staged files in your IDE, normally with color coding.
    - Git allows you to list staged files by issuing the `status` command.
      - In the clone type `git status`
  - The add process in Git has a second purpose.
    - It *stages* changed files that are used to update a repository.
    - You could add everything to the stage or just select files.
    - Again, use `git status` to tell you want files have been changed.
  - To get the staged changes ready to update a repository you *commit* to the staged files.

        * When you commit you create what is called a *change set.*
        * A change set is a collection of related changes.
        * When you commit to your changes you must provide a commit message.
            · For our purposes, we will take the commit message to always bee the issue number followed by the synopsis line of the issue.
    - Note: If you were to type `git status` or otherwise check the status after a commit, there will be nothing staged.

- The final step is to update a remote repository.
    - This is done using a *push.*
    - A push submits a committed set of changes to a remote repository.
    - Can be done in your IDE or from the command line using `git push`.
    - **If things could go wrong, this is where they go wrong.**
- It is a common occurrence that you will want to update your clone with changes from a remote repository.
    - This is done using the `pull` command.
    - As a warning **If things could go wrong, this is where they go wrong.**

0.3 Merging

- There were two warnings around push and pulls.
- This warning is because of the merge process.
- Since multiple developers may work on the same remote repository and thus can make changes to the same files, there is a case where the system must reconcile the change sets.
    - This process is called *merging.*
- The merge process always happens in the clones (local repository).
- There are several cases that could occur when you need to push.
    - The easy case (fast-forward merge)
        * The changeset you are pushing are not in the remote repository. There are no conflicts!
        * The files you are pushing haven not been modified in the remote repository. There are no conflicts.
    - The harder case
        * There are changes in the remote repository that affect files contained in your changeset.
    - The easy case is, well, easy.
        1. Pull any changes (using `git pull`).
        2. Push your changeset to the remote repository.
    - In the harder case, there have been changes to files in the remote repository that your changeset modifies.
        1. Pull any changes (using `git pull`).
            * The pull will try to merge your changeset with the remote repository.
        2. If the automatic merge fails, you need to resolve all of the conflicts in the listed files.
            * Open the file with conflicts look for `>>>>` and `<<<<<` and determine what is the correct change.
            * Once this has been done for all files in conflict, go through the process of staging, committing, and pushing.

To save yourself a headache make sure all of your staged changes are committed before you pull from a remote repository.

0.4 Branching

- Branching is a convenient way of isolate changes.
- You can easily create a branch in a clone by using the `branch` command to Git.
  - As part of the branch you must specify a branch name.
  - For example, `git branch issue3`.
- To list all the branches in the current repository you execute the command `git branch`
- To delete a branch you can execute the command `git branch -d branchname`.
  - You should only delete a branch once it has rejoined the mainline.
- To swap between branches the `checkout` command is used.
  - e.g., `git checkout branchname`.
  - Before you use a newly created branch, you *must* check it out.
  - You can create a new branch and check it out in one step if you issue the command: `git checkout -b branchname`. The `-b` option tells `checkout` to create the new branch.
- When you want your branch to rejoin the main line, you will undergo a *merge* operation.
  - Just like in the case of merging with remote repository changes, all merge will be done locally.
- The process for merging branches is fairly straight forward.
  1. Make sure all of the changes in your source branch have been committed.
  2. Checkout the branch you want to merge to.
     - It is referred to as the receiving branch.
  3. Merge your source branch into the receiving branch by using the `merge` command.
     - e.g., `git merge srcbranch`.

0.5 History

- One of the useful ideas that all SCMs have in common is *file history.*
- Every file managed by Git has history associated with it.
- Every time you do a pull or clone you get a copy of all the changes and the associated history.
- To see the history of the repository you use the `log` command.
  - e.g., `git log`
  - To see more information about the files involved in each changeset use the `--stat` subcommand.
    * e.g., `git log --stat`
- As an example of a log entry (with `--stat`) we have:

```
commit 0994f88a772fe62757164adef9eb1aa43a848133
Author: Zach Kissel <kisselz@merrimack.edu>
Date:   Sun Sep 6 21:38:24 2020 -0400

    #7 Implement nextword text generation lab.

nextword/index.html  |  42 ++++++++++++++
nextword/nextword.js | 148 ++++++++++++++++++++++++++++++++++++++++++++++++----
2 files changed, 181 insertions(+), 9 deletions(-)
```

  - The hexadecimal number after the word "commit" is the changeset identifier
  - The "Author" field tells us who made the commit.
  - The "Date" field tells use the date and time that the commit was made to the repository

  - The commit message is the next line. Notice the issue tracker number.
  - The remaining lines tell us what was changed
    * two files were modified
    * 42 changes in `nextword/index.html`
    * 148 changes in `nextword/nextword.js`

- If we want to see the changes that were made by the commit, we can use the `show` command.
  - This requires the changeset number.
  - e.g., `git show 0994f88a772fe62757164adef9eb1aa43a848133`

0.6 Backing Out Changes and Removing Files

- Occasionally you may have a change in the repository that is problematic.
- If the change is at the `HEAD` of the branch, you can remove it using a `revert` process.
  - This requires the clone to have no staged changes.
- The command for reverting changes in Git is the unsurprisingly named `revert`.
  - e.g., `git revert HEAD` will undo the previous commit/push.
  - You can also revert early commits by providing a changeset number, however, this error prone.
    * Just file a bug and fix it with a new commit.
- At times you may need to remove a file from source control.
  - In Git this is done using the `rm` command.
  - For example, `git rm foo.txt` will remove the file `foo.txt` from the repository.
  - Any remove will have to be staged and committed, just like any other change.

0.7 SCM Workflows

- How you utilize an SCM is called its *workflow*.
- There are several common types of workflows:
  1. Centralized
  2. Managed
  3. Advocate Based or Gated Check In

0.7.1 Centralized Workflow

- This is one of the oldest models.
- There is one master repository that all developers interact with.
- Generally treated as the master copy of the bits, any copy the developers have are just that, copies.
- Some of the systems have the notion of file checkouts that allow only one developer to work on a file at a time.
  - This has the advantage or reducing problems with merging.
  - The disadvantage is only one person can work on the file at a time.

0.7.2 Managed Workflow

- There is one master copy of the source code.
- Each developer has their own copy of the master repository.
- Developers make changes only commit to their local repository, never the master.
- The integration manager, a specially designated engineer, will pull in changes they think are good.
  - These changes then become part of the master repository.

0.7.3 Advocate Based/Gated Check In

- This is similar, in a way, to the managed workflow.
- Every developer has a copy of the master repository.
- A developer can update the master repository only when a designated *change advocate* agrees with the changes.
  - Generally enforced by setting the master repository to read only unless your change has been approved.
- This process is used in very large projects.
  - This is the model that was used a companies I have worked for.

0.8 Practice

1. Install git if you have not already done so.
2. Partner up with 1 or 2 classmates.
3. Each of you, create an account on github (if you haven't already). You should use your Merrimack account name if possible.
4. Create a single repository for a mock project.
   - Add each member of your group to the repository
   - Add me (stuetzlec@merrimack.edu) to your repository (as an editor with write access)
5. In VS Code, have *each member* clone your newly created repository
6. Now that each member has their own cloned copy of the repository (repo):
   - Each team member should add their own Class file to the project.
   - Add to the `.gitignore` any files *that are not .java files* in your project. Repositories should mostly contain only files necessary to run the project on each person's architecture (such as source code, art assets), as well as github
7. From inside VS Code, commit and push your changes.
8. From inside VS Code, pull your teammates' changes.
9. Practice working simultaneously, creating and editing files, staging and pushing commits, and pulling your partners' commits.
10. Make sure you are comfortable with this workflow before the end of class today.