Version Control

1. Intro to Version Control
   1. What it is
      1. Version control systems regulate changes so that developers can keep track of their own work and collaborate with others on the same project at the same time.[[1]](#footnote-1)
   2. Purpose/Benefits
      1. It allows developers to know who made what changes and when so that everything is organized and controlled. At any time, developers can go back and redo a project or file from any point in ‘history’. Additionally, VCSs allows developers to compare different versions and not override other developer’s code too easily. [[2]](#footnote-2)
   3. Types
      1. Local
         1. On each PC there is a directory where the files are saved every so often. This is not ideal because it is only available on one computer and it’s very easy to make mistakes.[[3]](#footnote-3)
      2. Centralized
         1. Figure 1
         2. One server with all the versions saved. Each developer ‘checks-out’ the versions that they need.[[4]](#footnote-4)
         3. Advantages are that there is communication between developers. Everyone can see what others are doing because there is one centralized server. Administrators have more control. It’s easier to work with.
         4. Disadvantages are that if the central server goes down or is compromised, all the data could potentially be lost. [[5]](#footnote-5)
      3. Distributed
         1. Figure 2
         2. Developers checkout all versions every time they clone.
         3. Advantages are if a server goes down, they can copy from one of the other developers back to the server. Additionally, developers can connect to more than one repository at once.[[6]](#footnote-6)
         4. Disadvantages are that it could take up more space.
      4. In short, local is one copy on one machine, centralized is one copy that can be accessed from many machines, and distributed is many copies on many machines.[[7]](#footnote-7)
2. Version Control Terminology
   1. Repository
      1. A group of files that is under version control
      2. Other older VCSs it means the central code[[8]](#footnote-8), which there is no such thing in distributed systems.
   2. Checkout- checkout a specific branch from central repository
   3. Revert- reverts to initial state and deletes pending changes[[9]](#footnote-9)
   4. Working copy- developers personal copy of the central repository.[[10]](#footnote-10)
   5. Trunk- main branch meaning the most up to date branch where all developers push to and pull from
   6. Branches- divide from the trunk to attempt something new[[11]](#footnote-11)
   7. Patch- A change done to one version that produces a new version.[[12]](#footnote-12)
   8. Commit-
      1. Save new changes to original on your computer[[13]](#footnote-13)
      2. CVS- Send changes to central repository[[14]](#footnote-14)
   9. Pull/ Update- update your current branch by ‘pulling’ from the shared central repository. Used when other developers made changes to central repository. Needs to be done before merging changes to central repository so that there is no conflicts.[[15]](#footnote-15) Does fetch and merge together.
   10. Merge- merge working copy to central repository[[16]](#footnote-16)
   11. Fork- branch off the main code to produce entirely new file. Will not be merged back. Now owned by you and does slight or big differences from the original.[[17]](#footnote-17)
   12. Status- new, changed, and deleted files[[18]](#footnote-18)
   13. Log- See who did what[[19]](#footnote-19)
3. Comparing Version Control software and how to use them
   1. Local
      1. RCS
         1. RCS saves just patches so if a developer wants a specific version the computer combines the patches.[[20]](#footnote-20)
   2. Centralized
      1. CVS
         1. Can ‘rollback’ and completely delete[[21]](#footnote-21)
         2. Not so easy to store any type of file and metadata[[22]](#footnote-22)
      2. Subversion
         1. Can’t ‘rollback’ meaning you can change it back but will always be in history.[[23]](#footnote-23)
         2. Can store all types of files without user specifying the type and can store metadata.[[24]](#footnote-24)
         3. Easier tracking
         4. Better revision numbering
         5. Can list branches
         6. Delete branches and if necessary revert
         7. Have copy of repository that is under control on personal PC (takes up a lot of room)[[25]](#footnote-25)
   3. Distributed
      1. Git
      2. Mercurial
4. Comparing the different VCSs
5. An Example of a Popular VC: Git and GitHub in Detail
   1. Installing Git
      1. Git can be installed in a few ways, I will be using the command prompt which can be accessed in all three installation methods.
         1. Git Bash- a version of the command prompt with git already preconfigured.
         2. Git- the actual source code which can we used from the command prompt.
         3. GitHub Desktop- a graphical interface that works on with your local repository to track changes and differences between your local repository and the repository stored on GitHub.
   2. Creating repositories
      1. Once you have git installed, you want to start keeping track of files. You can either
         1. Clone an existing repository from GitHub
            1. GitHub is a website that stores repositories, allows the developers to do all git functions, and much more. Developers use GitHub like a central repository, in that it is the most up to date branch. All developers can access it and update their local repositories from it when necessary. They also merge their own updates to the repository stored on GitHub.
            2. To clone an existing repository, open the command prompt and cd to the directory where you want your local repository to exist.
            3. Clone the repository by git clone <URL>.
            4. You should get back something like the following:
            5. Cloning into '<Name of Repository>'...  
               remote: Counting objects: 97, done.  
               remote: Compressing objects: 100% (10/10), done.  
               remote: Total 97 (delta 1), reused 0 (delta 0), pack-reused 87  
               Unpacking objects: 100% (97/97), done.
         2. Create a new repository
            1. Open the command prompt and move to the directory where your new repository is stored. Ex: cd c:/gitTutorial.
            2. Then git init to initialize the repository as a git-controlled repository. This will produce a .git folder within your repository.
            3. Add and commit all the existing files to your repository. See adding and committing
            4. Now you should have a version-controlled repository on your local computer.
            5. To clone your new repository to GitHub, open the command prompt and cd into your local repository.
            6. Create a completely empty repository in GitHub.
            7. In the command prompt, add the GitHub remote repository named and name it.
            8. git remote add origin git@github.com:<username>/<repository name>.git
            9. Push your current repository (master) to the remote repository (called origin)
            10. git push origin master
            11. You will be asked for your username and password of your GitHub account. Keep in mind that when entering your password the command prompt will appear as if nothing is being entered.
            12. fatal: HttpRequestException encountered.  
                   An error occurred while sending the request.  
                Username for 'https://github.com': <username>  
                Password for 'https://<username>@github.com':  
                Counting objects: 53, done.  
                Delta compression using up to 4 threads.  
                Compressing objects: 100% (51/51), done.  
                Writing objects: 100% (53/53), 55.79 KiB | 5.58 MiB/s, done.  
                Total 53 (delta 2), reused 0 (delta 0)  
                remote: Resolving deltas: 100% (2/2), done.  
                To https://github.com/elishevastrauss1/SchoolWork.git  
                \* [new branch]      master -> master
   3. Committing
      1. Figure 3
      2. There are many steps to change and save changes to a git repository.
      3. First do git status to find out what was changed. The output will be:
      4. On branch master  
         Your branch is up to date with 'origin/master'.  
           
         Changes not staged for commit:  
           (use "git add <file>..." to update what will be committed)  
           (use "git checkout -- <file>..." to discard changes in working directory)  
           
                 modified:   <file name>
      5. Add those files to the pending list to be committed by typing git add <file name> or git add . . This is called staging a commit. Those files are now staged and ready for committing. They have not been saved to your local repository and you can still change the files.
      6. Commit the changes to your local repository by git commit.
      7. A message editor will pop-up to write a detailed message about your commit. Save the message and close the editor.
      8. Once the editor is closed, the command prompt will say something like:
      9. [master (root-commit) <Commit Log Number>] <Your Message>  
         40 files changed, 1136 insertions(+)  
         create mode 100644 .classpath  
         create mode 100644 .project  
         create mode 100644 .settings/org.eclipse.jdt.core.prefs  
         create mode 100644 MusicLessons.html  
         create mode 100644 MusicNotes.jpg  
         create mode 100644 StudentLists.txt  
         create mode 100644 bin/dataStructures/LinkList.class
      10. To unstage a file that was already added to the pending list:
      11. git reset HEAD <file name>
      12. To undo all changes since the last commit
      13. git checkout -- <file>
   4. Updating, merging, fetching
   5. Reverting, cherry-picking
      1. git log to find the commit that you want to go back to.
      2. Revert
         1. Copy the log number and then git revert <log number>. This will make a new commit that does the opposite of the reverted commit. All + become – and visa-versa. You only need the first eight numbers.
   6. Branching
   7. Using GitHub
6. Full Version Control Plan
   1. If a system goes down
   2. If a version blows up
      1. How to revert
         1. Check the log. Developers should have made sure to write thorough messages which will describe the changes made. Find which commit you want to go back to.
         2. git revert <log number>
7. Conclusion
   1. Reinforce why we need VC (Card 22)
      1. Track code revisions
      2. Show history
      3. Support multiple users on same code at once
      4. Support branching
      5. Support merging of branches
      6. Reveal conflicts and allow to fix them
      7. Support going back in history



Figure 1 From Pro Git



Figure 2 From Pro Git



Figure 3

1. Source 1 (Card 1) [↑](#footnote-ref-1)
2. 1 (2) [↑](#footnote-ref-2)
3. 1 (4) [↑](#footnote-ref-3)
4. 1 (6) [↑](#footnote-ref-4)
5. 1 (7) [↑](#footnote-ref-5)
6. 1 (8) [↑](#footnote-ref-6)
7. 3b (13) [↑](#footnote-ref-7)
8. 3c (16) [↑](#footnote-ref-8)
9. 2 (24) [↑](#footnote-ref-9)
10. 3b (14) [↑](#footnote-ref-10)
11. 3c (16) [↑](#footnote-ref-11)
12. 1 (3) [↑](#footnote-ref-12)
13. 3b (14) [↑](#footnote-ref-13)
14. 5 (20) [↑](#footnote-ref-14)
15. 3b (14) [↑](#footnote-ref-15)
16. 3b (14) [↑](#footnote-ref-16)
17. 3b (14) [↑](#footnote-ref-17)
18. 7 (23) [↑](#footnote-ref-18)
19. 2 (24) [↑](#footnote-ref-19)
20. 1 (5) [↑](#footnote-ref-20)
21. 9 (26) [↑](#footnote-ref-21)
22. 9 (27) [↑](#footnote-ref-22)
23. 9 (26) [↑](#footnote-ref-23)
24. 9 (27) [↑](#footnote-ref-24)
25. 10 (28) [↑](#footnote-ref-25)