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 when first clone the database to their local machine. There is usually a central server that all developers can access and is the version kept most up to date. Each developer can work locally, make changes, and test their work, and then just push to the central server for the other developers to access.[[6]](#footnote-6)
         3. Advantages are
            1. Very fast
            2. Don’t need internet[[7]](#footnote-7)
            3. 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.[[8]](#footnote-8)
         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.[[9]](#footnote-9)
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[[10]](#footnote-10), 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[[11]](#footnote-11)
   4. Working copy- developers personal copy of the central repository.[[12]](#footnote-12)
   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.[[13]](#footnote-13) Branching will completely change your working directory to be identical to your new branch.[[14]](#footnote-14)
   7. Patch- A change done to one version that produces a new version.[[15]](#footnote-15)
   8. Commit-
      1. Save new changes to original on your computer[[16]](#footnote-16)
      2. CVS- Send changes to central repository[[17]](#footnote-17)
   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.[[18]](#footnote-18) Does fetch and merge together.
   10. Merge- merge working copy to central repository[[19]](#footnote-19)
   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.[[20]](#footnote-20)
   12. Status- new, changed, and deleted files[[21]](#footnote-21)
   13. Log- See who did what[[22]](#footnote-22)
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.[[23]](#footnote-23)
   2. Centralized
      1. Like RCS in way stores data
         1. Uses patches, file based, saves each file change and has to calculate all the changes
         2. Only use <version system> add to add a file the first time. After that, you just commit[[24]](#footnote-24)
      2. CVS
         1. Can ‘rollback’ and completely delete[[25]](#footnote-25)
         2. Not so easy to store any type of file and metadata[[26]](#footnote-26)
      3. Subversion
         1. Can’t ‘rollback’ meaning you can change it back but will always be in history.[[27]](#footnote-27)
      4. Perversion
         1. Can store all types of files without user specifying the type and can store metadata.[[28]](#footnote-28)
         2. Easier tracking
         3. Better revision numbering
         4. Can list branches
         5. Delete branches and if necessary revert
         6. Have copy of repository that is under control on personal PC (twakes up a lot of room)[[29]](#footnote-29)
   3. Distributed
      1. Git
         1. Open-source
         2. Never deletes history, just moves pointer to new place
         3. Snapshots
            1. git add is done whenever you make changes because your saying you want this new change to be included in next commit/ snapshot.
            2. git commit produces new snapshots with different hash codes for each new snapshot. Can compare different versions of the same file because your just comparing snapshots.
            3. When sending the files, it is stored as changes not as whole files to minimize content and maximize speed and efficiency.[[30]](#footnote-30)
      2. Mercurial
      3. Bazaar
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 ex: 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 later. (git add . and git commit)
            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 as a remote repository and name it. In this example origin is the name of your new remote repository.
            8. git remote add origin git@github.com:<username>/<repository name>.git
            9. Push your current repository (here 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
            13. As we mentioned earlier, when cloning and passing files, git compresses them by saving the changes (called Deltas) to maximize speed. As shown in:
            14. Delta compression using up to 4 threads.  
                Compressing objects: 100% (51/51), done.
   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. The first two lines are saying that your local repository, master is “up to date” with your remote repository, origin. If the other developers changed the remote repository, you would have to update your local repository before committing. See updating your local repository.
      6. 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.
      7. Commit the changes to your local repository by git commit.
      8. A message editor will pop-up to write a detailed message about your commit. Save the message and close the editor.
      9. Another option would be to just add the message after the commit command by saying git commit -m "<message>"
      10. This will not open an editor and will just continue as if the editor was opened, saved, and closed.
      11. Once the editor is closed, the command prompt will say something like:
      12. [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
      13. To unstage a file that was already added to the pending list:
      14. git reset HEAD <file name>
      15. To undo all changes since the last commit
   4. Finding differences
      1. git diff <file name>
   5. Updating your local repository, fetching
      1. git remote update
      2. git fetch
      3. git pull
   6. 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.
   7. Branching, merging
      1. As a reminder, branching will completely change your working directory to be identical to your new branch.
      2. git branch to determine what branches exist and what is your current branch.
      3. git branch <new branch name> to create a new branch
      4. git checkout <file> to checkout a specific branch into your working directory
      5. git checkout -b <new branch name> to make a new branch and switch to it at the same time
      6. “Working directory needs to be clean”, meaning no pending changes or unstaged changes, if you want to switch to another branch.
      7. git stash will clean out the working directory and save your unstaged changes.
         1. To get back previously stashed changes
         2. git stash show will give a list of the branches and files that have stashed changes.
         3. git stash apply will apply stashed changes to your current branch.
   8. Using GitHub
      1. Pull requests
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>
   3. If a new software or version is introduced to your company
      1. Trainings
         1. Old people
         2. New people
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 From Pro Git



Figure From Pro Git



Figure

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. 11 [↑](#footnote-ref-6)
7. 11 [↑](#footnote-ref-7)
8. 1 (8) [↑](#footnote-ref-8)
9. 3b (13) [↑](#footnote-ref-9)
10. 3c (16) [↑](#footnote-ref-10)
11. 2 (24) [↑](#footnote-ref-11)
12. 3b (14) [↑](#footnote-ref-12)
13. 3c (16) [↑](#footnote-ref-13)
14. 11 [↑](#footnote-ref-14)
15. 1 (3) [↑](#footnote-ref-15)
16. 3b (14) [↑](#footnote-ref-16)
17. 5 (20) [↑](#footnote-ref-17)
18. 3b (14) [↑](#footnote-ref-18)
19. 3b (14) [↑](#footnote-ref-19)
20. 3b (14) [↑](#footnote-ref-20)
21. 7 (23) [↑](#footnote-ref-21)
22. 2 (24) [↑](#footnote-ref-22)
23. 1 (5) [↑](#footnote-ref-23)
24. 11 [↑](#footnote-ref-24)
25. 9 (26) [↑](#footnote-ref-25)
26. 9 (27) [↑](#footnote-ref-26)
27. 9 (26) [↑](#footnote-ref-27)
28. 9 (27) [↑](#footnote-ref-28)
29. 10 (28) [↑](#footnote-ref-29)
30. 11 [↑](#footnote-ref-30)