Kaushal Mangipudi, Gerard Gaimari, Robert Kolmos, Kamden Chew

GitUp: A Simple, Portable Backup Manager with Practical Version Control

Motivation:

Backing up any project is essential to its reliable completion, and software projects are no exception. Today, more and more inexperienced non-developers work on software projects, and while they are the only coder and their projects are relatively small, they still need to backup their work. Due to their lack of experience with software development, non-developers need different things in a backup system than experienced programmers. A backup system that satisfies all the needs of non-developers must:

1. **Offer long-term tracking of many past versions of files and allow easy reverting to said past versions.** It can be difficult to remember which past versions of files have which changes, especially when the files are large, or the file system itself is large. There are tools that allow users to quickly lookup a file's changelog (the command git blame if they're using git), but these typically take the form of command-line interfaces which serve to confuse a non-developer.
2. **Offer a quick and easy way to compare past versions with one another.** Often times, the differences between versions of a file can be very small and hard to see with the naked eye, as bug fixes often involve changing only a couple lines.
3. **Be easy to use.** Non-developers are already short for time. Due to their inexperience, they already spend far more time writing their actual code than actual developers. Even worse, since coding isn't their primary job, they have less time to code than full time developers. They do not have the time to learn a complicated system, so any backup system they use must be fast to learn.
4. **Be as automated and out of the way as possible**. Non-developers are already out of their depth. Unlike software developers who use version control systems, they are not used to using some third party system to back up their work after saving their work. As a result, it is unreasonable to expect them to constantly remember to take the extra step of manually backing up their work after they save it regularly.

There are a plethora of backup tools available for non-developers to use, but unfortunately, none of them meet all of their needs. Existing backup tools can be divided into four broad categories:

1. **Cloud Sync services (OneDrive[7], Google Drive[8], DropBox[9], etc.)**. They are very easy to use and automatically save a user's work and sync it with other copies of the user's work on other machines and the cloud. Some of them, like Google Drive, let you compare a version with the directly preceding version to see what changes are made. However, Cloud Sync services do not have reliable, long-term version tracking. They save a few of the past versions and give you the option of reverting to them. However, they only save the past versions for a couple of weeks at the longest. If a change you want to undo happened a while back, or if the change you want to undo isn't cleanly isolated to one of the past versions, it will be impossible to easily undo the change.
2. **Cloud Backup services (BackBlaze[10], Mozy[11], Carbonite[12], etc.).** Like Cloud Sync services, for the most part, they are easy to use and automatically backup a user's work. They store more past versions of a user's work for a long time, so they allow a user to more easily undo changes they made, even if the changes happened a while ago. However, they do not provide the user with an easy way to identify what the changes were. If a user wanted to compare two past versions of a file with one another, they'd need to open up a text comparison tool and paste both versions of their file into it, which is a waste of time and very disruptive to their workflow.

3. **Dedicated Version Control Systems (git[13], Mercurial[14], Subversive[15], etc.).** They offer comprehensive, near eternal version tracking and allow a user to easily compare past versions of a file with each other. However, they are very difficult to use and require users to interrupt their work flow to backup their work. Git, by far the most popular version control system[1], was reported to have a very steep learning curve and an inadequate UI and documentation by over 70% of its users[2]. If experienced developers are overwhelmed by git's enormous amount of functionality, non-developers will certainly struggle even more. Additionally, these version control systems use command line interfaces, which can be very hard to use if you have limited experience with them.
4. **VCS 'Simplifiers' (GitKraken[16], Gitless[17], TortoiseHg[18], SmartGit[19], Tower[20], GitHub Desktop[21], etc.).** These tools purport to make VCSs easier to use, and they do, but only for developers already familiar with how VCSs work. Some of them still rely on command line interfaces, and all of them still require a user to understand how VCSs work. That isn't a problem for developers, but non-developers do not have the understanding, the time, or a reason to obtain it.

Proposed Solution:

GitUp aims to preserve the reliability, automatic nature, and security that existing cloud-based backup systems provide while also providing users with the robust version tracking of files they need for software projects in an intuitive way. GitUp is designed for users working on single-developer projects, with a focus on inexperienced users who need to code but work outside of software. To use GitUp, a user opens the application and logs in to their GitHub account. Once they are logged in, they select the project they want to work on. GitUp automatically tracks project file versions and updates the backup whenever it detects a change. It provides simple version restoration with granularity ranging from the entire project to individual file changes.

No other system has been created with the idea of single user projects in mind. Users who work outside of software and have never used version control systems won't be able to use them without investing considerable amounts of time to learning them. GitUp allows these inexperienced users to enjoy robust version tracking without having to learn about the underlying architecture of git or some other version control system.

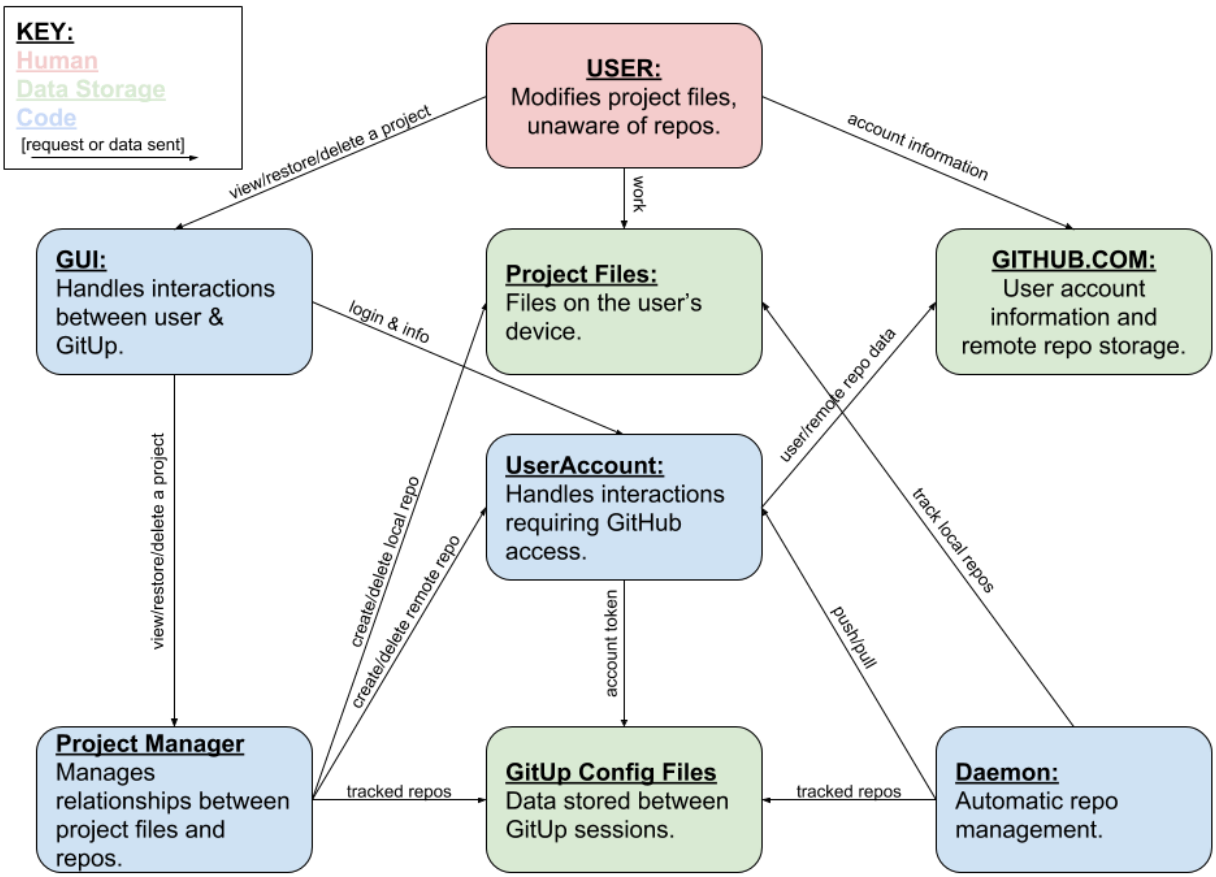
GitUp's Functionality:

1. The user can create a new GitHub project or open an existing one
2. Any locally saved changes are automatically backed up on the remote repository
3. The local version of the project is periodically synced with the remote repository
4. Automatically backed up past versions are grouped by files modified and/or time modified before becoming visible to users
5. The user can view past versions of files and revert a file to a past version if they desire to
6. The risk of merge conflicts is almost completely eliminated, and the user is warned in advance about any potential merge conflicts that could occur
7. A very inexperienced user with minimal programming experience and no experience with version control systems can do all of the above with the help of the user manual

User Interface:

See GitUp's User Manual for a detailed explanation of what GitUp's UI looks like and how users interact with it: https://github.com/gerar231/GitUp/blob/master/README.md

Architecture:



To save development resources GitUp's architecture was designed to make the most of the following existing libraries:

1. Tkinter[4] to create the GUI.
2. GitPython[5], to control local repositories in Project Manager
3. PyGithub[22], to control remote repositories and user account information on GitHub in User Account
4. Linux's inotify[6] library to watch project files through the Daemon.

It uses a version control system behind the scenes. This could be any version control system, but we chose git since it is the version control system we are most familiar with. The architecture assumes that the user will not attempt to modify/manage either local or remote repositories and excludes a module to repair repositories. The user can still technically interact with their project using git in the command line, but the user will never need to, as GitUp will handle all interactions between the local and remote repositories.

Use of git command line is discouraged by the user manual for this reason. The user can work on a project using multiple devices, but each project can only be worked on by a single user, and the user should only use one device at a time.

The GUI aims to emulate the interface of cloud-based backup systems like Google Drive and Dropbox, with additional functionality involving viewing and reverting past versions of files. User interactions are passed from the GUI to either the User Account or Project Manager module depending on their intention. User Account handles any requests that would require direct access to GitHub such as login information, and stores an access token on disk. Project Manager handles any requests that would require access to local or remote files to view/restore/delete a "project" as specified in the User Manual, forwarding requests to create/delete/modify a remote repository through User Account for access to GitHub. Additionally, Project Manager communicates with the Daemon through local files to ensure the correct repositories are being tracked. The Daemon runs in the background as long as there are repositories that need to be tracked. Functionally, the Daemon automatically creates meaningful commits and requests to push/pull from local repositories to remote repositories when necessary.

GitUp does not attempt to manage local project files, only repositories, to prioritize the abstraction of version control from the User and keep local and remote repos current and synchronized. GitUp automatically creates local and remote repositories whenever a user views a new project and handles many cases in between:

1. Project files not a part of an existing repository → Create a new local repository with a remote to a new remote repository.
2. Project files are part of an existing repository without a remote repository backed by GitUp → Create a new remote to a new remote repository.
3. Project files are part of an existing repository with a remote backed by GitUp → ensure they are tracked by the Daemon.

If GitUp detects an inconsistency in local and remote repositories that it cannot resolve then it issues a warning that the file is out of sync and prompts a user response.

Implementation:

The UI is implemented with Tkinter[4] for Python since this library is platform independent allowing a single build to produce a GUI for OS X, Linux and Windows 10 if we manage to find the time to support operating systems other than Linux. Tkinter provides a basic UI kit that should be sufficient to create a UI resembling the above mockups. Python was chosen for GUI implementation so we can easily interface with the GitPython[5] API that GitUp uses to utilize git in order to interact with the local project files and the remote repository. GitUp automatically pushes by becoming aware of file opens and closes through Linux's inotify API[6]. When a user sets up a project, GitUp begins monitoring file opens, and closes in the directory associated with this project and it's subdirectories. When GitUp sees that a file has been opened it stores the current state of the file, and begins checking intermittently to monitor the magnitude of changes the user has made. When the user closes the file, or GitUp determines that a significant amount of changes have been made, GitUp automatically commits the changes to this file. If the file is still open

when a commit is made, GitUp updates the stored current state of the file, and proceeds monitoring changes. When a file is closed GitUp stops intermittently monitoring it for changes. GitUp displays a warning to a user about potential merge conflicts whenever an automatic push or pull fails, and to do so, it uses a popup implemented with Tkinter to do so. We do not put any hard blocks on modifications that could lead to merge conflicts, because the user is guaranteed to have seen a warning about merge conflicts since they are the only user working on the project. If the user still causes a merge conflict despite the warning, a pop up window appears asking them to resolve this conflict. With regards to grouping commits together to give the user a big picture of changes made to their project over time, we plan on grouping commits together based on the time they were created in increments of 1 day. We have also implemented commit messages that contain the method modified, and we plan to group commits according to this information as well.

Challenges and Risks:

Design Challenges:

One fundamental design risk that will need to be constantly addressed during the development of GitUp is that its design does not satisfy the problem it aims to solve, i.e. GitUp does not provide single users with almost all the functionality they need and/or it is bloated with too much functionality and difficult/unintuitive to use. To mitigate this risk, the GitUp team will conduct comprehensive user research at all stages of the development process using a group of developers familiar with git and a group of inexperienced developers with little to no version control experience who have kindly agreed to help us. When collecting information, we will focus on determining the most useful version control features needed in a backup system, user desires with respect to backup frequency, size and speed, and how we can ensure that user interaction with the system is simple and automated.

One technical challenge is how we plan on handling merge conflicts when the user causes one despite numerous warnings to be careful. We see three potential solutions to this. The first is to automatically replace the file with the version that is being pushed. A second possibility would be to save a renamed version of the file to the repository that the user could look at. The third would be to utilize a GUI based merge conflict resolution tool to help the user manually resolve the conflict. Given the complexity of difficulty of the third option, and the danger of overwriting files the user wanted to keep, we are leaning towards either the second option.

Testing & Evaluation:

These use cases are intended as examples of specific behavior that should work in order for the relevant phase of the project to be considered finished. These are not the exact tests that will be written as part of the test suite, but are meant as high level behavioral requirements. It is assumed that test suites for each phase will be written in order for the phase to be considered completed.

[1] Basic project use case: This test is intended to ensure that the basic functionality of automatic pushing/pulling works. In order for this part of the project to be considered successful a user should be able to:

- Open GitUp on machine 1
- Login to GitUp using a GitHub account

- Create a project “hello world” on machine 1.
- Add file helloworld.txt to project, and make changes on machine 1, and save the changes
- Verify that the changes were automatically pushed.
- Close GitUp on machine 1
- Open GitUp on machine 2
- Open the “hello world” project on machine 2.
- Verify that the version of “hello world” on machine 2 is up to date
- Make changes to helloworld.txt on machine 2, and save
- Close GitUp on machine 2
- Open GitUp on machine 1
- Verify that helloworld.txt is up to date

[2] Merge conflict use case:

- Create a project “parallel test” on machine 1.
- Add file “paralleledit.txt” to project, and make changes on machine 1
- Turn off internet on machine 1
- Save the changes on machine 1
- Verify that the user is warned that the automatic push failed
- Close GitUp, verifying that the user is warned once again that their changes haven’t been backed up
- Turn on the internet in machine 1
- Verify that the changes are pushed to the remote repository
- Close GitUp
- Verify that there are no warnings about merge conflicts
- Turn off the internet on machine 2
- Open GitUp on machine 2, and open parallel test
- Verify that a warning appears about not being able to sync the local version of parallel test with the remote version

[3] Reverting files use case:

- Create a project “Test revert” on machine 1
    - Create a file “version-test.txt” in the project with the line “version 1”.
    - Commit/push all changes on machine 1.
- Open “Test revert” on machine 2
    - Add the line “version 2” to version-test.txt
    - Create the file “extra-file.txt” with the line “Extra unrelated file”
    - Commit/push all changes on machine 2.
- Open “Test revert” on machine 1
    - Edit the file “version-text.txt” by adding the line “version 3” to it
    - Commit/push all changes on machine 1.
    - View the past version of version-text.txt with line “version 1”
    - View the past version of version-text.txt with lines “version 1” and “version 2”
    - Revert version-test.txt to the past version with lines “version 1” and “version 2”
- Open “Test revert” on machine 2

- Validate that version-test.txt has been successfully reverted back to the version with lines "version 1" and "version 2"

[4] Smart grouping case:
We are choosing to delay defining what smart change grouping means exactly until we receive feedback from users as to how they value various styles of grouping commits (grouping based on files, grouping based on time changes have been made, etc.).

[5] Final use case:
- Run all permutations of orderings of previous phase tests, and verify that all tests still pass when called in any order.

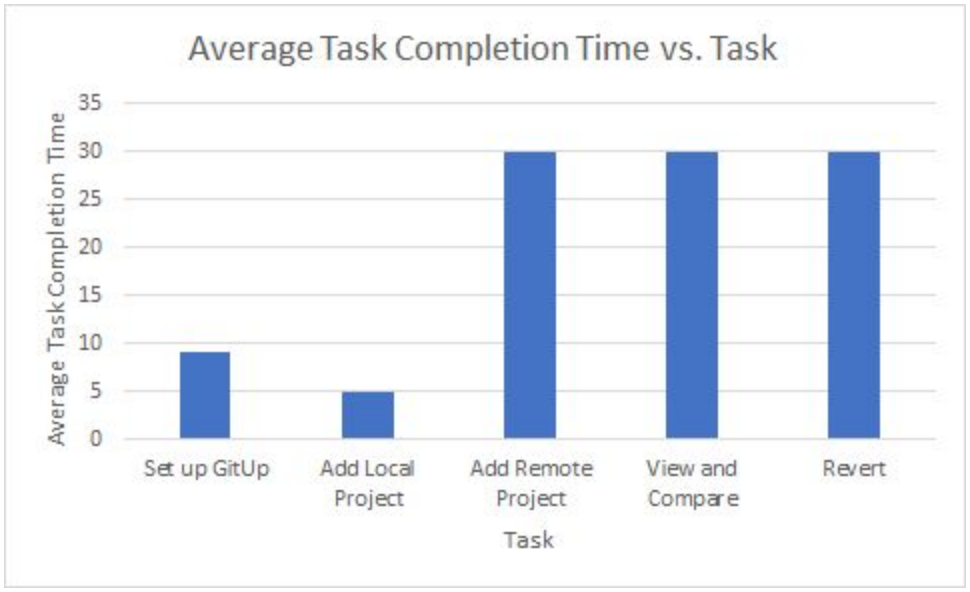
User Testing:

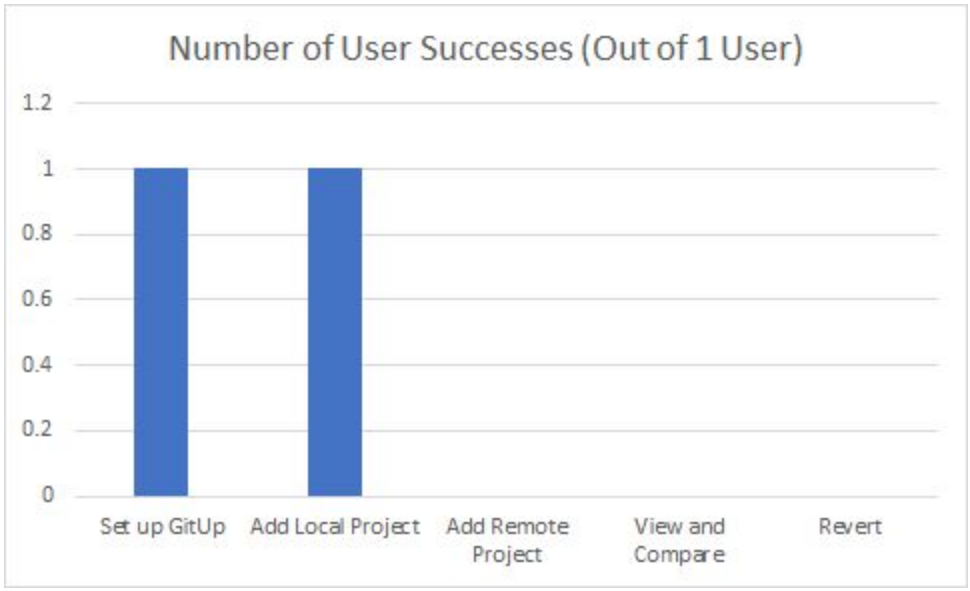
Evaluating GitUp's success required that we not only test that the implementation was working as expected, but that it achieved the goal of making project management for non-developers easy and intuitive. To make sure that our product hit this mark, we had a pool of non-developers with little to no version control experience who were willing to try out GitUp. We evaluated if our product was useful to its target audience by seeing if our pool of novice testers were able to navigate the system easily, or at least more easily than competitors.

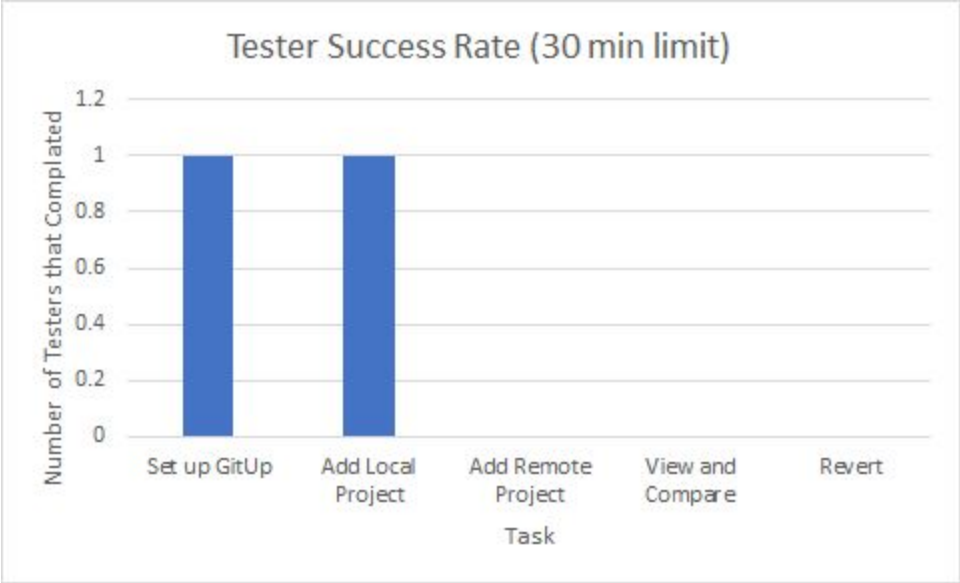
To that end, we had them fill out the form at https://goo.gl/forms/q37QkzF7sXDW0SHH3. The instructions in our survey gave general goals that we wanted the testers to be able to achieve, and they attempted to satisfy those goals in a time limit using GitUp. After the tester finished attempting the tasks, they answered general questions about their experiences with the tool. We transcribed the results from our survey to 'data/Survey Data Analysis.xlsx', which automatically generated graphs that visually represent our results.

In addition to this ease of use test, we also had a couple of testers more experienced with version control systems attempt to use GitUp to backup a software project of their own. They jotted down any interactions they had with GitUp, whether they thought those interactions were necessary, and if they ever wanted/needed to do something that they couldn't do with GitUp. This served as a functionality test of sorts, just to make sure that GitUp wasn't missing any essential functionality. Each response in our functionality test was saved as a text file in the data directory of the format 'Functionality_Test_[#]_Feedback.txt'.

Ease of Use Results:


Average Task Completion Time vs. Task
Average Task Completion Time
35
30
25
20
15
10
5
0
Set up GitUp
Add Local Project
Add Remote Project
View and Compare
Revert
Task


Number of User Successes (Out of 1 User)
1.2
1
0.8
0.6
0.4
0.2
0
Set up GitUp
Add Local Project
Add Remote Project
View and Compare
Revert



These results indicate that we need to update either our testing methodology, our user manual, or both. While dogfooding we were more easily able to navigate GitUp, but we as developers are biased with our information of the system. Likely what needs to be done is we need to update our User Manual to be more user friendly.

Functionality Test Results:
The file viewer felt cumbersome when there were files with super long lines. Users weren't able to revert files - that's a huge disadvantage. Also, since automatic pushing/pulling did not work, GitUp was not able to be utilized to back-up a project.

Lessons Learned:
Overall, our development schedule seemed to be far too ambitious. While we managed to get back on track in terms of our reports, our actual implementation rate did not match up with our schedule. We ended up falling fairly far behind in the later weeks. To combat this, we had some extra in-person meetings where we were able to bounce ideas and questions off eachother much more quickly and effectively. Seeing as most of our implementation issues stemmed from connecting our individual modules, this was incredibly useful time spent. If we had to do it over again, we may employ some form of pair-programming so that there's a higher cross-module understanding between team members.

Improvements:
In this section, we'll document our responses to feedback we receive from testers. This is separate from the Feedback Response section, which covers our response to course feedback.

Future Improvements:
Fully implement commit grouping.

Fix bugs with automatic pushing/pulling
Fully implement file reverting

Feedback Response:
We have responded to all the feedback we have received last week.

Work Used:

[1]**https://insights.stackoverflow.com/survey/2018 Accessed 2/12/19**
[2] **What's Wrong with Git? A Conceptual Design Analysis. Santiago Perez De Rosso, Daniel Jackson. Computer Science and Artificial Intelligence Lab at MIT.** ***Presented at the Onward 2013 Conference, p. 37-51.***
Accessed 2/12/19 at https://spderosso.github.io/onward13.pdf
[3]**https://www.gitkraken.com/git-client Accessed 2/12/19**
[4]**https://docs.python.org/3/library/tk.html#tkinter Accessed 2/12/19**
[5]**https://gitpython.readthedocs.io/en/stable/ Accessed 2/12/19**
[6]**http://man7.org/linux/man-pages/man7/inotify.7.html**
[7]**https://onedrive.live.com/about/en-us/**
[8]**https://www.google.com/drive/**
[9]**https://www.dropbox.com/**
[10]**https://www.backblaze.com/**
[11]**https://mozy.com/#slide-11**
[12]**https://www.carbonite.com/**
[13]**https://git-scm.com/**
[14]**https://www.mercurial-scm.org/**
[15]**https://subversion.apache.org/**
[16]**https://www.gitkraken.com/**
[17]**https://gitless.com/**
[18]**https://tortoisehg.bitbucket.io/**
[19]**https://www.syntevo.com/smartgit/**
[20]**https://www.git-tower.com/windows**
[21]**https://desktop.github.com/**
[22]**https://pygithub.readthedocs.io/en/latest/index.html**

Hours worked on report (team total): 101

Appendix:
Implementation Schedule:

Date	Goal(s)	Test(s)	Implementers	Evaluators	Achieved?

02/04	Finish preliminary user usage data and finish designing a preliminary UX/UI.	Be able to run example use cases through UX flowchart. Results of survey collected.	Rob, Kamden	Gerard, Kaushal	Yes
02/11	Automated pushing/pulling and OAuth.	Be able to complete the basic project use case [1].	Kaushal, Gerard	Rob, Kamden	Yes 3/1
02/18	Edge case avoidance/File reverting.	Be able to complete the syncing warnings use case [2] and reverting files use case [3].	Gerard, Kaushal	Rob, Kamden	No
02/25	Begin work on smarter grouping of changes.	Collect customer data on what users think a 'smart grouping' of changes are, and write a smart grouping use case/test based off the customer data.	Kaushal, Kamden	Rob, Gerard	No
03/04	Automated pulling and GUI completely hooked to back-end.	Users should be able to use GitUp to edit and backup files. Users will take a survey to gauge GitUp's simplicity compared to other VCS tools.	Kaushal, Kamden	Rob, Gerard	No Everything works separately, but the combined version doesn't 3/5
03/11	Implement initial version of commit groupings.	Smart grouping test case [4].	Rob, Gerard	Kaushal, Kamden	No
03/18	Obtain user feedback on commit grouping, and feedback implementation.	Dependent on feedback received. Fixes implemented address most important user concerns with GitUp.	Rob, Gerard, Kaushal	Kamden	No