Chapter 1

Introduction

In the winter spanning the years of 2014 and 2015, I worked as a teaching assistant for an introductory course in Python programming. The course was short, only four weeks long. I met with multiple groups of students as they worked their way towards their final project. Immediately the issue of collaboration came up. How can they divide work in a way that makes sense? How would they share code? Can two team members work on the same file at the same time?

The main question, of course, was how to share their code. In such as short course, learing about Git was out of the question. We were more worried about teaching students what a function was than showing them the finer points of branches and merging. Dropbox was an easy alternative, but after a half-dozen hours trying to set up a useful, collaborative folder, they realized that the complexities of sharing in Dropbox were not worth the trouble. Dropbox would not let them efficiently share portions of their codebase, disallowing things like nested shared folders. The timeline of this project was very short, and parallel development with Dropbox was very difficult with everyone writing code at once.

In the end, they decided to use email to share code. This decision was made in order to reduce general administrative overhead and the possibilities of conflicts. But it came at the cost of speed and efficiency.

Version control shouldn't be confined to a small subset of power users in the software industry, as is the case with Git. File sharing shouldn't be obscured with

confusing design concepts, as is the case with Dropbox's shared folder model. Users from any discipline should be able to start an application and intuitively share files and utilize version control in minutes. That is the vision of this paper.

1.1 File Syncing and Version Control Systems

1.1.1 File Syncing Systems

File syncing systems have become popular over the past few years. Systems like Dropbox and Google Docs are simple systems that allow users to backup their files and share files with other users.

These systems are often characterized by how easy it is for users to learn and use them effectively. Some, such as Dropbox, are popular for situations in which concurrent work is not being done because Dropbox does not merge conflicting edits to a file. Others, like Google Docs, do allow concurrent editing, but they do so in a way that can drastically change the file's composition. Sometimes, Google Docs will create new lines and extra white space to accomodate the conflict.

1.1.2 Version Control Systems

Version Control Systems (VCSs) are also common today in certain industries. Systems such as Git and Mercurial are often used by software developers to maintain their projects and share code.

Like file syncing systems, VCSs provide data backup and data sharing. However, they also provide more functionality to help with project management and efficient development. This extra functionality comes at a cost, though. VCSs are often labeled as difficult to learn; many users resign themselves to using a few basic commands and do not explore the full extent of these systems.

1.1.3 Flaws

Due to their overlapping nature, this paper explores both file syncing systems and VCSs, finding benefits and flaws within each set. Those flaws come in two major categories.

The first issue these systems have is they are too narrow in their domain. Rarely do software developers use Dropbox to collaborate on software. Dropbox does not have the functionality required by most complex software projects. The ability to work on independent lines, merge those independent lines, and perform tasks such as grouping and labeling changes are not possible.

It is even rarer that a non-technical industry would use a system like Git for sharing files. Git is a complex system with a huge learning curve, and it's advanced functionality, branching for example, would not be useful for a non-technical team trying to share files. Unfortunately, it is difficult to use Git at a basic level without somewhat exposing yourself to the complexities of its more advanced functions.

In both cases, the system was designed for a specific user group, not for the underlying purposes the users needed the system to fullfill. Technical systems are built for technical people, with a steep learning curve only conquerable by power users. More basic systems are too simple and lack the functionality to support the requirements of more complex projects.

The second main issue these systems have is their design. Even Git, perhaps the most popular, prototypical VCS, suffers from a lack of robustness in its design. Users are often frustrated by Git's complicated and opaque design. Novices, especially, find some of Git's design choices confusing such as its inability to allow a committing of an empty directory [3]. Supposedly simpler file syncing systems like Dropbox have similar design issues. The way that Dropbox's shared folder model operates has left many users confused[6].

A new system is proposed, a VCS that can also serve as a capable file syncing system. We believe that all of the issues associated with current VCSs and file syncing systems can be solved by focusing on the design of our VCS at the conceptual level.

This system leverages the best from the technical and non-technical systems available today. It is designed using essential purposes and concepts in order to make user startup as fast and as easy as possible. It promotes the idea of "opt-in complexity". This ensures that basic users can effectively use the system on day 1 while advanced users can learn the entire system to unlock its full set of features. We hope this system can bridge the gap between technical and non-technical industries.

1.2 Conceptual Design

In the field of software design, there is little agreement concerning how a designer should structure the software they build. Notions of conceptual integrity and concept-based design are nothing new. Leaders in the field of user interface have noted the important of the connection between the mental model that a user has of a piece of software with its underlying software-based concepts.

Conceptual design is a design theory that brings together all of these past, disjointed ideas. It calls for a conceptual model, designed to fulfill a set of purposes [4]. Within the conceptual model is a set of concepts. These concepts represent essential ideas that a system deals with, and their creation and refinement are the central activity of software design.

The designer should be designing the system with these concepts as their vocabulary. Then, the user can use the system with this conceptual model as their mental model of the system. This shared model connects the designer with the user, making it easier to understand. Any system needs an unspoken medium of communication between the designer and the user. Conceptual design gives that medium a language.

A given concept is accompanied by a motivating purpose, its reason for existing. A purpose is a desired result. It is not a piece of code, a design detail, or a way to achieve a desired result. The purpose behind the trash can, for example, on your computer's operating system is to be able to undo file deletions.

There are four properties that concepts must have in order to be strong and viable. Concepts should have **motivation**, meaning they fulfill an articulated purpose. No

two concepts should be **redundant**, or fulfill the same purpose. Concepts should not be **overloaded** and fullfill more than one purpose. Finally, concepts should be **uniform** and, when possible, variant concepts should behave similarly.

Conceptual design was used in this paper to help guide the design of Snapstore from the beginning. We believe that its use will make for an easier mental model of the system for the user. One of the goals of this system, after all, is to be accessible to non-technical users. Conceptual design simplifies this task by pruning unnecessary, complicating concepts and creating simple, purpose-driven ones.