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 should they 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 of 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 Version Control Systems

Version control systems (VCSs) are ubiquitous today, though they come in different forms. Software developers have been using systems like SVN and Git for years for version control. However, there are also file sharing systems such as Dropbox and Google Docs that perform simpler tasks and are geared towards less technical users. While version control systems and file sharing systems are very different, they do fulfill some of the same needs: data storage and data sharing. The version control system, then, is a file sharing system that contains even more power and functionality.

Due to their overlapping nature, this paper compares the two sets of systems, finding benefits and flaws within each set. Each system tends to have their faults, and those faults come in two major categories.

The first issue these systems have is they are too narrow in their domain. Rarely are software developers using Dropbox to collaborate on software. It is even rarer that a non-technical industry, such as law, would use a system like Git for sharing files. In both cases, the system was designed for a specific user group instead of underlying purposes. Technical systems are built for technical people, with a steep learning curve only conquerable by power users. More basic systems are too simple, lacking 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. They donâĂŹt understand so many of its functions that they often resign themselves to using a few basic commands. Novices, especially, find some of Git's design choices confusing such as its inability to allow a committing of an empty directory [2]. Even supposedly simpler systems like Dropbox have similar conceptual issues. Dropbox's shared folder

model has left many users confused [8].

We believe that both of these categories of issues can be solved by focusing on the design of a VCS at the conceptual level. A new system is proposed, one that leverages the best from the technical and non-technical systems available today. It is robustly designed using essential purposes and concepts in order to make the user learnability 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 system more in-depth 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 reduces all of these past, disjointed theories. It calls for a conceptual model, designed to fulfill a set of purposes [5]. 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 built needs an unspoken medium of communication between the designer and the user. Conceptual design gives that medium a singular language.

A given concept is accompanied by a motivating purpose, its reason for existing. A purpose is a desired result. It is not piece of code, a design detail, or a way to achieve

a desired result. The purpose behind the trash can 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.

In [2], 6 purposes associated with VCSs are enumerated.

1. Making a set of changes to a file persistent
2. Represent and record coherent points in history
3. Group logically related changes together
4. Synchronize changes of collaborators
5. Support parallel lines of work
6. Do work in disconnected mode

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.