Chapter 3

Design

The design of Snapstore had two goals. The first was to create a conceptual model composed of concepts that fulfilled the purposes of a version control system. The seven purposes of a version control system identified by de Rosso were used. Then, concrete concepts were defined that fulfilled those purposes in a way that is aligned with conceptual design theory.

The second goal was to prune unnecessary concepts from the design. Excessive concepts would only lead to more cloudy system with a more difficult mental model for the user. The goal of this, and any, system should be simplicity when possible.

3.1 Conceptual Design

The conceptual design of Snapstore follows the theory put forth by Jackson in his work on software design. A graphical representation of the conceptual model is included in the appendix.

3.1.1 Data Storage - Snapshot

Persistent data storage in Snapstore is achieved with the notion of a *snapshot*. The snapshot is a saved state of a file. Snapshots record edits, renames, moves, deletes, and merges. Records of the many operations enable all VCS features of Snapstore.

The *head* snapshot for any given file is the most recent snapshot made for that file and the current content of that file.

All snapshots have parent snapshots and child snapshots. The snapshots of a file are related by the tree they create with these parent/child relationships. This ordering forms the snapshot tree described in the tutorial section. The snapshot tree is guaranteed to be an in-order description of snapshots a specific client has made to a file in a branch. Each unique (branch, file) tuple is represented by its own snapshot tree.

3.1.2 Grouping Changes - Group

A *group* is an assembly of related snapshots. A group must contain at least one snapshot, but there are no restrictions on what kinds of snapshots can be in the group or what their relationship must be. The same snapshot can exist in more than one group. It is up to the user to decide what makes a group of snapshots logically related. This allows flexibility in projects and development strategy.

Groups are an attribute of a specific branch. Even if a group contains the same snapshots across different branches, they are treated as different concepts because they exist on independent lines of development.

3.1.3 Recording Coherent Points - Tag

The notion of a *tag* allows users to label logical milestones in their work. They are similar to a group but provide an added function: they describe the status of the branch as coherent. Here, coherent means that the project is state that is ready for further development or work.

A tag is used to describe a group and give it a level of meaning beyond just a group name. For example, a group can be formed with the title ‘finished version 1.1‘. But, there is no way, using only the title, to compare that group to another group with the title ‘There’s a bug in here somewhere‘. Instead, assigning a pre-made tag of ‘v1.1‘ gives it both a level of importance and a designation that the project is in a suitable state.

Tags are also an attribute of the branch. This means that they must be created inside a of an independent line of development. They can be copied across branches when merging and cloning, but they stay a fundamental attribute of the branch concept.

3.1.4 Support Parallel Lines - Branch

In Snapstore, the concept of a *branch* supports parallel and independent lines of development. These branches are completely separate from each other. The facilitate the appropriate partitioning of data.

The branch houses three of the other main concepts in Snapstore: snapshots, groups, and tags. All three of those concepts exist within the confines of a branch. All of these things together consitute a line of development, and so the branch is the conceptual representation of that line.

Branches make up a repo, whether that repo is local or upstream. Switching between them constitutes changing the line of development, project, folder, or anything that delineates the user's work. Switching between two branches on a single local repo that are stored on different upstream repos has no averse effects due to their independence.

Branches can be merged together, synchronizing the parallel development. This simply involves combining each branches individual snapshot, group, and tag data together.

3.1.5 Synchronize Changes of Collaborators - Upstream Repo

Snapstore uses a centralized data storage system that holds all connected branch data, called an *upstream*. While users do not necessarily have to use an upstream repo for a given branch, it is the only way to collaborate with other users on that branch.

Every branch can have multiple upstream repos. This allows the user to distribute where their snapshots, groups, and tags are saved and backed up. These upstream repos can then be shared with other Snapstore users, provided that the upstream repo is connected to those users.

All changes that occur at the branch level (branches, snapshots, groups, tags) are reflected in any connected upstream repo. There, the upstream can see if any other users have access to the branch and it can push the changes down to them.

3.1.6 Disconnected Work - Local Repo

The ability to leverage the benefits of a version control system without needing an internet or network connection is handled by another kind of repo - the *local repo*. The local repo affords all of the same relationships to other concepts as the upstream repo. That is, it is a collection of branches, which are in turn collections of snapshots, groups, and tags.

The local repo has one other important relationship, it can have zero or more upstreams attached to it. Zero or more upstreams are then mirroring the data housed by the local repo.

3.2 Design Tradeoffs

During the design process, there were many decisions made that had lasting tradeoffs for Snapstore. Two of those tradeoffs are explored below.

3.2.1 Granularity of a Snapshot

The decision of what a snapshot would represent was one of the first design decisions that we encountered. In many systems, the act of saving looks at the entire branch or repository and saves that state. In Git, for example, the act of commit is the act of committing an entire repository branch. There is no concept of a file, and so there is no concept of saving a file.

However, we decided to make the snapshot a file-based object for a few reasons. The first was that it was more intuitive to a typical user. If a user was to save a file and create a snapshot, they would expect that snapshot to relate to the thing they just interacted with, the file. They would not expect it to relate to the entire branch.

The second reason for this choice was our realization for the necessity of an additional concept, the group. If the snapshot pertained to the entire branch of files, we would have to lump snapshots together implicitly. In an effort to decouple concepts that are not essentially bound, we separated the idea of saving changes with grouping those changes. Because of this, we thought it prudent to make snapshots relate to files because now users could group together changes from a specific file over time, without the unnecessary overhead of including the entire project.

The final reason for this change is that it allows us, from a conceptual standpoint, to easily reason about different versions of individual files in the project. Because all snapshots of a file are saved, we can simply find a past version of that file and load it into the branch. This eliminates the need for calculating diffs between project versions and lightens the conceptual load on the user.

By making the snapshot refer to an individual file, we are adding overhead to system by disallowing clever saving strategies such as those used by Git. However, it does not otherwise affect Snapstore, and we can mimic the functionality of Git using this model. In Git, the head is the commit that represents the current state of the branch. In Snapstore, we can easily calculate this by creating a group of all of the head snapshots of all of the files in a branch. The Snapstore model is both more modular and understandable for the user.

3.2.2 Nature of the Upstream

Different version control systems and file sharing systems have their own ways of handling storage. Git, for example, use a decentralized storage system. Dropbox and SVN, on the other hand, use a centralized system. When deciding which model to use for our upstream repo, we looked at both models' pros and cons.

Snapstore uses a centralized upstream repo model, and there are a few reasons why we chose this. The first is that a centralized model is easier for the majority of users to grasp. The popularity of Github, a centralized application using Git, shows this to be true.

Another reason we chose the centralized model is that we are able to leverage some of the benefits of a decentralized model even with a centralized model, due to the rest of our design. One of the main benefits of a decentralized model is that saving changes can be done locally, without anyone else seeing them. This is the case with Snapstore because, even though it is a centralized system that relies on network connection to synchronize, it still creates snapshots while you work offline.

We again used our goal of opt-in complexity to create a balance between the simplicity of a client/server, centralized model with some of the power of distributed systems. In Snapstore, the data on the upstream is indeed 'blessed', and so the model is centralized. However, by offering a local repo alongisde and potentially independent from the upstream repo, Snapstore has some characteristics of a decentralized system.