Chapter 3

Design

The design of Snapstore had two steps. The first was to identify the purposes of a VCS. The second was to create a conceptual model composed of concepts that fulfilled the purposes of a VCS.

3.1 Purposes of Version Control

The six purposes of a VCS identified in [3] were used in the design of Snapstore. A purpose graph is included in Figure 3-1, where each subpurpose points to a purpose. There are five classes of VCS purposes: data management, change management, collaboration, parallel development, and disconnected development.

Data Management deals with the notion of backup. In case of failure, these backups need to be stored persistently and be able to be retreived. This purposes allays risks associated with development such as accidental deletion and incorrect saves, not just machine failure. The ability to track/untrack files to track types of edits to files provides more granular persistent saves for the files.

The second class, **change management**, deals with managing these edits. Grouping changes allows the user to divide the history of a file or files in logical segments. Groups like current file state help users accurately model the state of their project. Tagging these groups as coherent points in development aid in administrative and managerial tasks associated with long or large projects. These coherent points in the

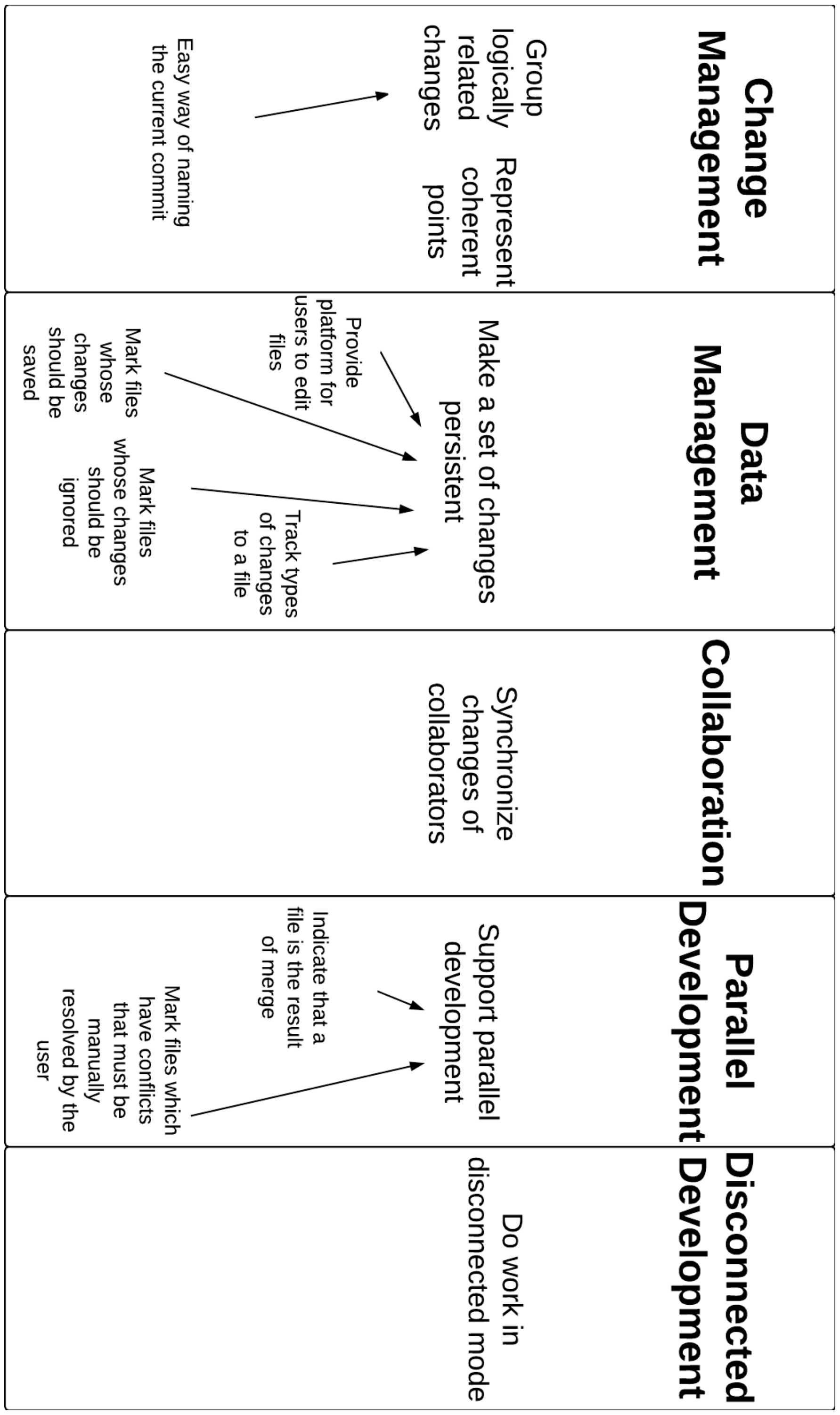
Change Management
Group logically related changes
Represent coherent points
Easy way of naming the current commit
Data Management
Make a set of changes persistent
Provide platform for users to edit files
Mark files whose changes should be saved
Mark files whose changes should be ignored
Track types of changes to a file
Collaboration
Synchronize changes of collaborators
Parallel Development
Support parallel development
Indicate that a file is the result of merge
Mark files which have conflicts that must be manually resolved by the user
Disconnected Development
Do work in disconnected mode

Figure 3-1: Purpose Graph of Version Control Systems.

project can then be returned to by reverting the project to reflect the versions of files in that collection.

Collaboration concerns a system shared by multiple users. Synchronizing the changes of collaborators on that system allows their edits to be amalgamated in such a way that conflicts are avoided when possible and made explicit when they cannot be avoided.

Parallel Development, the fourth class, mainly deals with supporting parallel lines of development. Switching between parallel lines, as well as merging parallel lines are needed to fully support parallel development. This purpose allows users more flexibility to isolate parts of their work from others, and to develop without affecting the main line.

The final class, **Disconnected Development**, allows operations to be performed in a disconnected mode. This motivates any work that a user can do in an offline setting or in a setting where a user is willingly disconnected from their collaborators.

3.2 Conceptual Model

A conceptual model of Snapstore was then created composed of single concepts. These concepts were invented in a way that is aligned with conceptual design theory [4]. They each had a motivating purposes, they were not redundant or overloaded, and they were uniform. A mapping of each Snapstore concept to its motivating purpose is shown in Table 3.1. Table 3.2 shows each Snapstore concept and its operational principle. A graphical representation of the conceptual model, using the notation for extended entity-relationship diagrams as used in [4], is shown in Figure 3-2.

3.2.1 Data Storage — Snapshot

Persistent data storage in Snapstore is achieved with the notion of a *snapshot*. A snapshot is a saved state of a file. Snapshots record updates, renames, moves, deletes, along with merges and conflicts. The *head* snapshot for any given file is the most recent snapshot made for that file and reflects the current content of that file on that

Concept	**Motivating Purpose**
Snapshot	Making a set of changes to a file persistent
Create, Update, Rename, and Delete Snapshot	Track various types of changes to a file
Snapstore Folder	Provide a platform for users to edit files
Tracked File	Mark files whose changes should be saved
Untracked File	Mark files whose changes should be ignored
Group	Group logically related changes together
Head Snapshot	Easy way of naming the current snapshot
Tag	Represent and record coherent points in history
Upstream Repository	Synchronize changes of collaborators
Branch	Support parallel lines of work
Conflict Snapshot	Mark files which have conflicts that must be manually resolved by the user
Merge Snapshot	Indicate that a file is the result of a merge
Local Repository	Do work in disconnected mode

Table 3.1: Concepts of Snapstore and their motivating purposes.

Concept	**Operational Principle**
Snapshot	Whenever a file is saved to disk, a snapshot containing that file's contents is created
Create, Update, Rename, and Delete Snapshot	Whenever a specific type of action on a file results in a disk save, the same type of snapshot is created
Snapstore Folder	If a user edits any files within the Snapstore folder, Snapstore will
Tracked File	When a user tracks a file within the Snapstore folder, snapshots can be created for that file
Untracked File	When a user untracks a file within the Snapstore folder, no snapshots will be created for that file
Group	When a user places a set of snapshots in a group, they can be found later with the group's name
Head Snapshot	Whenever a new snapshot is created for a file, it becomes the head snapshot for that file
Tag	When a user places a tag on a group, that group can be found using the tag's name, and every file within that group can be reverted to it's associated snapshot content within that group at the same time
Upstream Repository	Any change to the local repository made by a user is distributed by the upstream to all collaborators of that user
Branch	When the user switches branches, Snapstore hides all of the data associated with the previous branch and shows them all of the data associated with the current branch
Conflict Snapshot	If there is a conflict when merging two head snapshots, the result is a conflict snapshot, which shows where the conflict is and how to resolve it
Merge Snapshot	If there is no conflict when merging two head snapshots, the result is a merge snapshot
Local Repository	Whenever the user is offline, any changes to Snapstore are saved persistently in the local repository

Table 3.2: Concepts of Snapstore and their operational principles.

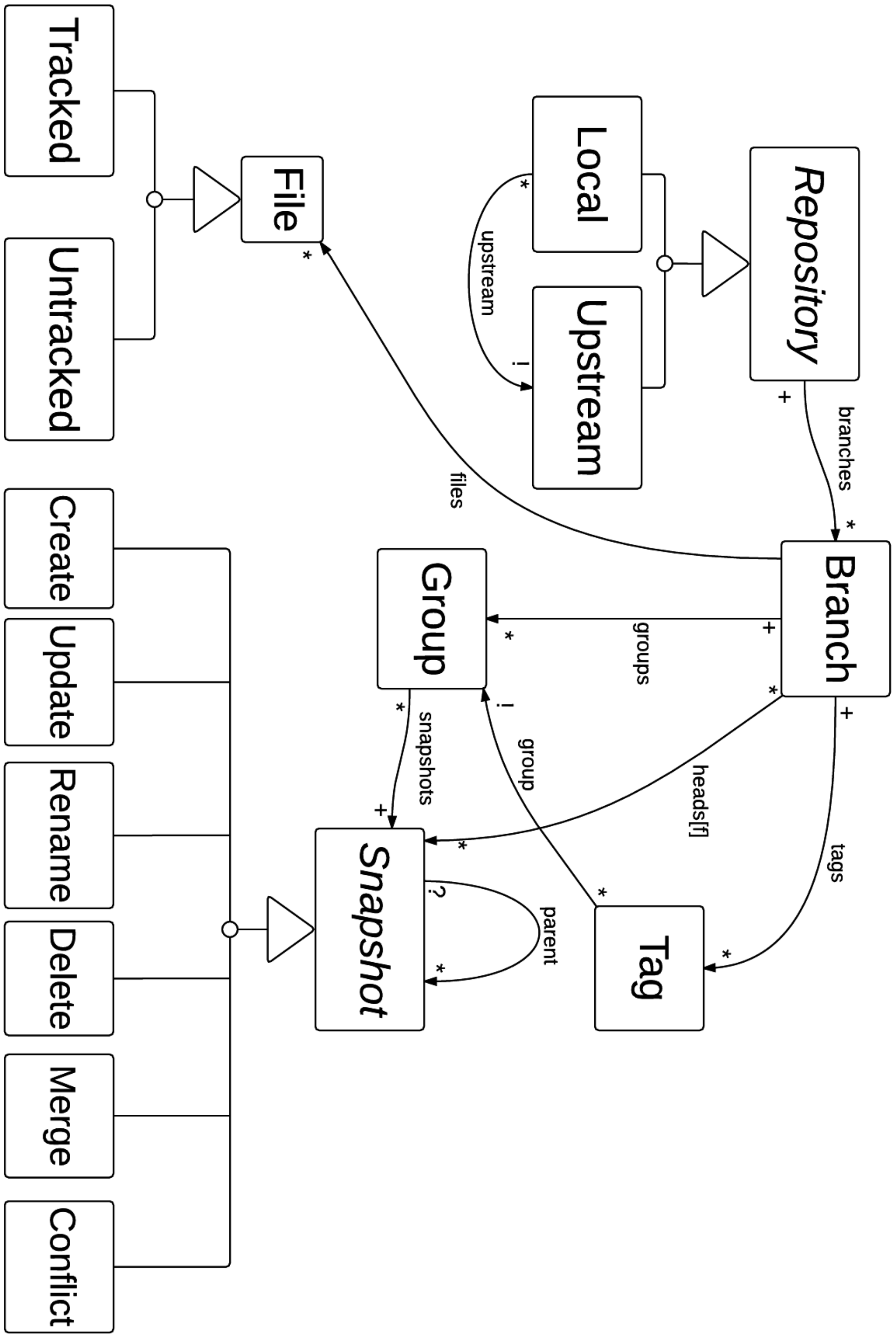

Figure 3-2: Concept Model of Snapstore.

machine.

This type of snapshot dictates the values of that snapshot's attributes. Create snapshots have no parent. Update snapshots have a parent, a child, and content. Rename snapshots have a different filename than their parent. Delete snapshots have no content, though they still have a parent and can therefore be placed in the snapshot graph. Merge snapshots have more than one parent, and conflict snapshots are merge snapshots that have conflict markers in their data.

All snapshots have parent snapshots and child snapshots. The snapshots of a file are related by the graph they create with these parent/child relationships. This ordering forms the snapshot graph described in section 2.1.1. Each unique (branch, file) tuple is represented by its own snapshot graph. A snapshot can have multiple parents if it is the result of a merge operation.

The snapshot graph is guaranteed to be an in-order description of snapshots a specific client has made to a file in a branch. There is no operation on the client that distorts the ordering of snapshots in the graph. The only operation that can alter the graph occurs when a snapshot graph is being shared and a collaborator inserts their snapshots somewhere in the graph. However, even if snapshots are inserted into a user's snapshot graph, the user's ordering of snapshots made by them stays intact.

Any file, identified by its snapshot graph, can either be tracked or untracked. Untracked files will not create snapshots and will not affect the local or upstream repositories.

3.2.2 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 two group contain the same snapshots across different branches, those groups are different because they exist on different lines of development. Any group can be given a name for identification for the user.

3.2.3 Recording Coherent Points — Tag

The notion of a *tag* allows users to label logical milestones in their work. They describe groups but have an added function over a group's name: they describe the status of the group as representing a coherent point. Here, coherent means that the project is in a state that is ready for further development or work, though the definition will differ from team to team[3].

Tags will always describe groups that are perfectly vertical. That is, at most one snapshot from any file is in the group. An example of this is tagging every head snapshot in a branch at a given time with the tag "Submitted to Scientific Journal" or "Version 1.0".

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.

3.2.4 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. They 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 repository, whether that repository 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 repository that are stored on different upstreams has no averse effects due to their independence.

Branches can also be shared between multiple users. User on a shared branch can work independently, confident that any changes they make will not be lost. These changes, whether they deal with snapshots, groups, or tags, are be persistent. New changes can come in through the network while a user is working, but it won't affect their ability to send their own changes.

Branches can be merged together, synchronizing the parallel development. This simply involves combining each branch's individual snapshot, group, and tag data together as explained in chapter 2.2.3.

3.2.5 Synchronize Changes of Collaborators — Upstream Repository

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

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

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

3.2.6 Disconnected Work — Local Repository

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

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

3.2.7 Discussion

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

Granularity of a Snapshot

The decision of what a snapshot would represent was the first design decision we encountered. Either a snapshot could represent a file, or it could represent every file in a branch. In many VCSs such as Git and Mercurial, saving changes couples together the act of saving changes with the act of grouping changes, resulting in an overloaded concept [4]. We separated the saving of changes with grouping those changes, so the snapshot represents a single file.

Another reason the snapshot describes only a single file 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 object they just interacted with, that file. They would not expect it to relate to every file in the branch.

One downside of this approach is the additional computation needed to compute the current state of a branch. In Git, the current state is the head commit. In Snapstore, we need to calculate this by grabbing all of the head snapshots for a branch. This is typically trivial, so the tradeoff is beneficial.

The Upstream

Different VCSs 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, we looked at both models' pros and cons.

Snapstore uses a hybrid centralized/decentralized upstream model. On the surface, it is centralzied because all collaboration must take place via the upstream repository. That is, any data a user wants to collaborate on with another user must first go through a centralized repository.

But, Snapstore is also decentralized because users have a local repository, where actions can be made without requiring an active connection to the upstream. Snapstore users each have an entire history of their branch on their own system, just like in a decentralized VCS. Because of this, users can work offline, without checking out a central repository.

There are downsides to this hybrid model. Initially, there is only one upstream server, and so the location of the data is held by a single entity. If users want to set up their own server, they must use a machine to do so, and there is the overhead of setting up that server. Because there is no push/pull model in Snapstore, this machine must always be online in order to facilitate collaboration. Further, users cannot share directly with subsets of users on a given branch. They can, however, work around this limitation by creating new branches with new sets of users.

We 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 a decentralized system. The centralized model is easier for the majority of users to use, and it has a faster learning curve [1]. In Snapstore, the data on the upstream is indeed "blessed", and so it is centralized. However, by offering a local repository alongisde and potentially independent from the upstream, Snapstore has some characteristics of a decentralized system.

File Names

Whether or not to make the file name a property of a file or the identifier for a file was an important decision for Snapstore. Systems like Git use the file name as an identifier for a file. Because of this decision, renames to a file are sometimes processed as deleting that file and creating a new file, causing much consernation among users, especially novices[3].

We wanted to be able to fully support renaming files in Snapstore, so file names in Snapstore are simply a property of the file. Each snapshot in a given snapshot graph will have the same file id. When branch merging occurs, only snapshot graphs with the same file id will be merged. The merge operation will look back over a file's two snapshot graphs for a common ancestor and perform a standard three-way merge. This allows Snapstore to accurately handle renames and merges.