DVCS Design Analysis

Jiao Mingyang, Simon Budker, Aeshaan Wahlang, Wei Zhang

**Overview**

Before deciding to pursue this DVCS design we weighed several potential options. In this document we try to give background for the design we believe is best for our constraints.

## **Module Structure**

## A: Top Level Decomposition

This DVCS system includes two major modules.

### A:1 Behavior Hiding Module

This module includes programs that deal with how the DVCS system works at a high level based on user commands. More specifically, these programs allow for the documented functionality to work. This means the main secret of this module is the translation of basic text commands into some high level software orchestrator and the orchestrator itself. This set of programs will need to change if new commands or behavior are added or changed. For instance, if a new command like "log-x" was to be added to the requirement this module would need to be updated to support that change.

### A:2 Software Decision Module

This module hides software decisions that focus on programming considerations like hardware limitations and algorithm efficiency. The secrets of this module are the underlying data structures and their interaction with the OS and actual files.

### B: Second Level Decomposition

### B:1 Behavior Hiding Module Decomposition

This decomposition contains two modules described below.

### B:1.1 User Interface Module

The module must be able to accept user input and translate it into commands for a high level software orchestrator (the repository module). This module should be simple and act almost as a one to one map from user text to repository behavior.

The commands this module must support are:

* init
* clone
* add
* remove
* status
* heads
* diff
* checkout
* cat
* log
* merge
* pull
* push

When the module receives one of these arguments it will use the repository module to actually perform the command. In addition, this module is responsible for providing some user specific response messages. In short, this module is responsible for handling what the user does with the system and what the user sees from the system.

### B:1.2 Repository Module

This module is extremely important and acts as the main command module. It serves to orchestrate what is happening within the software decision module. More clearly, the repository module is responsible for actually coordinating what should happen in the abstract decision module.

This module hides how the majority of the system is connected. It is the bridge between the users’ understanding of the system and most of the actual software implementation.

While this module will undoubtedly have some software considerations, it is grouped into the high level behavior hiding module because it will surely need to change if the behavior changes. Specifically, since this module coordinates and makes behavior happen, if behavior changes, the hidden secrets in this module will also need to change.

One down side to this module decomposition is that it is difficult to write independently. Since this module needs to know a lot about the interfaces of modules in the high level software decision module it will be hard to write unless things in that module are well specified and documented.

On the other hand, because this module will link a lot of the system together it will be extremely compressible since it should closely follow the intuition of how the general system works.

### B:2 Software Decision Module Decomposition

This decomposition contains four main module that are discussed below.

### B:2.1 Changeset log module

This module serves to support the concept of change history by maintaining a log of change sets and represents revisions of the repository. It hides everything about how the log is stored or actually manipulated. This module contains many decisions that relate to efficiency.

### B:2.2 Manifest module

The Manifest module contains all work related to maintaining and interacting with manifests which are file contents of the repository at a particular changeset.

### B:2.3 File log module

The filelog module is a module to maintain all version of a specific file and the changes made to each file.

### B:2.4 Directory state module

This module hides the maintenance and status of the working directory. For instance it must track currently checked out revisions, copied/renamed files, and system specific files that have system specific information.

### B.2.5 RevLog module

The revlog module is the collection of work to create the main data structure supporting logs in the system. The changeset log, manifest, and file log modules all use this module. This module hides how versions of files are stored. Moreover, the module is responsible for storing in a space efficient manner. In addition this module must support file level differences that higher level modules need.

This module has a significant amount of work involved, but provides great support for system changeability. Specifically, because it hides everything about how file versions are stored, and much of how the system relies on file storage, it only requires changes in one module.

Not only does separating this module help a lot with changeability, but it also helps to allow for added comprehensibility and independent development. The module is more comprehensible because it is an abstract data type. It does not need to know about how the entire system works or much about how it will be used. This fact also means that the module is easier to understand since it can be separated from the rest of the system.

### B.2.6 Diff module

This module serves to separate diff manipulation. This module aims to support comprehensibility by hiding code that would obfuscate pieces of the larger DVCS system. Again, this separation also helps for team work since it is distinctly separate from other modules.

**Testing Strategy**

Since we have several very different modules, we will employ various testing methods to ensure the functionality and quality of our software. The methods we will use will depend on the module and its integration into our system. Below we outline what these testing methods are.

**Unit Testing**: We will isolate and test complex methods individually. To do this we will call our complex methods and assert we get the expected values.

**Integration Testing**: We will design diversified integration test plans for modules in our DVCS system. Using top-down or bottom-up integration testing approaches we will combine individual modules and test them as a group in order to see if modules can work together properly. For example, combining User Interface Module and Repository Module together as a group, User Interface Module takes an input that have passed the unit test, and then check the output of the Repository Module.

**End to End Testing**: We will test the DVCS system as if we were real world users. I In simpler terms, we will use the version control system to manage the change of documents or computer program projects.

**Module Testing**

User Interface Module - B.1.1

Addresses requirements (9)

* Unit tests to ensure proper output to user
* Integration testing to ensure proper connection to repository module

Repository Module - B1.2

Addresses requirements (1-8)

* Unit tests will be difficult for this module but may still provide value if functions become complex
* Integration tests will be the main place where we can ensure quality because this orchestrator is so connected to other modules
* End to end tests will be essential because this is the software orchestrator of the entire system

ChangeLog Module - B2.1

Addresses requirements: (1,2,5,8)

* Unit test to cover all functions and methods and ensure that they perform as expected and return the right values
* Integration tests will ensure we can use the ChangeLog in our system

ManifestLog Module - B2.2

Addresses requirements: (1,2,3,4,6)

* Unit tests will make sure we can do manifest version manipulations properly
* Integration tests will ensure we can use the ManifestLog in our system

FileLog Module - B2.3

Addresses requirements: (1,2,3)

* Unit tests will make sure we can do file version manipulations properly
* Integration tests will ensure we can use the FileLog in our system

Directory State Module - B2.4

Addresses requirement: (6)

* Unit test will cover writing and reading from file
* Integration test will make sure that our system can be built from files

RevLog Module - B2.5

Addresses requirement: (1,2,8)

* Unit tests will be essential to ensure that our key tree base data structure works as expected
* Integration tests will also be useful in ensuring all extend use relations work properly

Diff Module - B2.6

Addresses requirement: (6,7)

* Unit tests will play a major role in this module to make sure our complex functions like file diff actually return proper diff in the form we explicitly define ahead of time