DVCS Architecture

Jiao Mingyang, Simon Budker, Aeshaan Wahlang, Wei Zhang

**Overview**

In this document we breakdown our design of a DVCS system. We approach this by first giving an overview of how we imagine our DVCS functioning. Next, we break our system in distinct modules discussing the tradeoffs of our modularization. We also do some surface level thought about implementing these modules by including interfaces for each module. Last, we try to wrap our design together by walking through a few scenarios end to end.

**System Overview**

In order to meet the requirements specified, we have structured our system based on three types of tree data structure. In the first tree, ChangeLog, each node represents a ChangeSet which is a revision of the repository. When a new revision is created, a new ChangeSet is added to the tree as a child of the previous ChangeSet each of which have unique hash identifications. Since all ChangeSets except for the root revision will have at least one parent the chronological order of revisions is maintained. The second tree, ManifestLog, has nodes which represent a manifest and is associated with at least one ChangeSet. The Manifest contains the names of the files in that ChangeSet and their versions. Each Manifest is associated with at least one Changeset and are similarly identified with unique hashes. The third tree, FileLog, contains changes for a particular file in its nodes. These changes are with respect to the parent node. They represent a file version and are also uniquely identified with hashes. The tree structure maintains the order of the sequence of changes and allows for any file version to be reconstructed. Interestingly, since ChangeSets, Manifests, and File version have a tree structure we will follow the builder pattern and support all three trees with an abstract tree called a RevLog.

|  |
| --- |
|  |
| Version Control Simple Scenario with System Design Applied  Note: Numbers represent hashes |

**Module Overview**

|  |
| --- |
|  |
| Module Decomposition Diagram |

1. Behavior Hiding Module
   1. Repository Module
      1. init(directory)→Repository   
         Build an empty repository.
      2. clone(address, directory)→Repository   
         Clone the repository from the given address.
      3. add(self,ﬁle name)   
         Add a new ﬁle to the repository or update an existing ﬁle.
      4. remove(self,ﬁle name)   
         Stop tracking a ﬁle and remove it from the current snapshot.
      5. status(self)   
         Report the changes since the last commit.
      6. heads(self)→ List   
         All the head revisions in the repository.
      7. diﬀ(self[, ﬁle, revision1, revision2])   
         Print the diﬀerences between two revisions.
      8. checkout(self, revision num)   
         Switch to a revision according to the revision num.
      9. cat(self, ﬁle, revision num)   
         Output the ﬁle under a revision.
      10. commit(self, commit message)   
          Commit the changes with the given commit message. Build a new revision and Switch to it.
      11. log(self)   
          Print the history of the repository.
      12. merge(self, revision num1, revision num2)   
          Merge two revisions.
      13. pull(self, address)→Repository   
          Pull the changes from another repository.
      14. push(self, address)   
          Push the changes to another repository.
   2. User Interface Module
      1. main(command)  
         Call the function corresponding to the command.
   3. Changeset Log Module
      1. init→Changelog   
         Create a new Changelog.
      2. clone→Changelog   
         Clone this change log and return it.
      3. add(self, ﬁle name, begin position, data)→ true or false   
         Record adding a sequence of data to a ﬁle.
      4. erase(self, ﬁle name, begin position, length)→ true or false Record removing a sequence of data from a ﬁle.
      5. apply(self, ﬁle name, ﬁle)→ﬁle   
         Apply all the changes of a ﬁle stored in this revision to the given ﬁle.
      6. reverse(self, ﬁle)→ﬁle   
         Undo the changes happens in this reversion to the ﬁle and return the result.
      7. iter\_over\_manifest(self, function)   
         Internal iterator over all the pairs in the manifest. Directly call the each function of Manifest.
   4. Manifest Module
      1. init([[ﬁle name, revision num]\*])→Manifest   
         Create an Manifest
      2. checkout   
         Checkout the snapshot represented by this Manifest
      3. each(self, function)   
         Internal iterator over all the pairs of ﬁle name and revision number.
   5. RevLog Module
      1. init()→Revlog   
         Create an empty Revlog for a ﬁle.
      2. clone(self)→Revlog   
         Clone the Revlog and return it.
      3. add ﬁle(ﬁle)   
         Add a new ﬁle to the Revlog start with the current revision.
      4. add(self, Changelog)→ true or false   
         Add a Changelog to the Revlog of this ﬁle.
      5. erase(self, Changelog)→ true or false   
         Remove a Changelog from the Revlog of this ﬁle.
      6. apply(self, ﬁle[, revision num])→ﬁle   
         Apply the changes [in a revision] to the ﬁle and return the result.
      7. ancestors(self, revision num)→[revision num]\*   
         Find all the ancestors of the revision speciﬁed.
      8. heads→revision\_num   
         Return all the heads.
   6. Directory State Module
      1. read(File)→String

Read a file and return the content

* + 1. write(File, text)

Write into a file

* + 1. get\_repo\_folder→Dir  
       Return the folder of the repository
    2. get\_revision→revision\_num  
       Get the number of current revision.
    3. size(File)→size  
       Get the size of the File.
    4. state(File)→mode

Get the state of the File. The state can be n - normal, a - added, r - removed, m - 3-way merged.

* 1. File Log Module
     1. init()→File\_Log

Init a file log for a new file.

* + 1. apply(id,File)  
       Apply the changes recorded in the log with the index as id to the given File.
    2. find(id)→Node

Find the change by the id.

* 1. Diff Module
     1. linesplit(line)→Array  
        Split text to an array of lines.
     2. textdiff(texta, textb)  
        Return differences between two texts.
     3. diff(texta, textb)

Return the differences between two lists of lines.

* + 1. patch(a, bin)

Split the text into patches with length of 12.

**Module Structure Use Relationships**

The sections aims to explain and justify each use relationship between the modules. Each use relationship is evaluated against Parnas' criteria of module decomposition. It is important to note that because of the relative size of this project many module structures could work. As Parnas specifies in his writings, his software design approaches mainly apply to large scale projects with many thousands of lines. Since this is a relatively small project, many of these design breakdowns are not entirely necessary.

|  |
| --- |
|  |
| Module Use Relationship Diagram |

### **Relationship Justification**

Overall this system design and its use relationships are justified when applied against Parnas' four criteria of use. While some modules are not perfect under the criteria, the systems challenges make other design decisions less optimal.

#### **User Interface Module**

The user interface module is very independent. It relies on a single use relationship through the repository module. As discussed more thoroughly above this module acts as the layer between the DVCS and the user. This means all valid user input and output messages are determined by the module and are therefore extremely well defined. This also means that as long as this module's dependency behaves as specified this module can be developed completely independently from the rest of the system. Again, this separation of mainly user interface related functionality means the module can be tested on its own. Lastly, this module contains little system logic so errors should be very distinct and traceable.

### **Repository Module**

The repository module is one of the most complex modules in the DVCS design. It acts a software orchestrator which means it depends on several modules. It is distinct enough from the other modules, however, to be tested and developed on its own. In addition, because it is a high level controller it has well defined inputs only coming from the User Interface Module. The module outputs are slightly more complex since they depend on its data dependencies. The one place where this module really struggles according to the criteria is in tracing errors. These errors could be generated from some of the dependencies which means it makes is hard to trace.

### **Directory State Module**

This module is responsible for maintaining system files and the working directory. It is pretty independent and has no use relationships with other modules. Its relative isolation means that input and outputs are well defined, it can be developed on its own, tested on its own, and errors will be easy to trace.

### **Changeset Log/Manifest/File log Modules**

These three modules each support a type of revision log for their particular type of data. Each of these three files has one extends use relationship which means they are relatively independent modules. That means the inputs and outputs are well defined, they can be tested and developed independently, and errors should be contained.

### **Revision Module**

This module excels under the four criteria of module decomposition. It defines a fundamental data structure which is used by several modules. It has only a single use relationship which exists to separate distinct secrets. All in all, this module is structured in a way that allows for well defined input and outputs, independent testing and development, and errors that are easy to trace. Moreover, the existence of this module significantly benefits the entire system in terms of the DRY coding principle. By distilling all main logs into this fundamental data structure we are able to change all log structures by only changing code in the revision module.

### **Diff Module**

One of the lowest level modules is the diff module which is used by the revision module. The reason this module is beneficial is because it hides very different secrets than the revision module. This module aims to hide work involved in working with files and getting diffs between files. These types of secrets are good to separate from a data structure design. Having these secrets (i.e this module) separate allows for a collection of programs that have well-defined behavior with clear input and outputs. Its separation also allows for independent development and testing. Finally, its separation allows for easily traceable errors.