

---

---

# Distributed Version Control System

- Specification and Implementation -

---

---

CSC453 Project Report  
Group Four

University of Rochester  
Computer Science

Copyright © University of Rochester 2018

The Distributed Version Control System (DVCS) is developed in Ruby-2.5.1 and test cases are provided in Bash script.



**Computer Science**  
University of Rochester  
<https://www.cs.rochester.edu/>

**Title:**

Distributed Version Control System -  
Specification and Implementation

**Theme:**

Software Development

**Project Period:**

Fall Semester 2018

**Project Group:**

Group Four

**Participant(s):**

Fengxiang Lan (team leader)

Hao Huang

Jing Shi

**Supervisor(s):**

Chen Ding

**Copies:** 1

**Page Numbers:** 27

**Date of Completion:**

November 22, 2018

**Abstract:**

Mercurial is a cross-platform distributed version control system (DVCS). It is developed in Python. Based on Mercurial-0.1 version, we develop a Ruby version distributed version control system. It includes most of functionality of Mercurial-0.1, e.g., add, delete, merge, commit, etc. Our DVCS can be divided into two levels of modules. The top-level are five modules and each of them corresponds to a Ruby class. The bottom level of modules are functions contained in the top-level modules. Our DVCS also provides a command-line style user interface to handle users' commands. In this report, we provide a thorough description of our DVCS, including system functionality, module specification, function design, test cases and development process.

*The content of this report is freely available, but publication (with reference) may only be pursued due to agreement with the author.*



# Contents

<b>Preface</b>	<b>vii</b>
<b>1 Specification</b>	<b>1</b>
1.1 Software Functionality . . . . .	1
1.2 Functionality Specification . . . . .	2
<b>2 Module Analysis</b>	<b>5</b>
2.1 Class Diagram . . . . .	5
2.2 Module Analysis . . . . .	6
2.2.1 Top-level Modules . . . . .	6
2.2.2 Bottom-level Modules . . . . .	6
<b>3 Test</b>	<b>13</b>
3.1 Unit Test . . . . .	13
3.1.1 Revlog . . . . .	13
3.1.2 Manifest . . . . .	14
3.1.3 Changelog . . . . .	14
3.1.4 Repository . . . . .	15
3.1.5 mdiff . . . . .	16
3.2 Integration Test . . . . .	16
<b>4 Development Phase</b>	<b>21</b>
4.1 Schedule Plan . . . . .	21
4.2 Development Record . . . . .	22
<b>5 Conclusion</b>	<b>27</b>



# Preface

Here is the preface. You should put your signatures at the end of the preface.

University of Rochester, November 22, 2018



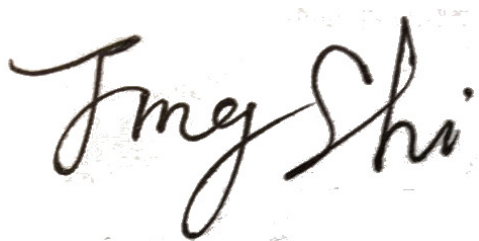
---

Fengxiang Lan (team leader)  
<flan@ur.rochester.edu>



---

Hao Huang  
<hhuang40@ur.rochester.edu>



---

Jing Shi  
<jshi31@ur.rochester.edu>





# Chapter 1

## Specification

The Distributed Version Control System (DVCS) is a file version control program written in Ruby language. It is developed mainly based on Mercurial-0.1 which is written in Python.

### 1.1 Software Functionality

The DVCS supports most commands contained in Mercurial-0.1. A full list of supported commands are listed in Table 1.1.

Number	Command	Description
1	init	Create an empty repository
2	add	Add specific files to be tracked
3	delete	Remove specific files from tracking list
4	clone	Copy an existing repository
5	stat	Check the current status of the repository
6	heads	Display the current heads
7	diff	Check changes between different revisions
8	checkout	Recover a specific revision
9	cat	Inspect a file of a given revision
10	commit	Commit the current changes
11	log	View commit history
12	merge	Merge two repositories

**Table 1.1:** All user commands supported by the DVCS.

## 1.2 Functionality Specification

Each command listed in Tabel 1.1 is further specified in this subsection. Arguments in “[]” are optional.

### **init**

Create a new empty repository in the current directory. If the current directory is already a (part of) repository, this command will fail. This command should be executed before all other commands.

### **add <files>**

Add the specified file or files to a list for the next commit. All arguments should be delimited by one or more spaces.

### **delete <files>**

Delete the specified file or files from a list for for the next commit. All arguments should be delimited by one or more spaces. This command will use the working directory as the default path to search for files.

### **clone <dir\_1> [,<dir\_2>]**

Make a copy of repository in dir\_1 to the dir\_2 directory. If the second argument is not given, copy the repository in current directory to dir\_1.

### **stat**

Display the status of all files in the current repository. “A” stands for the all un-tracked files. “C” denotes the tracked file that are changed but not yet commit. “D” represents the tracked file that deleted in working directory.

### **heads**

Display the heads which are descendants of the latest revision.

### **diff <rev\_1>, <rev\_2>**

Display differences of all files content between the two given revisions. Two valid revision numbers are required.

**checkout <rev>**

Restore the working directory to the specified revision. One argument is required. If the provided revision number is invalid, the command will fail.

**cat <file>, <rev>**

Display the content of the file of the specified revision. If the provided file or revision number is invalid, the command will fail. This command will use the working directory as the default path to search for files.

**commit**

Commit all staged files as a new revision which will become the parent of the working directory. Each commit will generate a new revision number starting from 0 sequentially along with a unique global id. The sequential revision number will be used to list history.

**log**

View revision history of all files in the current repository.

**merge <dir>**

Merge the repository specified by the directory argument to the current repository. One argument is required.

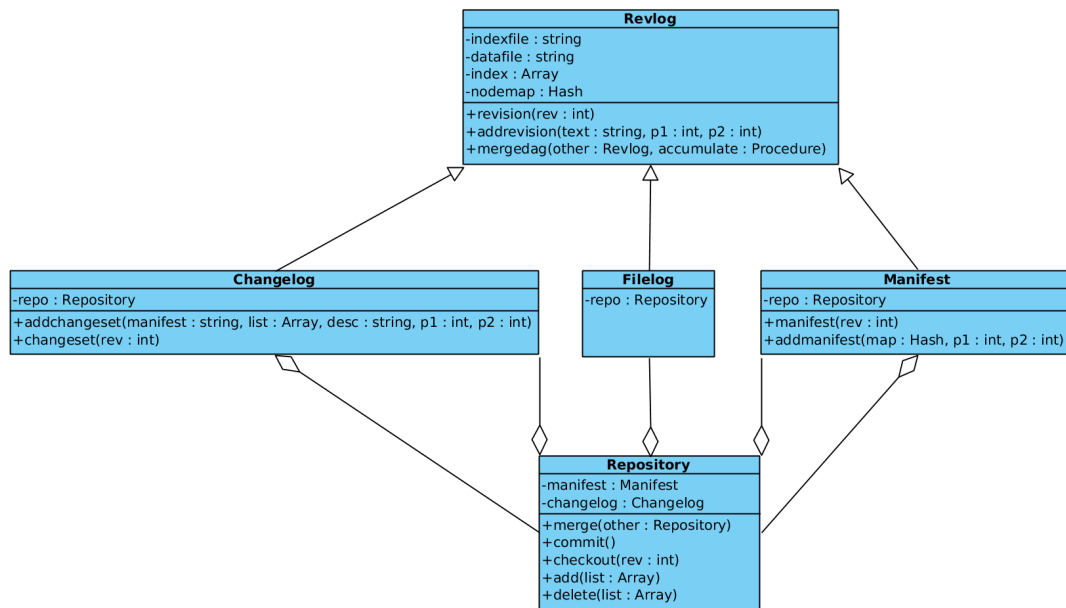


## Chapter 2

# Module Analysis

### 2.1 Class Diagram

“Revlog” is the base class for “Changelog”, “Filelog” and “Manifest”. “Repository” is aggregated in “Changelog”, “Filelog” and “Manifest”. Meanwhile, “Changelog” and “Manifest” are also aggregated in “Repository”. Their relations are illustrated in Figure 2.1.



**Figure 2.1:** UML class diagram of the DVCS. Some important attributes and member functions are also listed within each class.

## 2.2 Module Analysis

This part we will introduce the two-level module architecture of our DVCS. Firstly, a conceptual description of the top-level module will be given; then a detailed specification of each bottom-level module (function) will be described.

### 2.2.1 Top-level Modules

The DVCS has five top-level modules which are all listed in Table 2.1. Each module is a Ruby class and defined in a file. They correspond to the UML class diagram presented in Figure 2.1.

Name	Description	File
Repository	Accept the users' command and maintain a repository	repository.rb
Revlog	Track the history of each revision	revlog.rb
Manifest	Store the mapping from file to revision node id	manifest.rb
Changelog	Store history of user commit information	changelog.rb
Filelog	Add file revision information to the repository	filelog.rb

**Table 2.1:** All top level modules' names, descriptions and corresponding files

### 2.2.2 Bottom-level Modules

For the seek of saving space, we only include functions that have a direct relation with top-level module's functionality.

#### 1. Repository

This top-level module is to handle users' commands and maintains a repository. It includes all the basic data structure of a repository. The second-level modules of this top-level module are functions to execute these commands.

##### (a) initialize (path, create):

**Purpose:** to create a new repository.

**Arguments:** "path" is the path where the repository will be at; "create" means whether the user will create a new repository or not.

**Operation:** If the current directory is not a part of a repository, then create a new one with a manifest and a changelog for this repository. It will also create a data file and an index file to store the information of this repository.

**Return:** boolean. If a new valid repository is created successfully, returns true; otherwise returns false.

- (b) **add (list):**
  - Purpose:** to add a list of specified files for the next commit.
  - Arguments:** “list” is a list of files to be added to the repository.
  - Operation:** Add a list of files for the next commit (but not commit them yet).
  - Return:** nil. If the file does not exist, throw an error.
- (c) **delete (list):**
  - Purpose:** to delete a list of specified files.
  - Arguments:** “list” is a list of files to be deleted from the repository.
  - Operation:** Find the specified staged files and add them to a “to-delete” list.
  - Return:** nil. If the specified files are not found, throw an error.
- (d) **checkout (rev):**
  - Purpose:** to restore the current repository to the specified reversion.
  - Argument:** “rev” is a revision number.
  - Operation:** Restore the repository to the specified reversion. And use manifest, changelog and filelog to restore files contents.
  - Return:** nil. If the provided reversion number does not exist, throw an error.
- (e) **commit:**
  - Purpose:** to commit the changes, e.g., add and delete.
  - Argument:** None.
  - Operation:** Commit the file according to the “to-add” and “to-delete” lists. Manifest, changelog and filelog will also be modified and a new revision will be created.
  - Return:** nil. If no file is staged, nothing will happen.
- (f) **clone (org\_dir, new\_dir):**
  - Purpose:** to clone the repository specified by “org\_dir” to the current directory specified by “new\_dir”.
  - Arguments:** “org\_dir” is the repository to be cloned; “new\_dir” is the destination directory.
  - Operation:** Call system function to clone the specified repository.
  - Return:** nil. If “org\_dir” does not exist, throw an error.
- (g) **diff (rev1, rev2, f):**
  - Purpose:** to check changes between the two specified revisions.
  - Argument:** “rev1” and “rev2” are two revision numbers; “f” is the filename to be compared.
  - Operation:** Check the manifest and changelog of the two repository revisions and find out differences between them.
  - Return:** string. If either of the revision numbers is invalid, throw an

error.

(h) **diffdir (path):**

**Purpose:** Find differences between files in the specified path with files tracked in the current repository revision.

**Argument:** "path" is the path to be compared.

**Operation:** "A" stands for the all untracked files. "C" denotes the tracked file that are changed but not yet commit. "D" represents the tracked file that deleted in working directory.

**Return:** nil.

(i) **merge (other):**

**Purpose:** to merge two repositories.

**Argument:** "other" is another repository to be merged into the current repository.

**Operation:** It finds the nearest common parent and then starts merging manifests from this parent. If there are differences, it will modify changelog and continue merging.

**Return:** nil.

## 2. Revlog

This top-level module is used to track the history of each revision. This module is core part of the whole system, and it will not use any other top-level module.

(a) **initialize (indexfile, datafile):**

**Purpose:** to create a Revlog instance with the specified index file and data file.

**Arguments:** "indexfile" is the filename of the index file which the Revlog points to; "datafile" is the filename of the data file which the Revlog points to.

**Operation:** create a new Revlog with the indexfile and datafile which are used to record the index and data information.

**Return:** nil. If indexfile or datafile does not exist, it will throw an error.

(b) **ancestor (rev1, rev2):**

**Purpose:** to return the ancestor of the two specified revisions.

**Arguments:** "rev1" is the one revision number; "rev2" is the another revision number.

**Operation:** Find out the nearest ancestor of two revisions.

**Return:** revision number of the ancestor.

(c) **revision (rev):**

**Purpose:** to return the content of files under the specified revision.

**Arguments:** "rev" is the number of a revision.



**Operation:** Find the node of the specified revision and then decodes the node to retrieve the content of this revision.

**Return:** string. If no such revision, throw an error.

(d) **addrevision (text, p1, p2):**

**Purpose:** to add a new revision to the current repository.

**Arguments:** “text” is the content of a revision; “p1” and “p2” are two revisions.

**Operation:** Find the tip revision and add “text” as a new revision; write this revision information into indexfile and datafile.

**Return:** revision number.

### 3. Manifest

This module is applied to store the mapping between each single file and its committed revision id. When a new manifest is created, file information must be maintained by a Revlog.

(a) **initialize (repo):**

**Purpose:** to create a manifest for a repository in the given path.

**Arguments:** “repo” is the name of the repository which the manifest points to.

**Operation:** create a new manifest with the index file and data file, record these two files to Revlog.

**Return:** nil.

(b) **manifest (rev):**

**Purpose:** to parse the mapping relation between each file and its revision id.

**Arguments:** “rev” is the number of the revision we want to see.

**Operation:** call *revision* to get the encrypted relation between file and its version number and then parse it to Hash data structure.

**Return:** map which includes contents of manifest under the given revision.

(c) **addmanifest (map, p1, p2):**

**Purpose:** to pass the encrypted code standing for filename, and parent to addrevision.

**Arguments:** “map” is the Hash map from filename to its revision hash id. “p1” is the first parent’s revision id and “p2” is the second parent’s revision id.

**Operation:** Pass the encrypted code standing for filename, and parent to addrevision.

**Return:** a new revision number.

#### 4. Changelog

Changelog records auxiliary information of each modification, such as the user, date, modification operation of such modification.

(a) **initialize (repo):**

**Purpose:** to create a changelog for a repository in the given path.

**Arguments:** “repo” is the name of the repository which the manifest points to.

**Operation:** Construct a Revlog instance with changelog-style input.

**Return:** nil.

(b) **extract (text):**

**Purpose:** to extract the auxiliary change information from the encrypted string to a hash map.

**Arguments:** “text” is the encrypted text of file name, revision node, user, data, operation of an modification.

**Operation:** parse the auxiliary change information from from an encrypted string into a comprehensible array

**Return:** an array of revision hash id, user, data, files and operation.

(c) **changeset (rev):**

**Purpose:** to get the auxiliary change information from a certain revision.

**Arguments:** “rev” is a revision number.

**Operation:** Receive the encrypted auxiliary change information form revision and then call extract to get a readable structure of the auxiliary change information.

**Return:** an array of change information.

(d) **addchangeset (manifest, list, desc, p1, p2)**

**Purpose:** to add up user, data and operation to the revision information.

**Arguments:** “manifest” is string representing the hash node of a revision; “list” is a list of file names that is committed; “desc” is a string indicating the operation of the modification; “p1” is the first parent’s revision number; “p2” is the second parent’s revision number.

**Operation:** Grab the date and user information of the new revision then call *addrevision* to record such information.

**Return:** a revision number.

#### 5. Filelog

Filelog stores the change-part of the content in each revision.

(a) **initialize (repo, path):**

**Purpose:** to create a filelog-style revision that contain the content of files in each revision.

**Arguments:** “repo” is the name of the repository which the filelog refer to; “path” is the path of the file in working directory.

**Operation:** Construct a Revlog instance with filelog-style input.

**Return:** nil.



## Chapter 3

# Test

### 3.1 Unit Test

For each top-level module (except for “Filelog” module since it only contains an *open* function), we design one or more unit test cases to verify its correctness. Note that since there is only a *open* function in Filelog module, so we do not provide a unit test for this module.

#### 3.1.1 Revlog

Unit test for revlog Test is in ‘Unit\_Test1’ folder, go into the ‘Unit\_Test1’ by Terminal, then “ruby revlog\_test.rb”, you will get the result of revlog Test.

To test Revlog, we apply Python version Mercurial-0.1 to generate a fake *.hg* folder denote as “*revlog\_test*” and put it into the unit test folder. We organize all test commands in a single test function. In this function, we can test the four most important functions (revision, addrevision, mergedag, merge) in a single pass.

```
1 class RevlogTest < Test::Unit::TestCase
2 def test_all_in_one
3   path_prefix = '.hg_fake/'
4   FileUtils.mkdir_p path_prefix
5   FileUtils.cp_r './unit_test/revlog_test/.', path_prefix
6   revlog = Revlog.new(path_prefix+"00manifest.i", path_prefix+"00
  manifest.d")
7   revlog.addrevision("text1-content\n")
8   revlog.revision(-1)
9   revlog = Revlog.new(path_prefix+"00manifest.i", path_prefix+"00
  manifest.d")
10  revlog.addrevision("text2-content\n")
11  revlog.revision(0)
12
13  revlog1 = Revlog.new(path_prefix+"00changelog.i", path_prefix+"00
  changelog.d")
```

```

14     revlog1.addrevision("revlog1")
15     revlog2 = Revlog.new(path_prefix+"01changelog.i", path_prefix+"01
changelog.d")
16     revlog2.addrevision("revlog2")
17     revlog2.mergedag(revlog1)
18     revlog2.merge(revlog1)
19
20     FileUtils.rm_rf path_prefix
21 end

```

Listing 3.1: Test for Revlog

### 3.1.2 Manifest

Unit test for manifest Test is in 'Unit\_Test1' folder, go into the 'Unit\_Test1' by Terminal, then "ruby manifest\_test.rb", you will get the result of manifest Test. Here is the file to test the manifest

```

1 class ManifestTest < Test::Unit::TestCase
2
3     def test_manifest()
4         # test pack and unpack
5         text = "de400bee8d5079600fe1fbb36b3f93673de82bb0 foo.txt\n"
6
7         map = Hash.new()
8         Mdiff.linesplit(text).each do |||
9             map[||[41...-1]] = [||[0...40]].pack('H*')
10        end
11
12        gt_map = {"foo.txt" => ["de400bee8d5079600fe1fbb36b3f93673de82bb0"
].pack('H*')}
13
14        assert_equal(gt_map, map)
15    end
16 end

```

Listing 3.2: Test for Manifest

We test whether our generated map, which is to unpack the binary revision node id and file name string to hexadecimal revision node id and file name map. This process is very confusion and thus we test the correctness of it.

### 3.1.3 Changelog

Unit test for changelog Test is in 'Unit\_Test1' folder, go into the 'Unit\_Test1' by Terminal, then "ruby changelog\_test.rb", you will get the result of changelog Test. Changelog will collect the user modification auxiliary information, such as user, midification time and so on. It is basically simple but the transform form binary string to hexadecimal string is error-prone, hence we test it here.

```

1 class ChangelogTest < Test::Unit::TestCase
2   def test_pack_unpack()
3     # test pack and unpack
4     str1 = Digest::SHA1.digest('')
5     strH = str1.unpack('H*').first
6     str2 = [strH].pack('H*')
7     assert_equal(str1, str2)
8   end
9 end

```

Listing 3.3: Test for Changelog

### 3.1.4 Repository

Unit test for repository Test is in 'Unit\_Test1' folder, go into the 'Unit\_Test1' by Terminal, then "ruby repo\_unit\_test.rb", you will get the result of repository Test. The following is the unit test and result of unit test.

```

1 require 'minitest/autorun'
2 require './repository.rb'
3 require 'fileutils'
4 require './lhs.rb'
5 include LHS
6
7 class RepoUnitTest < Minitest::Test
8   def test_all
9     FileUtils.rm_rf ".hg/"
10    #test for initilaize function of repository
11    LHS.init()
12    # test for add function for repository
13    LHS.add(['./T1.txt'])
14    # test for commit function for repository
15    LHS.commit()
16    File.open("./T1.txt", "w").write("World")
17    # test for add function for repository
18    LHS.add(['./T1.txt'])
19    # test for commit function for repository
20    LHS.commit()
21    # test for cat function for repository
22    assert_equal LHS.cat(['./T1.txt', 0]).to_s, "hello"
23    # test for diff function for repository
24    LHS.diff([0, 1, './T1.txt'])
25    # test for delete function for repository
26    LHS.delete(['./T1.txt'])
27    # test for commit function for repository
28    LHS.commit()
29    # test for heads function for repository
30    LHS.heads()
31    # test for log function for repository
32    LHS.log()

```

```

33 # test for difffdir function for repository
34 LHS.difffdir()
35 LHS.checkout([1])
36 end
37 end

```

Listing 3.4: Test for Repository

### 3.1.5 mdiff

Unit test for mdiff Test is in 'Unit\_Test1' folder, go into the 'Unit\_Test1' by Terminal, then "ruby mdiff\_test.rb", you will get the result of mdiff Test.

Functions in *mdiff.rb* are all deal with strings and we design three test cases to test them. Strings used in these cases are generated by Python version Mercurial-0.1. For illustration simplicity, we replace actual strings in test code with '...'.

```

1 def test_linesplit()
2   text = "..." # We use '...' to replace the actual text string here to
   save space
3   res = Mdiff.linesplit(text)
4   gt = ["...", "...", "...", "...", "...", "...", "..."]
5   assert_equal(gt, res)
6 end
7
8 def test_textdiff()
9   a = "..."
10  b = "..."
11  res = Mdiff.textdiff(a, b)
12  gt = "..."
13  assert_equal(gt, res)
14 end
15
16 def test_patch()
17   b = "..."
18   text = "..."
19   res = Mdiff.patch(text, b)
20   gt = "..."
21   assert_equal(gt, res)
22 end

```

Listing 3.5: Test for mdiff

## 3.2 Integration Test

Unit test for Integration Test is in 'Unit\_Test2' folder, this folder has three sub-folder, the test.sh in 'ori\_dir' used to test the integration, go into the 'ori\_dir' by Terminal, then run "./test.sh", you will get the result of Integration Test.

The integration test is tested as following.



```

1 echo "hello foo" > foo.txt
2 echo "hello bar" > bar.txt
3 # test create
4 ./lhs init
5 # test add
6 ./lhs add foo.txt bar.txt
7 # test commit
8 ./lhs commit
9 # update file and commit
10 echo "world" >> foo.txt
11 ./lhs add foo.txt
12 ./lhs commit

```

The above code create repository and add two files 'foo.txt' and 'bar.txt', and then commit. Then we test cat and diff as follows

```

1 # test cat
2 ./lhs cat foo.txt 0
3 # test cat
4 ./lhs cat foo.txt 1
5 # test diff
6 ./lhs diff 0 1 foo.txt

```

the output is

```

1 "hello foo\n"
2 "hello foo\nworld\n"
3 ["=", [0, "hello foo\n"], [0, "hello foo\n"]]
4 ["+", [1, nil], [1, "world\n"]]
5 ["=", [0, "hello foo\n"], [0, "hello foo\n"]]
6 ["+", [1, nil], [1, "world\n"]]

```

The 1,2 lines show the content of 'foo.txt' in revision 0 and 1, respectively, indicating that we commit twice successfully, and cat correctly shows the content. The following output represents the difference of the content of 'foo.txt' in two revisions. '=' means no difference and '+' means change, the [number, filenames] means the revision number and the filenames belonging to it.

```

1 # test heads
2 ./lhs heads

```

the output is

```

1 1: 0 -1 504b3277296c05f84e5e9ec6caa1bf0c3c58e25c
2 manifest nodeid: 221a675f6d9301f19993be3c73e972e8131b4091
3 User: lanfengxiang@lanfengxiangdemacbook-pro.local
4 changed files:
5 foo.txt
6 description: commit

```

The heads shows the revision who has no child revision.

```

1 # test delete
2 ./lhs delete foo.txt
3 ./lhs commit
4 # test log
5 ./lhs log

```

The output is

```

1 0: -1 -1 6a9ecac3d0db7de94e29a322260367cdcc71c276
2 manifest nodeid:
3 User: lanfengxiang@lanfengxiangdemacbook-pro.local
4 changed files:
5 bar.txt
6 foo.txt
7 description: commit
8 1: 0 -1 504b3277296c05f84e5e9ec6caa1bf0c3c58e25c
9 manifest nodeid:
10 User: lanfengxiang@lanfengxiangdemacbook-pro.local
11 changed files:
12 foo.txt
13 description: commit
14 2: 1 -1 e7657c60050cc61f47e53d190779c7dce92dccdb
15 manifest nodeid:
16 User: lanfengxiang@lanfengxiangdemacbook-pro.local
17 changed files:
18 description: commit

```

After delete we commit the thrid time, so the log shows we have three revisions in total. Then we test stat

```

1 echo "new" >> foo.txt
2 # test stat
3 ./lhs stat

```

```

1 "A .DS_Store"
2 "A filelog.rb"
3 "A repository.rb"
4 "A revlog.rb"
5 "A walk.rb"
6 "A changelog.rb"
7 "C foo.txt"
8 "A mdiff.rb"
9 "A lhs"
10 "A manifest.rb"
11 "A test.sh"
12 "A .hg/00changelog.d"
13 "A .hg/00manifest.d"
14 "A .hg/current"
15 "A .hg/00changelog.i"
16 "A .hg/dircache"
17 "A .hg/00manifest.i"
18 "A .hg/index/kgasQrUy746YNHDCUfTho2X9Y2"

```

```

19 "A .hg/index/nWyqMPVNBa8O2xIL%iYTexCfIR "
20 "A .hg/data/kgasQrUy746YNHDCUfTho2X9Y2"
21 "A .hg/data/nWyqMPVNBa8O2xIL%iYTexCfIR "

```

The stats shows the status of the current working directory. “A” stands for the all untracked files. “C” denotes the tracked file that are changed but not yet commit. “D” represents the tracked file that deleted in working directory.

```

1 # test merge
2 ./lhs merge ../other

```

the output is

```

1 Begin merge changeset
2 Begin merge manifest
3 Begin resolve manifests
4 Committing merge changeset

```

‘other’ is another directory initialized as repository. And we merge ‘other’ into current working directory.

```

1 # test clone
2 ./lhs clone ../clone
3 ./lhs clone ../clone

```

the ouput is

```

1 "Copied!!!"
2 "Ops, you already have a repository in ../clone"

```

‘clone’ is another non-repository folder. We clone the current repository to directory ‘clone’ by copying all files in current repository to it. If the ‘clone’ is already a repository, other repository cannot clone to it.

```

1 # test checkout
2 ./lhs checkout 0

```

We checkout to revision 0.



## Chapter 4

# Development Phase

The unified process of software development of software is usually divided into four phases: inception phase, elaboration phase, construction phase and transition phase. During the inception phase, the project team aims at gaining an understanding of the project domain. In this DVCS project, the goal is to development a Ruby version of Mercurial-0.1. This goal clear and well-defined, therefore the inception phase is omitted in this report for simplicity.

### 4.1 Schedule Plan

Working as team, each member is in charge of one or more modules through elaboration, construction and transition phases. Table 4.1 lists the proposed schedule plan. **The workload for each member is roughly the same.**

Member	Goal	Person-hour	Phase
<i>Fengxiang Lan</i>	Design Repository module	10	Elaboration
<i>Hao Huang</i>	Design Revlog module	10	
<i>Jing Shi</i>	Design other modules	10	
<i>Fengxiang</i>	Complete Repository module	45	Construction
<i>Hao Huang</i>	Complete Revlog module	45	
<i>Jing Shi</i>	Complete other modules	45	
<i>Fengxiang Lan</i>	Unit test Repository module	20	Transition
<i>Hao Huang</i>	Unit test Revlog module	20	
<i>Jing Shi</i>	Unit test other modules	20	

**Table 4.1:** Note that during the transition phases, all members will participate in integration test.

## 4.2 Development Record

To collaborate and work remotely, all code is hosted on *Dropbox*. Parts of snapshots are provided in Figure 4.1, 4.2 and 4.3. In addition, during the development process, each member’s workload is recorded in separate tables, as shown in Table 4.2, 4.3 and 4.4.

{}	manifest.rb	1 hr ago by Shi skin	3 members
{}	mdiff.rb	1 hr ago by Shi skin	3 members
{}	repository_test.sh	2 hrs ago by Shi skin	3 members
{}	repository.rb	1 hr ago by Shi skin	3 members
{}	revlog.rb	1 hr ago by Shi skin	3 members

Figure 4.1: A snapshot of several project files hosted on *Dropbox*

{}	manifest.rb 6:28 PM	Edited by Shi skin Desktop
{}	manifest.rb 6:28 PM	Edited by Shi skin Desktop
{}	manifest.rb 4:47 PM	Added by Shi skin Desktop

Figure 4.2: A snapshot of the edit history of file *manifest.rb*

```
load "./revlog.rb"

class Manifest < Revlog
  def initialize(repo)
    @repo = repo
    super("00manifest.i", "00manifest.d")
  end

  def open(file, mode="r", &block)
    return @repo.open(file, mode, &block)
  end
end
```

Figure 4.3: A snapshot of the content of file *manifest.rb*

Date	Week	Hours	Work	Phase
10/22	MON	2.0	Analysis Python version Repository	Elaboration
10/23	TUE	1.0	Analysis Python version hg	
10/24	WED	2.0	Design DVCS module interfaces	
10/25	THU	2.0	Design repository.rb	
10/26	FRI	-	-	
10/27	SAT	3.5	Design third-party libraries	
10/28	SUN	-	-	Construction
10/29	MON	3.0	Complete one third of repository.rb	
10/30	TUE	2.5	Complete two thirds of repository.rb	
10/31	WED	-	-	
11/01	THU	2.5	Complete repository.rb	
11/02	FRI	-	-	
11/03	SAT	-	-	
11/04	SUN	3.0	Complete lhs.rb	
11/05	MON	-	-	
11/06	TUE	2.0	Develop unit test cases for repository.rb	
11/07	WED	2.0	Test half of repository.rb	Transition
11/08	THU	2.5	Complete testing repository.rb	
11/09	FRI	2.0	Test revlog.rb	
11/10	SAT	3.5	Debug revlog.rb (Part I)	
11/11	SUN	3.0	Debug revlog.rb (Part II)	
11/12	MON	1.5	Develop integration test cases (Part I)	
11/13	TUE	2.5	Develop integration test cases (Part II)	
11/14	WED	1.5	Test repository.rb	
11/15	THU	1.0	Debug revlog.rb	
11/16	FRI	-	-	
11/17	SAT	3.5	Debug changelog.rb and filelog.rb	Finish
11/18	SUN	5.0	Debug revlog.rb	
11/19	MON	3.0	Develop user interface (optional)	
11/20	TUE	3.5	Write report	Finish

Table 4.2: Work record for Fengxiang Lan

Date	Week	Hours	Work	Phase
10/22	MON	2.0	Analysis Python version Revlog	Elaboration
10/23	TUE	1.0	Analysis Python version mdiff	
10/24	WED	2.0	Design DVCS module interfaces	
10/25	THU	2.0	Design Revlog.rb	
10/26	FRI	-	-	
10/27	SAT	3.5	Design third-party libraries	Construction
10/28	SUN	-	-	
10/29	MON	2.0	Complete one third of revlog.rb	
10/30	TUE	2.5	Complete two thirds of revlog.rb	
10/31	WED	-	-	
11/01	THU	2.5	Complete revlog.rb	
11/02	FRI	-	-	
11/03	SAT	-	-	
11/04	SUN	3.0	Complete mdiff.rb	
11/05	MON	-	-	
11/06	TUE	2.0	Develop unit test cases for mdiff.rb	Transition
11/07	WED	2.0	Test mdiff.rb	
11/08	THU	2.5	Develop unit test cases for revlog.rb	
11/09	FRI	2.0	Test revlog.rb	
11/10	SAT	3.5	Debug revlog.rb (Part I)	
11/11	SUN	3.0	Debug revlog.rb (Part II)	
11/12	MON	1.5	Develop integration test cases (Part I)	
11/13	TUE	2.5	Develop integration test cases (Part II)	
11/14	WED	1.5	Test repository.rb	
11/15	THU	1.0	Debug revlog.rb	
11/16	FRI	-	-	Finish
11/17	SAT	3.5	Debug changelog.rb and filelog.rb	
11/18	SUN	5.0	Debug revlog.rb	
11/19	MON	3.0	Develop user interface (optional)	
11/20	TUE	3.5	Write report	

Table 4.3: Work record for Hao Huang



Date	Week	Hours	Work	Phase
10/22	MON	2.0	Analysis Python version filelog, manifest	Elaboration
10/23	TUE	1.0	Analysis Python version changelog, mdiff	
10/24	WED	2.0	Design DVCS module interfaces	
10/25	THU	2.0	Design mdiff	
10/26	FRI	-	-	
10/27	SAT	3.5	Design third-party libraries	
10/28	SUN	-	-	Construction
10/29	MON	2.0	Complete filelog.rb	
10/30	TUE	2.5	Complete changelog.rb	
10/31	WED	-	-	
11/01	THU	2.5	Complete manifest.rb	
11/02	FRI	-	-	
11/03	SAT	-	-	
11/04	SUN	3.0	Complete mdiff.rb	
11/05	MON	-	-	
11/06	TUE	2.0	Develop unit test cases for mdiff.rb	
11/07	WED	2.0	Test mdiff.rb	Transition
11/08	THU	2.5	Develop unit test cases for revlog.rb	
11/09	FRI	2.0	Test revlog.rb	
11/10	SAT	3.5	Debug revlog.rb (Part I)	
11/11	SUN	3.0	Debug revlog.rb (Part II)	
11/12	MON	1.5	Develop integration test cases (Part I)	
11/13	TUE	2.5	Develop integration test cases (Part II)	
11/14	WED	1.5	Test repository.rb	
11/15	THU	1.0	Debug revlog.rb	
11/16	FRI	-	-	
11/17	SAT	3.5	Debug changelog.rb and filelog.rb	
11/18	SUN	5.0	Debug revlog.rb	
11/19	MON	3.0	Develop user interface (optional)	Finish
11/20	TUE	3.5	Write report	

Table 4.4: Work record for Jing Shi



## Chapter 5

# Conclusion

In this report, we give a thorough description of our Distributed Version Control System developed in Ruby, including system functionality, module specification, function design, test cases and development process.

In case you have questions, comments, suggestions or have found a bug, please do not hesitate to contact us.