



Team 3

Cassios Marques – Danzel Capers – Romeo Bellon

Table of contents

1. **Introduction**
2. **Chapter 1**
 - 2.1. Experience
 - 2.2. Compiling Git
 - 2.3. Using git
3. **Chapter 2**
 - 3.1. Experience
 - 3.2. Git Test Plan
 - 3.2.1. Testing Process
 - 3.2.2. Requirements Traceability
 - 3.2.3. Tested Items
 - 3.2.4. Testing Schedule
 - 3.2.5. Test Recording Procedures
 - 3.2.6. Software Requirements
 - 3.2.7. Constraints
4. **Chapter 3**
 - 4.1. Experience
 - 4.2. Architectural Framework Description
 - 4.2.1. Test Template
 - 4.2.2. Compiling Tests
 - 4.2.3. Test Script
 - 4.3. Running Script
 - 4.4. Test Cases
5. **Chapter 4**
 - 5.1. Experience
 - 5.2. Test Cases
 - 5.3. Methods being tested
6. **Chapter 5**
 - 6.1. Experience
 - 6.2. Fault Injection
7. **Chapter 6**
 - 7.1. Overall Experience
 - 7.2. Team Evaluation
 - 7.3. Assignments Evaluation

1. Introduction

In this project, we designed and developed a special purpose test framework for Git, which is a open source version control system, and we wrote test cases intended to run in this test framework. We have chosen Git out of Lisa Lab - Deep Learning Tutorials and Ubuntu Software Center because its importance in the software development community.

This report is divided into 6 chapters. Chapter 1 describes what dependencies shall be installed, and how to download and compile Git's source code. Chapter 2 presents the test plan for Git that was followed in this project. Chapter 3 describes the test framework architecture, as well as its usage. Chapter 4 briefly describes all test cases, and it shows the methods that were tested. Chapter 5 shows our fault injection plan that were designed to demonstrate the framework's ability to get errors in the code. Finally, Chapter 6 presents our overall experience in doing this project, as well as the team and assignments evaluation.

2. Chapter 1

2.1. Experience

Danzel - Had a little trouble cloning and compiling the code because of little typing mistakes, not yet use to writing directly to a command line, made a "work space" directory to keep all my work organized. Learned to always be aware of what directory I'm in because it affects the commands I use. The cloning and building process took a while to finish, not sure if it has anything to do with the fact that I am using VirtualBox or just that there are a lot of files being managed.

2.2. Compiling Git

All commands shown here should be inserted in a unix terminal:

1. install the following libraries: curl, zlib, openssl, expat, and libiconv:
 - `sudo apt-get install libcurl4openssldev zlib1gdev openssl libexpat1dev libiconv*`
2. download the source code for git from github:
 - `git clone https://github.com/git/git.git`
3. enter in the repository:
 - `cd git/`
4. compile the code:
 - `make`
5. install it at the ~/bin/ repository
 - `make install`

2.3. Using Git

1. Go to your home directory:
 - `cd ~/`
2. Clone a repository from github:
 - `./git/git clone https://github.com/lisalab/DeepLearningTutorials.git`

3. Chapter 2

3.1. Experience

This part of the project was a little bit complicated because we were not sure how deep we should go in the plan. Furthermore, figuring out what should go in each section took some time.

3.2. Git Test Plan

3.2.1. Testing Process

- The open source project exposed to test is **git**. We have it cloned, compiled and run/used it in our machines.
- Secondly, we have checked the content of the project folder and examined some of the .c files contained.

3.2.2. Requirements Traceability

Users are most interested in the system meeting its requirements and testing should be planned so that all requirements are individually tested.

3.2.3. Tested Items

Since our test framework does not have the capability of mocks and stubs, we could not test functions that calls many other functions. Thus, the following modules were chosen to be tested since they have functions which matches the complexity level that our test framework can handle :

- credential.c
- url.c
- color.c
- bisect.c
- commit.c

3.2.4. Testing Schedule

The testing schedule is organized according and following each assignment's deadline.

3.2.5. Test Recording Procedures

The output of the test case will be an HTML file containing the results.

3.2.6. Software Requirements

- Ubuntu 14.04
- gcc 4.9.2

3.2.7. Constraints

No constraints.

4. Chapter 3

4.1. Experience

In completing deliverable 3 we had more experience working together as a team. We started off by defining what we needed to get done and separating the work amongst ourselves. Our divide and conquer approach helped knock out one task after another in a nice pace. When a team member got stuck or had issues with code they could get help when we meet together or by using slack. Slack is messaging app that let us communicate between the three of us. By having team meetings it makes it easy to get help with making the different sections work together while keeping us on the same page. Starting was hard because it was hard to figure out how we wanted to go about setting up the drivers and how to compile them with low overhead.

4.2. Architectural Framework Description

4.2.1. Test Template

We choose a template that is readable to the terminal but is also easily read by a person for easy understanding. It includes the component and function that is being tested, a description of what is being tested, the name of the driver, the arguments that must be passed to the driver, and the expected outcome.

4.2.2. Compiling Tests

Since the components that we testing have a lot of dependencies and also need some environment variables that are set during compilation, we decided to compile our drivers in the general git compilation. It adds a huge overhead for the first time that you compile the project (around 2 minutes). However, since it is compiled using a makefile, after the first compilation, only the files that have been changed are recompiled and linked with the others. Furthermore, we think that it is the most convenient approach since it is how native tests are compiled on git.

4.2.3. Test Script

The script copies all drivers from /testCasesExecutables to the project folder (project/git) and compiles all the project. As pointed above, it takes around 2 minutes for the first compilation, but in later executions it will compile only the files that have changed. After compile the project, the script takes all test cases. For it test case, it identifies the driver name, the arguments, and run the driver. Then, the output is stored in a variable and compared with the expected result to define whether the test passed or not.

4.3. Running Script

First, make sure that you have all the libraries needed by Git in our machine:

- `sudo apt-get install libcurl4-openssl-dev zlib1g-dev openssl libexpat1-dev libiconv*`

After install the dependencies, go to the top-level folder of the test framework and run:

- `./scripts/runAllScripts.sh`

4.4. Test Cases

The first 5 test cases are shown altogether with the other test cases in the next section.

5. Chapter 4

5.1. Experience

Once again, we had a good experience working as a team. We split the work in three main tasks: write documentation, improve script to run tests, and write the drivers for the remaining 20 test cases. Writing the tests cases was time consuming because it was hard to find functions that were neither too complicated or static (they can only be called inside the module where they were defined). We also changed our test cases output layout to a table view, so it's easier to read the information. By using the table view, the 25 test cases are shown in a cleaner view.


5.2. Test Cases

ID	Module / Function	Requirement	Driver	Arguments	Exp.	Res.
1	credential.c / credential_match	returns 1 credentials have the same protocol, otherwise 0	test-credentialMatchDriver	"https example.com foo.git bob http example.com foo.git bob"	0	0

2	credential.c / credential_match	returns 1 credentials have the same host, otherwise 0	test-credentialMatchDriver	"https example.com foo.git bob https otherExample.com foo.git bob"	0	0
3	credential.c / credential_match	returns 1 credentials have the same path, otherwise 0	test-credentialMatchDriver	"https example.com foo.git bob https example.com bar.git bob"	0	0
4	credential.c / credential_match	returns 1 credentials have the same username, otherwise 0	test-credentialMatchDriver	"https example.com foo.git bob https example.co foo.git mary"	0	0
5	credential.c / credential_match	returns 1 when every field of the credentials are equal	test-credentialMatchDriver	"https example.com foo.git bob https example.com foo.git bob"	1	1
6	url.c / is_urlschemechar	return 0 if '+' is the first character, otherwise 1	test-urlSchemecharDriver	"1 +"	1	0
7	url.c / is_urlschemechar	returns 0 if '-' as the first character, otherwise 1	test-urlSchemecharDriver	"1 -"	0	0
8	url.c / is_urlschemechar	returns 0 if '.' as the first character, otherwise 1	test-urlSchemecharDriver	"1 ."	0	0
9	url.c / is_urlschemechar	returns 1 if the special character is in other positions that is not the first one, otherwise 0	test-urlSchemecharDriver	"0 +"	1	1
10	url.c / is_urlschemechar	returns 1 if the character in the first position is a number	test-urlSchemecharDriver	"1/8/2015"	1	1
11	url.c / is_urlschemechar	allows numbers in the other positions of the url	test-urlSchemecharDriver	"0 8"	1	1
12	url.c / is_url	verifies if url is not empty	test-isUrlDriver	"" ""	0	0
13	url.c / is_url	return 0 if the first character of the url is a special character(+ - .)	test-isUrlDriver	"https://myurl.com"	0	0
14	url.c / is_url	verifies if url has the pattern '://'	test-isUrlDriver	"https:myurl.com"	0	0
15	bisect.c / estimate_bisect_steps	returns 0 if n is less than 3	test-estimateBisectStepsDriver	"2"	0	0
16	bisect.c / estimate_bisect_steps	returns the integer log e of n minus 1 ((logi n)-1) if ($2^e < 3*(n - 2^e)$)	test-estimateBisectStepsDriver	"10"	2	2
17	bisect.c / estimate_bisect_steps	returns integer log e of n if ($2^e < 3*(n - 2^e)$)	test-estimateBisectStepsDriver	"100"	6	6
18	color.c / git_config_colorbool	returns 0 if value is equal to 'never'	test-gitConfigColorboolDriver	"core.color never"	0	0
19	color.c / git_config_colorbool	returns 1 if value is equal to 'always'	test-gitConfigColorboolDriver	"core.color always"	1	1
20	color.c /	returns the automatic color(2) if the value is equal to 'auto'	test-gitConfigColorboolDriver	"core.color auto"	2	2


	git_config_colorbo ol					
21	color.c / git_config_colorbo ol	returns -1 if var and value are empty	test-gitConfigColorboolDriver	"returns -1 if var and value are empty"	-1	-1
22	color.c / git_config_colorbo ol	returns the automatic color(2) if var is not empty, but value is empty	test-gitConfigColorboolDriver	"core.color" ""	2	2
23	color.c / color_is_nil	returns 1 if color is nil	test-colorIsNil	"NIL"	1	1
24	color.c / color_is_nil	returns 0 if color is not nil	test-colorIsNil	"green"	0	0
25	commit.c / commit_list_count	returns 0 if commit list is empty	test-commitListCount	"0"	0	0
26	commit.c / commit_list_count	returns the number of items in the commit list if it is not empty	test-commitListCount	"4"	4	4

5.3. Functions being tested

 **credential_match.c** Raw

```

1  int credential_match(const struct credential *want, const struct credential *have){
2  #define CHECK(x) (!want->x || (have->x && !strcmp(want->x, have->x)))
3      return CHECK(protocol) && CHECK(host) && CHECK(path) && CHECK(username);
4      // return CHECK(protocol) && CHECK(path) && CHECK(username); //code for fail injection 1
5  #undef CHECK
6  }
```

 **is_urlschemechar.c** Raw

```

1  int is_urlschemechar(int first_flag, int ch) {
2      /*
3       * The set of valid URL schemes, as per STD66 (RFC3986) is
4       * '[A-Za-z][A-Za-z0-9+.-]*'. But use sightly looser check
5       * of '[A-Za-z0-9][A-Za-z0-9+.-]*' because earlier version
6       * of check used '[A-Za-z0-9]+' so not to break any remote
7       * helpers.
8       */
9      int alphanumeric, special;
10     alphanumeric = ch > 0 && isalnum(ch);
11     // alphanumeric = ch > 0 && isalpha(ch); //code for fail injection 2
12     special = ch == '+' || ch == '-' || ch == '.';
13     return alphanumeric || (!first_flag && special);
14 }
```

is_url.c

Raw

```

1  int is_url(const char *url) {
2      /* Is "scheme" part reasonable? */
3      if (!url || !is_urlschemechar(1, *url++))
4          // if (!url) //code for fail injection 3
5          return 0;
6      while (*url && *url != ':') {
7          if (!is_urlschemechar(0, *url++))
8              return 0;
9      }
10     /* We've seen "scheme"; we want colon-slash-slash */
11     return (url[0] == ':' && url[1] == '/' && url[2] == '/');
12 }

```

estimate_bisect_steps.c

Raw

```

1  int estimate_bisect_steps(int all) {
2      int n, x, e;
3      if (all < 3)
4          return 0;
5      n = log2i(all);
6      e = exp2i(n);
7      x = all - e;
8      return (e < 3 * x) ? n : n - 1;
9      // return n; //code for fail injection 4
10 }

```

git_config_colorbool.c

Raw

```

1  int git_config_colorbool(const char *var, const char *value) {
2      if (value) {
3          if (!strcasecmp(value, "never"))
4              return 0;
5          if (!strcasecmp(value, "always"))
6              return 1;
7          if (!strcasecmp(value, "auto"))
8              return GIT_COLOR_AUTO;
9      }
10     if (!var)
11         return -1;
12     /* Missing or explicit false to turn off colorization */
13     if (!git_config_bool(var, value))
14         return 0;
15
16     /* any normal truth value defaults to 'auto' */
17     return GIT_COLOR_AUTO;
18     // return GIT_COLOR_RED; //code for fail injection 5
19 }

```

 `color_is_nil.c`

Raw

```
1  int color_is_nil(const char *c) {  
2      return !strcmp(c, "NIL");  
3  }
```

 `commit_list_count.c`

Raw

```
1  unsigned commit_list_count(const struct commit_list *l) {  
2      unsigned c = 0;  
3      for (; l; l = l->next )  
4          c++;  
5      return c;  
6  }
```

6. Chapter 5

6.1. Experience

Since we have a well defined test framework and our test cases are testing specific parts of the code, defining the fault injections was straightforward. We just went through our test cases and found the most easily manipulated methods that could be isolated to affect only one or two test cases at a time. This made the overall task better and we completed deliverable 5 along with deliverable 4.

6.2. Fault Injection

When defining the fail injections, we tried to inject fails in different functions in order to verify our understanding of all tested functions. To make easier replicating the faults, we added the faults injection code in the functions as comments. Thus, when testing it, one only need to comment the line indicated in the table below, and uncomment the line right below that.

Fault ID	Module	Function	Changes in code	At line	Effects
1	credential.c	credential_match	Remove test for host. CHECK(host)	31	The test 002 shall fail
2	url.c	is_urlschemechar	Change the check for alphanumeric character to only letters. isalnum(ch) -> isalpha(ch)	14	The tests 010 and 011 shall fail
3	url.c	is_url	Remove test for first character. !is_urlschemechar(1, *url++)	23	The test 013 shall fail
4	bisect.c	estimate_bisect_steps	Remove conditional and only returns n.	1032	The test 016 shall fail
5	color.c	git_config_colorbool	As the default, returns the color red instead of the automatic color. GIT_COLOR_AUTO -> GIT_COLOR_RED	282	The test 022 shall fail

7. Chapter 6

7.1. Overall Experience

In this project we had the experience of testing a mature software. We could get a taste of the challenges faced by test framework developers. We also improved our teamwork skills and how to coordinate a software development project among developers. For some of us, it was the first time working with a version control system such as Github. Another experience taken from this semester is the ability to adapt quickly to new technology and a changing environment. Some of us had to refresh ourselves on using languages like C and python while learning how to write scripts in bash. Learning on one's own time the tools needed to accomplish tasks is something that can be taken into the real world.

7.2. Team Evaluation

7.2.1. Cássios

I had a great experience working with my team. Everyone helped in the assignments, and everyone was proactive.

7.2.2. Danzel

This has been one of the most productive teams I have been able to work with during college. Regular meetings and set goals for each member made the work flow very smooth without worry about others not completing assigned responsibilities.

7.2.3. Romeo

It was awesome work with this team. Every time we met we always conclude what we were suppose to finish, accomplished our goals, and try not to lose time.

7.3. Assignments Evaluation

7.3.1. Cássios

If the goal of the project was learning the test process, I do not think that it was achieved. First, we spent more time developing the test framework than actually testing the project. We could have used a mature test framework, so we could focused on the testing process. Furthermore, by using a developed test framework, we would have more advanced

capabilities available, such as mock and stub, resulting in designing better tests. Secondly, although we talked about testing, I do not think that enough time was spent on explaining the difference of the several kind of tests.

7.3.2. Danzel

Most of the assignments were focus on getting our test framework established more so than testing the source code or test framework. I would suggest adding in more assignments like deliverable 5 that tasked us with injecting faults into the code. I'm not sure about any other different ways of testing our framework but having more exposure and time thinking about where my framework may be lacking could help improve the experience overall.