ユーザーメモソフト my_help のテスト作成及びコマンドの改良

関西学院大学理工学部
情報科学科 西谷研究室
27016634
山口修平

2020 年 3 月

目 次

第 1 章　序論 **2**

第 2 章　開発手法 **3**
2.1　テストの種類 3
2.2　RSpec 3
2.3　aruba 4
2.4　command_line 5
2.5　生じた問題 5

第 3 章　my_help **6**
3.1　my_help のインストール 6
3.2　コマンド一覧 6

第 4 章　behavior test の作成 **8**
4.1　テストを作成するための my_help の設定 8
4.2　aruba を用いたテストの作成 10
4.3　command_line を用いたテストの作成 11
4.4　aruba と command_line の違い 13
4.5　結果 13

第 5 章　delete メソッドの改良 **14**
5.1　delete_help メソッドの書き換え 14
5.2　結果 14

第 6 章　git コマンドの改良 **16**
6.1　GitHub 16
6.2　my_help と GitHub の連携 16
6.3　開発者側とテスターの目的の相違による問題 17
6.4　結果 18

第 7 章　考察と今後の課題 **20**
7.1　考察 20
7.2　my_help の今後 20

第 8 章　総括 **21**

第1章　序論

使用者がコンピューター上で作業したい機能を直接的に実現するソフトウェアとしてアプリケーションソフトウェアがある．大抵のアプリケーションソフトウェアの場合，発展，進化させる上でプログラムコードを書き換えていくというのが常である．しかし，書き換えることにより，どこかの箇所が機能しなくなる恐れがある．それを正常に動作しているのかを検証する方法がテストである．

一番身近なテスト手法として，全ての機能を一つずつ手作業で実行し，確認していくという方法がある．短期的な目線で見ると実行結果を目視するだけで良いため楽に見える．しかし，プログラムコードの書き換えを行う中でテストの量が膨大になり，手作業だと時間がかかり大変になる．そのため大抵のアプリケーションソフトウェアには，コマンド一つでテストを行うことができるテストコードが書かれている．

西谷研で 2016 年以降に開発を進めているユーザーメモソフトである my_help にはテストコードが書かれていない [1]. そこで本研究では，my_help のテストコードを作成し，テストを自動化することを目的とした．また，その過程において，my_help のコード記述のルールとしてテストと整合しないところがあることが判明したので，それら全ての書き換えを行った．また，my_help が提供しているいくつかの振る舞いに利用者が誤解するところがあったので，その改善を行った．

第2章　開発手法

2.1　テストの種類

近年，アジャイル開発やリーン開発などの先進的なシステム開発においては，テスト駆動開発 (TDD) という開発方法が推奨されている [2].

テストをしながら開発を行っていくという方法であり，内容としては以下である．

- テストを書く
- 実行する
- エラーを見てテストが動くようにコードを作成する
- コードを綺麗にする

テストには様々な種類がある．中でも coding のもっとも初期に行われるテストが unit test である．unit test はクラスや関数といった単位ごとのテストであり，入力に対して，出力が仕様通りになっているかが確認できる．コード作成時の早い段階で開発者によって実施されることが多く，my_help の unit test は開発者である西谷が作成した.

一方，behavior test と呼ばれるテストがある．このテストは unit test などと同じように，作成したプログラムコードが正しく動作しているかを検証するためのテストであるが，それに加えテストするプログラムに期待する振る舞いを自然言語を併記して書いていくものである．

このテストを作成することにより，テストコードを見るだけで振る舞いを確認することができ，今後 my_help で新しい機能を開発する際，テストの雛形になると考えられる．そのため， 本研究では behavior test というテスト方法を採用した．

2.2　RSpec

behavior test を作成するにあたり，テストフレームワークに RSpec を用いることとした．aruba というコマンドラインインターフェイス (CLI) に特化したテストフレームワークや command_line という gem をインストールすることにより，Rspec の機能を拡張し，CLI のテストを簡単に作成できるという特徴がある．さらに，図 2.1のような基本系もあり，自然言語を用いてテストの整理，分類ができるためわかりやすいテストを作成することができる [3].

```
describe [仕様を記述する対象（テスト対象）] do

  context [ある状況・状態] do

    before do
      [事前準備]
    end

    it [仕様の内容（期待の概要）] do
      [期待する動作]
    end
  end
end
```

図 2.1: RSpec のテストコードの基本形.

2.3 aruba

コマンドラインで操作できる my_help の振る舞いをテストするためには実働環境で返ってきた結果を評価する必要がある．RSpec を用いてコマンドラインアプリケーションを実行するためには，まず下記のコードを spec_helper.rb に記述しなければならない．

```
RSpec.configure do |config|
  def capture(stream)
    begin
      stream = stream.to_s
      eval "$#{stream} = StringIO.new"
      yield
      result = eval("$#{stream}").string
    ensure
      eval("$#{stream} = #{stream.upcase}")
    end
    result
  end
```

それに加え，サブプログラムコードで書かれた関数を呼び出さなければならないため，サブプログラムコードの関数名などを確認しなければならないため，手間がかかる．そこで aruba というテストフレームワークを用いて RSpec の機能を拡張した．aruba ではメソッド一つでコマンドラインアプリケーションを実行できる [4]．しかし，aruba を用いると実行結果を比較できるものの結果やエラーを取り出すことはできない．そのため，エラーが起こるはずの部分でテストが正常に通ってしまうところを見つけることができない．

2.4 command_line

結果やエラーを取り出すためによく使われる標準ライブラリは，open3 で

```
require 'open3'
out, _err, _status = Open3.capture3('ls')
Open3.capture3("echo a; sort >&2", :stdin_data=>"foo\nbar\nbaz\n")
```

などとする．Open3.capture3(*cmd) は popen3(*cmd) を呼び出し，そこから Kernel.#spawn に渡される．そういう内部動作から見ると，外部コマンドを実行するには open3 を使うのがもっとも標準的であることがわかる．

しかし，コマンドが複雑になりがちである．そのため，コマンドラインアプリケーションを簡単に実行するための gem である command_line を用いることとした．使い方は次の通りである [5].

```
$ result = command_line('echo', 'hello')
 #=> #<CommandLine::Result ...>
$ result.stdout
 #=> "hello\n"
```

特徴としては実行したいコマンドの exitstatus,stdout,stderr などを返してくれるため、コマンドラインアプリケーションのテストには最適と思われる．

2.5 生じた問題

テストを進めていくうちに以下の問題が生じた．

- delete メソッドにおけるテストの途中終了
- git コマンドの開発者側とテスター（使用者）での使用目的の相違

以上の問題を解決するために my_help のプログラムコードの書き換えを行い，改善を提案した．

第3章　my_help

ユーザーメモソフトである my_help は西谷研究室で開発，使用している gem である．特徴としては，

- user が自分にあった man を作成
- 雛形を提供
- おなじ format,looks, 操作, 階層構造
- すぐに手が届く
- それらを追加・修正・削除できる

の 5 点があげられている [6]．my_help はコマンドラインインターフェース (CLI) で操作できるため，terminal 上で簡単に提示させることができる．それにより，プログラミングに集中することができる．また，my_help が提供する編集, 参照機能の高い利便性を使えば，メモとしての用途も可能である．

3.1　my_help のインストール

以下のコマンドを入力することで誰でも使用可能である．

```
$ gem install my_help
```

3.2　コマンド一覧

以下のコマンドを入力することで my_help のコマンド一覧を表示させることができる．

```
$ my_help help

Commands:
  my_help delete HELP                # delete HELP
  my_help edit HELP                  # edit HELP
  my_help git [push|pull]            # git push or pull
  my_help help [COMMAND]             # Describe available commands or one specific comman
  my_help list [HELP] [ITEM]         # list all helps, specific HELP, or ITEM
```

```
my_help new HELP                 # make new HELP
my_help set_editor EDITOR_NAME  # set editor to EDITOR_NAME
my_help setup                    # set up the test database
my_help version                  # show version
```

第4章　behavior test の作成

4.1　テストを作成するための my_help の設定

my_help はすでに GitHub で公開されている．このレポジトリを自分のパソコンに移し，そこで改良を加えていった．その手順は次の通りである．

- my_help の fork

  レポジトリを fork することにより，オリジナルのプログラムに影響を及ぼすことなく変更できるようになる．GitHub の daddygongon のページに行き，my_help を fork する．図 4.1の右上の fork 部分をクリックする．自分の GitHub のレポジトリ一覧に my_help が作成されていると fork 成功である．

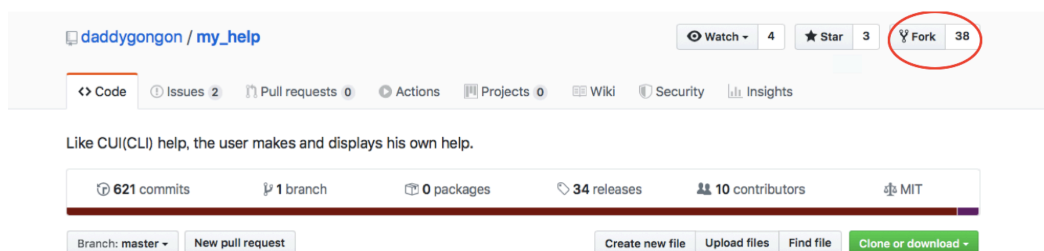

図 4.1: my_help のレポジトリ画面.

- my_help の clone

  現段階では my_help のレポジトリの fork はありますが，使用しているパソコンにはレポジトリ内のファイルはない．それを作成するために以下のコマンドをターミナル上で入力する．

```
$ git clone https://github.com/daddygongon/my_help.git
$ ls
```

  my_help という名前のディレクトリができていれば clone できている．

- オリジナルの my_help のレポジトリと fork したものを同期

同期させることにより，変更点を開発者に送ったり，新しい my_help を取り込むことができる．まず，ターミナルで clone で作成した my_help のディレクトリに移動し，以下のコマンドを入力する．

```
$ git remote -v
```

図 4.2のように upstream があれば完了である．ない場合，以下のコマンドを入力する．

```
$ git remote add upstream git@github.com:daddygongon/my_help.git
```

図 4.2のように upstream ができると完了である．

```
Shuhei@AdminnoMacBook-ea ~/my_help> git remote -v
origin  https://github.com/shuhei555/my_help.git (fetch)
origin  https://github.com/shuhei555/my_help.git (push)
upstream        git@github.com:daddygongon/my_help.git (fetch)
upstream        git@github.com:daddygongon/my_help.git (push)
```

図 4.2: git remote -v の実行結果.

- Rspec のインストール

  ターミナルで以下のコマンドを入力し，インストールを行う．

  ```
  $ gem install rspec
  ```

  次に，以下のコマンドを入力する．

  ```
  $ rspec -help
  Usage: rspec [options] [files or directories]
  ```

  このように，表示されていれば install できている．

- Rspec の初期設定

  my_help のディレクトリ内で以下のコマンドを入力する．

  ```
  $ rspec --init
  create   .rspec
  create   spec/spec_helper.rb
  ```

  このように表示されていれば Rspec を使うための準備は完了である．

- command_line を使用する場合

  ターミナルで以下のコマンドを入力し．インストールを行う．

```
$ gem install command_line
```

インストール後，my_help の Gemfile に以下を追記する．

```
$ gem 'command_line', '~> 1.1'
```

次に my_help/spec/spec_helper.rb に

```
require 'command_line'
  RSpec.configure do |config|
  config.include CommandLine
end
```

を追記することで使用可能となる．

- aruba を使用する場合

  ターミナルで以下のコマンドを入力し，インストールを行う．

  ```
  $ gem install aruba
  ```

  インストール後，my_help のディレクトリで

  ```
  $ aruba init --test-framework rspec
  ```

  というコマンドを入力する．

  ```
  append  spec/spec_helper.rb
  create  spec/support/aruba.rb
  append  Gemfile
  ```

  上記のような表示がされると使用可能となる．

4.2 aruba を用いたテストの作成

aruba を用いて，コマンドが正常に動作しているかを確認するテストコードを作成した．テスト項目は以下の 2 点である．

1. テストするコマンドが正常に終了しているか
2. コマンドを実行した結果の比較

以下が my_help list のテストコードである．

```
1  context 'list sample' do
2     expected = "- my todo\n     , head            : head\n     , license          \
3     : license\n   -d,  daily           : daily\n   -w, weekly                    \
4     : weekly\n   -s, sample test    : sample test"
5     before(:each) { run_command('bundle exec ../bin/my_help_thor list sample')}
6     it{expect(last_command_started).to be_successfully_executed}
7     it{expect(last_command_started).to have_output(expected)}
8   end
```

テストコードを参考にテストの作成手順を説明していく．1 行目では context を用いてどのコマンドをテストするか表記する．context を用いることによりコマンドのテストコードごとにブロック化されるため，テストコードが見やすくなる．2, 3, 4 行目は，コマンドの理想的な実行結果を変数に代入する．後ほど，コマンドの実行結果と比較するために用いる．5 行目では，before(:each) メソッドを使うことにより，6,7 行目にある it ブロックの前に my_help の test list のコマンドが実行されるようにコードを書いている．6 行目では.to be_successfully_executed を用いることにより，プログラムが正常に終了しているか検証を行っている．7 行目では.to have_output(expected) を用いることにより，2, 3, 4 行目で書いた理想の出力結果と比較を行っている．

以上の手順で my_help のコマンドのテストを作成した．

4.3 command_line を用いたテストの作成

command_line の特徴として，コマンドラインアプリケーションを実行した結果やエラーを取り出すことができる．そのため command_line を用いたテストでは，my_help のエラーが出るはずのところでエラーが出ていないなどの異常な点を見つけるという目的でテストを作成した．

command_line でのテスト項目は以下の 3 点である．

1. exit_status の値の比較
2. stdout の比較
3. stderr の比較

以下が my_help list sample のテストコードである．

```
1  context 'my_help list exist file' do
2    test_list = command_line('my_help','list','sample')
3    it 'exitstatus test' do
4      if test_list.exitstatus ==  0
5         puts clear
6      else
7         puts miss
```

```
 8      end
 9    end
10
11    it 'stdout test' do
12    message = <<'EOS'
13  - ヘルプのサンプル雛形
14  -   head に常に表示される内容を記述
15      , head           : head
16      , license        : license
17     -i, item_example   : item_example
18  EOS
19      if test_list.stdout == message
20         puts clear
21      else
22         puts miss
23      end
24    end
25
26    it 'stderr test' do
27       if test_list.stderr == ""
28         puts clear
29       else
30         puts miss
31       end
32    end
33  end
```

テストコードを参考にテストの作成を説明していく．1 行目で context を用いて my_help のテストするコマンドを示す．context でテストするコマンドをグループごとに分割することにより，テストコードを見やすくするという役割がある．2 行目で my_help list sample というコマンドを実行した時の exit_status,stdout,stderr を取り出し，変数 test_list に代入する．以降もコマンドを呼び出すことがあり，その作業をなくすため変数に代入しておくと良い．4, 12, 17 行目で it を用いることにより exit_status,stdout,stderr の 3 項目のグループに分割し，if 文を用いて比較していく．

以下からは分割した exit_statis のテスト (4?10 行目) を例に説明する．my_help list sample のコマンドを実行した際の exit_status は 0 である．そのため以下のような if 文になる．

```
if test_list.exitstatus ==  0
```

true なら clear,false なら miss という出力をして my_help list sample の exit_status のテストは完成である．

これをメモが存在する場合，しない場合の条件をつけて以上の手順でテストを作成していく．

4.4 aruba と command_line の違い

特徴として，どちらも実行したいコマンドを直接呼び出すことができるためテストで実行結果を比較する際に便利である．aruba を用いる場合，コマンドの呼び出し，比較など，フレームワーク独自のメソッドを覚えなければならない．さらに aruba を使用した参考文献が少ないため，テストを書くまでに比較的時間がかかる．それに比べ command_line を用いる場合，コマンドの呼び出し以外独自のメソッドがない．さらに exit_status,stdout,stderr などのコマンドの実行結果を受け取ることができるため比較などは if 文を用いて書くことが出来る．そのため既存の知識でメソッドを書くことが出来る．

4.5 結果

aruba，command_line を用いたテストは共に完成した．しかし，どちらのテストでも delete メソッドのテストを行うと停止した．delete メソッドでは 図 4.3のようなファイルを間違えて消さないための confirm 機能がある．my_help では confirm の値の受け渡しの

```
[Shuhei@AdminnoMacBook-ea ~/my_help> my_help delete sample
 Are you sure to delete /Users/Shuhei/.my_help/sample.org?[Yn]
```

図 4.3: confirm 機能.

手段としてキーボード入力を用いている．しかし，aruba や command_line ではキーボード入力をプログラムコードに受け渡す機能がない．これを改善するために

1. confirm 機能を削除
2. confirm 機能のテスト部分のみ RSpec にテストフレームワークを変更

1 つ目の confirm 機能を削除を採用した場合間違えてファイルを削除してしまう可能性がある．そのため 2 つ目の方法で行うこととした．

第5章　deleteメソッドの改良

5.1　delete_helpメソッドの書き換え

delete_help メソッドでは一部 exit 機能が使われている．しかし，exit を含むコードをテストすると ruby のプロセスが終了し，テストコード側に処理が戻ってこない．そのため，テスト継続が不可能である．これにより，exit をテスト対象のコードで使用しない設計が良いと考えた．exit_status を用いることにより，実行したコマンドに関するより良い情報を取得することが出来る [7]．そのため return 文を用いることとした．exit_status が 0 の場合，プログラムが正常に終了し，1 の場合，一般的なエラーが起こっているとされている [8]．さらに，存在しないファイルを消すように実行した場合も confirm 機能が出たため, エラーが出るように書き換えた．

変更前の delete_help メソッドは 図 5.1のようになる．

```
def delete_help(file)
  file = File.join(@local_help_dir,file+'.org')
  print "Are you sure to delete "+file.blue+"?[Ynq] ".red
  case STDIN.gets.chomp
  when 'Y'
    begin
      FileUtils.rm(file,:verbose=>true)
    rescue => error
      puts error.to_s.red
    end
  when 'n', 'q' ; exit
  end
end
```

図 5.1: 変更前の delete_help メソッド.

変更後の delete_help メソッドは図 5.2のようになる．

5.2　結果

delete_help メソッドのテスト結果は 図 5.3のようになる．テストした結果，失敗が 0 になっていることがわかる．

```ruby
def delete_help(file)
  file = File.join(@local_help_dir,file+'.org')
  if File.exist?(file) == true
    print "Are you sure to delete "+file.blue+"?[Yn] ".red
    case STDIN.gets.chomp
    when 'Y'
      begin
        FileUtils.rm(file,:verbose=>true)
        return 0
      rescue => error
        puts error.to_s.red
        return 1
      end
    when 'n', 'q' ; return 0
    end
  else
    print file + " is a non-existent file"
  end
end
```

図 5.2: 変更後の delete_help メソッド.

```
delete option sample
  exit file
    user input  n
Are you sure to delete /Users/Shuhei/.my_help/sample.org?[Yn]       exitstatus test
      file test
    user input  q
Are you sure to delete /Users/Shuhei/.my_help/sample.org?[Yn]       exitstatus test
      file test
    user input  Y
Are you sure to delete /Users/Shuhei/.my_help/sample.org?[Yn] rm /Users/Shuhei/.my_help/sample.org
      exitstatus test
      file test
  not exit file
/Users/Shuhei/.my_help/zzzzzz.org is a non-existent file     stdout test

Finished in 0.02612 seconds (files took 0.63034 seconds to load)
7 examples, 0 failures
```

図 5.3: delete_help メソッドのテスト結果.

第6章　git コマンドの改良

6.1　GitHub

本研究では my_help で作成されたメモを他デバイスに転送あるいは共有するために GitHub を用いた．Git の仕組みを利用し，各個人がプログラミングコードやメモを保存，公開できるようにしたウェブサービスが GitHub である．Github を利用することにより，デバイス間や，研究室の仲間内での共有も可能となる．

6.2　my_help と GitHub の連携

my_help の git コマンドを用いることで GitHub のレポジトリに作成したメモをあげることができる．そのコマンドを使うための設定が次の手順である．

1. GitHub アカウントを作成する．
2. GitHub のアカウントでレポジトリを作成する．
3. 作成したレポジトリに一度メモをあげる．my_help で作成した help(メモ) を保存しているディレクトリの .my_help に移動する以下のコマンドを入力する．

   ```
   $ git add (GitHubにあげたいfile名)
   $ git commit -m ’add file’
   ```

   リモートリポジトリに反映させる前に、リモートリポジトリの情報を追加する．以下のコマンドを入力する．

   ```
   $ git remote add origin (作成したレポジトリのアドレス)
   ```

   最後に GitHub にメモをあげるために以下のコマンドを入力する．

   ```
   $ git push origin master
   ```

以上で my_help と GitHub の連携は完了し，git コマンド使用可能となる．

6.3 開発者側とテスターの目的の相違による問題

開発者である西谷は図6.1のようにgitコマンドを用いて作成したメモをGitHub上にあげ，そのメモを個人のデバイス間での共有を目的としている．しかし，テスターである私は図6.2のようにGitHub上にあげたメモをゼミ間，開発者間などのpublic間での共有を目的としていた．

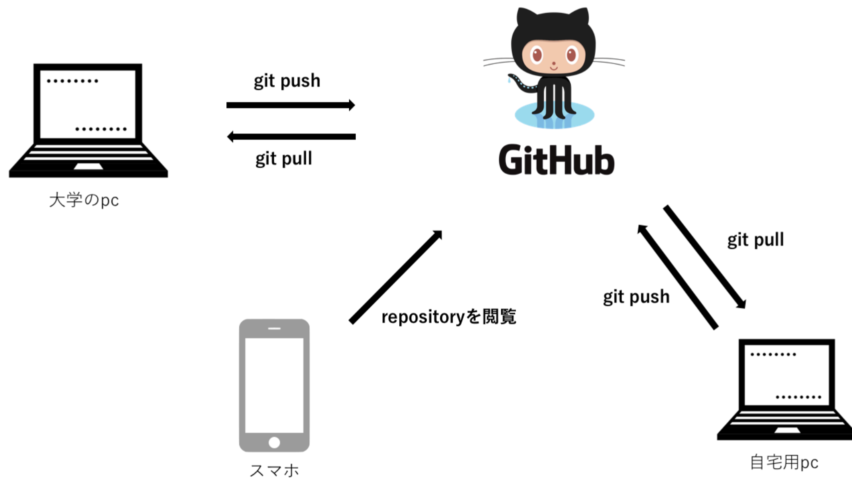

図 6.1: deveice 間での共有.

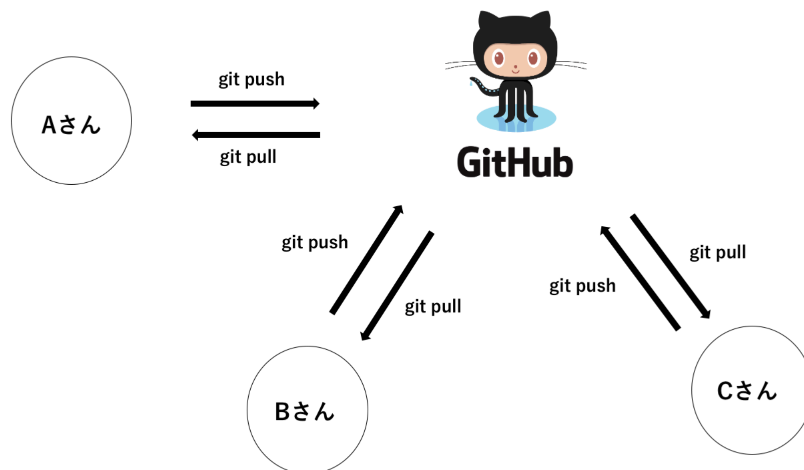

図 6.2: public 間での共有.

my_help のgitコマンドの機能は作成した全てのメモをGitHub上にあげるというものである．私の使用目的としてはpublic間でのメモの共有だったため，あげる必要のないprivateなメモをGitHub上にあげてしまうという問題が発生した．これを改善するために

1. メモをあげるレポジトリをprivateに設定する．

2. 指定したメモを GitHub 上にあげることが出来るようにコードを書き換える．

以上の 2 点を検討した．1 つ目の場合，レポジトリを作る際に private に設定すると，レポジトリを public に公開されないように出来る．さらに，コラボレーターとして設定した者は private のレポジトリを見ることが出来るため，開発者側とテスターのどちらの目的も果たすことが出来る．しかし，private なメモをコラボレーターは見ることが出来るため根本的な問題の解決とは言えない．そのため 2 つ目のメモを指定できるようにコードを変更するという方法をとった．

6.4 結果

以下が変更後の git コマンドのプログラムコードである．

```
 1  desc "git [push|pull]", "git push or pull"
 2  def git(push_or_pull,*args)
 3    p push_or_pull
 4    invoke :setup
 5    argument_size =  args.size
 6    Dir.chdir($control.local_help_dir) do
 7      case push_or_pull
 8      when 'push'
 9        if argument_size == 0
10   comms = ['git add -A',
11    "git commit -m 'git push from my_help'",
12    "git push origin master"]
13        else
14   p args
15   argument_size.times do |i|
16     orgfile = args[i] + '.org'
17     file = File.join($control.local_help_dir,orgfile)
18     if File.exist?(file) == true
19       puts orgfile.green
20       dir = $control.local_help_dir
21       Dir.chdir(dir) do
22         comm = 'git add ' + file
23         c = command_line(comm)
24         puts c
25         puts c.stdout.blue
26         puts c.stderr.red
27       end
28     else
```

```
29        puts (orgfile + " does not existed").red
30      end
31    end
32    comms = ["git commit -m 'git push from my_help'",
33     "git push origin master"]
34          end
35        when 'pull'
36          comms = ['git pull origin master']
37        else
38          raise "my_help git was called by the other than 'push or pull'"
39        end
40        comms.each do |comm|
41          puts comm
42          c = command_line(comm)
43          puts c.stdout.blue
44          puts c.stderr.red
45        end
46      end
47  end
```

コードを用いて変更点を説明する．はじめに GitHub にあげるメモを一つ指定，複数指定できる機能を付け足すために, 2 行目にある引数を配列に変更した．次に引数の要素数が 0 の場合，全てのメモを GitHub 上にあげ, それ以外の場合，指定された名前のメモがある場合のみ GitHub 上にあげ，名前のメモがない場合 29 行目のように,file 名　does not existed と表示されるように変更した．5 行目で配列の要素数を数えており，18 行目で指定された名前のメモが存在しているかを確認している．

第7章　考察と今後の課題

7.1　考察

my_help のテストを作成するにあたって以下のような問題が明らかとなった．

- ruby の知識が必要

RSpec を使用する際や my_help のコマンドがどういう動作をしているか，コードを読み取る際に知識がなければ一つ一つコマンドを実行していかなければならないため時間がかかると考える．そのため，ruby の知識は最低限必要だと感じた．

- テストフレームワークを使えるまでに時間がかかる

本研究ではテストフレームワークのである RSpec,aruba を用いたが，フレームワークならではのメソッドがあり，使用方法を覚えるのに手間取った．aruba においては参考にするものも少なく，知識がない方にとっては習得に時間がかかると考える．

7.2　my_help の今後

my_help は今後も開発者，テスターならびに使用者を通じてより良い機能を搭載し，発展していくと考える．今回 behavior test の記述をしたことにより，my_help のコマンドを実行した際にエラーが出るべきところなど細かい点を修正した．今後 my_help の使用できる環境を拡張する際や，既存のプログラムコードを書き換える際にテストがあることにより思った振る舞いをしているか確認する手助けになると考えている．

第8章　総括

本研究では my_help を発展させるため，開発を進めた．以下に開発内容を示す．

- aruba,command_line を用いたテストの作成
- delete メソッドの改良
- git コマンドの改良

上記の開発により，my_help のテストコードが作成されたため，これから my_help を発展，進化させるためにプログラムコードを書き換えた場合，正常に動作しているかコマンド一つで検証できるようになると考える．さらに，delete メソッドの改良により，テスト対象となるコードには exit を用いることを避けた方が良いことが判明した．

謝辞

本研究を進めるにあたり，様々なご指摘を頂いた西谷滋人教授に対し，深く感謝いたします．また，本研究の進行に伴い様々な助力，協力を頂きました西谷研究室の同輩，先輩方に心から感謝の意を示します．本当にありがとうございました．

参考文献

[1] my_help, Shigeto R. Nishitani, https://rubygems.org/gems/my_help, (accessed on 12 Feb 2020).

[2] データの時間, https://data.wingarc.com/lean-and-agile-14328, (accessed on 12 Feb 2020).

[3] プログラミングマガジン, http://www.code-magagine.com/?p=7745, (accessed on 25 Feb 2020).

[4] aruba, https://github.com/cucumber/aruba, (accessed on 25 Feb 2020).

[5] command_line, https://github.com/DragonRuby/command_line, (accessed on 1 Feb 2020).

[6] my_help, https://github.com/daddygongon/my_help, (accessed on 1 Feb 2020).

[7] "Build Awesome Command-Line Applications in Ruby2", David Bryant Copeland, (Pragmaic Bookshelf,2013).

[8] vagrant_download, https://www.gnu.org/software/libc/manual/html_node/Exit-Status.html, (accessed on 1 Feb 2020).