卒業論文

Arubaを使ったmy_helpのCLIテスト

関西学院大学理工学部

情報科学科 西谷研究室

27018542 東畑萌子

2022年3月

目 次

図 目 次

第1章　序論

1.1　背景

本研究室では主に Ruby を用いてソフトウェアの開発や物理シミレーションの研究を行なっている．その中に 2016 年から開発を進めている my_help というアプリケーションがある．

1.2　my_help とは

CUI(Charactar User Interface), あるいは CLI(Command Line Interface) で利用するヘルプソフトとして my_help がある．my_help は

> CUI(CLA) ヘルプの Usage 出力を真似て，user 独自の help を作成・提供する gem[1].

であり，講義のメモや自身の予定や課題進捗を保存することができる．世の中には多くのメモ機能があるが，その中でも my_help の特徴として

- 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

> memo ソフトでは，検索が必要となりますが，my_help は key(記憶のとっかかり) を提供することが目的です． RPG でレベル上げとかアイテムを貯めるようにして， プログラミングでスキルを発展させてください [1].

である．具体的に解消できる問題点としては，

CUI や shell，何かのプログラミング言語などを習得しようとする初心者は，command や文法を覚えるのに苦労します．少しの key(とっかかり) があると思い出すんですが，うろ覚えでは間違えて路頭に迷います．問題点は，

- man は基本的に英語
- manual では重たい
- いつもおなじことを web で検索して
- 同じとこ見ている
- memo しても，どこへ置いたか忘れる [1].

実際使用していて便利な点として以下の 2 つが挙げられる．

- どのディレクトリーにいても my_help というコマンドを打つといつでもアプリケーションを起動させることができ，自分が見たいメモを見ることができる．

```
> my_help list
```

- list を出力するコマンドを入力すると題名だけではなく，具体的に何が書かれてあるか 1 文でまとめて表示してくれる．

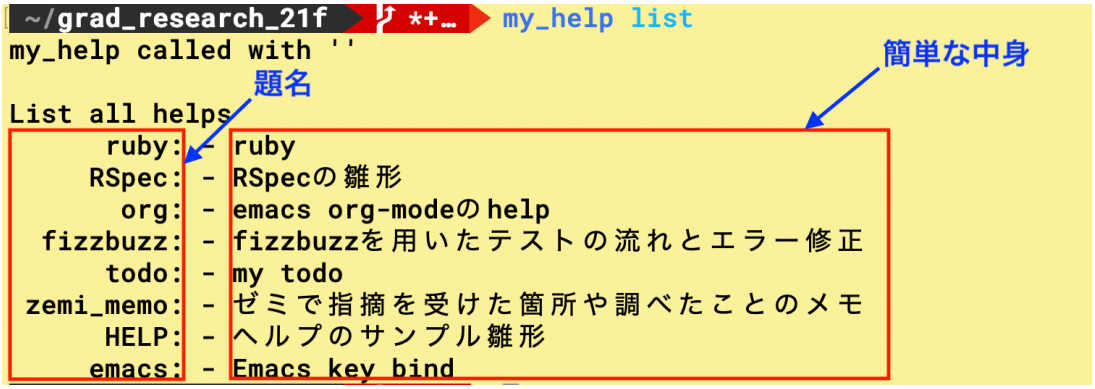

図 1.1: ：my_help list 表示例.

1.3 目的

my_help の問題点として my_help にはテストが存在しないということが挙げられる．しかしオープンソースソフトウェア (OSS) ではテストを書くことが必須である．なぜなら OSS ではシステムを更新していくにつれて，仕様が変更されていく．その変更によって元々正常に動作していた機能が動かなくなってないかを確認することができるからである．また，新しく追加した機能も正常に動作しているかテストを追加することによって確かめることができる．以上の理由から本研究では my_help というアプリケーションのテストを行なっていく．具体的には CLI アプリケーションのテスティングフレームワークの一つである Aruba を使ってテストを行なった．

第2章　手法

2.1　Testing Framework

Kent Beck らが提唱している eXtreme Programming(以下 XP) と呼ばれている新しい開発手法がある．この開発手法で重要なアイテムと言われているものが Testing Framework である．Testing Framework とは

> ソフトウェアのテスト用プログラムを簡単に作成し，テストの実行を支援するためのツールである．もっと言えばテストを楽しくする為のツールである [2].

XP には大きく 2 種類ある．

2.2　acceptance test と unit test

- acceptance test(受け入れテスト)

  ソフトウェアのユーザー視点で行われるテストだ．そのソフトウェアが実際に使われるストーリーを元にしてテストは実施される．テストはそのソフトウェアのユーザーの視点で行われるテストだ [2].

CLI を直接テストを行うものである．本研究では主にこの acceptance test を行なっていくことにする．

- unit test(ユニットテスト)

  ソフトウェアの開発者視点で行われる．Unit test では，ソフトウェアの個々の部品であるクラスがテストされる．テストはそのクラスの開発者が行う [2].

このテストにおいては同じ西谷研究室に所属している大寺が担当している．

2.3 RSpec

RSpec は，Ruby コードをテストする DSL(Domain-Specific Language) のテストツールである．数ある testing framework の中から本研究では RSpec を採用した．理由としては，

1. Ruby 言語を基盤として作られたツールであることから，Ruby の基本的な文法を学習している者にとって容易に使用できる．

2. DSL で記述することができることから，決まった形を使うことができる場合が多い．その為テストを初めて書く者にも適している．

2.4 Aruba

Aruba は testing framework で CLI をテストする際に簡単にテストが書けるように支援する拡張である．Bash、Python、Ruby、Java など、あらゆるプログラミング言語で実装されたコマンドライン・アプリケーションをテストを行うことができる．このような理由から本研究では CLI テストを支援する Aruba gem を用いたテストを記述していくこととする．

第3章　環境構築

3.1　my_help 環境構築

初めに my_help の中身を編集できるようにする．その為に github サイト内 `https://github.com/daddygongon/my_help`にある my_help のページを開く．サイト内の右上の code から，SSH アドレスをコピーする．

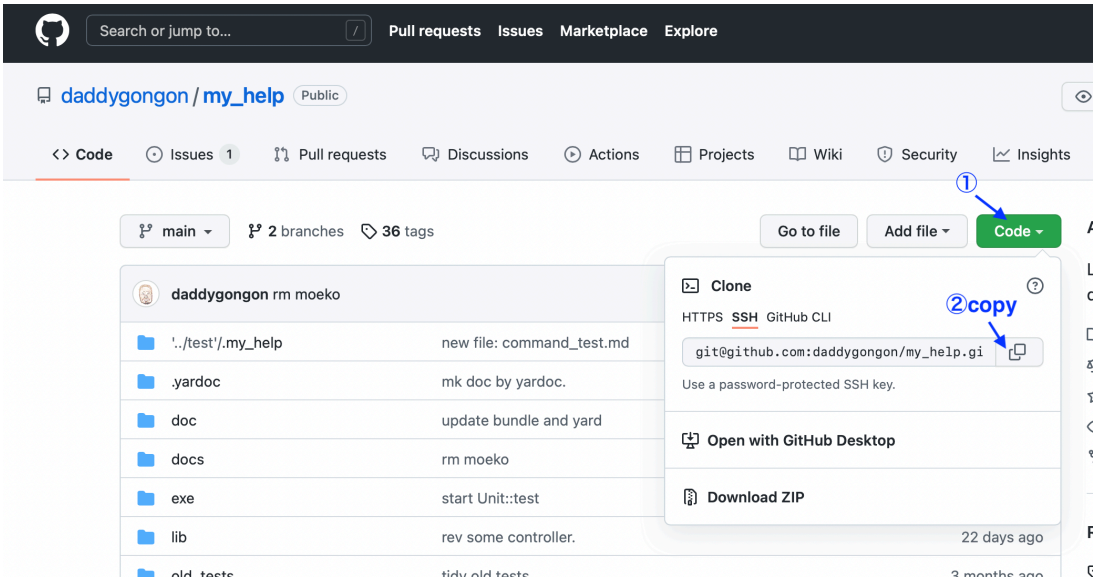

図 3.1: ：github にある my_help ページ．

`> git clone` 先ほどコピーした SSH のアドレスを貼り付ける

以上の作業で my_help を編集できる．しかし，複数の人が my_help を編集することが出来る為，pull した時に誰がどこの書き換えを行なったのか分からなくなってしまう．また，自分が編集することによって動かなくなってしまった時，そのまま pull してしまうと，主となるオリジナルが動かなくなってしまい，混乱が生じてしまう．これを防ぐために自身が編集したものを更新するか否かオリジナルに申請する機能が git_hub にはあり，その機能が pull request である．ここで状況確認を行う．

```
> git remote -v
origin  git@github.com:daddygongon/my_help.git (fetch)
origin  git@github.com:daddygongon/my_help.git (push)
```

上記の状態は origin が daddygongon(西谷) となっており，この状態では，my_help のコードを閲覧することはできるが，書き換えることができない．これを書き換えができるようにする作業が fork と呼ばれるものである．

> fork はユーザーが管理するリポジトリのコピーです．フォークを使えば，オリジナルのリポジトリに影響を与えることなくプロジェクトへの変更を行えます．オリジナルのリポジトリから更新をフィッチしたり，プルリクエストでオリジナルのリポジトリに変更をサブミットしたりできます [3].

fork をする手順として, my_help オリジナルのページを開く

`https://github.com/daddygongon/my_help` 以下の画像の右上の fork を押すとアカウントを選択する画面が出る．そこで自分のアカウントを選択すると fork が完了する．

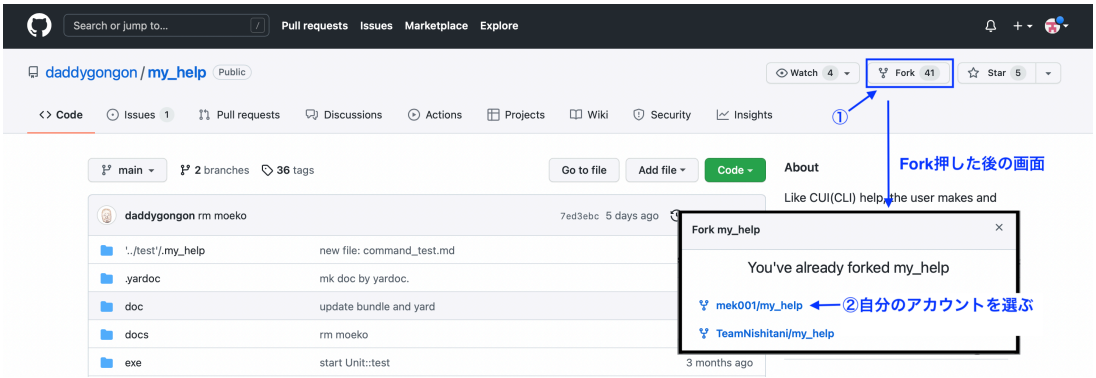

図 3.2: ：fork する際の github/my_help 画面.

ターミナルに戻り，origin を自分のアカウントにする為に，一度今ある origin(daddygongon) を消す．

```
> git remote rm origin
```

my_help の自分が fork したページを開き，`https://github.com/mek001/my_help` code の SSH をコピーする．origin を自分のアカウントに変えるために，以下を実行．

```
> git remote add origin [SSH コピーしたものを貼り付け]
```

fork したアカウントと同期する上流 (upstream) リポジトリ (ここでは daddygongon) を指定する．

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

確認すると origin が自分のアカウント．そして upstream が西谷のアカウントになっている為，西谷が merge した branch を upstream から pull することができるように設置しておく．

```
> get remote -v
origin  git@github.com:mek001/my_help.git (fetch)
origin  git@github.com:mek001/my_help.git (push)
upstream    git@github.com:daddygongon/my_help.git (fetch)
upstream    git@github.com:daddygongon/my_help.git (push)
```

3.2 RSpec 環境構築

次に RSpec の環境構築をしていく．my_help を編集できるようになっただけでは RSpec, Aruba が入っていない為，テストを行うことができない．bundle で test として RSpec をしてすると自動で生成されるが，本節では RSpec を手動で環境構築する手順を紹介しておく．まず，spec ファイルを作成．

```
> mkdir spec
```

この spec ファイルの中に 2 つファイルを作る．1 つ目が.spec という隠しファイルであり，ここには出力 format などを設定するものである．

```
  --format documentation
  --color
  --require spec_helper
```

2 つ目が cli_spec.rb という主に test coding していくファイルとなる．以下は version のテスト例とする．

```
require 'spec_helper'

RSpec.describe 'my_help', type: :aruba do
      context 'version option' do
before(:each){run_command('my_help version')}
it { expect(last_command_started).to be_successfully_executed}
it { expect(last_command_started).to have_output(/1.0b/) }
      end
  end
```

次に gemspec ファイルを開いて,rspec を使えるようにするためにコメントアウトを外す.

```
> s.add_development_dependency 'rspec'
```

そして本研究ではテストを Thor と Aruba を用いる為この 2 つを追加する.

```
> s.add_runtime_dependency 'thor'
> s.add_development_dependency 'aruba'
```

gemspec に書き込んだ内容を更新する.

```
> bundle install
```

3.3 Aruba のセットアップ

本研究では Aruba の RSpec 版を使用する．本節ではセットアップの手順を示す.

```
> bundle exec aruba init --test-framework rspec
```

すると自動的に Gemfile に Aruba が追加される．また，spec/support/aruba.rb が自動生成される．そこで前節で書いたテストを動かすと，以下のような Aruba が入っていないというエラーが出てしまう.

```
  $bundle exec rspec spec/cli_spec.rb
  An error occurred while loading spec_helper. - Did you mean?
```

```
    rspec ./spec/spec_helper.rb
Failure/Error: require 'aruba/rspec'
LoadError:
  cannot load such file -- aruba/rspec
~ 以下省略 ~
```

そこで install されているものを確認してみると，install していた aruba が消去されていた．

```
> bundle install
Using rake 13.0.6
Using bundler 2.2.23
Using thor 1.1.0
Using fizzbuzz 0.1.0 from source at `.` and installing its executables
Bundle complete! 2 Gemfile dependencies, 4 gems now installed.
Bundled gems are installed into `./vendor/bundle`
```

Gemfile の中身を以下のように変更するとテストが成功した．

```
source 'https://rubygems.org/'
gemspec
```

```
~/my_help  * bundle exec rspec
my_help called with ''

my_help
  version option
    is expected to be successfully executed
    is expected to have output: /1.0b/

Finished in 0.43257 seconds (files took 0.11245 seconds to load)
2 examples, 0 failures
```

図 3.3: ：テストが成功した時のターミナル表示．

以上の手順で Aruba gem を使ったテストを行う環境ができた [4]．

第4章　結果

4.1　テストするcommand

本研究ではCLIをテストすることから，最初にmy_help独自のcommand一覧を示す．

```
> my_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
  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
Options:
  d, [--dir=DIR]
```

これらのcommandを順にテストしていく．

4.2　versionテスト

まず環境構築の章の例で挙げたversionからテストを行なっていく．CLIテストでは実際に出力されるものと，出力して欲しいものが同じであるか否かを調査するという理由から，出力が短いversionテストが行い易いと考えた．なので最初のテストとしてあげた．以下は実際にversion commandを動かした際の出力である．

```
> my_help version
1.0b
```

この出力結果を参考にテストを書いていく．

```
RSpec.describe 'my_help', type: :aruba do
  context 'version option' do
    before(:each){run_command('my_help version')}
    it { expect(last_command_started).to be_successfully_executed}
    it { expect(last_command_started).to have_output(/1.0b/) }
  end
end
```

上記のコードの説明として一行目の RSpec.describe 'my_help', type: :aruba do は RSpec で my_help のテストを Aruba を用いて書くことを示している．次に context は条件が異なるテストをグループ分けする際に用いる．これを用いることによってどこにどの command のテストが書かれているか誰が見てもわかることが出来ることから非常に便利なものである．次の befor は次の 2 行を実行させるために書くものであり，実際に挙動させたい command を run_command の後に打つことによってその command が動く．次の 2 行では１つ目の it で前のコマンドを実行すると，うまく実行されることが期待され，２つ目の it では前のコマンドの出力が 1.0b を含んだ正規表現であることを期待している．以下は成功した際のテスト例である．

```
~/my_help  master *  bundle exec rspec
my_help called with ''

my_help
  version option
    is expected to be successfully executed
    is expected to have output: /1.0b/

Finished in 0.43257 seconds (files took 0.11245 seconds to load)
2 examples, 0 failures
```

図 4.1: : my_help version テストが成功した時のターミナル表示.

結果の表示については，アプリケーション名，テスト名，比較した出力が階層構造と

なっており，テストが増えていったとしても見やすい構造であると考える．また，テストが成功した際には緑色，失敗した際には赤色の文字で表示され一目見てわかるようになっている．そして，失敗したエラー内容も赤文字でテストを行なった順番で書かれている為，どのテストにどのエラーが書かれているか一目で判断することができる．

4.3　テストにおけるディレクトリー指定 (set_editor テスト)

次のテストとしてエディターの変更を行うコマンド set_editor についてテストを行なった．

このテストに限ったことではないが，受け入れテストでは CLI をテストすることから各々のテスターの環境に依存してしまう．このことによって同じテストを実行しているが，テスターによってはテストが成功する環境の方もいれば，失敗してしまう環境の方もいるという問題が発生してしまう．このような結果を招いてしまわないように-d でディレクトリーを統一することによって，環境に依存したテスト結果の差異が出ないようにすることにした．

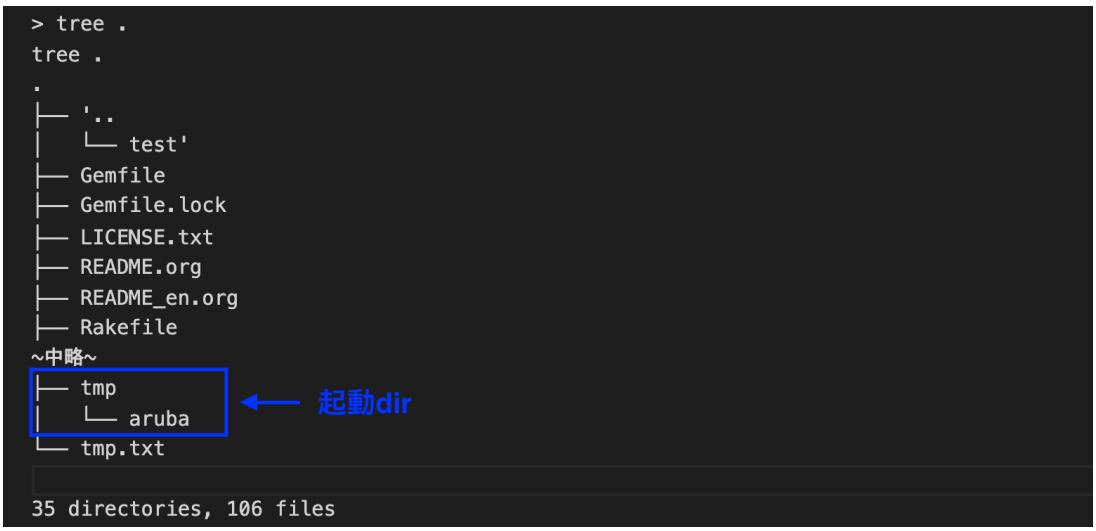

図 4.2: ：my_help における起動ディレクトリーの位置を tree 構造で表示させた．

aruba でテストを実行すると起動ディレクトリーは./tmp/arubaである．統一するディレクトリーとして，my_help の下の test へ指定する．このディレクトリーではテスター専用に作成された各コマンドの出力例がコーディングされている．その為入力するときは

```
my_help [実行したい command] -d='../../test'
```

といったように指定する．

例えば，エディターを emacs に変更したい際は

```
> my_help set_editor emacs -d=\'../../test\'
```

と入力すると，

```
my_help called with 'set_editor emacs -d='../../test''
> default target dir : '../../test'
set editor 'emacs'
```

と表示され，次回から編集を行う際には emacs が立ち上がることになる．そして，テストの中身は version テストを行なった時と同じように書く．

```
context 'set_editor option' do
    expected = <<~EXPECTED
   my_help called with 'set_editor emacs -d=../../test'
   > default target dir : ../../test
   set editor 'emacs'
EXPECTED
    before(:each){run_command("my_help set_editor emacs -d=\'../../test\'")}
    it { expect(last_command_started).to be_successfully_executed}
    it { expect(last_command_started).to have_output(expected) }
  end
```

しかし，これでは以下のようなエラーが出てしまう．

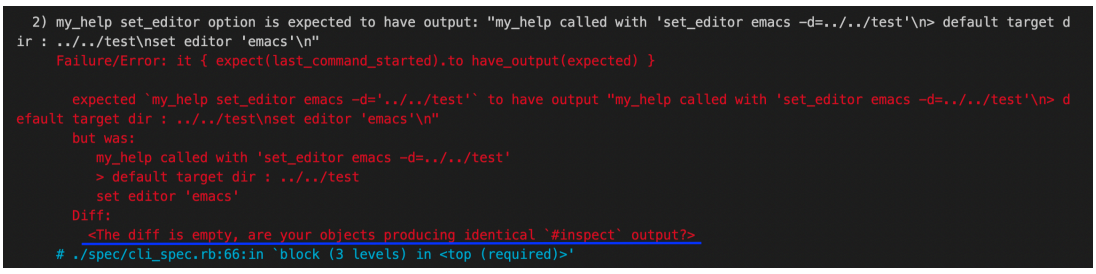

図 4.3: ：diff empty エラーが出た際のターミナルの表示．

diff(差分) は無いと表示されているがエラーが出てしまう．これは標準出力 (stdout) を用いることによって回避できる．stdout を使用するには次のような条件がある．

例えば#last_command_started という一つのコマンドの#stdout にアクセスする前には#stop_all_commands が必要になるだろう [5].

これに沿って次のようにテストの書き換えを行うとテストを成功させることができた.

```
1 context 'set_editor option' do
2     expected = <<~EXPECTED
3   my_help called with 'set_editor emacs -d=../../test'
4   > default target dir : ../../test
5   set editor 'emacs'
6 EXPECTED
7     before(:each){run_command("my_help set_editor emacs -d=\'../../test\'")}
8     before(:each) { stop_all_commands }
9     it { expect(last_command_started).to be_successfully_executed}
10    it { expect(last_command_started.stdout).to eq(expected) }
11   end
```

8 行目で記述した”stop_all_commands”が注にあった対処法である.

4.4 正規表現を用いたテスト (my_help テスト)

これまでのテストは比較的出力が短いものをテストしてきたが，テストには出力が長くなるものも存在する．この章の冒頭にあった my_help 内の command の一覧を表示する my_help コマンドをテストする．先ほどと同様に標準出力を用いた書き方で書くと，以下の画像のようなエラーが出てきてしまう．これは空白や改行が期待されているものと，出力結果を比較した際に差異が出てきてしまっている．このような現象を diff という．これが本テストでは様々な箇所で出てしまう為，長い出力を diff なしで通すことは難しい.

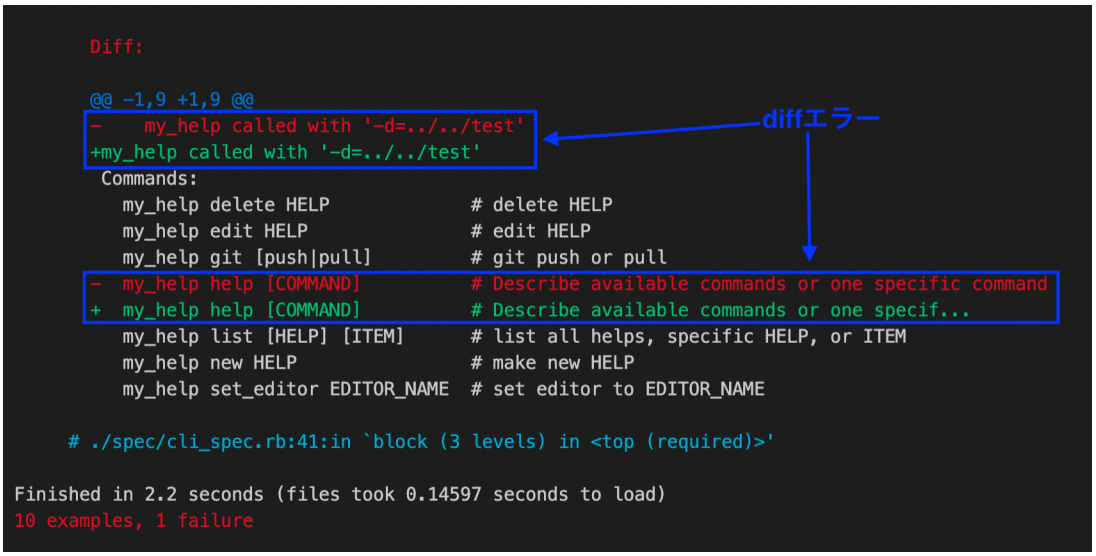

図 4.4: : diff エラーが出た際のターミナルの表示.

長い出力でも全て一致させなければならないテストもあるが，本節でのテストは前節にあったようなディレクトリーの指定をしてもしなくても出力結果が変わらないことから，出力の最初や一部を抜き出し，一致していればテストが成功するように書くことができる正規表現を用いることにした．この方法の利点として，

> どのサンプルも単体ではそれほど脆弱ではなく，無関係な変更によって失敗する可能性が低いという利点があります [6].

ここで表現されている無関係な変更による失敗というのはディレクトリーの指定に左右されず常に同じ出力をする箇所で起こる diff エラーのことである．

このような背景から my_help テストを書いた．

```
1   context "my_help option" do
2     expected = /^my_help called with ''/
3     let(:my_help){ run_command("my_help") }
4     it { expect(my_help).to have_output(expected) }
5   end
```

/で囲うことによって正規表現を用いることができる．正規表現には記号によって意味が決められている．例えば本テストで使用している”^”は文の冒頭を意味する．以下が記号とその意味の一覧である．

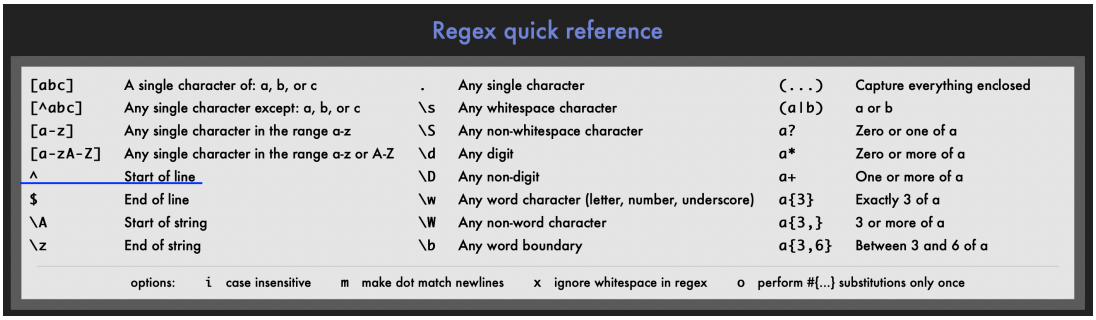

図 4.5: ：正規表現記号一覧 [7].

4.5 全文一致させるテスト (list テスト)

前節では全文一致させるには余計な手間と時間がかかる為，一部を抜き出して一致させる方法を取った．しかしこの list テストではディレクトリー指定をする場合と，ディレクトリーを指定せずに出力する場合とでは大きく異なってしまう．

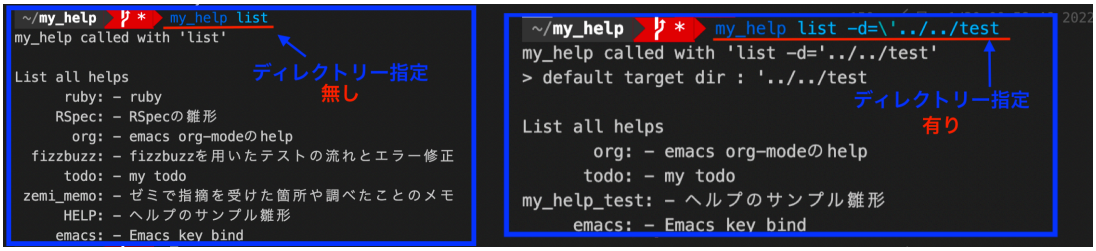

図 4.6: ：list コマンドにおけるディレクトリー指定の有無比較.

その為本テストではディレクトリー指定を行い全文一致が必要となる．どうしても diff エラーが出てしまうが，地道に改行や空白を調節してエラーを消していく．

```
context "command list" do
  expected = <<~EXPECTED
my_help called with 'list -d=../../test'
> default target dir : ../../test

List all helps
       org: - emacs org-mode の help
      todo: - my todo
my_help_test: - ヘルプのサンプル雛形
```

```
    emacs: - Emacs key bind
EXPECTED
  before(:each) { run_command ("my_help list -d=\'../../test\'") }
  before(:each) { stop_all_commands }
  it { expect(last_command_started).to be_successfully_executed }
  it { expect(last_command_started.stdout).to eq(expected) }
end
```

4.3 節で記述した通り本テストは出力が 1 行では収まらない為，EXPECTED で囲っている．図 9 のディレクトリー指定有りの画像を参照することや，diff エラーの中身を参照することで少しずつエラーを解消できる．手間が少々かかってしまうが，全文一致させなければならないテストも存在するのでこのような方法を用いた．

第5章　総括

本研究では my_help の中にある CLI を RSpec で Aruba という支援ソフトを用いることによってテストを書き進めてきた．書き進めてきた内容は以下の通りである．

- version のテストを行い，Aruba gem の振る舞い，RSpec の書き方を簡単にではあるが習得することができた.
- 環境依存してしまうテストにおいては test ディレクトリーを作成し，そこにディレクトリー指定を行うことで解消できた.
- 出力が長くなってしまう関係で，無関係な変更によって失敗する可能性がある my_help コマンドに関しては，重要な一部分のみを抽出し正規表現でマッチさせることにより解消することができた.
- 全文一致が必要なテストに関しては地道に diff エラーを解消することによってテストを成功させることができた.

テストを初めて書く者にとってはテストの仕組みや，挙動を理解するのに時間を要してしまうと考える．Aruba に関しての参考資料も日本語のものはほとんどなく，また英語の資料も他の gem と比べると少なく調べるのに手間がかかってしまう．しかし正規表現を用いたり，ディレクトリー指定をしたりと，ここで示した通り, ある程度雛形が定まってくると次々とテストが完成させることができる．

今後は，未だテストが行われていない edit や new コマンドのテストを進めていくことで，my_help の信頼性の向上が測れると考える．

謝辞

本研究を行うにあたり，西谷教授には研究の運びや方法の相談を沢山させていただきました．どのようなことでも丁寧かつ熱心なご指導を賜りました。深く感謝申し上げます。最後に、同研究室の皆様には、研究を進めるにあたり多大なご助言、ご協力頂きました。ここに感謝の意を表します。本当にありがとうございました.

参考文献

[1] githubmyhelp, - https://github.com/daddygongon/my_help/, (accessd on 20 Jan 2022).

[2] 助田雅紀,"Ruby を 256 倍使うための本",(アスキー,2001).

[3] GithubFork, - https://docs.github.com/ja/pull-requests/collaborating-with-pull-requests/working-with-forks/about-forks/,(accessd on 20 Jan 2022).

[4] Aruba gem で fizzbuzz テストを書いてみた, - https://qiita.com/mek001/items/4c46af014f66b201288c,(accessd on 26 Jan 2022).

[5] Relish stdout command, - https://relishapp.com/philoserf/aruba/docs/command/access-stdout-of-command, (accessd on 29 Jan 2022).

[6] David Chelimsky 他,"The RSpec Book",(翔泳社,2012).

[7] Rubular, - https://rubular.com,(accessd on 5 Feb 2022).