卒業論文

ユーザメモソフト my_help のビヘイビアテスト開発

関西学院大学　理工学部　情報科学科

2535 那須比呂貴

2017 年 3 月

指導教員　　西谷　滋人　教授

目次

- {{attach_anchor(my_help_nasu.pdf,my_help_nasu)}}

1 序論

プログラム開発では，統合開発環境がいくつも用意されているが，多くの現場では，terminal 上での開発が一般的です．ところが，プログラミング初心者は terminal 上での character user interface(CUI) を苦手としています．プログラミングのレベルが上がるに従って，shell command や file directory 操作, process 制御に CUI を使うことが常識です．

この不可欠な CUI スキルの習得を助けるソフトとして，ユーザメモソフト my_help が ruby gems に置かれています．この command line interface(CLI) で動作するソフトは，help を terminal 上で簡単に提示するものです．また，初心者が自ら編集することによって，すぐに参照できるメモとしての機能を提供しています．これによって，terminal 上でちょっとした調べ物ができるため，作業や思考が中断することなくプログラム開発に集中できること，さらに初心者のスキル習得が加速することが期待できます．

しかし，Ruby gems として提供されているこのソフトは，動作はしますがテストが用意されていません．熟練した開発者は，テストを見ることで仕様を理解するのが常識です．今後ソフトを進化させるために共同開発を進めていくには，仕様や動作の標準となるテスト記述が不可欠となります．

そこで，本研究では，ユーザメモソフトである my_help のテストを開発することを目的とします．本研究では、テスト駆動開発の中でも，ソフトの振る舞いを記述します．ビヘイビア駆動開発 (Behavior Driven Development:BDD) に基づいてテストを記述していきます．Ruby において、BDD 環境を提供する標準的なフレームワークである Cucumber と RSpec を用いて，my_help がどのような振る舞いをするのかを記述します．Cucumber は自然言語で振る舞いを記述することができるため，開発者や，ユーザにとっても，わかりやすく振る舞いを確認することができます．

2　先行研究，方法

ここでは、本研究で使用する cucumber の特徴について詳述する．cucumber はビヘイビア駆動開発 (bdd) を実現するフレームワークである．まずは bdd の現れた背景や現状を示した後、Cucumber の記述の具体例を示して，その特徴を詳述する．さらに，本研究の対象となる my_help の振る舞いを使用法とともに示す．

2.1 RSpec と BDD について

ビヘイビア駆動開発 (Behaviour-Driven Development： BDD) は，テスト駆動開発 (Test-Driven Development：TDD) の工程への理解を深め，それをうまく説明しようとして始まりました．TDD の持つ単語のイメージが構造のテストを中心とするべしというのに対して，BDD はソフトの振る舞いに中心をおきなさいという意図があります．この違いが，初めに考えるべきテストの性質を変化させ，構造ではなく振る舞いを中心にテストを構築するという意識をもたせてくれます．

さらに，ソフトの中で，オブジェクト同士がコミュニケーションをとるように，実世界において開発チームやテストチーム，あるいはドキュメントチーム間のコミュニケーションの取り方をシステムで提供しようというのが BDD のフレームワークです．Cucumber と RSpec はこれを実現する一つのシステムとして提供されています．

RSpec と Cucumber の関係を図に示しました．これは，RSpec 本から書き写した図です [1, pp.9]．RSpec でテストを書くと一つ一つの function あるいは method レベルで Red, Green, Refactoring を行うべしという意図があります．一方で，もっと大きな枠組み，つまりシステムレベルでもこれらのステップは必要です．ところが，それを RSpec で書くのには無理があります．このレベルのテスト記述をしやすくするのが，Cucumber です．そこでも Red, Green, Refactoring が必要で，そこでサイクルが回ることを意図しています．

BDD の基本的な考え方は次の通りまとめられています．

> BDD の目的は，ソフトウェアが使われる状況を説明するための言語を単純化することで，ソフトウェア開発チームのコミュニケーションを後押しすることです．つまり，あるコンテキストで (Given)，あるイベントが発生すると (When)，ある結果が期待されます (Then)．BDD における Given, When, Then の 3 つの単語は，アプリケーションやオブジェクトを，それらの振る舞いに関係なく表現するために使われる単純な単語です．ビジネスアナリスト，テスト担当者，開発者は皆，それらをすぐに理解します．これらの単語は Cucumber の言語に直接埋め込まれています [1, pp.3-6].

手順を書き直すと次の通りです．

まず Cucumber で一つのシナリオに焦点を当てて，その振る舞いを記述する feature を書きます．一つずつつぶしていくのがこつです．一つの feature が書けたら、次に，そ

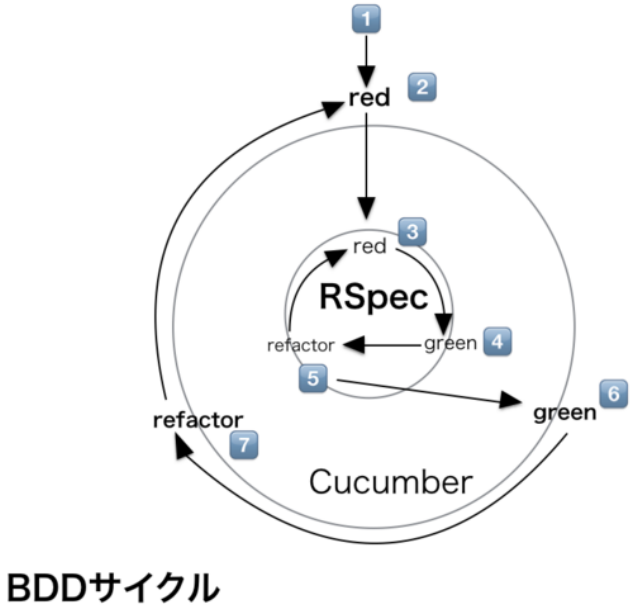

図 1　RSpec と Cucumber の Red-Green-Refactoring サイクル間の関係.

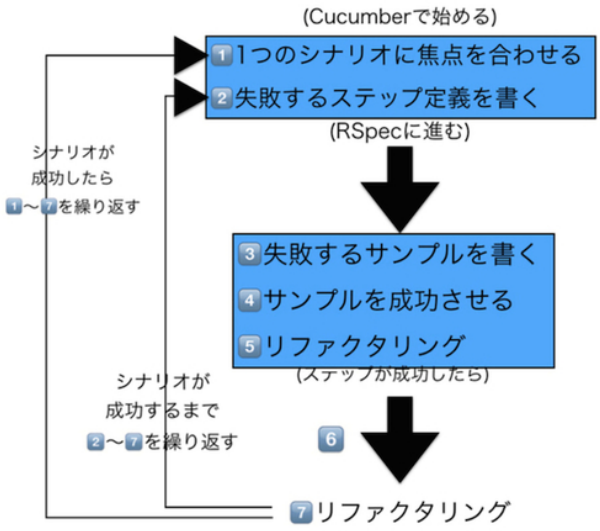

図 2　RSpec と Cucumber の手順.

れぞれ feature を実現するステップに分けて仕様を決めて行きます．これは TDD の red green refactoring の前に行う作業,「仕様をきめる」に対応しています．このプロセスが終了したら，RSpec に行きます．RSpec では実際にテストコードを書き，ここでも red, green, refactoring を行います．RSpec が成功したら，Cucumber の refactoring を行います．

2.2 Cucumber について

2.2.1 概要

Cucumber が提供する BDD の内容をまとめると

> BDD はフルスタックのアジャイル開発技法です．BDD は ATDP(Acceptance Test-Driven Planning) と呼ばれる Acceptance TDD の一種を含め，エクストリームプログラミングからヒントを得ています．ATDP では，顧客受け入れテストを導入し，それを主体にコードの開発を進めて行きます．それらは顧客と開発チームによる共同作業の結果であることが理想的です．開発チームによってテストが書かれた後，顧客がレビューと承認を行うこともあります．いずれにしても，それらのテストは顧客と向き合うものなので，顧客が理解できる言語とフォーマットで表現されていなければなりません．Cucumber を利用すれば，そのための言語とフォーマットを手に入れることができます．Cucumber は，アプリケーションの機能とサンプルシナリオを説明するテキストを読み取り，そのシナリオの手順に従って開発中のコードとのやり取りを自動化します [1, 7pp.].

と記されている．

2.2.2 features

Cucumber では先の引用にある通り、振る舞いをシナリオとしてまず記述します．シナリオとは，一つひとつの振る舞いを，主に Given(前提), Then(もし), When(ならば) に分けて features を記述します．次に，英語の features のひな形を示します．

```
1 % cat ./featrues/sample_e.feature
2 Feature: Description of feature
3
4 Scenario: Description of scenario
5   Given I want to explain scenario
6   Then I investigate
7   When I know the meaning
```

ファイルの先頭で,

```
# language: ja
```

と記すと日本語での keyword が認識されます．下記に my_todo に対する features ファイルの具体例を示します．

```
# language: ja

機能: todoの更新を行う
自分のするべきことを書き込むためにtodoを更新する

シナリオ: コマンドを入力してtodoを更新していく
 前提  todoを編集したい
 もし  "my_todo␣--edit"と入力する
 ならば editが開かれる
 かつ  自分のtodoを書き込む
```

このように Feature, Scenario, Given, Then, When などの cucumber が解釈する大文字で始まる keywords に対して，それぞれ機能，シナリオ，前提，もし，ならばなどの単語があてられています．この機能により，より自然な日本語で features を書くことができ，顧客にもわかりやすく，開発者も書きやすくなっています。

feature ファイルで用意されている keyword は

```
cucumber --i18n LANG
```

によって表示される．LANG=ja, en に対しては表 1 の通りになっています．

2.2.3 Cucumber,RSpec インストール

まず rspec を gem で install する．

1. gem install rspec –version 2.0.0
2. rspec –help

と入力して

```
/Users/nasubi/nasu% rspec --help
Usage: rspec [options] [files or directories]
```

のような表示がされていれば install ができている．次に，cucumber を install する

1. gem install cucumber –version 0.9.2
2. cucumber –help

と入力して

```
cucumber --help
Usage: cucumber [options] [ [FILE|DIR|URL][:LINE[:LINE]*] ]+
```

のような表示がされていれば install できている．

表 1　feature ファイルで用意されている keyword の日本語 (ja)，英語 (en) の対応表.

keyword	ja(japanese)	en(english)
feature	”フィーチャ”, ”機能”	”Feature”, ”Business Need”, ”Ability”
background	”背景”	”Background”
scenario	”シナリオ”	”Scenario”
scenario_outline	”シナリオアウトライン”, ”シナリオテンプレート”,”テンプレ”, ”シナリオテンプレ”	”Scenario Outline”, ”Scenario Template”
examples	”例”, ”サンプル”	”Examples”, ”Scenarios”
given	”* ”, ”前提”	”* ”, ”Given ”
when	”* ”, ”もし”	”* ”, ”When ”
then	”* ”, ”ならば”	”* ”, ”Then ”
and	”* ”, ”かつ”	”* ”, ”And ”
but	”* ”, ”しかし”, ”但し”, ”ただし”	”* ”, ”But ”
given (code)	”前提”	”Given”
when (code)	”もし”	”When”
then (code)	”ならば”	”Then”
and (code)	”かつ”	”And”
but (code)	”しかし”, ”但し”,”ただし”	”But”

2.2.4　ディレクトリー構造と使用手順

cucumber は Rubygems の提供する基本 directory 構造での作業を前提としています. その構造を表示すると次のようになります.

```
bob% tree .
.
├── Gemfile
├── Rakefile
├── features
```

```
│   ├── hogehoge.feature
│   ├── step_definitions
│   │   └── hogehoge_step.rb
│   ├── support
│       ├── env.rb
├── lib
│   ├── daddygongon
│   │   ├── emacs_help.yml
│   │   ├── my_todo.yml
├── pkg
├── spec
│   ├── my_help_spec.rb
│   ├── my_todo
│   │   ├── todo_spec.rb
│   ├── spec_helper.rb
│   └── support
│       └── aruba.rb
```

カレントディレクトリ (.) の中に features というサブディレクトリを作成します．その features の中に書きたいシナリオを書いた，hogehoge.feature を作成します．feature の具体例は上記に示してします．

次にシェルを開いて，カレントディレクトリで，

```
cucumber features hogehoge.feature
```

と入力します．そうすると以下のような出力が得られます．

```
Feature: Description of feature

  Scenario: Description of scenario  # features/hogehoge.feature:3
    Given I want to explain scenario # features/hogehoge.feature:4
    Then I investigate               # features/hogehoge.feature:5
    When I know the meaning          # features/hogehoge.feature:6

1 scenario (1 undefined)
3 steps (3 undefined)
0m0.066s

You can implement step definitions for undefined steps with these snippets:

```

```
14 Given(/^I want to explain scenario$/) do
15   pending # Write code here that turns the phrase above into concrete actions
16 end
17
18 Then(/^I investigate$/) do
19   pending # Write code here that turns the phrase above into concrete actions
20 end
21
22 When(/^I know the meaning$/) do
23   pending # Write code here that turns the phrase above into concrete actions
24 end
```

ここではステップ定義に使用することができるコードブロックが表示されています．ステップ定義はステップを作成するための方法です．このサンプルでは，Giver(), When(), Then() の 3 つのメソッドを使ってステップを記述します．これらのメソッドはそれぞれで囲まれた Regexp(正規表現) とブロックを受け取ります．Cucumber はシナリオの最初のステップを読み取り, そのステップにマッチする正規表現を持つステップ定義を探します．その中の対応するステップ定義のブロックを実行します．

これは features ディレクトリの下に step_definitions ディレクトリーにあることになっています．このシナリオを成功させるには，Cucumber が読み込めるファイルにステップ定義を保存する必要があります [1, pp15.]．その内容は次の通り cucumber から自動生成されます．

```
1
2 Given(/^I want to explain scenario$/) do
3   pending # Write code here that turns the phrase above into concrete actio\
4 ns
5 end
6
7 Then(/^I investigate$/) do
8   pending # Write code here that turns the phrase above into concrete actio\
9 ns
10 end
11
12 When(/^I know the meaning$/) do
13   pending # Write code here that turns the phrase above into concrete actio\
14 ns
15 end
```

pending を削除して，そこにあれば良いなと思うコードを記述していきます．ここまでが Cucumber の使用方法のテンプレートです．

2.3 my_help について

my_help は本研究室の西谷が開発したものです．my_help とはユーザメモソフトであり，CUI スキルの習得を助けてくれます．tarminal 上で簡単に提示させることができるため，プログラミングに集中することができるといった特徴があります．また，自分の見やすいように初心者でも簡単に編集することができ，すぐに参照できるメモとしても使うことができます．

2.3.1 my_help の開発動機と特徴

以下は my_help の README からの抜粋です [2].

■概要 my_help - CUI(CLI) ヘルプの Usage 出力を真似て，user 独自の help を作成・提供する gem.

■開発動機 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

などです．

■特徴 これらを gem 環境として提供しようというのが，この gem の目的です．仕様としては，

- 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

hiki でやろうとしていることの半分くらいはこのあたりのことなのかもしれません．

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．

2.3.2 my_help のインストール

github に行って daddygongon の my_help を fork します．

1. git clone git@github.com:daddygongon/my_help.git
2. cd my_help
3. rake to_yml
4. rake clean_exe

sudo bundle exec exe/my_help -m

5. source /.zshrc or source /.cshrc
6. my_help -l
7. rake add_yml

2.3.3 my_help の更新

git hub を用いて my_help を新しくします．

1. git remote -v をする (remote の確認).
2. (upstream がなければ)git remote add upstream git@github.com:gitname/my_help.git
3. git add -A
4. git commit -m 'hogehoge'
5. git push upstream master(ここで自分の my_help を upstream に送っとく)
6. git pull origin master(新しい my_help を取ってくる)

次にとってきた.yml を /.my_help に cp する．

1. cd my_help で my_help に移動．
2. cp hogehoge.yml /.my_help

それを動かすために (sudo)bundle exec ruby exe/my_help -m をする．ここで過去に sudo をした人は permission が root になっているので，sudo をつけないと error が出ます．(sudo で実行していたら権限が root に移行される)

1. 新しいターミナルを開いて動くかチェックする．

3　ビヘイビア駆動開発の実践

Cucumber と RSpec を用いて BDD で my_help のテスト開発を進めて行きました．ビヘイビア駆動開発のコツとして「焦点を合わせる」ことが強調されています．ここでは，焦点を合わせた my_help の中での一つの振る舞いである「todo の更新」を例として詳しく見て行きます．

3.1　todo の更新マニュアル

最初に，todo を更新するときの手順を示します．

1. my_todo –edit を入力して /.my_help/my_todo.yml を開く
2. editor で todo を書き込む（今週やることなら weekly という item を作ってそこに書き込む）
3. 保存して /.my_help/my_todo.yml を閉じる

この振る舞いがきちんとできているのかを BDD でテストを書いていきます．

3.2　Cucumber

以下は todo の更新を行うときの feature です．まず，適当なディレクトリに features というディレクトリを作成します．次に先ほど作成した，features ディレクトリに my_todo.feature を作成します．

```
# language: ja

機能: todoの更新を行う
自分のするべきことを書き込むためにtodoを更新する

 シナリオ: コマンドを入力してtodoを更新していく
          前提 todoを編集したい
          もし "my_todo␣--edit"と入力する
          ならば editが開かれる
          かつ 自分のtodoを書き込む
```

機能とは，このシステムの機能のことを記述します．ここでは，todo を更新するシステムですので，「todo の更新を行う」です．機能の下には，機能の補足説明を記述します．機能の補足説明では，ルールがないので自分がわかりやすいように，記述するのが常識です．シナリオは，その名の通り todo を更新する時のユーザの行動やシステム振る舞いを

前提，もし，ならば，かつ，しかしに分類して記述します．

ここまで feature が記述できたら，次は cucumber コマンドを実行してみます．コマンドは以下の通りです．

```
nasu% cucumber features/my_todo.feature
```

features ディレクトリにある my_todo.feature ファイルを cucumber で実行するという意味です．

実行すると図 3 のようになります．

```
/Users/nasubi/my_help% cucumber features/my_todo.feature
# language: ja
機能: todoの更新を行う
自分のするべきことを書き込むためにtodoを更新する

  シナリオ: コマンドを入力してtodoを更新していく # features/my_todo.feature:7
    前提todoを編集したい                  # features/my_todo.feature:8
    もし"my_todo --edit"と入力する  # features/my_todo.feature:9
    ならばeditが開かれる                  # features/my_todo.feature:10
    かつ自分のtodoを書き込む              # features/my_todo.feature:11

1 scenario (1 undefined)
4 steps (4 undefined)
0m0.049s

You can implement step definitions for undefined steps with these snippets:

前提(/^todoを編集したい$/) do
  pending # Write code here that turns the phrase above into concrete actions
end

もし(/^"([^"]*)"と入力する$/) do |arg1|
  pending # Write code here that turns the phrase above into concrete actions
end

ならば(/^editが開かれる$/) do
  pending # Write code here that turns the phrase above into concrete actions
end

ならば(/^自分のtodoを書き込む$/) do
  pending # Write code here that turns the phrase above into concrete actions
end
```

図 3　step 定義をする前の cucumber の実行結果．

ここでは，1 つの scenario と 4 つの step が失敗しています．まだ step 定義を記述していないので当たり前です．

一度 cucumber を実行したのには理由があります．feature を書いた時点で cucumber を実行すると，ステップ定義の元となるコマンドを，cucumber が自動的に作成してくれるからです．

以下が cucumber から出力されたステップ定義の元となる部分です．

```
前提(/^todoを編集したい$/) do

end
```

```
もし(/^"([^"]*)"と入力する$/) do |command|

end

ならば(/^editが開かれる$/) do

end

ならば(/^自分のtodoを書き込む$/) do

end
```

これをコピーして，features ディレクトリの中で step_definitions ディレクトリを作成し，その中に my_todo_spec.rb を作成し，そこに貼付けます．

ここでもう一度 cucumber を実行してみると図??のように変化が出てきます．

```
/Users/nasubi/my_help% cucumber features/my_todo.feature
# language: ja
機能: todoの更新を行う
自分のするべきことを書き込むためにtodoを更新する

  シナリオ: コマンドを入力してtodoを更新していく # features/my_todo.feature:7
    前提todoを編集したい                  # features/step_definitions/my_todo_spec.rb:1
      TODO (Cucumber::Pending)
      ./features/step_definitions/my_todo_spec.rb:2:in `/^todoを編集したい$/'
      features/my_todo.feature:8:in `前提todoを編集したい'
    もし"my_todo --edit"と入力する  # features/step_definitions/my_todo_spec.rb:5
    ならばeditが開かれる              # features/step_definitions/my_todo_spec.rb:9
    かつ自分のtodoを書き込む          # features/step_definitions/my_todo_spec.rb:13

1 scenario (1 pending)
4 steps (3 skipped, 1 pending)
0m0.030s
```

図 4　default の step 定義を貼り付けた後の cucumber の実行結果．

これは 1 つのシナリオがあり，1 つが pending であり，4 つの step の内 1 つが pending で 3 つが skip したことを表しています．step_definitions の my_todo_spec.rb の pending 部分を書き換えて，step_definitions の記述を進めて行きます．

まず，「前提」を見てみると my_help が何か振る舞いをすることはありません．よって，このままにしておきます．「もし」もユーザが入力するコマンドであり，my_help が何か

振る舞いをすることはないのでこのままにしておきます．次に，「ならば」を見てみるとmy_help が edit を開くという振る舞いをしています．ステップ定義では，あれば良いと思うコードを記述するので私は下記のように記述しました．

```
ならば(/^editが開かれる$/) do
  Mytodo::Edit.new.open
end
```

pending の部分が書けたので，もう一度 cucumber features/my_todo.feature を実行します．すると，図 5 のような結果が返ってきました．

図 5　pending の step 定義を書き込んだ後の cucumber の実行結果．

Cucumber は，エラーが出たステップのすぐ後ろにエラーを表示してくれます．ここで Cucumber でエラーが出たので，この「ならば　edit が開かれる」のシナリオに注目して RSpec に進むことにします．

3.3　RSpec

次に RSpec を使って実際に todo を更新する振る舞いをするコード書いていきます．

そのための準備として，まず spec というディレクトリを作成し，my_todo というサブディレクトリを追加します．次に，このサブディレクトリに todo_spec.rb というファ

イルを追加します．作業を進める過程で，lib/my_todo/my_todo.rb ソースファイルと spec/my_todo/todo_spec.rb スペックファイルが 1 対 1 に対応するといった要領で，並列のディレクトリ構造を築いていきます．この機能は my_help –edit と入力されれば，/.my_help/my_todo.yml が開かれるのでその振る舞いをするコードを書きます．まず todo_spec.rb は下記の通りになります．

```
1  require 'spec_helper'
2
3
4  module Mytodo
5    describe Todo do
6      describe "#open" do
7        it "open␣file␣my_todo.yml"
8      end
9    end
10 end
```

describe() メソッドは，RSpec の API にアクセスして RSpec::Core::ExampleGroup のサブクラスを返します．ExampleGroup クラスはオブジェクトに期待される振る舞いのサンプルを示すグループです．it() メソッドはサンプルを作成します．

完成したコードを下記の通りです．

```
1  require 'spec_helper'
2
3
4  module Mytodo
5    describe Todo do
6      describe "#open" do
7        it "open␣file␣my_todo.yml" do
8            system("emacs␣~/.my_help/my_todo.yml")
9        end
10     end
11   end
12 end
```

spec ディレクトリの my_todo ディレクトリを rspec で実行すると下記のような結果がでました．--color を付け加えるとわかりやすく色づけをしてくれて，見やすくなります．

図 6 を見るとエラーが出てしまっているのがわかります．

```
'<module:Mytodo>': uninitialized constant Mytodo::Edit (NameError)
```

上記のエラーを解決するために，spec ディレクトリの一つ上の構造のディレクトリに lib ディレクトリを作成します．その中に my_todo というディレクトリを作成し，my_todo.rb を作成します．

構造を表示すると以下のようになっています．

```
/Users/nasubi/my_help% rspec spec/my_todo --color
/Users/nasubi/my_help/spec/my_todo/todo_spec.rb:6:in `<module:Mytodo>': uninitiali
zed constant Mytodo::Edit (NameError)
        from /Users/nasubi/my_help/spec/my_todo/todo_spec.rb:5:in `<top (required)
>'
        from /Users/nasubi/.rbenv/versions/2.2.2/lib/ruby/gems/2.2.0/gems/rspec-co
re-3.5.4/lib/rspec/core/configuration.rb:1435:in `load'
        from /Users/nasubi/.rbenv/versions/2.2.2/lib/ruby/gems/2.2.0/gems/rspec-co
re-3.5.4/lib/rspec/core/configuration.rb:1435:in `block in load_spec_files'
        from /Users/nasubi/.rbenv/versions/2.2.2/lib/ruby/gems/2.2.0/gems/rspec-co
re-3.5.4/lib/rspec/core/configuration.rb:1433:in `each'
        from /Users/nasubi/.rbenv/versions/2.2.2/lib/ruby/gems/2.2.0/gems/rspec-co
re-3.5.4/lib/rspec/core/configuration.rb:1433:in `load_spec_files'
        from /Users/nasubi/.rbenv/versions/2.2.2/lib/ruby/gems/2.2.0/gems/rspec-co
re-3.5.4/lib/rspec/core/runner.rb:100:in `setup'
        from /Users/nasubi/.rbenv/versions/2.2.2/lib/ruby/gems/2.2.0/gems/rspec-co
re-3.5.4/lib/rspec/core/runner.rb:86:in `run'
        from /Users/nasubi/.rbenv/versions/2.2.2/lib/ruby/gems/2.2.0/gems/rspec-co
re-3.5.4/lib/rspec/core/runner.rb:71:in `run'
        from /Users/nasubi/.rbenv/versions/2.2.2/lib/ruby/gems/2.2.0/gems/rspec-co
re-3.5.4/lib/rspec/core/runner.rb:45:in `invoke'
        from /Users/nasubi/.rbenv/versions/2.2.2/lib/ruby/gems/2.2.0/gems/rspec-co
re-3.5.4/exe/rspec:4:in `<top (required)>'
        from /Users/nasubi/.rbenv/versions/2.2.2/bin/rspec:23:in `load'
        from /Users/nasubi/.rbenv/versions/2.2.2/bin/rspec:23:in `<main>'
```

図 6　pending の step 定義を書き込んだ後の cucumber の実行結果.

```
/Users/nasubi/my_help/lib/my_todo% ls
my_todo.rb
```

my_todo.rb に先ほどのエラーで Mytodo という module がないといわれているので，ここで作成します.

/Users/nasubi/my_help/lib/my_todo/my_todo.rb の中に以下のコードを記述します.

```
module Mytodo
  class Edit
    def open

    end
  end
end
```

また，これを require しないといけないので，lib/todo.rb として，以下を追加します.

```
require 'my_todo/my_todo'
```

これだけでは，rspec が lib/my_todo/my_todo.rb を読み込んでいないため，このスペックを実行するために，spec ディレクトリに spec_helper.rb に以下を追加します.

```
$LOAD_PATH.unshift File.expand_path('../../lib', __FILE__)
require 'todo'
```

これでもう一度 rspec を実行してみます．

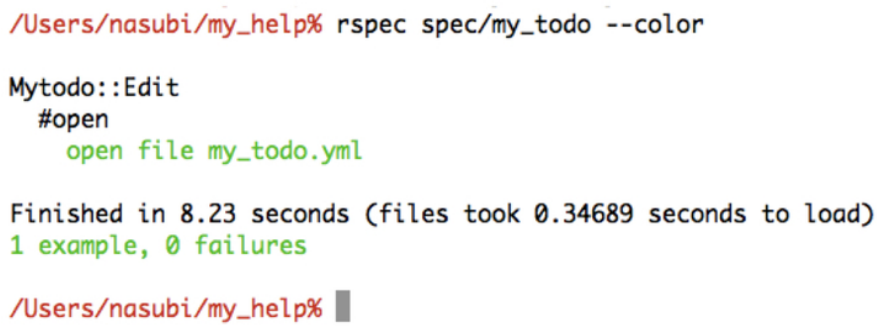

図 7　red を修正して green になった rspec の実行結果

エラーが消えて成功しているのがわかります．これで「ならば　edit が開かれる」のシナリオの RSpec 部分が成功しました．Red, Green と進めたので次は Refactoring をするのですが，ここではあまり必要のないので省略します．RSpec が終わったので，Cucumber に戻ります．

先ほど lib ディレクトリで lib/my_todo/my_todo.rb を作成したので，cucumber でも読み込むために，以下を作成します．

`/Users/nasubi/my_help/features/support/env.rb`

env.rb の中は以下の通りです．

```
1 $LOAD_PATH.unshift File.expand_path('../../../lib', __FILE__)
2 require 'todo'
```

これで cucumber も lib/の中身を読み取ってくれます．

もう一度 cucumber を実行してみると，

図 8　lib の中身を読み込むようにした cucumber の実行結果.

エラーが消えています.

これで BDD が成功しました. 残りの「自分の todo を書き込む」も my_help が何か振る舞いをするわけではないので,「コマンドを入力して todo を更新する」シナリオ全てのテスト開発が終わりました. このように BDD で my_help のテスト開発を行っていきました.

3.4 features での記述とその意味

features での記述は，コマンドの振る舞いを説明する自然な記述です．その様子を specific_help が用意しているデフォルトのコマンドについて説明します．specific_help とは，ユーザが作成するそれぞれのヘルプです．speific_help の–help を表示させると，

```
--edit                          edit help contentsを開く
--to_hiki                       hikiのformatに変更する
--all                           すべてのhelp画面を表示させる
--store [item]                  store [item] でback upをとる
--remove [item]                 remove [item] back upしてるlistを消去する
--add [item]                    add new [item]で新しいhelpを作る
--backup_list [val]             back upしているlistを表示させる
```

が得られます．これらの項目について順に詳細な振る舞いとそれを記述するシナリオを検討していきます．

3.4.1 –add [item]

このコマンドは新しい item を specific_help に追加します．提供される機能をシナリオの先頭に内容をかいつまんでこの振る舞いが記述されています．実装では，ヘルプの内容は /.my_help/emacs_help.yml に元 data があります．

```
1  nasu% cat add.feature
2  #language: ja
3
4  #--add [item]
5  機能: 新しいitemをspecific_helpに追加する
6  specific_helpとは，ユーザが作成するそれぞれのヘルプである
7  新しいhelp画面を追加したい
8
9  シナリオ: コマンドを入力してspecific_helpにitemを追加する
10         前提 新たなhelpコマンドを追加したい
11         もし emacs_help --add[item]を入力する
12         ならば ~/.my_help/emacs_help.ymlに新しいitemが自動的に追加される
```

3.4.2 –all

このコマンドは specific_help にある help を一度に全て表示させます．すると specific_help にある item を全て表示します．実装では，元データが /.my_help/emacs_help.yml にあるので，それを全て表示します．

```
1  nasu% cat all_help.feature
2  #language: ja
3
```

```
#--all
機能: 全てのhelp画面を見る
複数のhelp画面を一度に見たい時に便利である

シナリオ: コマンドを入力してすべてのhelpを見る
        前提 複数のhelp画面を表示したい
        もし emacs_help --allと入力する
        ならば すべてのhelp画面が表示される
```

3.4.3 –backup_list

このコマンドは過去にバックアップをとったことのある item を表示させます．自分がどの item をバックアップしたのかの確認を行えます．また，その item のバックアップをとった時間も表示されるので，いつバックアップをとったのかの確認も行えます．

```
nasu% cat backup_list.feature
#language: ja

#--backup_list
機能: 過去にバックアップしてあるitemのリストを表示させる
何をバックアップしたかの確認をしたい

シナリオ: コマンドを入力してバックアップのリストを見る
        前提 バックアップのリストを見たい
        もし emacs_help --backup_listを入力する
        ならば バックアップしているitemのリストが表示される
```

3.4.4 –edit

このコマンドは specific_help の item の編集ができます．実装では，元データのある /.my_help/emacs_help.yml が emacs で開かれ，自分で編集ができます．編集作業が終了したら，emacs で開かれているので， [C-x C-s] で保存，[C-x C-c] で edit を閉じます．

```
nasu% cat edit_help.feature
# language: ja
#--edit
機能: helpコマンドの追加や削除，編集をするためのeiditを開く
emacs_helpと入力したときに出てくるhelpのコマンドの追加や削除，編集ができる

シナリオ: コマンドを入力してeditを開く
        前提 emacs_helpのコマンドの編集がしたい
        もし emacs_help --editと入力する
        ならば ~/.my_help/emacs_help.ymlがemacsで開かれる
```

3.4.5 –remove [item]

このコマンドは新しい item を specific_help から削除します．提供される機能をシナリオの先頭に内容をかいつまんでこの振る舞いが記述されています．実装では，ヘルプの内容は /.my_help/emacs_help.yml に元 data があります．

```
nasu% cat remove.feature
#language: ja

#--remove [item]
機能: specific_helpのitemを消す
いらなくなったitemを消したいときに使う

シナリオ: コマンドを入力してitemを消す
        前提 いらないitemを消したい
        もし emacs_help remove [item]
        ならば ~/.my_help/emacs_help.ymlからitemが消える
```

3.4.6 –store [item]

このコマンドは specific_help の item のバックアップをとります．少し help を新しくしたいが，過去の help を残しておけば安心できます． /.my_help ディレクトリーの中に，同じ名前の先頭に'.' をつけたファイルに追記されていきます．

```
-rw-r--r--   1 bob  501   3231  2 10 17:11 my_todo.yml
-rw-r--r--   1 bob  501  31925  2  3 19:49 .my_todo.yml
```

などとしてます．

```
nasu% cat store.feature
#language: ja

#--store [item]
機能: itemのバックアップを取る
バックアップとして残したいitemがあるときに使う

シナリオ: コマンドを入力してitemのバックアップをとる
        前提 バックアップをとっておきたい
        もし emacs_help --store [item]と入力する
        ならば 入力したitemのバックアップが作られる
```

3.4.7 –to_hiki

このコマンドは format を hiki に変更します．

:**カーソル移動**:cursor

*C-f, move Forwrard, **前 or 右へ**

*C-b, move Backwrard, **後 or 左へ**

*C-a, go Ahead of line, **行頭へ**

*C-e, go End of line, **行末へ**

*C-n, move Next line, **次行へ**

*C-p, move Previous line, **前行へ**

などと hiki 記法で出力されます．これを hiki の本文中に入れると

カーソル移動 cursor

- C-f, move Forwrard, 前 or 右へ
- C-b, move Backwrard, 後 or 左へ
- C-a, go Ahead of line, 行頭へ
- C-e, go End of line, 行末へ
- C-n, move Next line, 次行へ
- C-p, move Previous line, 前行へ

などと web 上で表示されます．

```
nasu% cat to_hiki.feature
# language: ja

#--to_hiki
機能:formatをhikiモードに変更する
一つ一つエディタで開いて変更するのがめんどくさい時に有益である

シナリオ: コマンドを入力してformatをhikiモードに変える
        前提 hikiモードに変更したい
        もし emacs_help --to_hikiと入力する
        ならば formatがhikiモードに変更される
```

3.4.8 todo の更新

まず，todo を書き込むために，edit を開くために，my_todo –edit と入力します．–edit コマンドは上記で説明した動きをします．しかし，ここでは元データは /.my_help/todo_help.yml です．edit が開かれたら，todo を書き込みます．（今週やることなら weekly という item を作ってそこに書き込む）保存方法や，edit の閉じ方は emacs と同じ操作方法です．

```
nasu% my_todo.feature
# language: ja

機能: todoの更新を行う
自分のするべきことを書き込むためにtodoを更新する

シナリオ: コマンドを入力してtodoを更新していく
          前提 todoを編集したい
          もし "my_todo␣--edit"と入力する
          ならば editが開かれる
          かつ 自分のtodoを書き込む
```

4　考察と今後の課題

4.1　考察

my_help を対象とした具体的なビヘイビア駆動開発の実践を通して，以下のような問題点が明らかになってきました．

4.1.1　Cucumber を記述できるようになるまで時間がかかる

日本語で書くことができるので，記述されているコードを「理解する」のは容易で，「記述する」ことも可能であると考えていました．ところが，The RSpec Book[1] に記載されているテンプレートが少なく，理解することにも非常に困難を覚えました．自力で何も前例のない状況から作成することはできますが，やはり時間がかかってしまいます．ネットで検索しても，日本語のサイトではほぼ The RSpec Book[1] と類似したシナリオしかのっていないのも問題の一つであり，違った種類の書き方のテンプレートがもっと必要です．地道に幾度も Cucumber を叩いて，features や rspec の記述すること，またそのデータベースを蓄えていくことが必要です．また，BDD では Cucumber でエラーが出ている状況が正しく，Cucumber を放置して RSpec へと進みます．今までのプログラミングとは少し違う感覚に陥り，不安がよぎります．これは慣れるまでの辛抱ではありますが，この問題も時間との戦いです．

4.1.2　BDD を行うときに ruby の知識が必要

Cucumber のステップ定義や RSpec において ruby を用いることが多々あります．スッテプ定義を記述するときに，あればよいと思うコードを書くのですが，具体的なプログラムを書くよりも難易度が高いと感じました．部分的なプログラムを作成するので，知識がないと正しいかどうかの判断が最後にしかできません．BDD を通して ruby を学べるという考え方もできますが，BDD で ruby を学ぶには物足りなさがあり，不向きです．したがって，BDD を使用する前に ruby の基礎の勉強は不可欠です．

4.2　my_help の今後

my_help のテスト記述が完了するに伴って，仕様や動作の標準が確定しました．

my_help は今後も開発者，ならびに多くのユーザの使用を通じて，どのように進化させれば便利なのかが徐々にわかってくることで，my_help はまだまだ進化する機会が出て

くると思われます．つまり，ビヘイビアの記述を利用することで，今後 my_help を進化させるための共同開発が円滑に進める手助けになると推測します．また，my_help の進化の開発だけに関わらず，my_help の使用方法が明確になったことで，本研究で作成した my_help の features を読めば初心者でも my_help の振る舞いが容易に理解できます．my_help は本研究室で今後使われていくと予想し，本研究はこれからの研究の手助けになります．今後の本研究室の研究生が，CUI などの習得に時間をかけずにプログラミングに集中でき，研究の質が向上すると予想できます．

5 謝辞

本研究を進めるにあたり，様々なご指導を頂きました西谷滋人教授先生に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，知識の供給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] ”The RSpec Book”, David Chelimsky, Dave Astels, Zach Dennis, 訳, 株式会社クイーブ, 監修, 角谷信太郎, 豊田裕司 (翔泳社, 2012).

[2] my_help README, Shigeot R. Nishitani, `http://www.rubydoc.info/gems/my_help/0.4.3` 2017/2/11 アクセス.