卒業論文

ユーザメモソフト my_help の開発

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

目次

本研究ではユーザメモソフトである my_help の開発において，BDD を取り入れることにより my_help の向上を目指した．

my_help とは，ユーザメモソフトであり，user 独自の help を作成・提供することができる gem である．しかし，これらの仕様方法を初心者が理解すること自体に時間がかかってしまうという問題点がある．そこで，cucumber を用いる．cucumber は Ruby で BDD を実践するために用意された環境である．したがって，cucumber は振る舞いをチェックするために記述するが，そこで日本語がそのまま用いることが可能であるため，その記述を読むだけで，my_help の振る舞いを理解することが可能となる．

cucumber は実際にソフトウェア開発の現場において，ユーザーとプログラマがお互いの意思疎通のために利用される．テストはプログラムがチェックしてくれるが，記述は人間が理解できなければならない．この二つの要求を同時に叶えようというのが，BDD の基本思想である．これらは，研究室の知識を定着させることに有益であり，研究室の役に立つと考えた．

1 目的

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ある．ところが，プログラミング初心者は terminal 上での character user interface(CUI) を苦手としている．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 を使うことが常識となる．

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 に置かれている．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ある．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る．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が期待でき，初心者のスキル習得が加速することが期待できる．

しかし，Ruby gems として提供されているこのソフトは，動作はするがテストが用意されていない．慣れた開発者は，テストを見ることで仕様を理解するのが常識である．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る．

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する．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する Behavior Driven Development(BDD) に基づいてテストを記述していく．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する．Cucumber は自然言語で振る舞いを記述することができるため，ユーザにとって，わかりやすく振る舞いを確認することができる．

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

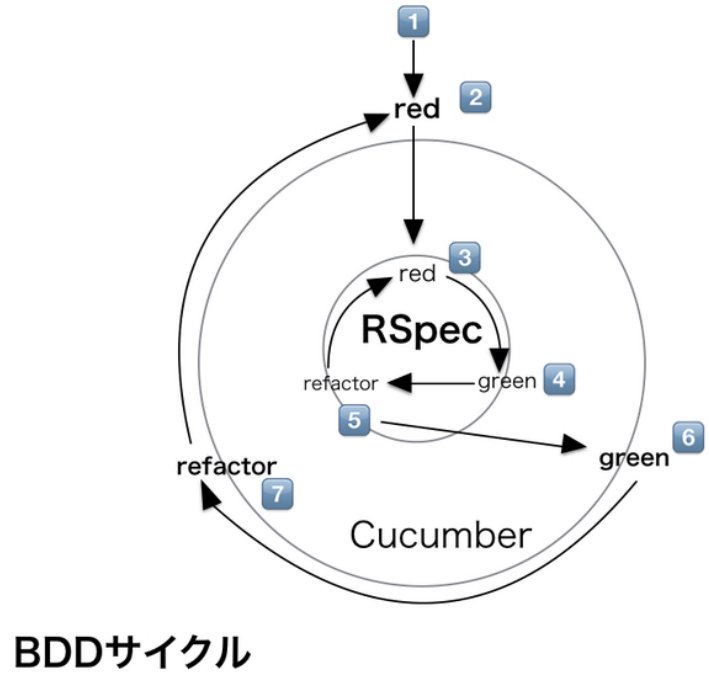

図 1　RSpec と Cucumber の Red-Green-Refactoring サイクル間の関係.

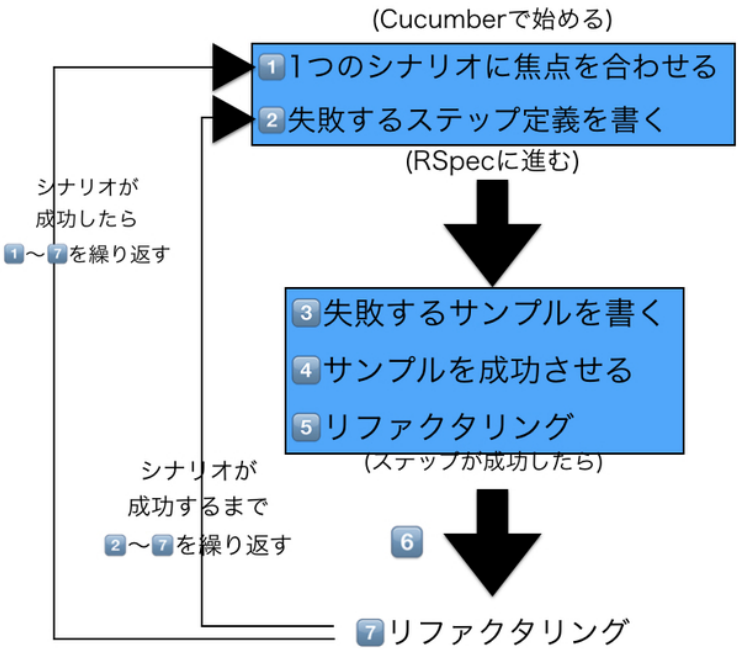

図 2　RSpec と Cucumber の手順.

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．次に，英語の features のひな形を示します．

```
% cat ./featrues/sample_e.feature
Feature: Description of feature


Scenario: Description of scenario
  Given I want to explain scenario
  Then I investigate
  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す．

```
# language: ja

機能: todo の更新を行う
todo は更新していくものであり，新しく書いたり終わったものを消したいので
バックアップをとって，過去の todo を残しておく

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

シナリオ: コマンドを入力してバックアップをとる
 前提 todo の編集が終わった
 もし "my_todo --store [item]"と入力する
 ならば item のバックアップを取る
```

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 に対しては下記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

表 1

keyword	ja(japanese)
feature	”フィーチャ”, ”機能”
background	”背景”
scenario	”シナリオ”
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
examples	”例”, ”サンプル”
given	”* ”, ”前提”
when	”* ”, ”もし”
then	”* ”, ”ならば”
and	”* ”, ”かつ”
but	”* ”, ”しかし”, ”但し”, ”ただし”
given (code)	”前提”
when (code)	”もし”
then (code)	”ならば”
and (code)	”かつ”
but (code)	”しかし”, ”但し”,”ただし”

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．

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

```

```
3   Scenario: Description of scenario  # features/hogehoge.
      feature:3
4     Given I want to explain scenario # features/hogehoge.
        feature:4
5     Then I investigate       # features/hogehoge.feature:5
6     When I know the meaning # features/hogehoge.feature:6
7
8 1 scenario (1 undefined)
9 3 steps (3 undefined)
10 0m0.066s
11
12 You can implement step definitions for undefined steps
     with these snippets:
13
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

2.3.1 使用法

my_help は本研究室の西谷が開発したものです.

以下は my_help の README です [2].

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.

1. 問題点

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.

1. 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

2.3.3 github に行って daddygongon の my_help を fork する

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

2.3.4 my_help の更新

■git を用いて my_help を新しくする.

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 が出る.

(sudo で実行していたら権限が root に移行される)

新しいターミナルを開いて動くかチェックする.

3 結果

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．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
4. my_todo と打ち込んで更新されていたら完成
5. my_todo –store [item] を入力して item のバックアップをとる

この振る舞いがきちんとできているのかを実際にテスト構築して行きます．

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．

```
1  # language: ja機能
2
3  : の更新を行うtodoは更新していくものであり
4  todo新しく書いたり終わったものを,消したいのでバックアップをとって，過去のを残しておく
5  todoシナリオ
6
7
8    : コマンドを入力してを更新していくtodo前提を編集したい
9                todoもし
10               "my_todo ――editと入力する"ならばが開かれる
11               editかつ自分のを書き込む
12               todoシナリオ
13
14 : コマンドを入力してバックアップをとる前提の編集が終わった
15               todoもし
16               "my_todo ――store [itemと入力する]"ならばのバックアップを取る
17               item
```

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．シナリオは，一つの機能に対して複数書くことが可能です．ここでは、「todo を更新する」と「バックアップをとる」という二つのシナリオをたてています．

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．　/Users/nasubi/nasufeatures 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．

実行すると以下のようになります．

```
1  # language: ja機能
2  : の更新を行うtodoは更新していくものであり
3  todo新しく書いたり終わったものを消したいのでバックアップをとって，過去の,を残しておくtodoシナリオ
4
5     : コマンドを入力してを更新していくtodo # features/my_todo.feature:6前提を編集したい
6       todo                  # features/my_todo.feature:7もし
7       "my_todo --editと入力する"  # features/my_todo.feature:8ならばが開かれる
8       edit                  # features/my_todo.feature:9かつ自分のを書き込む
9       todo               # features/my_todo.feature:10シナリオ
10
11    : コマンドを入力してバックアップをとる            # features/my_todo.feature:12前提の編
          集が終わった
12      todo                     # features/my_todo.feature:13もし
13      "my_todo --store [itemと入力する]" # features/my_todo.feature:14ならばのバッ
            クアップを取る
14      item                  # features/my_todo.feature:15
15
16 2 scenarios (2 undefined)
17 7 steps (7 undefined)
18 0m0.080s
19
20 You can implement step definitions for undefined steps with these
       snippets:前
       提
21
22 (/^を編集したいtodo$/) do
23   pending # Write code here that turns the phrase above into concrete
         actions
24 endもし
25
26 (/^"([^"]*)と入力する"$/) do |arg1|
27   pending # Write code here that turns the phrase above into concrete
         actions
28 endならば
29
30 (/^が開かれるedit$/) do
31   pending # Write code here that turns the phrase above into concrete
         actions
32 endならば自分の
33
34 (/^を書き込むtodo$/) do
35   pending # Write code here that turns the phrase above into concrete
         actions
36 end前提
37
38 (/^の編集が終わったtodo$/) do
39   pending # Write code here that turns the phrase above into concrete
         actions
40 endならば
41
```

```
42 (/^のバックアップを取るitem$/) do
43   pending # Write code here that turns the phrase above into concrete
         actions
44 end
```

ここでは，2 つの scenario と 7 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．

```
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

前提(/^todo の編集が終わった$/) do
  pending # Write code here that turns the phrase above into concrete actions
end

ならば(/^item のバックアップを取る$/) do
```

```
  pending # Write code here that turns the phrase above into concrete actions
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

```
/Users/nasubi/nasu% cucumber features/my_todo.feature
# language: ja
機能: todo の更新を行う
todo は更新していくものであり，新しく書いたり終わったものを消したいので
バックアップをとって，過去の todo を残しておく

  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:6
    前提 todo を編集したい                  # features/step_definitions/my_todo_spec.r
      TODO (Cucumber::Pending)
      ./features/step_definitions/my_todo_spec.rb:2:in `/^todo を
編集したい$/'
      features/my_todo.feature:7: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
    ならば edit が開かれる                  # features/step_definitions/my_todo_spec.r
    かつ自分の todo を書き込む               # features/step_definitions/my_todo_spec

  シナリオ: コマンドを入力してバックアップをとる          # features/my_todo.featur
    前提 todo の編集が終わった                  # features/step_definitions/my_to
      TODO (Cucumber::Pending)
      ./features/step_definitions/my_todo_spec.rb:18:in `/^todo の
編集が終わった$/'
      features/my_todo.feature:13:in `前提 todo の編集が終わった'
    もし"my_todo --store [item]"と入力する # features/step_definitions/my_todo_
    ならば item のバックアップを取る              # features/step_definitions/my_

2 scenarios (2 pending)
7 steps (5 skipped, 2 pending)
```

```
0m0.045s
```

と変化が出てきます.

```
2 scenarios (2 pending)
7 steps (5 skipped, 2 pending)
```

これは 2 つのシナリオの内 2 つが pending であり，7 つの step の内 2 つが pending で 5 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.

cucumber が成功すると下記のような結果となります.

```
1  /Users/nasubi/my_help% cucumber features/my_todo.feature
2  # language: ja機能
3  : の更新を行うtodoは更新していくものであり
4  todo新しく書いたり終わったものを消したいのでバックアップをとって，過去の
       ,を残しておくtodoシナリオ
5
6      : コマンドを入力してを更新していく
         todo # features/my_todo.feature:6前提を編集したい
7        todo                # features/step_definitions/
             my_todo_spec.rb:2も
             し
8        "my_todo --editと入力する
             "  # features/step_definitions/my_todo_spec.rb:6な
             らばが開かれる
9        edit                # features/step_definitions/
             my_todo_spec.rb:10かつ自分のを書き
             込む
10       todo             # features/step_definitions/
             my_todo_spec.rb:14シナリ
             オ
11
12     : コマンドを入力してバックアップをとる
                       # features/my_todo.feature:12前提の編集が終
         わった
13       todo                      # features/step_definitions/
             my_todo_spec.rb:18も
             し
14       "my_todo --store [itemと入力する
             ]" # features/step_definitions/my_todo_spec.rb:6な
```

```
        らばのバックアップを取る
15      item                    # features/step_definitions/
        my_todo_spec.rb:22
16
17 2 scenarios (2 passed)
18 7 steps (7 passed)
19 0m0.030s
```

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く．

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する．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する．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く．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．

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 を追加します．中身は下記の通りです．

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

これで事前準備は完成でコードを書いていきます.

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
```

これで rspec を実行すると以下のような結果が表示される.

```
1  /Users/nasubi/my_help% rspec spec --color
2
3  my_help command
4    version option
5      should be successfully executed
6      should have output: "my_help 0.4.3"
7    help option
8      should be successfully executed
9      should have output: "Usage: my_help [options]\n    -v
         , --version                      show program
         Version.\n    -l, --list ...                  install
         local after edit helps\n        --delete NAME
                        delete NAME help"
10
11 Mytodo::Todo
12   #open
```

```
13       open file my_todo.yml
14
15 Finished in 3.87 seconds (files took 0.30703 seconds to
       load)
16 5 examples, 0 failures
```

これで todo を更新するときの振る舞いのテストはうまく成功した.

この方法で BDD で my_help のテスト開発を行い，仕様を決定していった.

3.3.1 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となる．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する．specific_help とは，ユーザが作成するそれぞれのヘルプである．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

が得られる．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く．

3.3.2 my_help の features

下記は私が作成した my_help の一部を features で書いたものである．

■–add [item]　このコマンドは新しい item を specific_help に追加する．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る．実装では，ヘルプの内容は /.my_help/emacs_help.yml に元 data がある

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

#--add [item]
機能：新しい item を specific_help に追加する
specific_help とは，ユーザが作成するそれぞれのヘルプである
新しい help 画面を追加したい
```

シナリオ: コマンドを入力して specific_help に item を追加する

前提 新たな help コマンドを追加したい

もし emacs_help --add[item] を入力する

ならば ~/.my_help/emacs_help.yml に新しい item が自動的に追加される

■全ての help 画面の表示

```
1
2 #language: ja
3
4 #—all機能
5 : 全ての画面を見るhelp複数の画面を一度に見たい時に便利である
6 helpシナリオ
7
8 : コマンドを入力してすべてのを見るhelp前提複数の画面を表示したい
9             helpもし
10            emacs_help ——と入力するallならばすべての画面が表示される
11            help
```

シナリオ：コマンドをニュ力してすべての help 画面を見る

コマンド：emacs_help –all

■過去にバックアップしてある item のリストの表示

```
1 #language: ja
2
3 #—backup_list機能
4 : 過去にバックアップしてあるのリストを表示させるitem何をバックアップしたかの確認をしたいシナリオ
5
6
7 : 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
8
9             emacs_help ——を入力するbackup_listならばバックアップしているのリストが表示される
10            item
```

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コマンド：emacs_help –backup_list

■help コマンドの追加や削除，編集をするファイルの開示

```
# language: ja
#—edit機能
: コマンドの追加や削除，編集をするためのを開くhelpeiditと入力したとき
    に出てくるのコマンドの追加や削除，編集ができる
emacs_helphelpシナリオ

: コマンドを入力してを開くedit前提のコマンドの編集がしたい
        emacs_helpもし
        emacs_help ——と入力するeditならば
        ~/.my_help/emacs_help.がで開かれるymlemacs
```

シナリオ：コマンドを入力して edit を開くコマンド：emacs_help –edit

元 data である /.my_help/emacs_help.yml を開く．

ここで編集を行い，emacs で開いているので C-x,C-s で保存する．

■specific_help の item の消去

```
#language: ja

#—remove [item]機能
: のを消すspecific_helpitemいらなくなったを消したいときに使う
itemシナリオ

: コマンドを入力してを消すitem前提いらないを消したい
        itemもし
        emacs_help remove [item]ならば
        ~/.my_help/emacs_help.からが消えるymlitem
```

シナリオ：コマンドを入力して item を消す

コマンド：emacs_help –remove

■item のバックアップ

```
#language: ja

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

: コマンドを入力してのバックアップをとるitem前提バックアップをとっておき
    たいもし
```

```
8
9               emacs_help --store [itemと入力する]ならば入力したの
                    バックアップが作られる
10              item
```

シナリオ：コマンドを入力してバックアップをとる

コマンド：emacs_help –store [item]

■hiki への format の変更

```
1 # language: ja
2
3 #—to_hiki機能
4 :をモードに変更するformathiki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
5
6
7 : コマンドを入力してをモードに変えるformathiki前提モードに変更したい
8               hikiもし
9               emacs_help --と入力するto_hikiならばがモードに変更される
10              formathiki
```

シナリオ：コマンドを入力して format を hiki モードにする

コマンド：emacs_help –to_hiki

■todo の更新

```
1 # language: ja機能
2
3 : の更新を行うtodoは更新していくものであり
4 todo新しく書いたり終わったものを消したいのでバック,\アップをとって，過去
      のを残しておく
5 todoシナリオ
6
7 : コマンドを入力してを更新していくtodo前提を編集したい
8                   todoもし
9                   "my_todo --editと入力する"ならばが開かれる
10                  editかつ自分のを書き込む
11                  todoシナリオ
12
13 : コマンドを入力してバックアップをとる前提の編集が終わった
14                  todoもし
```

```
15                  "my_todo --store [itemと入力する]"ならばのバックアッ
                        プを取る
16                  item
```

シナリオ１：コマンドを入力して todo を更新するシナリオ２：コマンドを入力してバックアップをとる

コマンド１：my_todo –edit コマンド２：my_todo –store [item]

my_todo –edit で /.my_help/my_todo.yml を開く．

ここで編集を行い，emacs で開いているので C-x,C-s で保存する．

my_todo –store [item] で todo の item をバックアップとっておく．

この動作により過去のバックアップを閲覧することができ，どんどん更新することが可能である．

4　考察

今回の開発において，ユーザメモソフトである my_help のテスト開発は成功した．つまり，my_help の仕様と動作の標準が確定したことで，今後のソフトを進化させるための共同開発がスムーズにいくと推測される．先に述べた通り，ソフト開発は一人でせず，複数人で開発することが普通である．その時に起こる障害として，意思の疎通ができていないことがあげられる．振る舞いが標準化されていないと，どのような振る舞いをするのかが，プログラムを見るだけでは，ずれが生じてしまう．また，開発者の意図が読めていないと，コードの意味も変わってくるので，テスト開発をして，仕様と動作の標準を確定することは重要であった．加えて，初めて my_help を使うユーザでも仕様と動作の標準がわかっていれば，短い時間で my_help を使いこなせるし，そのことによって，ユーザそれぞれに自分にあった my_help に変更することも容易になる．

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．

6　参考文献

1. The RSpec Book　著者：David Chelimsky Dave Astels Zach Dennis ほか　翻訳：株式会社クイーブ　監修：株式会社クイーブ　角谷信太郎　豊田裕司.
2. Shigeot R. Nishitani, my_help の README, http://www.rubydoc.info/gems/my_help/0.4.