

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

1	序論	5
2	先行研究, 方法	6
2.1	RSpec と BDD について	7
2.2	Cucumber について	10
2.2.1	概要	10
2.2.2	features	10
2.2.3	Cucumber, RSpec インストール	11
2.2.4	ディレクトリー構造と使用手順	12
2.3	my_help について	16
2.3.1	my_help のインストール	16
2.3.2	github に行って daddygongon の my_help を fork する	16
2.3.3	my_help の更新	16
	git を用いて my_help を新しくする.	16
	次にとってきた.yml を /.my_help に cp する.	16
3	実践	18
3.1	todo の更新マニュアル	18
3.2	Cucumber	18
3.3	RSpec	21
3.3.1	features での記述とその意味	26
3.3.2	my_help の features	26
	-add [item]	26
	全ての help 画面の表示	27
	過去にバックアップしてある item のリストの表示	27
	help コマンドの追加や削除, 編集をするファイルの開示	28
	specific_help の item の消去	28
	item のバックアップ	28
	hiki への format の変更	29
	todo の更新	29

4	結果と今後の課題	31
4.1	本研究の成果	31
4.2	今後の課題	31
5	謝辞	33
6	参考文献	33

目次

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。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 my_help 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。これにより、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、動作はするがテストが用意されていません。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 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、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ます。ここでは、実際に BDD の流れにそって、my_help のある一例を取り上げてのテスト開発を進めていきます。また、my_help の頻繁に使うコマンドの振る舞いを記述しています。このことにより、来年の本研究室の研究生のスキルがより向上することが期待されると考えています。

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、動作はするがテストが用意されていません。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。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、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。

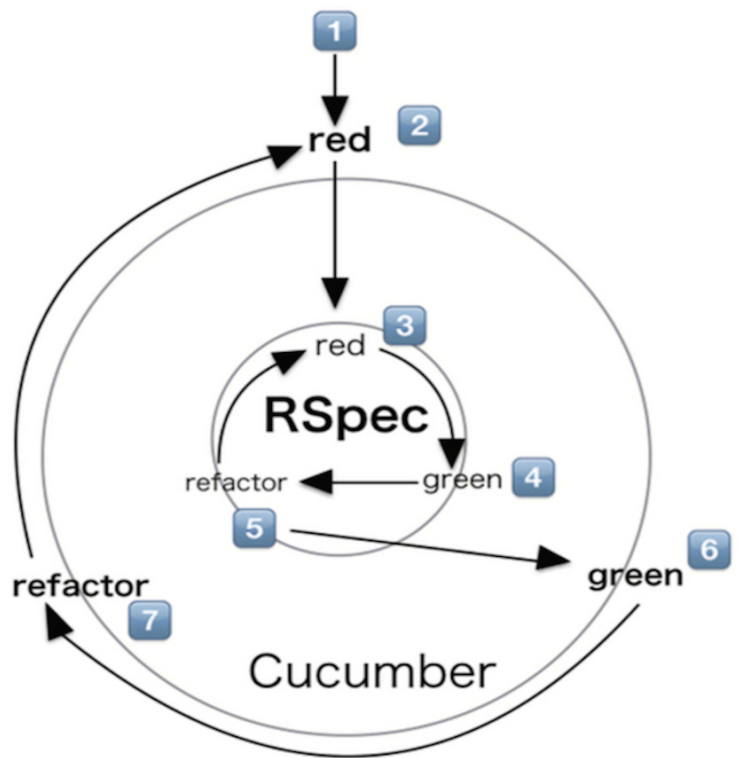
RSpec と Cucumber の関係を図に示しました。これは、RSpec 本から書き写した図です [1, pp.9]。RSpec でテストを書くと一つ一つの function あるいは method レベルで Red, Green, Refactoring を行うべしという意図があります。一方で、もっと大きな枠組み、つまりシステムレベルでもこれらのステップは必要です。ところが、それを RSpec で書くのには無理があります。このレベルのテスト記述をしやすくするのが、Cucumber です。そこでも Red, Green, Refactoring が必要で、そこでサイクルが回ることを意図して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

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



BDDサイクル

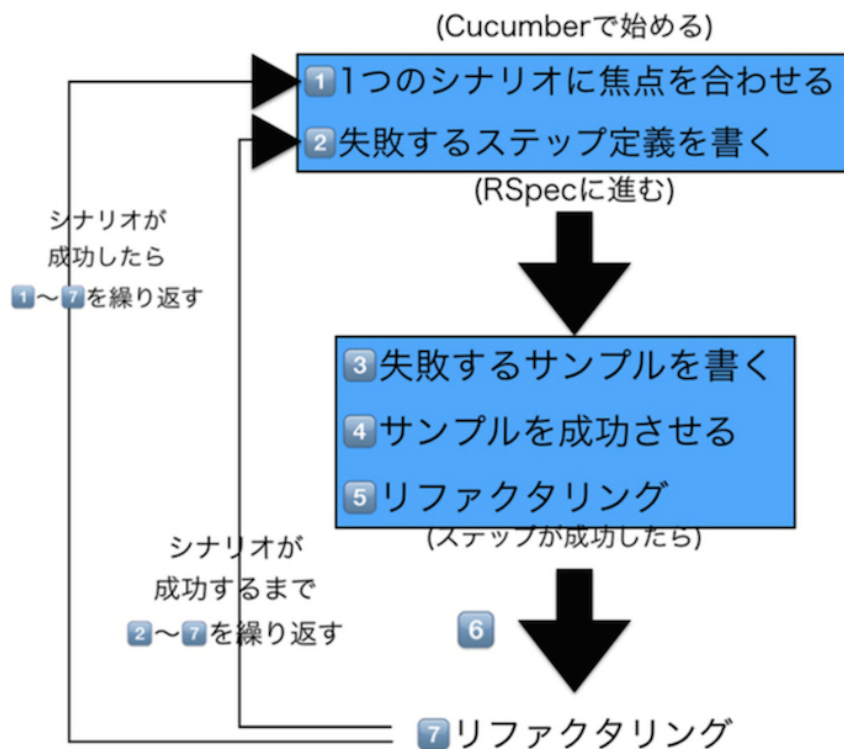


図2 RSpec と Cucumber の手順.

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 ./features/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

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

表 1

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”

```
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
|   |—— hoge hoge.feature
|   |—— step_definitions
|   |   |—— hoge hoge_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、hoge hoge.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 hoge hoge.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。

```

1 Feature: Description of feature
2
3   Scenario: Description of scenario # features/hoge hoge.
4             feature:3
5   Given I want to explain scenario # features/hoge hoge.
6             feature:4
7   Then I investigate # features/hoge hoge.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、Given(), 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。terminal 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 のインストール

2.3.2 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_yaml`
4. `rake clean_exe`

`sudo bundle exec exe/my_help -m`

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

2.3.3 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.

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- 同じところ見ている- 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.

3 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。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. /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

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

```
/Users/nasubi/my_help% cucumber features/my_todo.feature
# language: ja
機能: todoの更新を行う
自分のす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

と変化が出てきます。

```
1 scenarios (1 pending)
4 steps (3 skipped, 1 pending)
```

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ので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 の部分を書けたので、もう一度 cucumber features/my_todo.feature を実行します。すると、下記のような結果が返ってきました。

```
/Users/nasubi/my_help% cucumber features/my_todo.feature
# language: ja
機能: todoの更新を行う
自分のす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
  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:8
    uninitialized constant Mytodo (NameError)
    ./features/step_definitions/my_todo_spec.rb:9:in `(/^editが開かれる$/'
    features/my_todo.feature:10:in `ならばeditが開かれる'
    かつ自分のtodoを書き込む # features/step_definitions/my_todo_spec.rb:12

Failing Scenarios:
cucumber features/my_todo.feature:7 # シナリオ: コマンドを入力してtodoを更新していく

1 scenario (1 failed)
4 steps (1 failed, 1 skipped, 2 passed)
0m0.025s
-
```

図 5

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.

```
/Users/nasubi/my_help% rspec spec/my_todo/ --color
/Users/nasubi/my_help/spec/my_todo/todo_spec.rb:6:in '<
  module:Mytodo>': uninitialized 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-core-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-core-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-core-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-core-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-core-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-core-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-core-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-core-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-core-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>'

```

エラーが出てしまっているのがわかると思います。

`{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/lib/my_todo

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
end

```



```
end
```

また、これを require い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'
```

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

これでもう一度 rspec を実行してみます。

```
/Users/nasubi/my_help% rspec spec/my_todo --color
```

```
Mytodo::Edit
```

```
#open
```

```
open file my_todo.yml
```

```
Finished in 8.23 seconds (files took 0.34689 seconds to load)
```

```
1 example, 0 failures
```

```
/Users/nasubi/my_help% █
```

図 6

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,

```
/Users/nasubi/my_help% cucumber features/my_todo.feature  
# language: ja  
機能: todoの更新を行う  
自分のす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  
  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:8  
  かつ自分のtodoを書き込む # features/step_definitions/my_todo_spec.rb:12  
  
1 scenario (1 passed)  
4 steps (4 passed)  
0m0.024s  
.....■
```

図 7

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.

3.3.1 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。specific_help の-help を表示させると、

<code>--edit</code>	edit help contents を開く
<code>--to_hiki</code>	hiki の format に変更する
<code>--all</code>	すべての help 画面を表示させる
<code>--store [item]</code>	store [item] で back up をとる
<code>--remove [item]</code>	remove [item] back up してる
list を消去する	
<code>--add [item]</code>	add new [item] で新しい help を作る
<code>--backup_list [val]</code>	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
5 機能: 全ての help 画面を見る
6 複数の help 画面を一度に見たい時に便利である
7
8 シナリオ: コマンドを入力してすべての help を見る
9     前提 複数の help 画面を表示したい
10    もし emacs_help —all と入力する
11    ならば すべての help 画面が表示される
```

シナリオ: コマンドをニュカしてすべての 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
4 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
5 何をバックアップしたかの確認をしたい
6
7 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
8     前提 バックアップのリストを見たい
9     もし emacs_help —backup_list を入力する
10    ならば バックアップしている
        item のリストが表示される
```

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

```
1 # language: ja
2 #—edit
3 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
4 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
5
6 シナリオ: コマンドを入力してeditを開く
7     前提 emacs_helpのコマンドの編集がしたい
8     もし emacs_help —editと入力する
9     ならば ~/.my_help/emacs_help.ymlがemacsで開かれる
```

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

```
1 #language: ja
2
3 #—remove [item]
4 機能: specific_helpのitemを消す
5 いらなくなったitemを消したいときに使う
6
7 シナリオ: コマンドを入力してitemを消す
8     前提 いらないitemを消したい
9     もし emacs_help remove [item]
10    ならば ~/.my_help/emacs_help.ymlからitemが消える
```

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

```
1 #language: ja
2
3 #—store [item]
4 機能: itemのバックアップを取る
5 バックアップとして残したいitemがあるときに使う
6
```

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

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
4 機能: formatをhikiモードに変更する
5 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
6
7 シナリオ： コマンドを入力してformatをhikiモードに変える
8 前提 hikiモードに変更したい
9 もし emacs_help --to_hikiと入力する
10 ならば formatがhikiモードに変更される

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
2
3 機能: todoの更新を行う
4 自分のすべきことを書き込むためのtodoを更新する
5
6 シナリオ： コマンドを入力してtodoを更新していく
7 前提 todoを編集したい
8 もし "my_todo --edit"と入力する
9 ならば editが開かれる
10 かつ 自分のtodoを書き込む

シナリオ1：コマンドを入力してtodoを更新する

コマンド1：my_todo --edit

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．

4 結果と今後の課題

4.1 本研究の成果

my_help のテスト開発が成功したため、仕様や動作の標準が確定しました。私は、my_help は何度も使用することで、どのように進化させれば便利なのかがわかってくると考えているので、my_help はまだまだ進化する機会が出てくると思います。つまり、テスト開発をしたことで、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 の進化の開発だけでなく、my_help の使用方法が明確になったことで、私が作成した 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ると考えています。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。

4.2 今後の課題

my_help のテスト開発は成功したが、以下のような問題点が出てきました。

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にできました。しかし、日本語で書くことができるので容易に記述することも可能であると考えていましたが、The RSpec Book に記載されているテンプレートが少なく、理解が難しく感じました。自力で作成することはできますが、やはり時間がかかってしまうと考えます。ネットで検索しても、The RSpec Book と類似したシナリオが多いのも問題の一つであり、書き方のテンプレートがもっと必要です。地道に Cucumber を使用して、データベースのようなものを作成するのがよいと考えました。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。

1. BDD を行うときに ruby の知識が必要である。

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。ステッ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。

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：株式会社クイープ 監修：株式会社クイープ 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.