

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

0.1	my_help というソフト	4
0.2	共同開発のためには, テストが必要だが, 不十分.	4
0.3	BDD に従ってテストを書いていく	4
1	BDD	5
2	Cucumber と features の説明	6
3	my_help について	8
4	my_help のインストール	8
4.1	github に行って daddygongon の my_help を fork する	8
5	my_help の更新	9
5.1	git を用いて my_help を新しくする.	9
5.2	次にとってきた.yml を /.my_help に cp する.	9
6	実際に BDD を使ってコードを書く流れを説明する.	11
7	todo を更新するときのマニュアル	11
8	インストール	11
8.1	まず rspec と cucumber を gem で install する	11
9	Cucumber	12
9.1	step	16
10	RSpec	17
11	新しい item を specific_help に追加	19
12	全ての help 画面の表示	19
13	過去にバックアップしてある item のリストの表示	19

14	help コマンドの追加や削除, 編集をするファイルの開示	19
15	specific_help の item の消去	19
16	item のバックアップ	19
17	hiki への format の変更	20
18	todo の更新	20
	ユーザメモソフト my_help の開発那須比呂貴	

目次

0.1 my_help というソフト

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。

0.2 共同開発のためには、テストが必要だが、不十分。

しかし、この Ruby で書かれたソフトは動作するが、テストが用意されていない。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。

0.3 BDD に従ってテストを書いていく

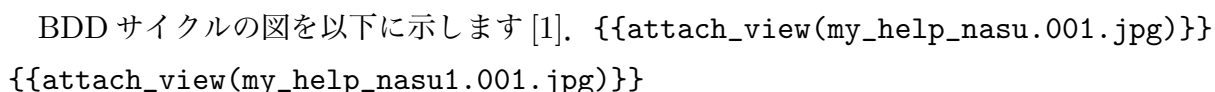
本研究の目的は、ユーザメモソフトである my_help のテスト開発である。ここでは、テスト駆動開発の中でも、ソフトの振る舞いを記述する Behavior Driven Development(BDD) に基づいてテストを記述していく。そこで、my_help がどのような振る舞いをするのかを Cucumber と RSpec を用いて BDD でコードを書いていく。Cucumber は自然言語で振る舞いを記述することができるため、ユーザにとって、わかりやすく振る舞いを確認することができる。

1 BDD

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めました。

BDD は構造ではなく振る舞いに焦点を合わせます。それは開発のすべてのレベルでいっかんしてそうなります。2つの都市の間の距離を計算するオブジェクトのことであっても、サードパーティのサービスに検索を委任する別のオブジェクトのことであっても、あるいはユーザーが無効なデータを入力したときにフィードバックを提供する別の画面であっても、それはすべて振る舞いなのです。これを飲み込んでしまえば、コードに取り組むときの考え方が変わります。オブジェクトの構造よりも、ユーザーとシステムの間でのやり取り、つまりオブジェクトの間でのやり取りについて考えるようになります。

ソフトウェア開発チームが直面する問題のほとんどは、コミュニケーションの問題であると考えています。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かすることで、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]。

BDD サイクルの図を以下に示します [1]。{{attach_view(my_help_nasu.001.jpg)}}
{{attach_view(my_help_nasu1.001.jpg)}}


2 Cucumber と features の説明

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]。下記に例を示しています。

```
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、過去を残し
   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
```

このように日本語でシナリオを書くことができ、顧客にもわかりやすく、開発者も書きやすくなっている。

feature ファイルに書くキーワードは下記の通りになっている。

表 1

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”

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.

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.

4 my_help のインストール

4.1 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`

5 my_help の更新

5.1 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)

5.2 次にやってきた.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.

BDD の具体的な記述と，想定した使用法との対比を次に示す．

6 実際に BDD を使ってコードを書く流れを説明する.

ここでは，卒業研究の目的の意義を理解してもらうために，全体の概要を説明する.

7 todo を更新するときのマニュアル

1. `my_todo -edit` を入力して `/.my_help/my_todo.yml` を開く
2. `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

これを実際に振る舞いがきちんとできているのかを確認する.

8 インストール

8.1 まず `rspec` と `cucumber` 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.

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.

9 Cucumber

以下は todo の更新を行うときの feature である.

1. 適当なディレクトリに features というディレクトリを作成する.
2. その features ディレクトリに my_todo.feature を作成する.

```
1
2 # language: ja 言語の設定（ここでは日本語に設定している）#機能
3
4 : の更新を行うtodoは更新していくものであり
5 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
   去を残しておく
6 todoシナリオ
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
```

feature を書けたら次は cucumber を実行してみる

```
/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/my_todo.feature:7

```

もし"my_todo --edit"と入力する # features/my_todo.feature:8
ならば edit が開かれる          # features/my_todo.feature:9
かつ自分の todo を書き込む      # features/my_todo.feature:10

```

```

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.feature:11
  前提 todo の編集が終わった # features/my_todo.feature:13
  もし"my_todo --store [item]"と入力する # features/my_todo.feature:14
  ならば item のバックアップを取る # features/my_todo.feature:15

```

```

2 scenarios (2 undefined)
7 steps (7 undefined)
0m0.080s

```

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

```

```
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
```

と表示される.

次に features ディレクトリの中で step_definitions ディレクトリを作成する.
step_definitions ディレクトリの中に my_todo_spec.rb を作成する. 中身は以下の通りである.

```
1 前提
2 (/^を編集したいtodo$/) do
3   pending # Write code here that turns the phrase above
         into concrete actions
4 endもし
5
6 (/^"(["])*"と入力する"$/) do |arg1|
7   pending # Write code here that turns the phrase above
         into concrete actions
8 endならば
9
10 (/^が開かれるedit$/) do
11   pending # Write code here that turns the phrase above
         into concrete actions
12 endならば自分の
13
14 (/^を書き込むtodo$/) do
15   pending # Write code here that turns the phrase above
         into concrete actions
16 end前提
17
18 (/^の編集が終わったtodo$/) do
19   pending # Write code here that turns the phrase above
         into concrete actions
```

```

20 endならば
21
22 (/^のバックアップを取るitem$/ ) do
23   pending # Write code here that turns the phrase above
           into concrete actions
24 end

```

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.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: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:3

ならば edit が開かれる # features/step_definitions/my_todo_spec.rb:4

かつ自分の todo を書き込む # features/step_definitions/my_todo_spec.rb:5

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.feature:7

前提 todo の編集が終わった # features/step_definitions/my_todo_spec.rb:6

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_spec.rb:7

ならば item のバックアップを取る # features/step_definitions/my_todo_spec.rb:8

```
2 scenarios (2 pending)
7 steps (5 skipped, 2 pending)
0m0.045s
```

と変化が出てくる。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 skipped したことを表している。step_definitions の my_todo_spec.rb の pending 部分を書き換えて進行していく。下記が書き直したコードである。

9.1 step

1 完成したやつ

ここで cucumber を実行すると全て成功しているのがわかります。

```
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 0m0.029s

```

10 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 完成したやつ
```

このように他の振る舞いのコードも書き進めていくのが BDD であり、今回のシステムの開発です。

11 新しい item を specific_help に追加

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

コマンド：emacs_help -add [item]

specific_help とは，ユーザが作成するそれぞれのヘルプである．

ヘルプの内容は /.my_help/emacs_help.yml に元 data がある．

12 全ての help 画面の表示

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

13 過去にバックアップしてある item のリストの表示

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

14 help コマンドの追加や削除，編集をするファイルの開示

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．

15 specific_help の item の消去

シナリオ：コマンドを入力して item を消す

コマンド：emacs_help -remove

16 item のバックアップ

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]

17 hiki への format の変更

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

18 todo の更新

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.

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.