

卒業論文

hikiutils を用いた 卒業論文作成

関西学院大学 理工学部 情報科学科

1234 西谷滋人

2017 年 3 月

指導教員 西谷 滋人 教授

目次

1	序論	3
2	方法	4
2.1	ruby_novice の設計仕様	4
2.2	コードテスト環境	5
3	結果と分析	8
3.1	ruby_novice の振る舞いと意義	8
3.2	ruby_novice の仕組み	8
3.3	ruby_novice の現状	13
3.4	ruby_novice の作業の流れ	14
3.5	ruby_novice の使用法	15
3.6	全章のテストの仕方	19
3.7	各章ごとのテストの仕方	19
4	考察	21
4.1	なぜ aruba? (aruba vs test::unit)	21
5	結論	25
6	謝辞	26
7	参考文献	26
	GitHub を利用した Ruby 初心者学習ソフトの開発	

目次

- `{{attach_anchor(ruby_novice_koki.pdf,ruby_novice_koki)}}`

1 序論

Ruby は本格的なオブジェクト指向プログラムが記述できる汎用性の高い日本発のオープンソースである。Ruby は初心者に分かり易く、プログラム教育にもスムーズに活用できるメリットがある [1]。西谷研究室に在籍している学生は、Ruby プログラミングを修得するために初心者向けの問題集を使って学習している。

ところが開発現場においては単に文法やプログラミングの書き方を知っているだけでは未熟で、より多くのスキルが要求される。典型的なものがバックアップに対するスキルである。バックアップをとるあるいはおいておくことはプログラミングの初心者には強調されるが、実際にバックアップのスキルを具体的に指示する指導は行われていない。現在のプログラミング環境においては Github がその標準となりつつある。Github はバックアップだけでなく、進捗確認、バージョン管理やプルリクエストといった、チームによるプログラミングを促進するサービスが提供されている。

一方で、プログラミング開発の最先端の技法として Test 駆動開発 (Test Driven Development:TDD) が奨励されている。TDD では仕様を満たすテストを書く (Red)、テストと通るコードを書く (Green)、コードを読みやすく直す (Refactoring) というステップでプログラミングを進めていくことを基本としている。それぞれの段階でなにに目標をおいて集中するかが明確になり、コード開発の効率が上がるとされている。

「初学者がこれらのスキルを自然と身につけることはできないか？」という問いに対する一つの答えとして `ruby_novice` を開発する。`Ruby_novice` が目指すものは、学習者自身が出力チェックできるようにし Ruby プログラミングにおけるテスト実行に自然と慣れるような学習形態を目指している。さらに、進捗状況の管理や指導者からの添削をより容易におこなえるように改善するため、バージョン管理ソフト GitHub を利用するシステム (`ruby_novice`) を開発している。本研究は Ruby 初心者が文法だけでなく、プログラミングにおける振舞いを身につけるための支援ソフトを開発することを目的としている。

2 方法

2.1 ruby_novice の設計仕様

ruby_novice が想定している操作法について概略を記す.

2.1.1 Github

本研究では Github を使用し, 進捗状況の管理や指導者からの添削をより容易できるようにする. Github は, コンピュータプログラムの元となるソースコードをインターネット上で管理するためのサービスである. 複数人が携わるソフトウェア開発において, ソースコードの共有や, バージョン管理といった作業は必要不可欠となる [2]. 本研究では, 下記の図のように Github を利用している.

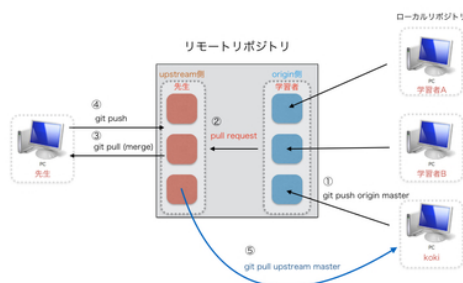


図1 Github のしくみ

ここからは, 上記の図を参考にしながら Github を利用した作業の流れを段階を踏んで示します.

2.1.2 進捗状況の報告

まずは本研究での進捗状況の報告までの簡単な流れは以下の通りである.

1. ファイルを作成する.

2. `git remote -v`: origin が自分のアドレスで upstream が先生のアドレスであるか確かめる.
3. `git add -A`: 編集操作を local の repository に登録.
4. `git commit`: ファイルの追加や変更の履歴をリポジトリに保存.
5. `git push origin master`: 図の操作により, Github の origin へ master を push.
6. pull request: 図の操作により, Github で自分のサイトに載せた変更を, 先生のサイトに変更希望として出す. コメント欄で変更詳細を伝えることが可能.

基本的にローカルリポジトリで作業を行い, その作業内容をリモートリポジトリ (Github) へプッシュする流れで行う.

2.1.3 添削後の作業の流れ

1. 先生がファイルを添削後, 図の操作により, リモートリポジトリ (Github) に `git push`.
2. `git pull upstream master`: 図の操作により, 自分の開発中のファイルに反映.

このサイクルを繰り返して, 研究または, 課題を進めていきます.

それぞれの用語の説明は以下の通りである.

- リポジトリ: ファイルやディレクトリの状態を保存する場所.
- ローカルリポジトリ: 自分のマシン内にあるリポジトリ.
- リモートリポジトリ: サーバなどネットワーク上にあるリポジトリ.
- コミット (commit): ファイルの追加や変更の履歴をリポジトリに保存すること.
- origin: リポジトリの場所 (URL) の別名.
- master: ブランチの名前.
- プッシュ (push): ファイルの追加や変更の履歴をリモートリポジトリにアップロードするための操作.

2.2 コードテスト環境

`ruby_novice` では提出されたコードを開発現場で使用されている一般的なテスト環境でテストする. 本研究でモデルとしたテスト駆動開発ならびに比較検討したフレームワークを示す.

2.2.1 TDD (Test Driven Development)

2000 年代初期に開発手法として確立された「テスト駆動開発」(Test Driven Development) は, その後 10 年もの間で普及が進み, 今や珍しくない開発スタイルの 1 つとなっている. 国内でも「アジャイルアカデミー」「TDD Boot Camp」などによる推進・普及活動が各地で活発化し, 認知が広がっている [3].

テスト駆動開発は, 簡単に言うとプログラムを書く前にテストコードを書くということです. プログラムが完成した後 にテストコードを書くのではなく, テストコードを先に書くことに大きな意味があります. それは先に仕様を決め, テストコードを書くことによって自分が次にやることが明確になるためです. これにより作業効率も上がります. 最初にいきなりプログラムを書くと, 整理されていないプログラムが出来てしまいます. しかしはじめにテストコードを書くことによって何をすべきか明確になるのでプログラムが書きやすくなります. 他に TDD の目的としては, 軽快なフィードバックの確保, きれいで動くコードの確保などによる開発の改善が挙げられます. テスト駆動開発は, テストファーストによる追加・変更とリファクタリングによる設計改善という 2 つの活動で構成されます. 継続的にユニットテストを使って設計検討やチェック, リファクタリングを行うことにより, テスタビリティに優れバグの少ないソースコードを実現することができます.

2.2.2 test::unit とは

Ruby 用の xUnit 系の単体テストフレームワークである. Ruby1.8 までは Ruby 本体に標準添付されていたが, Ruby1.9.1 からは minitest というフレームワークが標準添付されている. test-unit が Ruby1.8 に標準添付されていた頃はほとんど機能拡張などがされず, RSpec など新しいテストフレームワークから見劣りするものとなっていた. しかし, Ruby 標準添付ではなく, 1 つのプロジェクトとして開発が進められるようになってからは活発に開発が進められている. Ruby 本体のバージョンアップに関係なく新しいバージョンをリリースできるようになったことも開発が活発になった理由の一つである [4].

2.2.3 aruba とは

Aruba は Cucumber, RSpec, Minitest のような人気のある TDD/BDD フレームワークでコマンドラインアプリケーションのテストを簡単で楽しいものにする拡張である. 特徴としては以下の通りである [5].

- どんな言語で実装されたコマンドラインツールでもテスト可能.

- テスト自体は Ruby で書くが, テスト対象は, Python の CLI ツールでも Golang の CLI ツールでもよい.
- ファイルシステムやプロセス環境をヘルパーによって操作できる.
 - 例えば, read でファイルを読み込みできる.
 - 例えば, run で外部コマンドを実行し, その結果を have_output matcherなどで検証できる.
- ファイルシステムやプロセス環境はテストのたびにリセットされるので, leaking state がない.
 - 例えばテスト中に作成されたファイルはテスト終了後には消えている.
- コミュニティーサポートが手厚い.
- ドキュメントにあるとおりに動作することが期待できる [5].

3 結果と分析

3.1 ruby_novice の振る舞いと意義

ruby_novice は、情報環境である GitHub を利用し Ruby 初心者が文法だけでなく、Ruby プログラミングにおける振る舞いを身につけるための支援ソフトを開発する。また Ruby プログラミングで重要となるテスト駆動をおこなえる環境を提供している。これにより、学習者自身が出力チェックできるようにし Ruby プログラミングにおけるテスト実行に自然と慣れるような学習形態を目指している。

3.2 ruby_novice の仕組み

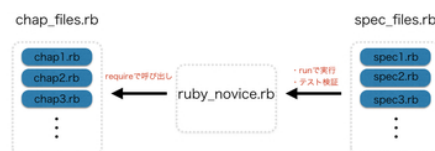


図 2 ruby_novice の構造.

ruby_novice の構造は、上記の図のように 3 つに分かれています。

chap_files.rb (chap1.rb, chap2.rb ...) Text(たのしい Ruby) のコードを書く部分.

ruby_novice.rb chap_files.rb を呼び出している.

spec_files.rb run で外部コマンドを入力して、出力結果 = 期待している値の検証.

テストコードが書いている spec ファイルを各章ごとに分け、ruby_novice.rb で呼び出すことにより、章ごとにテストを実行することを可能にした。

以下が ruby_novice.rb のコードの中身である。「たのしい Ruby」の1章に対応するコードのみを抜粋している。

```
1  #ruby_novice.rb
2
3  $LOAD_PATH.unshift File.expand_path("../../lib/#{ENV['
    RUBYNOVICE_NAME']}", __FILE__)
4  begin
5    require "chap_files"
6  rescue LoadError
7    p "Load_Error_of_ex_files_in_rubynovice.rb."
8    p File.expand_path("../../lib/#{ENV['RUBYNOVICE_NAME']}",
        __FILE__)
9    exit
10 end
11
12 require "ruby_novice/version"
13 require 'thor'
14 #require "code"
15
16 module RubyNovice
17   # Your code goes here...
18
19   class CLI < Thor
20     # class_option :help, type: :boolean, aliases: '-h',
21       desc: 'help.'
22     # class_option :debug, type: :boolean, aliases: '-d',
23       desc: 'debug mode'
24
25   =begin
26     desc 'hello', 'print_hello'
27     def hello
28       my_hello
29     end
30   =end
31
32     desc 'my_helloruby', 'print_helloruby'
33     def my_helloruby
34       helloruby
35     end
36
37     desc 'my_puts_and_p', 'print_puts_and_p'
38     def my_puts_and_p
39       puts_and_p
40     end
41
42     desc 'my_kiritsubo', 'print_kiritsubo'
```

```

41     def my_kiritsubo
42         kiritsubo
43     end
44
45     desc 'my_area_volume', 'print_area_volume'
46     def my_area_volume
47         area_volume
48     end
49
50     desc 'my_comment_sample', 'print_comment_sample'
51     def my_comment_sample
52         comment_sample
53     end
54
55     desc 'my_greater_smaller', 'print_greater_smaller'
56     def my_greater_smaller
57         greater_smaller
58     end
59
60     desc 'my_greater_smaller_else', 'print_
        greater_smaller_else'
61     def my_greater_smaller_else
62         greater_smaller_else
63     end
64
65     desc 'version', 'version'
66     def version
67         puts RubyNovice::VERSION
68     end
69
70     private
71
72     def output_error_if_debug_mode(e)
73         return unless options[:debug]
74         STDERR.puts(e.message)
75         STDERR.puts(e.backtrace)
76     end
77 end
78 end

```

以下は chap1_spec.rb のコードの中身である。「たのしい Ruby」の1章に対応する spec code を書き出している。

```

1  #spec_chap1.rb
2  require 'spec_helper'
3
4  RSpec.describe 'ruby_novice_command', type: :aruba do

```

```

5     context 'version_option', type: :version do
6       before(:each) { run('ruby_novice_v') }
7       it { expect(last_command_started).to
          be_successfully_executed }
8       it { expect(last_command_started).to have_output("0.1.0
          ") }
9     end
10
11    context 'help_option', type: :help do
12      expected = 'bundle exec exe/ruby_novice help'
13      before(:each) { run('ruby_novice_help') }
14      it { expect(last_command_started).to
          be_successfully_executed }
15      # it { expect(last_command_started).to have_output(
          expected) }
16    end
17
18    =begin
19      context 'print_hello', type: :hello do
20        before(:each) { run('ruby_novice_hello') }
21        expected = "Hello."
22        it { expect(last_command_started).to
            be_successfully_executed }
23        it { expect(last_command_started).to have_output(
            expected) }
24      end
25    =end
26
27    context 'helloruby', type: :helloruby do
28      before(:each) { run('ruby_novice_my_helloruby') }
29      expected = "Hello, Ruby."
30      it { expect(last_command_started).to
          be_successfully_executed }
31      it { expect(last_command_started).to have_output(
          expected) }
32    end
33
34    context 'puts_and_p', type: :puts_and_p do
35      before(:each) { run('ruby_novice_my_puts_and_p') }
36      expected = "Hello,\n\tRuby.\n\"Hello,\n\tRuby.\""
37
38      it { expect(last_command_started).to
          be_successfully_executed }
39      it { expect(last_command_started).to have_output(
          expected) }
40    end
41

```

```

42     context 'kiritsubo', type: :kiritsubo do
43       before(:each) { run('ruby_novice_my_kiritsubo') }
44       expected = "いづれの御時にか女御更衣あまたさぶらいたまいけるなかに\いと
45         やn\
46         \むごとなき際にはあらぬがすぐれて時めきたまふありけり"
47       it { expect(last_command_started).to
48         be_successfully_executed }
49       it { expect(last_command_started).to have_output(
50         expected) }
51     end
52
53     context 'area_volume', type: :area_volume do
54       before(:each) { run('ruby_novice_my_area_volume') }
55       expected = "表面積=2200\体積n=6000"
56
57       it { expect(last_command_started).to
58         be_successfully_executed }
59       it { expect(last_command_started).to have_output(
60         expected) }
61     end
62
63     context 'greater_smaller', type: :greater_smaller do
64       before(:each) { run('ruby_novice_my_greater_smaller') }
65       expected = "greater"
66
67       it { expect(last_command_started).to
68         be_successfully_executed }
69       it { expect(last_command_started).to have_output(
70         expected) }
71     end
72
73     context 'greater_smaller_else', type: :
74       greater_smaller_else do
75       before(:each) { run('ruby_novice_
76         my_greater_smaller_else') }
77       expected = "greater"
78
79       it { expect(last_command_started).to
80         be_successfully_executed }
81       it { expect(last_command_started).to have_output(
82         expected) }
83     end
84   end
85 end

```

chap1_spec.rb などが呼び出す spec_helper.rb は以下の通りである。 \$LOAD_PATH に

gem の標準構造の場合に配置される lib を入れている。また、その後は support directory であるが、RUBY の version が古い場合にも対応するように設定している。

```
1 #spec_helper.rb
2 $LOAD_PATH.unshift File.expand_path('../../lib', __FILE__)
3 require 'ruby_novice'
4 #require 'aruba/rspec'
5 $LOAD_PATH.unshift File.expand_path('../../lib', __FILE__)
6
7 if RUBY_VERSION < '1.9.3'
8   ::Dir.glob(::File.expand_path('../support/*.rb', __FILE__))
9     .each { |f| require File.join(File.dirname(f), File.
10       basename(f, '.rb')) }
11   ::Dir.glob(::File.expand_path('../support/**/*.rb',
12     __FILE__)).each { |f| require File.join(File.dirname(f),
13       File.basename(f, '.rb')) }
14 else
15   ::Dir.glob(::File.expand_path('../support/*.rb', __FILE__))
16     .each { |f| require_relative f }
17   ::Dir.glob(::File.expand_path('../support/**/*.rb',
18     __FILE__)).each { |f| require_relative f }
19 end
```

3.3 ruby_novice の現状

現状は、「たのしい Ruby」の第 1 章 第 7 章までのテストコードを書き実装できる。各章の概要は、以下の通りである。

- 第 1 章 (list1.1 1.7): puts メソッドや p メソッド
- 第 3 章 (list3.1 3.11): ファイルの読み込み
- 第 4 章 (list4.1): ローカル変数とグローバル変数
- 第 5 章 (list5.1 5.5): 条件判断 (if, unless など)
- 第 6 章 (list6.1 6.13): 繰り返し (for,times,while など)
- 第 7 章 (list7.1 7.4): メソッド

3.3.1 注意

「たのしい Ruby」の課題では、通し番号以外に、コードに対応する適当なプログラム名が付されている。しかし、Rub 言語の予約語 (for,while など) はコード中で使えないため、以下の問題は名前を変更して”1”をつけている。

- list5.3: unless.rb → unless1.rb に変更.
- list5.4: case.rb → case1.rb に変更.
- list6.4: for.rb → for1.rb に変更.
- list6.6: while.rb → while11.rb に変更.
- list6.9: until.rb → until1.rb に変更.
- list7.4: myloop.rb → myloop1.rb に変更.

3.4 ruby_novice の作業の流れ

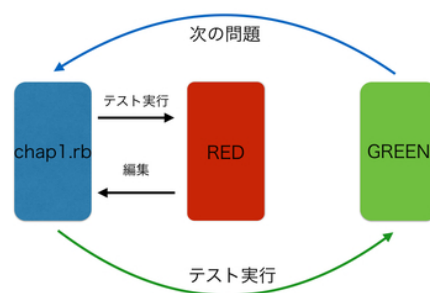


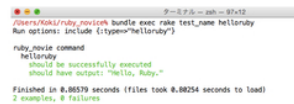
図3 学習の流れ.

図3のようにRuby学習者はRed, Greenという作業サイクルを繰り返してプログラミングを進めていきます.

1. 作成したいプログラムの仕様を明確にする.
2. Red (テストに失敗)
3. Green(Redの状態ならば, 編集しテストを成功させるコードを書く)
4. Greenになると次の問題に進む.

Red, Greenという言葉は, TDDで多用されるテストフレームワークの多くがテスト失敗を赤色表示で, テスト成功を緑色表示で通知することに由来している. 図4がテストにパスした時の出力結果で, 図5がテストに失敗した時の出力結果である. 色を見るだ

けでテストをパスしているか失敗しているか一目瞭然である。

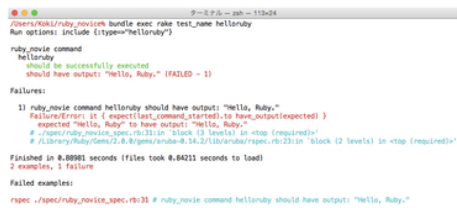


```
ターミナル - ssh - 87x12
/Users/koki/ruby novice bundle exec rake test_name helloruby
Run options: include {:types=>helloruby}

ruby novice command
helloruby
should be successfully executed
should have output: "Hello, Ruby."

Finished in 0.00279 seconds (files took 0.00254 seconds to load)
2 examples, 0 failures
```

図 4 Green の出力結果。



```
ターミナル - ssh - 113x24
/Users/koki/ruby novice bundle exec rake test_name helloruby
Run options: include {:types=>helloruby}

ruby novice command
helloruby
should be successfully executed
should have output: "Hello, Ruby." (FAILED - 1)

Failures:
  1) ruby novice command helloruby should have output: "Hello, Ruby."
     Failure/Error: it { expect(last_command_started).to have_output(expected) }
       expected "Hello, Ruby" to have output: "Hello, Ruby."
       # ./spec/ruby novice_spec.rb:22:in "block (2 levels) in <top (required)>"
       # /Library/Ruby/Gems/2.0.0/gems/rspec-0.14.2/lib/rspec/rspec.rb:22:in "block (2 levels) in <top (required)>"

Finished in 0.00981 seconds (files took 0.04211 seconds to load)
2 examples, 1 failure

Failed examples:
rspec ./spec/ruby novice_spec.rb:22 # ruby novice command helloruby should have output: "Hello, Ruby."
```

図 5 Red の出力結果。

3.5 ruby_novice の使用法

1. 自分の好きな名前 (koki) をつけたディレクトリを作成する。
2. ./lib/koki/chap_files.rb を準備する。

3. chap_files.rb の中に require "chap1" と書く.
4. chap1.rb というファイルを作り, そのファイルにたのしい Ruby 1 章の list(1.1 1.7) のコードを書いていく.
5. rspec で, 個人ごとの検査を実行する場合, 環境変数 RUBYNOVICE_NAME にディレクトリ名 (koki) を入れる.

- (csh,tcsh)setenv RUBYNOVICE_NAME koki
- (bash,zsh)export RUBYNOVICE_NAME=koki

コード例

```
1  #!/Users/Koki/ruby_novice% cat lib/koki/chap_files.rb
2
3      require "chap1"
4  #require "chap3"
5  #require "chap4"
6  #require "chap5"
7  #require "chap6"
8  #require "chap7"注
9
10      () # はコメントアウト.
```

コード例 (たのしい Ruby 第 1 章)

```
1  #!/Users/Koki/ruby_novice% cat lib/koki/chap1.rb
2
3  def helloruby
4    print("Hello , Ruby.\n")
5  end
6
7  def puts_and_p
8    puts "Hello ,\n\tRuby ."
9    p "Hello ,\n\tRuby ."
10 end
11
12 def kiritsubo
13   print "いづれの御時にか女御更衣あまたさぶらいたまいけるなかに\n"
14   print "いとやむごとなき際にはあらぬがすぐれて時めきたまふありけり\n"
15 end
16
17 def area_volume
18   x = 10
19   y = 20
20   z = 30
21   area = (x*y + y*z + z*x) * 2
```



```

22     volume = x * y * z
23     print "表面積=", area, "\n"
24     print "体積=", volume, "\n"
25 end
26
27 def comment_sample
28 =begin 「たのしい
29     Ruby 第5版」サンプルコメントの使い方の例
30
31     2006/06/16 作成
32     2006/07/01 一部コメントを追加
33     2015/10/01 第5版用に更新
34 =end
35
36     x = 10 # 縦
37     y = 20 # 縦
38     z = 30 # 高さ
39     # 表面積と体積を計算する
40     area = (x*y + y*z + z*x) * 2
41     volume = x * y * z
42     # 出力する
43     print "表面積=", area, "\n"
44     print "体積=", volume, "\n"
45 end
46
47 def greater_smaller
48     a = 20
49     if a >= 10 then
50         print "greater\n"
51     end
52     if a <= 9 then
53         print "smaller\n"
54     end
55 end
56
57 def greater_smaller_else
58     a = 20
59     if a >= 10
60         print "greater\n"
61     else
62         print "smaller\n"
63     end
64 end

```

3.5.1 tag の表示の仕方

1. `grep type spec/ruby_novice_spec.rb` で全ての context と type を表示. type は各章の各問題名に相当する. 各問題ごとにテストする時の便宜となる.

```
1 context 'version_option', type: :version do
2 context 'help_option', type: :help do
3 context 'print_hello', type: :hello do
4 context 'helloruby', type: :helloruby do
5 context 'puts_and_p', type: :puts_and_p do
6 context 'kiritsubo', type: :kiritsubo do
7 context 'area_volume', type: :area_volume do
8 context 'comment_sample', type: :comment_sample do
9 context 'greater_smaller', type: :greater_smaller do
10 context 'greater_smaller_else', type: :
    greater_smaller_else do
11 context 'print_argv', type: :print_argv do
12 context 'happy_birth', type: :happy_birth do
13 context 'arg_arith', type: :arg_arith do
14 context 'read_text', type: :read_text do
15 context 'read_text_simple', type: :read_text_simple do
16 context 'read_text_online', type: :read_text_online do
17 context 'read_line', type: :read_line do
18 context 'simple_grep', type: :simple_grep do
19 context 'hello_ruby2', type: :hello_ruby2 do
20 context 'use_grep', type: :use_grep do
21 context 'scopetest', type: :scopetest do
22 context 'ad2heisei', type: :ad2heisei do
23 context 'if_elsif', type: :if_elsif do
24 context 'unless1', type: :unless1 do
25 context 'case1', type: :case1 do
26 context 'case_class', type: :case_class do
27 context 'times', type: :times do
28 context 'times2', type: :times2 do
29 context 'times3', type: :times3 do
30 context 'for1', type: :for1 do
31 context 'for_names', type: :for_names do
32 context 'while1', type: :while1 do
33 context 'while2', type: :while2 do
34 context 'while3', type: :while3 do
35 context 'until1', type: :until1 do
36 context 'while_not', type: :while_not do
37 context 'each_names', type: :each_names do
38 context 'each', type: :each do
39 context 'break_next', type: :break_next do
40 context 'times_with_param', type: :times_with_param do
```

```
41 context 'hello_with_name', type: :hello_with_name do
42 context 'hello_with_default', type: :hello_with_default do
43 context 'myloop1', type: :myloop1 do
```

3.6 全章のテストの仕方

1. bundle exec rspec すべての章のテストを一括して実行できる.

3.7 各章ごとのテストの仕方

例: 1 章 (chap1) のテストをしたい時. 1. bundle exec rspec spec/chap1_spec.rb 2.
bundle exec rake chap 1

実行例

```
1 /Users/Koki/ruby_novice% bundle exec rake chap 1
2
3 ruby_novie command
4   version option
5     should be successfully executed
6     should have output: "0.1.0"
7   help option
8     should be successfully executed
9   helloruby
10    should be successfully executed
11    should have output: "Hello, Ruby."
12  puts_and_p
13    should be successfully executed
14    should have output: "Hello, \n\tRuby.\n"Hello, \n\tRuby
15    .\n"
16  kiritsubo
17    should be successfully executed
17    should have output: "いづれの御時にか女御更衣あまたさぶらいたまいけ
    るなかに\いとやむごとなき際にはあらぬがすぐれて時めきたまふありけりn"
18  area_volume
19    should be successfully executed
20    should have output: "表面積=2200\体積n=6000"
21  comment_sample
22    should be successfully executed
23    should have output: "表面積=2200\体積n=6000"
24  greater_smaller
25    should be successfully executed
26    should have output: "greater"
27  greater_smaller_else
28    should be successfully executed
```

```
29         should have output: "greater"
30
31 Finished in 7.61 seconds (files took 1.03 seconds to load)
32 17 examples, 0 failures
```

3.7.1 各問題ごとのテストの仕方

例: 各問題 (helloruby) ごとにテストをしたい時. 1. `bundle exec rspec -tag type:helloruby spec/ruby_novice_spec.rb` (helloruby は問題名) 2. `bundle exec rake test_name helloruby`

実行例

```
1 /Users/Koki/ruby_novice% bundle exec rake test_name
  helloruby
2 Run options: include {:type=>"helloruby"}
3
4 ruby_novie command
5   helloruby
6     should be successfully executed
7     should have output: "Hello, Ruby."
8
9 Finished in 0.87128 seconds (files took 0.81684 seconds to
  load)
10 2 examples, 0 failures
```

問題名は, 上記の `grep type spec/ruby_novice_spec.rb` で調べることができる. `type` が各問題の名前になる. また `text` の問題名 (例えば `puts_and_p.rb`) が, そのまま使えるのでテストも簡単にでき, 問題名で中身のコードの内容も把握できる.

3.7.2 各問題ごとの実行結果の出力

例: helloruby の実行結果の出力

1. `bundle exec exe/ruby_novice my_helloruby` 2. `bundle exec rake/output helloruby`

実行例

```
1 /Users/Koki/ruby_novice% bundle exec rake output helloruby
2 Hello, Ruby.
```

4 考察

4.1 なぜ aruba? (aruba vs test::unit)

Cucumber,RSpec,Minitest のような人気のある TDD/BDD フレームワークの中でも aruba を使用した理由は以下の通りである． test:unit や aruba で書くとどうなるかを具体的に書いたコードを比べて示していきます．

4.1.1 test::unit で書いたテストコード

たのしい Ruby のテキストに記載されている問題で比較していきたいと思います． テキストの最初の問題は、Hello, Ruby を出力するプログラムです．

```
print("Hello, Ruby.\n")
```

まず、出力される Hello, Ruby をテストする場合のコードです．

```
1 #helloruby.rb
2
3 def helloruby
4   return "Hello, Ruby.\n"
5 end
```

- test::unit で書いたテストコード

```
1 require 'test/unit'
2 require './helloruby'
3
4 class Test_Sample < Test::Unit::TestCase
5   def test_helloruby
6     assert_equal("Hello, Ruby.\n",helloruby)
7   end
8 end
9 print("Hello, Ruby.\n")
```

テストコードの内容は以下の通りである． Ruby で代表的な test/unit という gem が提供されています． このプログラムの始め（require 'test/unit'）で、 test/unit を呼び出します． Test::Unit::TestCase を継承したクラスを用意し、 test_xxx というメソッドを定義するとそのメソッドがテストの実行対象になり、ここではそれぞれ Test_Sample クラスと test_helloruby メソッドがそれに該当します． クラス名は大文字から始めるという規則が

ありますので注意してください。またメソッド名は、必ず `test_` から始めなくてははいけません。ここでは単純に `test_helloruby` としています。実行してみると分かりますが、`test_` がないとちゃんと動いてくれません。テストコードは、`assert_equal(期待値),(実際の値)` で実行結果を検証します。`assert_equal` は、ふたつの引数を取り、第1引数は期待している結果で、第2引数はテストの対象です。両者が一致すればテストをパスし、一致しない場合はテストが失敗する。補足ですが、`test_xxx` というメソッドはクラス内に複数あっても構いません。また、1つのテストメソッド内に `assert_equal` を複数書くのも OK です。（とはいえ、原則として1テストメソッドにつき1アサーションとするのが望ましい）

このテストを実行すると以下のような出力になります。

```
1 /Users/Koki/rubynovice/spec/test_unit/list1% ruby
   test_helloruby.rb
2 Hello , ruby.
3 Loaded suite test_helloruby
4 Started.
5
6 Finished in 0.000982 seconds.
7
8 1 tests , 1 assertions , 0 failures , 0 errors , 0 pendings ,
   0 omissions , 0 notifications
9 100% passed
10
11 1018.33 tests/s, 1018.33 assertions/s
```

4.1.2 test::unit での問題点

この場合だと初心者である Ruby の学習者がスクリプトとテストコードを同時に書かなければならない。学習者は、テストコードの書き方も学ぶ必要があるので、学習コストや間違えるリスクが大きくなる。一番の問題点は、テキストを見ながら、その問題通りに書けないということです。先ほどの問題で説明すると、コードに `return` を付け加えなければならないことや、`print` メソッドは `return` できないので、テストするときは `return "Hello, Ruby.`

`n"`と書き換えなければなりません。このように `test::unit` だとメソッドを書き換えないといけないことや、`print` メソッドを `return` で返すことができないというデメリットがある。そこで `aruba` は `print` をそのまま出力できテストが可能である。学習者が `text`（たのしい Ruby）を見ながら書いていけるというメリットがあるので学習コストや間違えるリ

スクを削減できます。実際に aruba で書いたコードを元にして具体的に示します。

4.1.3 aruba で書いたテストコード

先ほどと同じ Hello, Ruby を出力するプログラムをテストします。

```
1 # code.rb
2
3 def helloruby
4   print("Hello, Ruby.\n")
5 end
```

```
1 #ruby_novice.rb
2
3 require 'thor'
4 require "code.rb"
5
6 module RubyNovice
7   class CLI < Thor
8     desc 'my_helloruby', 'print helloruby'
9     def my_helloruby
10      helloruby
11    end
12  end
```

require で, thor と code.rb を呼び出しています。thor は, コマンドラインツールを作るための gem です。引数の受け渡しを簡潔に書くことができ, オプションのパーズや Usage Message の表示など簡単に作成できます。

次にテストコードですが, aruba の場合 print メソッドを return せずにそのままテストが可能になります。下記がこの問題でのテストコードです。

```
1 #ruby_novice_spec.rb
2
3 require 'spec_helper'
4
5 RSpec.describe 'ruby_novice_command', type: :aruba do
6   context 'helloruby', type: :helloruby do
7     before(:each) { run('ruby_novice_my_helloruby') }
8     expected = "Hello, Ruby."
9     it { expect(last_command_started).to be_successfully_executed }
10    it { expect(last_command_started).to have_output(expected) }
11  end
12 end
```

テストコードの意味は次の通りです.

`run('ruby_novice my_helloruby')` `ruby_novice` の `my_helloruby` を実行する.

`expected = "Hello, Ruby."` 期待している結果. `test::unit` でいう第 1 引数である.

`expect(last_command_started).to be_successfully_executed` `status 0` で終了していることを確認. このコードでエラーなく終了したことを確認する.

`expect(last_command_started).to have_output(expected)` 出力が `contents` であることを確認, 正規表現も使用可能である. このコードで期待値=実際の値であることを検証します. 両者が一致すればテストをパスし, 一致しない場合はテストが失敗する.

5 結論

同じ問題に対して、実際に aruba でのテストコードと test::unit でのテストコードを書き、具体的に出力結果やコードを比較したことによって双方の良い点や問題点を考察することができた。今回の開発目的としては、「Ruby 初心者が文法だけでなく、Ruby プログラミングにおける振舞いを身につけるための支援ソフトの開発」ということなので、aruba でテストコードを書いた。なぜ aruba なのか以下に簡単にまとめてみました。

test::unit だとテストコードとスクリプトを同時に書かないといけないので、Ruby 初心者にしては学習コストや間違えるリスクが大きくなる。また text(たのしい Ruby) で描かれているコードに return を付け加えなければならないというデメリットがある。それに比べて aruba だと text(たのしい Ruby) のコードをそのまま写すだけでよく、そのコードを実行するだけでテストをすることができる。テスト環境としては、環境変数 RUBYNOVICE_NAME にディレクトリ名を入れるだけで、個人ごとにテストすることができる。また章ごとにテストコードを書いているので、各章ごとや各問題ごとにテストができ、1 問ずつ確認しながらコードを書いていくことが可能である。

今後の課題としては、現段階で text の 7 章までしかテストコードを書けていないので引き続き書くことであったり、慣れてきたら text の問題だけでなく応用の問題もテストコードを書いていくことである。

6 謝辞

本研究を進めるにあたり, 終始多大なるご指導, 御鞭撻をいただいた西谷滋人教授に対し, 深くご御礼申し上げます. また, 同研究室に所属する先輩方, 同輩達からの様々な助力, 知識の共有があり, 本研究を大成することができました. この場をお借りして心から深く感謝いたします.

7 参考文献

- [1]「Ruby 入門教育」, 池本有里, 山本耕史, <http://www.shikoku-u.ac.jp/education/docs/Ser.A%20>
- [2]「GitHub」, 横田一輝, <https://kotobank.jp/word/GitHub-1725201>.
- [3]「テスト駆動開発／振る舞い駆動開発を始めるための基礎知識」, 井芹洋輝,
http://www.atmarkit.co.jp/ait/articles/1403/05/news035_3.html.
- [4]「test-unit - Ruby 用単体テストフレームワーク」, <https://test-unit.github.io/ja/>
- [5]「QiitaAruba gem で CLI のテストを支援する」, tbpgr, <http://qiita.com/tbpgr/items/41730ec>