卒業論文

GitHub を利用した Ruby 初心者学習ソフトの開発

関西学院大学　理工学部　情報科学科

2549 浦田 航貴

2017 年 3 月

指導教員　　西谷　滋人　教授

目次

目次

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

1 序論

1.1 目的

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．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) を開発しているそこでは，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．本研究は 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を開発する．

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，コンピューターブ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.

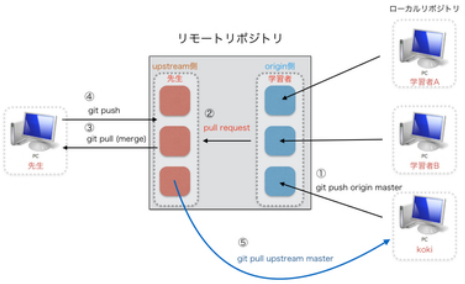

図 1　Github のしくみ

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: 図のの操作により,Github の origin へ master を push.
6. pull request: 図のの操作により,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 など新しいテスティング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.
- コミュ二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 の概要

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

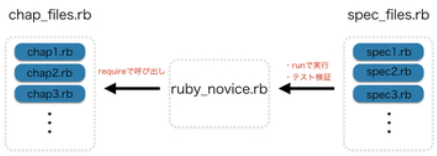

図 2　ruby_novice の構造．

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 章のみ抜粋)

```
#ruby_novice.rb

$LOAD_PATH.unshift File.expand_path("../../lib/#{ENV['RUBYNOVICE_NAME']}", _
begin
  require "chap_files"
rescue LoadError
  p "Load Error of ex_files in rubynovice.rb."
  p File.expand_path("../../lib/#{ENV['RUBYNOVICE_NAME']}", __FILE__)
  exit
end

require "ruby_novice/version"
require 'thor'
#require "code"

module RubyNovice
  # Your code goes here...

  class CLI < Thor
#    class_option :help, type: :boolean, aliases: '-h', desc: 'help.'
#    class_option :debug, type: :boolean, aliases: '-d', desc: 'debug mode'

=begin
    desc 'hello', 'print hello'
    def hello
      my_hello
    end
=end
```

```
desc 'my_helloruby', 'print helloruby'
def my_helloruby
  helloruby
end

desc 'my_puts_and_p', 'print puts_and_p'
def my_puts_and_p
  puts_and_p
end

desc 'my_kiritsubo', 'print kiritsubo'
def my_kiritsubo
  kiritsubo
end

desc 'my_area_volume', 'print area_volume'
def my_area_volume
  area_volume
end

desc 'my_comment_sample', 'print comment_sample'
def my_comment_sample
  comment_sample
end

desc 'my_greater_smaller', 'print greater_smaller'
def my_greater_smaller
  greater_smaller
end

desc 'my_greater_smaller_else', 'print greater_smaller_else'
def my_greater_smaller_else
  greater_smaller_else
```

```
    end

    desc 'version', 'version'
    def version
      puts RubyNovice::VERSION
    end

    private

    def output_error_if_debug_mode(e)
      return unless options[:debug]
      STDERR.puts(e.message)
      STDERR.puts(e.backtrace)
    end
  end
end
```

- 以下が chap1_spec.rb のコードの中身である. (たのしい Ruby の 1 章のみ抜粋)

```
#spec_chap1.rb
require 'spec_helper'

RSpec.describe 'ruby_novie command', type: :aruba do
  context 'version option', type: :version do
    before(:each) { run('ruby_novice v') }
    it { expect(last_command_started).to be_successfully_executed }
    it { expect(last_command_started).to have_output("0.1.0") }
  end

  context 'help option', type: :help do
    expected = `bundle exec exe/ruby_novice help`
    before(:each) { run('ruby_novice help') }
    it { expect(last_command_started).to be_successfully_executed }
#   it { expect(last_command_started).to have_output(expected) }
```

```
  end

=begin
  context 'print hello', type: :hello do
    before(:each) { run('ruby_novice hello') }
    expected = "Hello."
    it { expect(last_command_started).to be_successfully_executed }
    it { expect(last_command_started).to have_output(expected) }
  end
=end

  context 'helloruby', type: :helloruby do
    before(:each) { run('ruby_novice my_helloruby') }
    expected = "Hello, Ruby."
    it { expect(last_command_started).to be_successfully_executed }
    it { expect(last_command_started).to have_output(expected) }
  end

  context 'puts_and_p', type: :puts_and_p do
    before(:each) { run('ruby_novice my_puts_and_p') }
    expected = "Hello,\n\tRuby.\n\"Hello,\n\tRuby.\""

    it { expect(last_command_started).to be_successfully_executed }
    it { expect(last_command_started).to have_output(expected) }
  end

  context 'kiritsubo', type: :kiritsubo do
    before(:each) { run('ruby_novice my_kiritsubo') }
    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
\n 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"

    it { expect(last_command_started).to be_successfully_executed }
```

```
    it { expect(last_command_started).to have_output(expected) }
  end

  context 'area_volume', type: :area_volume do
    before(:each) { run('ruby_novice my_area_volume') }
    expected = "表面積=2200\n 体積=6000"

    it { expect(last_command_started).to be_successfully_executed }
    it { expect(last_command_started).to have_output(expected) }
  end

  context 'greater_smaller', type: :greater_smaller do
    before(:each) { run('ruby_novice my_greater_smaller') }
    expected = "greater"

    it { expect(last_command_started).to be_successfully_executed }
    it { expect(last_command_started).to have_output(expected) }
  end


  context 'greater_smaller_else', type: :greater_smaller_else do
    before(:each) { run('ruby_novice my_greater_smaller_else') }
    expected = "greater"

    it { expect(last_command_started).to be_successfully_executed }
    it { expect(last_command_started).to have_output(expected) }
  end
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

(注意)
予約語 (for,while) などは使えないため，以下の問題は名前を変更している．

- 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

上記の図のように 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 で多用されるテスティング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.

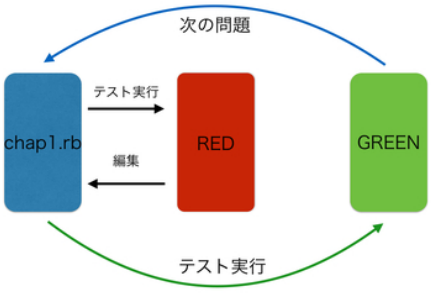

図 3　学習の流れ.

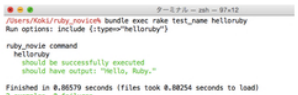

図 4　Green の出力結果.

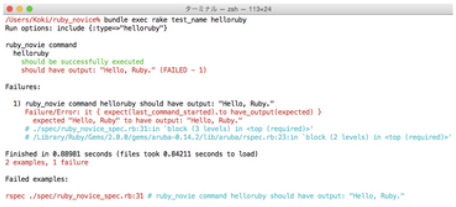

図 5　Red の出力結果.

3.5　ruby_novice の使用法

- 自分の好きな名前 (koki) をつけたディレクトリを作成し，”./lib/koki/chap_files.rb”を準備.

コード例

```
#/Users/Koki/ruby_novice% cat lib/koki/chap_files.rb

require "chap1"
 #require "chap3"
 #require "chap4"
 #require "chap5"
 #require "chap6"
 #require "chap7"
```

（注）# はコメントアウト.

- 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.

コード例 (たのしい Ruby 第 1 章)

```
#/Users/Koki/ruby_novice% cat lib/koki/chap1.rb

def helloruby
  print("Hello, Ruby.\n")
end

def puts_and_p
  puts "Hello,\n\tRuby."
  p "Hello,\n\tRuby."
end

def kiritsubo
  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"
  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"
end

def area_volume
  x = 10
  y = 20
  z = 30
  area = (x*y + y*z + z*x) * 2
  volume = x * y * z
  print "表面積=", area, "\n"
  print "体積=", volume, "\n"
end

def comment_sample
=begin
  「たのしい Ruby 第５版」サンプル
```

```
    コメントの使い方の例
    2006/06/16 作成
    2006/07/01 一部コメントを追加
    2015/10/01 第５版用に更新
=end

  x = 10 # 縦
  y = 20 # 縦
  z = 30 # 高さ
  # 表面積と体積を計算する
  area = (x*y + y*z + z*x) * 2
  volume = x * y * z
  # 出力する
  print "表面積=", area, "\n"
  print "体積=", volume, "\n"
end

def greater_smaller
  a = 20
  if a >= 10 then
    print "greater\n"
  end
  if a <= 9 then
    print "smaller\n"
  end
end

def greater_smaller_else
  a = 20
  if a >= 10
    print "greater\n"
  else
    print "smaller\n"
```

```
  end
end
```

- rspec で, 個人 (koki) ごとのテストを実行するには, 環境変数 RUBYNOVICE_NAME にディレクトリ名を入れる.
  - (csh,tcsh)setenv RUBYNOVICE_NAME koki
  - (bash,zsh)export RUBYNOVICE_NAME=koki

3.6 ruby_novice のコマンド

3.6.1 tag の表示の仕方

- grep type spec/ruby_novice_spec.rb で全ての context と type を表示.

type は各章の各問題名の相当する. 各問題ごとにテストする時に便利になる.

```
context 'version option', type: :version do
context 'help option', type: :help do
context 'print hello', type: :hello do
context 'helloruby', type: :helloruby do
context 'puts_and_p', type: :puts_and_p do
context 'kiritsubo', type: :kiritsubo do
context 'area_volume', type: :area_volume do
context 'comment_sample', type: :comment_sample do
context 'greater_smaller', type: :greater_smaller do
context 'greater_smaller_else', type: :greater_smaller_else do
context 'print_argv', type: :print_argv do
context 'happy_birth', type: :happy_birth do
context 'arg_arith', type: :arg_arith do
context 'read_text', type: :read_text do
context 'read_text_simple', type: :read_text_simple do
context 'read_text_oneline', type: :read_text_oneline do
context 'read_line', type: :read_line do
context 'simple_grep', type: :simple_grep do
```

```
context 'hello_ruby2', type: :hello_ruby2 do
context 'use_grep', type: :use_grep do
context 'scopetest', type: :scopetest do
context 'ad2heisei', type: :ad2heisei do
context 'if_elsif', type: :if_elsif do
context 'unless1', type: :unless1 do
context 'case1', type: :case1 do
context 'case_class', type: :case_class do
context 'times', type: :times do
context 'times2', type: :times2 do
context 'times3', type: :times3 do
context 'for1', type: :for1 do
context 'for_names', type: :for_names do
context 'while1', type: :while1 do
context 'while2', type: :while2 do
context 'while3', type: :while3 do
context 'until1', type: :until1 do
context 'while_not', type: :while_not do
context 'each_names', type: :each_names do
context 'each', type: :each do
context 'break_next', type: :break_next do
context 'times_with_param', type: :times_with_param do
context 'hello_with_name', type: :hello_with_name do
context 'hello_with_default', type: :hello_with_default do
context 'myloop1', type: :myloop1 do
```

3.7 全章のテストの仕方

- bundle exec rspec

すべての章のテストを一括して実行できる.

3.8 各章ごとのテストの仕方

例: 1 章 (chap1) のテストをしたい時.

- bundle exec rspec spec/chap1_spec.rb
- bundle exec rake chap 1

実行例

```
/Users/Koki/ruby_novice% bundle exec rake chap 1


ruby_novie command
  version option
    should be successfully executed
    should have output: "0.1.0"
  help option
    should be successfully executed
  helloruby
    should be successfully executed
    should have output: "Hello, Ruby."
  puts_and_p
    should be successfully executed
    should have output: "Hello,\n\tRuby.\n\"Hello,\n\tRuby.\""
  kiritsubo
    should be successfully executed
    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\n 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"
  area_volume
    should be successfully executed
    should have output: "表面積=2200\n 体積=6000"
  comment_sample
    should be successfully executed
    should have output: "表面積=2200\n 体積=6000"
```

```
  greater_smaller
    should be successfully executed
    should have output: "greater"
  greater_smaller_else
    should be successfully executed
    should have output: "greater"

Finished in 7.61 seconds (files took 1.03 seconds to load)
17 examples, 0 failures
```

3.8.1 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.

- bundle exec rspec –tag type:helloruby spec/ruby_novice_spec.rb (helloruby は問題名)
- bundle exec rake test_name helloruby

実行例

```
/Users/Koki/ruby_novice% bundle exec rake test_name helloruby
Run options: include {:type=>"helloruby"}

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

Finished in 0.87128 seconds (files took 0.81684 seconds to load)
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.

3.8.2 各問題ごとの実行結果の出力

例: helloruby の実行結果の出力

- bundle exec exe/ruby_novice my_helloruby
- 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，ふたつの引数をとり，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．

```
# code.rb

def helloruby
  print("Hello, Ruby.\n")
end
```

```
#ruby_novice.rb

require 'thor'
require "code.rb"

module RubyNovice
  class CLI < Thor
    desc 'my_helloruby', 'print helloruby'
    def my_helloruby
      helloruby
    end
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，オプションのパースや 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_novie␣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=実際の値であるかを検証します. 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　おわりに

6　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%2

[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