Code Submission for 8th Light

Table of Contents

Usage

1. Install Ruby
2. Install dependencies: `bundle install`
3. Run the program: `ruby bin/bookview.rb`

Set-Up Instructions

To use this application, clone this repo to your local with the following command:

`git clone https://github.com/BenjaminNeustadt/8th_light_test.git`

Ensure that you have Ruby downloaded on your system, you can check by inputting `ruby -v` into your terminal.

Run `bundle install` to ensure you have all the correct dependencies of this project.

Using the Program

From within the root directory of the program, in your terminal:

```
ruby bin/bookview.rb
```

Follow the prompts to search for a book query. Insert any word to get a return of 5 books that match your query. Follow the prompts to add the book to a reading list.

(Please note, the queries will only be matched via the Google Books API if you set the program to online - instructions below; else it will use persistent test data.)

To use this program live

If you would like to use this program online, the steps will be as follows:

1. Setup your own google Books **API key**, this article is very helpful
2. create a '.env' file in the root directory with `touch .env` and use the example layout given below
3. In the executable file ('bin/bookview.rb'), comment out line 64, and uncomment the corresponding line below (line 66). In short, `BookSearch.new` should be given an argument of `BookData.new(query).parse`
4. You can now use the program live

Example `.env`

```
#.env file
{ "API_KEY" : "[YOUR_API_KEY_HERE]" }
```

Testing

The testing framework used is Rspec.

To run the tests:

```
rspec -fd
```

Code coverage is reported when running the tests (currently 100%).

Process

Following is some of the initial planning, followed by some revisoinary CRC modelling post review.

CRC Model Planning

Initially the following design was considered.

Following a Model/View/Controller design

```
        Front-end                              Back-end
- - - - - - - - - - - - - -    - - - - - - - - - - - - - - - - - - - - - - - -
```

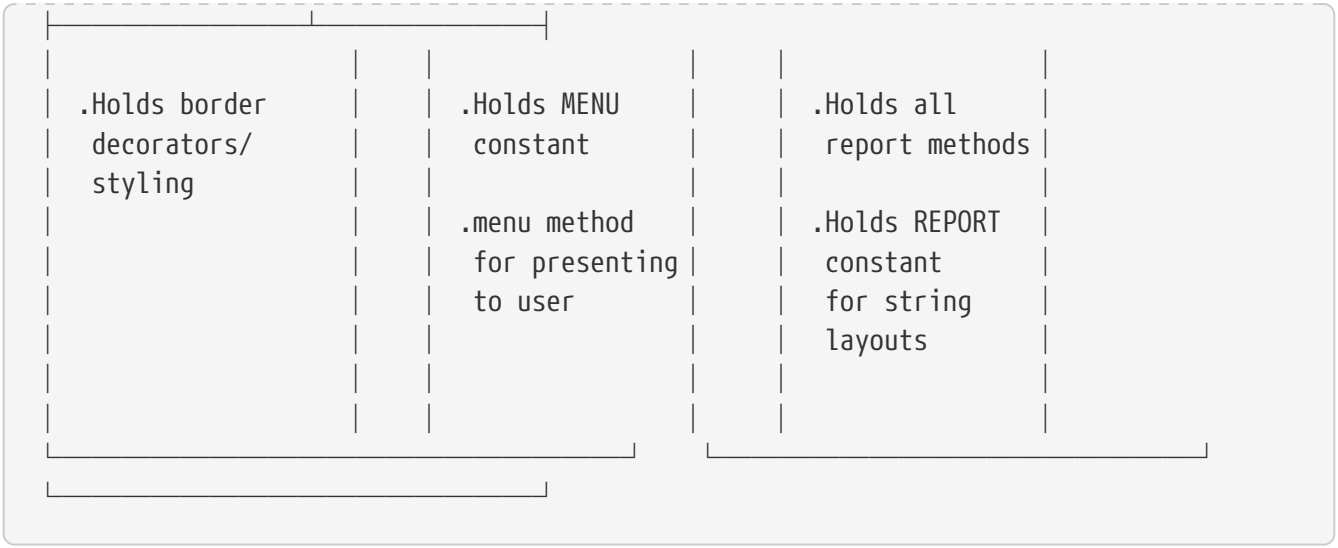
```
 Public facing interface          Operation                Storage
 (Command line interface)
 ┌──────────────────────────────────────────────────┐
 ┌──────────────────────────────────────────────────┐
 ┌──────────────────────────────────────────────────┐
 |   BookSearch            |      |   BookList             |     |   UserList
 |
 |               BookList  |      |            BookSearch  |     |
BookSearch,BookList
 ├─────────────────────────────┬──────────────────┤
 ├─────────────────────────────┬──────────────────┤
 ├─────────────────────────────┬──────────────────
 | .Give          |         |      | .grab_books    |         |     | .Store users
 | .BookList
 |  instructions  | .BookList     |              |         |     |  chosen books  |
 |
 |  to user       |         |      | .search        |         |     |
 | .BookSearch
 | .Pass          |         |      |              |         |     | .Pass array    |
 |
 |  instructions  |         |      | .add (instantiates     |     |  of books to   |
 |
 |  to BookList   |         |      | instance of UserList   |     |  BookSearch    |
 |
 | .Retrive Userlist       |      |              |         |     |                |
 |
 |  (stored book data)     |      |              |         |     |                |
 |
 |                |         |      |              |         |     |                |
 |
 └─────────────────────────────┴──────────────────┘
 └─────────────────────────────┴──────────────────┘
 └─────────────────────────────┴──────────────────┘
```

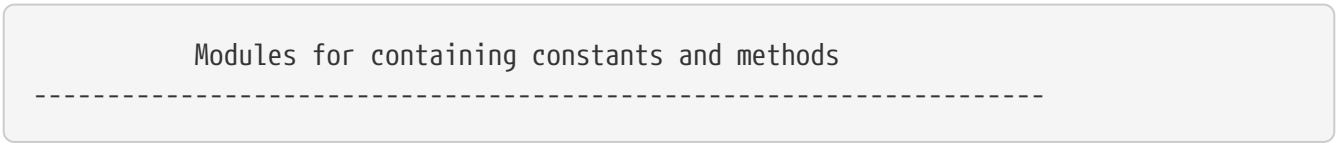
```
VIEW                          CONTROLLER                    MODEL
```

Post review revisions

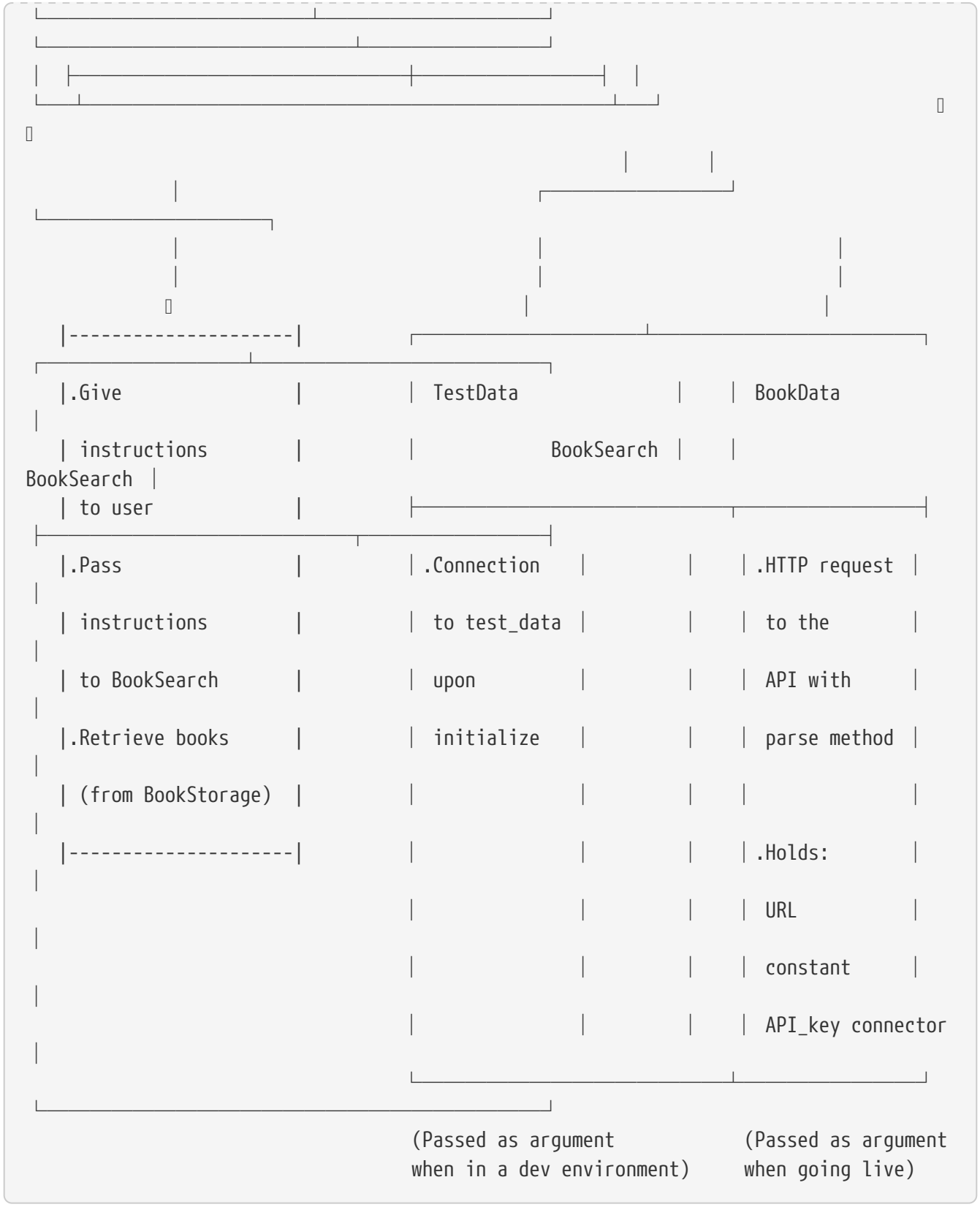
Not all of these constants have been implemented (specifically the 3 modules); and BookView has not been refactored accordingly. However, this is a design that feels like it would adhere more correctly to the Single Responsibility Principle, and other SOLID priniciples.

```
 ┌──────────────┬──────────────────────────┐    ┌──────────────┬──────────────────┐
 ┌──────────────┬──────────────┐
 |  module | Decorators  |    |  module  | Prompt  |     |  module  | Report  |
 ├──────────────┬──────────────────────────┤    ├──────────────┬──────────────────┤
```


.Holds border
decorators/
styling
.Holds MENU
constant
.menu method
for presenting
to user
.Holds all
report methods
.Holds REPORT
constant
for string
layouts


Modules for containing constants and methods


PUBLIC FACING INTERFACE
BookView
BookSearch
BookStorage
BookSearch
BookStorage
.look_up_books
BookSearch
.add_book
.includes 3
modules:
Report
Prompt
Decorators
.After
receiving
data
extract
attributes
BookStorage
TestData
BookData
.Store
users
chosen books
.Can
add
book



Post review

Question

> With the Single Responsibility Principle in mind, what are all of the responsibilities of the bin/bookview.rb script in its current form? Given the time, how might you refactor so that the

script has a single responsibility?

Currently the responsibilities of bin/bookview.rb are numerous:

- running the script/loop
- holding two constants for later use within this file
- instantiating an instance of the 'BookStorage' class
- containing methods used within script

Extracting and Refactoring

Initially when writing this script I followed a sort of procedural programming process. However, I would definitely like to improve this.

I have diagramed what I feel would be my ideal finished program (revised CRC above). I would change the permissions so that the script executable is `bin/bookfind` instead of `ruby bin/bookview.rb`, and give an instruction in the README for the next dev to implement this command in their terminal.

The script would comprise calling an instance of `BookView.new`, called 'session', and then use methods on that instance to output the respective 'reports' within each `when` conditional inside the loop; this way, we would only need to use `puts` once to output the return of each method. We would also DRY the code considerably. `BookView.new` would be placed inside the lib folder. The script's responsibility would therefore be to execute an instance of `BookView`, getting input and passing that input to BookView through its methods. In a way, the script would therefore be analogous to a front end, if we follow the design analogy initially used.

I would have to switch off the colorization, and use testing to match the current outputs. Case 4 and 0 would both use the same method, though could be given an argument defining the border "style" to be used, these styles would be stored inside a module I called "Decorators" (in revised CRC model). I like this approach as then the borders become customisable, and we can easily add a new style to these borders later on.

I would favour using heredocs to build the strings inside of these reports, as then I feel the string format would be easier to modify from within constants containing the string format. In this way I think we could more easily adhere to SRP and have modularity, as we every module would and class would only hold methods that serve a specific type of functionality.

I also think that heredocs would promote better readability for the next dev joining the project.

The trade-off I can imagine having to face is the colorization of specific lines for these reports; we would only be able to colorize the string in its entirety.

Ultimately, the refactor would comprise of:

- class BookView
- module Report

- module Decorators
- module Prompt
- bin/bookfind script

BookView would have access to all the report methods on the Report module. I believe it only needs be a module, as we do not need to create state for it.

Automating the environement outside of the code

Currently the environement (test or live data) is set by choosing which class object to instantiate when using 'BookSearch' from within the loop.

My reasoning for this change:

I hesitated on this point, though decided that the functioanlity of the `OFFLINE' constant (to toggle the integration on and off) could be achieved in another way. I used The Twelve-Factor App III.Config to partly inform this decision, as well as the open/close principle. It felt like having constants that changed the environement configuration of the program inside the code was not ideal, and I know you alluded to this also in the beginning of the review.

Ideally, a degree of automation in regards the "connection" to test-data or live data could be achieved by using a config file, in order to minimize time and cost for new developers joining the project; and to offer maximum portability between execution environements.

Currently I have tried to adher to the Open/Close principle by using the strategy pattern that was encouraged in the code review. Initially, what I had done was include an `if else` statement in 'BookData', the class was therefore initially responsible for checking whether the `OFFLINE` constant was set to `true` or `false`.

However, following the recommendation to use the strategy pattern in the review, I thought it would be beneficial to extract these even further into separate classes, and pass them in as arguments to 'BookSearch',

So ideally, I think that using 'dotenv' would be a suitable compromise. In so doing, we could set the "connection" configuration in the top of the script file 'bin/bookview.rb' (that would at this point in the refactor be 'bin/bookfind'). We could then choose which object to create (and thereby which data to retrieve, live or test) depending on whether the ENV is existant in the '.env' file.

In order to not stray from SRP we could then extract this "connection" configuration into its own file as a simple method, and then call it from within the script (i.e. `session.connection_environement`): this would then pass in the appropriate argument to `BookSearch.new(choice, connection_type)` (on line 64 in the script).

Reappropriate tests to `booksearch_spec.rb`

As part of the post-review revision I have moved the tests previously in `end_to_end_spec.rb`, as the behaviours we are testing there seem to be the responsibility of `BookSearch`.