Lab 6E Check-Off

Student Name: _____

Date of Lab: _____

Check-Offs

_____ Date validations are set up

_____ All unit tests created and pass

TA (signed): _____

TA (print): _____

Lab 6E Instructions

Lab Objectives: The primary objectives of this lab are (1) to help students learn how to write basic unit tests using factories rather than fixtures, and (2) to handle date validations in Rails, since there are no built-in validators for dates in the framework.

Directions:

1. Acme Publishing loved last week's lab and hired you to customize it to track their needs as a publishing house specializing in technical books. The project is a little different in that we have dropped publisher as a separate entity *(this is for just one publisher)*, but added category as a new entity. Moreover, we need to track books through the various stages of the publishing process: from the initial proposals to signing a contract with author(s) to finally publishing the book. To understand this better, get the BookManager_2 project by cloning it from Prof. H's account (`profh`) on github.com. Once you cloned the project, run `bundle install` in make sure you have the gems required such as 'validates_timeliness'. After that, migrate the database and when that is done, populate it with fictitious data by running the command `rake db:populate`. Feel free start up the server and experiment with the application briefly so you understand what is happening before we dive into testing.

2. Of course, before we dive into the unit testing, we are going to complete some of the missing validations in the book model. Going to that model, you will see a large open section for adding date validations. Since Rails does not have built-in date validations, we are using the `validates_timeliness` gem; find the documentation for this gem online at https://github.com/adzap/validates_timeliness. After reading through this documentation, you will need to add three validations:

   a) validate that the proposal_date is a legitimate date and that it is either the current date or some time in the past. (You

shouldn't be allowed to record a proposal you haven't yet received.)

b) validate that contract_date again is a legitimate date and that it is either the current date or some time in the past. (You shouldn't be allowed to record a contract you haven't yet signed.) Also make sure that the contract_date is some time after the proposal_date as you can't sign contracts for books yet to be proposed. Finally allow the contract_date to be blank as not all books we are tracking have contracts yet.

c) validate that published_date also is a legitimate date and that it is either the current date or some time in the past. Also make sure that the published_date is some time after the contract_date as you can't publish books without contracts. Finally allow the published_date to be blank as not all books we are tracking are published yet.

3. Now we are going to test the book model by doing some unit tests. Go into test/test_helper.rb and set the use of transactional fixtures to false, comment out the line `fixtures :all`. (The fixture files have been removed from the version of the BookManager application provided.) At the top of file, add in the support for simple_cov by adding the lines:

```
require 'simplecov'
SimpleCov.start 'rails'
```

4. If you have forgotten to do so by this point, then make sure you are committing your work to git. (This is your only reminder this lab, but hopefully it is becoming second nature now.) It is a great idea to be regularly saving your code to git or some other form of source code control and to take advantage of branching when appropriate. (Your discretion today when to do these things…)

5. Getting back to the code, within the `test/` directory, there is a file called factories.rb which you need to open. In the factories.rb file, we need to complete the author factory. Look at the other factories provided and the database schema to help guide building

this factory. Have a TA verify that the factories are correct before proceeding.

6. Now we are ready to create unit tests for the Book model. *(Note: we've removed all other unit tests at this time so we can focus on this one model and speed up testing; feel free to go back later and practice writing tests for other models.)* Open the book_test.rb file within the test/unit/ directory. In the first section, we will add some shoulda matcher, beginning with relationship matchers:

```
should belong_to(:category)
should have_many(:book_authors)
should have_many(:authors).through(:book_authors)
```

Run these three tests on the command line by executing `rake test:units`. This will create a test db and execute these tests. Assuming they pass, now add the following validation matcher:

```
should validate_presence_of(:title)
```

And while we're at it, let's check the format of units_sold:

```
should allow_value(1000).for(:units_sold)
should_not allow_value(-1000).for(:units_sold)
should_not allow_value(3.14159).for(:units_sold)
should_not allow_value("bad").for(:units_sold)
```

Test these by running `rake test:units` on the command line. Feel free to add any other tests on units_sold you deem necessary.

7. We want to use the matchers to test the validity of dates passed into the system. For the proposal_date, try the following:

```
should allow_value(1.year.ago).for(:proposal_date)
should_not allow_value(1.week.from_now).for(:proposal_date)
should_not allow_value("bad").for(:proposal_date)
should_not allow_value(nil).for(:proposal_date)
```

Run these tests and then try something similar for contract_date:

```
should allow_value(1.year.ago).for(:contract_date)
```

Why did we get a failure when we run this test? Remember that contract_date's validity is dependent on the value of the

proposal_date, but incorporating this is beyond this simple matcher's ability. Matchers are easy and nice when they work and there are lots of uses for them, but there are also plenty of times when we will need more – we cannot limit our testing to just what matchers provide us.

8. Now we will have to set up some contexts for more complex unit tests. Essentially a context is where we set up a situation were different aspects of the project can be tested and we know what the correct result should be. For example, we know there are five books in the system, three of which are published because we have created these books as part of the context. We can then execute tests on model methods within the context and confirm the results are as expected.

   To save you time and typing, we've provided you with the setup and teardown methods as well as a simple test of each factory created to verify it is working. Uncomment the lines within setup and teardown and the initial factories test. While doing that, review this code so you understand how it works and ask a TA for help if you are unclear about any aspect of it.

9. Looking at the first test, we need to test the 'by title' scope which essentially alphabetizes the books in order of title. We know what the result should be for these five books and can compare that with what the method returned with the following line:

   ```
   assert_equal ["Agile Testing", "Rails 3 Tutorial", "Ruby for Masters", "The RSpec Book", "The Well-Grounded Rubyist"], Book.by_title.map{|b| b.title}
   ```

   Run the unit tests and verify this is the case. Drop one of the book titles from the expected array and rerun to see what a failing test would look like. After observing, restore the missing book title so the test passes.

10. Looking at the second test, we need to test the 'by category' scope which essentially alphabetizes the books in order category name first, then book title. (You might want to look at that scope briefly as it uses a `.joins()`) We know what the result should be

for these five books and can compare that with what the method returned with the following line:

```
assert_equal ["Rails 3 Tutorial", "Ruby for Masters", "The Well-Grounded
Rubyist", "Agile Testing", "The RSpec Book"], Book.by_category.map{|b| b.title}
```

Run the unit tests and verify this test passes.

11. Looking at the third test, we need to test the 'published' scope which returns all the books that have been published to date. We know that there are only three books that are published and can compare that with what the method returned with the following line:

```
assert_equal ["Rails 3 Tutorial", "The RSpec Book", "The Well-Grounded
Rubyist"], Book.published.by_title.map{|b| b.title}
```

Note that we also used the 'by_title' scope since it was already tested and we wanted to be sure that the order in the two arrays was the same. Run the unit tests and verify this test passes, then create similar tests for the next two scopes.

12. Now to test the 'for_category' scope you will need to specify a category_id. This can be done as follows:

```
assert_equal ["Rails 3 Tutorial"],
Book.for_category(@rails.id).by_title.map{|b| b.title}
```

Write another assertion that tests for the `@ruby` category and check to make sure your tests pass.

13. Now to test the contract_date validation works allows good dates we can take advantage of the fact that we know by default the factory sets the proposal_date to one year ago and thus write the following assertion:

```
big_ruby_book = Factory.build(:book, :contract_date => 50.weeks.ago,
:category => @ruby, :title => "The Big Book of Ruby")
 assert big_ruby_book.valid?
```

Also make sure that nil values are acceptable in the next test with the following assertion:

```
 big_ruby_book = Factory.build(:book, :contract_date => nil,
:published_date => nil, :category => @ruby, :title => "The Big Book of
Ruby")
 assert big_ruby_book.valid?
```

14. Check to make sure that the contract date doesn't precede the proposal date with the following:

```
big_ruby_book = Factory.build(:book, :contract_date => 14.months.ago,
:category => @ruby, :title => "The Big Book of Ruby")
 deny big_ruby_book.valid?
```

    Also verify that the contract date can't be in the future with the following assertion:

```
 big_ruby_book = Factory.build(:book, :contract_date => 1.month.from_now,
:category => @ruby, :title => "The Big Book of Ruby")
 deny big_ruby_book.valid?
```

    Do you remember what was said in class about why we had to *build* these models rather than *create* them? If not, ask someone now and refresh your memory. To confirm your understanding, go back later to the lab and complete the tests for the published_date validations.

15. Finally, we need to test the custom validation 'category_is_active_in_system'. This can be done with the following code:

```
@python = Factory.create(:category, :name => "Python", :active => false)
 python_book = Factory.build(:book, :category => @python, :title => "Python!")
 deny python_book.valid?
 @python.destroy
```

    [*Note: remember that we have to destroy the factory after using it, otherwise it could cause problems in later tests.*]

16. Confirm that all tests pass and get the TA to check this portion off and verify that you've been using git appropriately.

plt1h

Shoulda::ActiveRecord::Matchers

(Textmate shortcut in parens; assumes you have the Textmate bundle installed)

```
should belong_to (sbt)
should have_one (sho)
should have_many (shm)
should have_many(:model).through(:reference) (shm)
should_have_readonly_attributes (shra)
should allow_value("var").for(:model) (sav)
should_not allow_value("var").for(:model) (snav)
should allow_mass_assignment_of (same)
should_not allow_mass_assignment_of (snama)
should validate_presence_of (svpo)
should validate_uniqueness_of (svuo)
should validate_acceptance_of (svao)
should ensure_length_at_least (selat)
should ensure_inclusion_of (sei)
should ensure_length_of (sel)
should have_db_column (shdc)
should_not have_db_column (snhdc)
should have_db_index (shdi)
should validate_format_of(:var).with('123').not_with('12D')
```