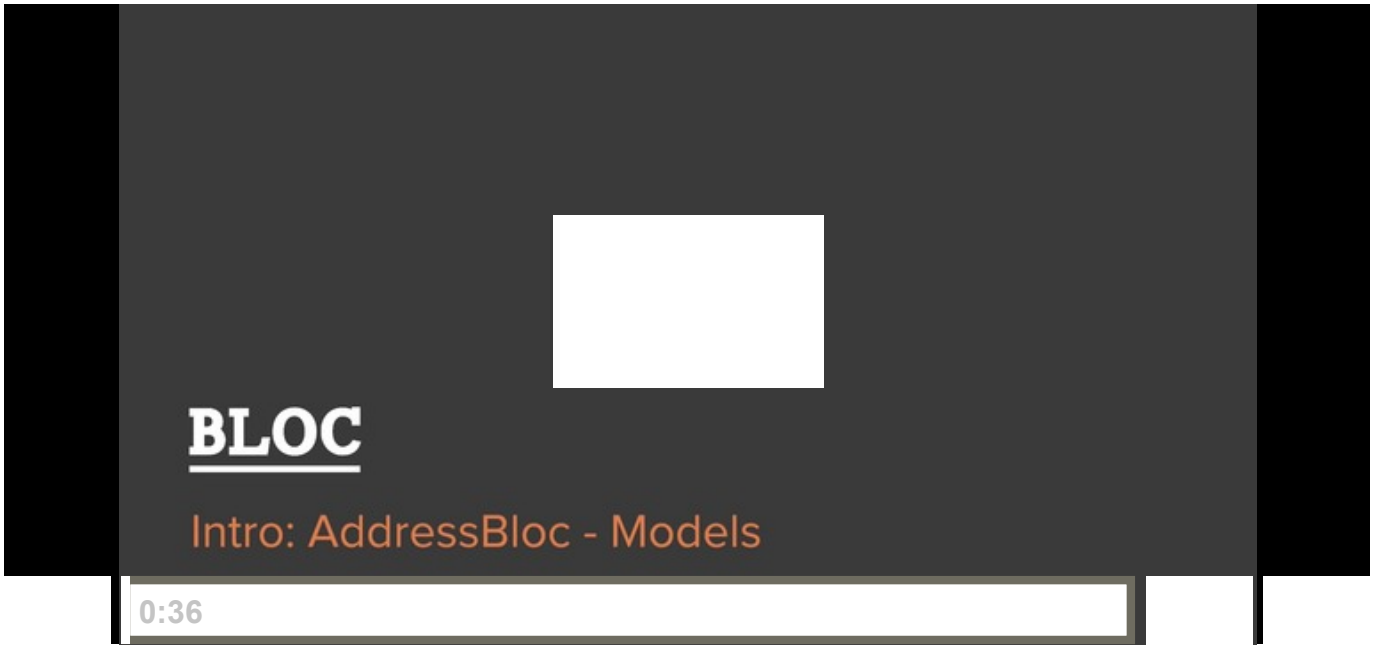
19 Address Bloc: Models

> “Maybe there's more to life than being really, really, really ridiculously good looking?”
>
> — Derek Zoolander, Model

Introduction



Address Bloc will need to **model** an internal address book and all its entries. Models provide a structured interface to the data source in a project which allows for easier manipulation of the underlying data.

We'll practice **Test-Driven Development (TDD)** by creating tests before we write functional code. TDD might seem like a backward way to write code, but programming this way encourages us to think carefully about expected behavior and desired design. The

resulting code base tends to be very modular and easy to change, making the app's development smoother and more sustainable. It keeps our work's scope tight, making it hard to drift into tangential or irrelevant changes.

For testing, we'll use the popular and powerful testing framework **RSpec**. Install it using the `gem` command:

Terminal

```
$ gem install rspec
```

> After installing the gem, consider running the `rspec --init` command to generate an `.rspec` file, which should contain the default color setting of `--color`. This will colorize the RSpec output. Additionally, consider installing the **Nyan Cat Formatter** because it's fun.

Git

Create a new Git feature branch for this checkpoint. See **Git Checkpoint Workflow: Before Each Checkpoint** for details.

The `Entry` Model

We'll start by creating a model to store our address book entries, `Entry`.

Let's create directories to store our models and specs (RSpec's names for tests):

Terminal

```
$ cd address-bloc
$ mkdir models
$ mkdir spec
```

Create two files: one for `Entry`, and one for `entry_spec.rb`:

Terminal

```
$ touch models/entry.rb
$ touch spec/entry_spec.rb
```

`Entry` Attributes

We'll keep our tests in the `spec` directory. Open `entry_spec.rb` and add the following:

spec/entry_spec.rb

```
+ require_relative '../models/entry'
+
+ RSpec.describe Entry do
+
+ end
```

All we have here is the skeleton of our test file. We'll use **RSpec** to test our code. We can run the spec by using `rspec` with the following command:

Terminal

```
$ rspec spec/entry_spec.rb
entry_spec.rb:3:in `<top (required)>': uninitialized constant Entry (NameError)
...
```

Our output shows a `NameError` which indicates that our spec failed when we executed it. This is because we reference `Entry` in the spec, but `Entry` is not defined anywhere in Address Bloc. Let's define `Entry` in `models/entry.rb` by creating a *stub* of the class. A **stub** in programming terminology is an empty class, or method. A stub is sort of like a pencil sketch for a painting: we create a sketch of what we're going to paint first and then fill in the rest with actual paint later.

models/entry.rb

```
+ class Entry
+
+ end
```

Let's execute our spec again:

Terminal

```
$ rspec spec/entry_spec.rb
No examples found.

Finished in 0.00015 seconds (files took 0.1186 seconds to load)
0 examples, 0 failures
```

RSpec tells us that there are no examples found. This is because we haven't added a test to `entry_spec.rb`. The term `example` in this context is synonymous with test.

Let's create our first set of tests for `Entry`. `Entry` will need attributes, including `name`, `phone_number`, and `email`. Our first specs will test that `Entry` responds to these attributes. Open `entry_spec.rb` and add the following:

spec/entry_spec.rb

```
  require_relative '../models/entry'

  # #1
  RSpec.describe Entry do
  # #2
+   describe "attributes" do
  # #3
+     it "responds to name" do
+       entry = Entry.new
  # #4
+       expect(entry).to respond_to(:name)
+     end
+
+     it "responds to phone number" do
+       entry = Entry.new
+       expect(entry).to respond_to(:phone_number)
+     end
+
+     it "responds to email" do
+       entry = Entry.new
+       expect(entry).to respond_to(:email)
+     end
+   end
  end
```

#1 is the standard first line of an RSpec test file. We are saying that the file is a spec file and that it tests `Entry`.

At **#2**, we use `context` to give our test structure and communicate that all specs inside it belong to the same context. In this case, we're using it to communicate that the specs in the context will test `Entry` attributes.

At **#3**, we separate our individual tests using the `it` method. Each `it` represents a unique test.

At **#4** each RSpec test ends with one or more `expect` method, which we use to declare the expectations for the test. If those expectations are met, our test passes, if they are not, it fails.

> Under the hood, `expect(entry).to respond_to(:name)` will check whether `entry.name` is valid code. One could have tried to run `entry.name` directly, but RSpec provides us with a set of methods, such as `expect` , `to` and `respond_to`, to better express our intents. It is known as a **domain-specific language** (DSL). A DSL is a specialized language in which a class of problems (here, testing) is expressed succinctly: the method names match the vocabulary of the problem domain and the resulting code is highly descriptive.

Run the entry specs from the command-line using the `rspec` command, the spec we want to run, and the file we're testing:

Terminal

```
$ rspec spec/entry_spec.rb
```

You should see three failing tests, each with output similar to the following:

Terminal

```
  1) Entry attributes should respond to name
     Failure/Error: expect(entry).to respond_to(:name)
       expected #<Entry:0x007fcb0340c790> to respond to :name
```

Failing tests are what we expect to see since we haven't added those attributes to `Entry`. We're currently in the "red" phase of the TDD **Red-Green-Refactor cycle**. Now it's time to turn our tests green. We expect an `Entry` instance to have a few methods for accessing a name, a phone number and an email. Let's leverage `attr_accessor` to handle those attributes:

models/entry.rb

```
  class Entry
+   # These must be accessors since we mutate them
+   attr_accessor :name, :phone_number, :email
  end
```

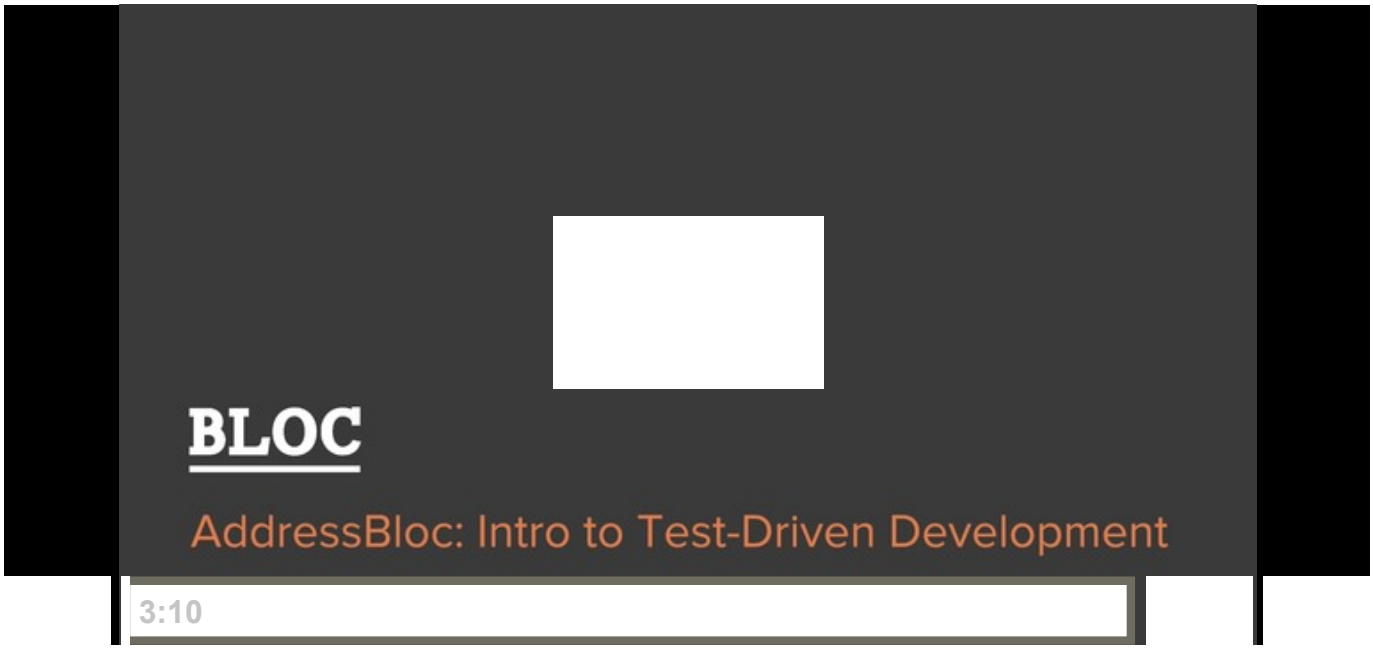
Run your tests again and watch them pass!

Terminal

```
$ rspec spec/entry_spec.rb
...

Finished in 0.00122 seconds (files took 0.1039 seconds to load)
3 examples, 0 failures
$
```

Watch the following video to see the above steps in practice:



`Entry` Initializer

An entry without initialized attributes isn't useful. Modify the tests so they assume entries will always be created with values for the `name`, `phone_number`, and `email` attributes:

spec/entry_spec.rb

```
  require_relative '../models/entry'

  RSpec.describe Entry do
    describe "attributes" do
      it "responds to name" do
+       entry = Entry.new('Ada Lovelace', '010.012.1815', 'augusta.king@lovelace.com'
-       entry = Entry.new
        expect(entry).to respond_to(:name)
      end

      it "responds to phone_number" do
+       entry = Entry.new('Ada Lovelace', '010.012.1815', 'augusta.king@lovelace.com'
-       entry = Entry.new
        expect(entry).to respond_to(:phone_number)
      end

      it "responds to email" do
+       entry = Entry.new('Ada Lovelace', '010.012.1815', 'augusta.king@lovelace.com'
-       entry = Entry.new
        expect(entry).to respond_to(:email)
      end
    end
  end
```

Running the tests again will result in failures because we haven't defined an `initialize` method that takes three arguments:

Terminal

```
rspec spec/entry_spec.rb
FFF

Failures:

  1) Entry attributes should respond to name
     Failure/Error: entry = Entry.new('Ada Lovelace', '010.012.1815', 'augusta.king@
     ArgumentError:
     ...
```

Add the `initialize` method to `Entry`:

models/entry.rb

```
  class Entry
    # These must be accessors since we mutate them
    attr_accessor :name, :phone_number, :email
+
+   def initialize(name, phone_number, email)
+   end
+
  end
```

Run the tests again to confirm that they pass. We wrote only the *minimal* implementation required for the tests to pass. Although the code doesn't do anything with the arguments, we are able to call `Entry.new` with three parameters.

Testing the attributes

The core idea is to be able to read an attribute's value. Let's add three tests:

spec/entry_spec.rb

```
  RSpec.describe Entry do
    describe "attributes" do
      it "responds to name" do
        entry = Entry.new('Ada Lovelace', '010.012.1815', 'augusta.king@lovelace.com'
        expect(entry).to respond_to(:name)
      end
+
+     it "reports its name" do
+       entry = Entry.new('Ada Lovelace', '010.012.1815', 'augusta.king@lovelace.com'
+       expect(entry.name).to eq('Ada Lovelace')
+     end

      it "responds to phone_number" do
        entry = Entry.new('Ada Lovelace', '010.012.1815', 'augusta.king@lovelace.com'
        expect(entry).to respond_to(:phone_number)
      end
+
+     it "reports its phone_number" do
+       entry = Entry.new('Ada Lovelace', '010.012.1815', 'augusta.king@lovelace.com'
+       expect(entry.phone_number).to eq('010.012.1815')
+     end

      it "responds to email" do
        entry = Entry.new('Ada Lovelace', '010.012.1815', 'augusta.king@lovelace.com'
        expect(entry).to respond_to(:email)
      end
+
+     it "reports its email" do
+       entry = Entry.new('Ada Lovelace', '010.012.1815', 'augusta.king@lovelace.com'
+       expect(entry.email).to eq('augusta.king@lovelace.com')
+     end
    end
  end
```

Running the tests again will report three failures because we never set the attributes. Let's add the required code to make them pass:

models/entry.rb

```
  class Entry
    # These must be accessors since we mutate them
    attr_accessor :name, :phone_number, :email

    def initialize(name, phone_number, email)
+     @name = name
+     @phone_number = phone_number
+     @email = email
    end

  end
```

Make sure to run the tests and check they all pass.

After we go from red to green, it's beneficial to ask whether there's some refactoring we can do. This time, the `Entry` class looks clean and consistent with the tests, so there's no need for any refactoring.

The tests now showcase serious duplication though, as we are repeating the same `Entry.new(…)` instruction. Fortunately, RSpec allows us to keep our tests DRY (Don't Repeat Yourself) by defining efficient helper methods using `let`. Instead of creating an `entry` local variable within each test, let's define an `entry` method once:

spec/entry_spec.rb

```
  RSpec.describe Entry do
    describe "attributes" do
+     let(:entry) { Entry.new('Ada Lovelace', '010.012.1815', 'augusta.king@lovelace.
+
      it "responds to name" do
-       entry = Entry.new('Ada Lovelace', '010.012.1815', 'augusta.king@lovelace.com'
        expect(entry).to respond_to(:name)
      end

      it "reports its name" do
-       entry = Entry.new('Ada Lovelace', '010.012.1815', 'augusta.king@lovelace.com'
        expect(entry.name).to eq('Ada Lovelace')
      end

      it "responds to phone_number" do
-       entry = Entry.new('Ada Lovelace', '010.012.1815', 'augusta.king@lovelace.com'
        expect(entry).to respond_to(:phone_number)
      end

      it "reports its phone_number" do
-       entry = Entry.new('Ada Lovelace', '010.012.1815', 'augusta.king@lovelace.com'
        expect(entry.phone_number).to eq('010.012.1815')
      end

      it "responds to email" do
-       entry = Entry.new('Ada Lovelace', '010.012.1815', 'augusta.king@lovelace.com'
        expect(entry).to respond_to(:email)
      end

      it "reports its email" do
-       entry = Entry.new('Ada Lovelace', '010.012.1815', 'augusta.king@lovelace.com'
        expect(entry.email).to eq('augusta.king@lovelace.com')
      end
    end
  end
```

The term "refactoring" only applies to code, but enhancing tests like that is a good habit to make. Run them all again to ensure they still pass.

The `to_s` Method

To build the menu we'll need a method to return a string representation of an `Entry`, to be displayed in the terminal. It's standard convention to name such a method `to_s`, short for "(convert) to string". Create a test for `to_s`:

spec/entry_spec.rb

```
  require_relative '../models/entry'

  RSpec.describe Entry do
    describe "attributes" do
      # …
    end

 # #5
+   describe "#to_s" do
+     it "prints an entry as a string" do
+       entry = Entry.new('Ada Lovelace', '010.012.1815', 'augusta.king@lovelace.com'
+       expected_string = "Name: Ada Lovelace\nPhone Number: 010.012.1815\nEmail: aug
 # #6
+       expect(entry.to_s).to eq(expected_string)
+     end
+   end
  end
```

At **#5**, we use a new context to separate the `to_s` test from the initializer tests. The `#` in front of `to_s` indicates that it is an instance method.

At **#6**, we use `eq` to check that `to_s` returns a string equal to `expected_string`.

Run the tests and we'll see that three still pass, but our new test fails because we haven't defined `to_s`. Let's do that now:

models/entry.rb

```
  class Entry
    # These must be accessors since we mutate them
    attr_accessor :name, :phone_number, :email

    def initialize(name, phone_number, email)
      @name = name
      @phone_number = phone_number
      @email = email
    end
# #7
+   def to_s
+     "Name: #{name}\nPhone Number: #{phone_number}\nEmail: #{email}"
+   end
end
```

At **#7**, we create the string we want. The `\n` character is the `newline` character. It prints a **newline** to the console.

Run the tests again to confirm that they pass.

The `AddressBook` Model

We need an address book model to store entries. Create the address book model and specs:

Terminal

```
$ touch models/address_book.rb
$ touch spec/address_book_spec.rb
```

Create the skeleton of `AddressBook`:

models/address_book.rb

```
+ class AddressBook
+
+ end
```

`AddressBook` Attributes

`AddressBook` needs an attribute to hold the array of entries. The `entries` array will store entries. It also provides an easy way to add, remove, and count entries.

Create a test to confirm that `AddressBook` responds to `entries`:

spec/address_book_spec.rb

```
+ require_relative '../models/address_book'
+
+ RSpec.describe AddressBook do
+   describe "attributes" do
+     it "responds to entries" do
+       book = AddressBook.new
+       expect(book).to respond_to(:entries)
+     end
+   end
+
+ end
```

Run the specs and we'll see the expected failure:

Terminal

```
$ rspec spec/address_book_spec.rb
```

Make this spec pass by adding a read-only `entries` attribute to `AddressBook`:

models/address_book.rb

```
  class AddressBook
+   attr_reader :entries

  end
```

Run the specs again and see that they pass.

`AddressBook` should initialize `entries` as an empty array so that it's ready to be used. Add a couple new tests to confirm this behavior:

spec/address_book_spec.rb

```
  require_relative '../models/address_book'

  RSpec.describe AddressBook do
    describe "attributes" do
      it "responds to entries" do
        book = AddressBook.new
        expect(book).to respond_to(:entries)
      end

+     it "initializes entries as an array" do
+       book = AddressBook.new
+       expect(book.entries).to be_an(Array)
+     end
+
+     it "initializes entries as empty" do
+       book = AddressBook.new
+       expect(book.entries.size).to eq(0)
+     end
    end

  end
```

Running the specs will confirm that our two new tests fail. Let's make them pass:

models/address_book.rb

```
  class AddressBook
    attr_reader :entries

+   def initialize
+     @entries = []
+   end
  end
```

Having an array of entries is only useful if we can add items to that array. We need a method, `add_entry`, to update our `entries` array. Let's add some tests to define the expected behavior:

spec/address_book_spec.rb

```
  require_relative '../models/address_book'

  RSpec.describe AddressBook do
    describe "attributes" do
      # …
    end

+   describe "#add_entry" do
+     it "adds only one entry to the address book" do
+       book = AddressBook.new
+       book.add_entry('Ada Lovelace', '010.012.1815', 'augusta.king@lovelace.com')
+
+       expect(book.entries.size).to eq(1)
+     end
+
+     it "adds the correct information to entries" do
+       book = AddressBook.new
+       book.add_entry('Ada Lovelace', '010.012.1815', 'augusta.king@lovelace.com')
+       new_entry = book.entries[0]
+
+       expect(new_entry.name).to eq('Ada Lovelace')
+       expect(new_entry.phone_number).to eq('010.012.1815')
+       expect(new_entry.email).to eq('augusta.king@lovelace.com')
+     end
+   end
  end
```

We've wrapped these tests in a separate context to keep our specs organized. We've also included multiple expects in the second test, if any of them fail, then the entire `it` test will fail.

Run the specs again to confirm that the new tests fail. Now let's turn them green by creating `add_entry`. To keep `entries` nicely ordered we'll code `add_entry` to insert entries in **lexicographical order**.

models/address_book.rb

```
  # #8
+  require_relative 'entry'
+
  class AddressBook
    attr_reader :entries

    def initialize
      @entries = []
    end
+
+   def add_entry(name, phone_number, email)
  # #9
+     index = 0
+     entries.each do |entry|
  # #10
+       if name < entry.name
+         break
+       end
+       index += 1
+     end
  # #11
+     entries.insert(index, Entry.new(name, phone_number, email))
+   end
  end
```

At **#8**, we tell Ruby to load the library named `entry.rb` relative to `address_book.rb`'s file path using `require_relative`. At **#9**, we create a variable to store the insertion `index`. At **#10**, we compare `name` with the name of the current `entry`. If `name` lexicographically proceeds `entry.name`, we've found the `index` to insert at. Otherwise we increment `index` and continue comparing with the other entries. At **#11**, we insert a new entry into `entries` using the calculated `index.

Run the specs a final time and confirm that they pass.

Conclusion

Nice work! We've now created both models AddressBloc will need using TDD. TDD allowed us to keep our code focused and ensured complete test coverage. With our `Entry` and `AddressBook` models, we're ready to start creating a menu for users to use.

Git

Commit your checkpoint work in Git. See **Git Checkpoint Workflow: After Each Checkpoint** for details.

Recap

Concept	Description
Model	Models provide an interface for getting and setting data.
RSpec	RSpec is a **behavior-driven development (BDD)** framework for the Ruby programming language.
Test-Driven Development	Test-Driven development (TDD) is a software development process that relies on the repetition of a very short development cycle. First we write a failing test case. Then we produce the minimum amount of code to pass the test. Finally, we refactor the new code.

Assignment | Discussion | Submission

Create a new Git feature branch for this assignment. See **Git Checkpoint Workflow: Before Each Assignment** for details.

In addition to adding entries to `AddressBook`, let's create a method, `remove_entry(name, phone_number, email)`, to remove entries.

- Write your tests first. The tests should confirm that a single entry is removed when `remove_entry(name, phone_number, email)` is called. You can use `entries.size` to determine if an item was successfully removed.

- Iterate over `entries` until you find the match. Use the index of the match to remove the entry from `entries`. Use the following method definition for `remove_entry`:

models/address_book.rb

```
...

  def remove_entry(name, phone_number, email)
    # Implementation goes here
  end

...
```

- Confirm that all your tests pass.

Commit your assignment in Git. See **Git Checkpoint Workflow: After Each Assignment** for details. Submit your commit to your mentor.

Solution

Do not watch this video until after you've attempted to complete the assignment. If you struggle to complete the assignment, submit your best effort to your mentor *before watching a solution video*.

Media Queries Solution

assignment completed

?