The DRY testing pattern

Treating automated tests with the same care and respect as the application that we are trying to test is the key to long-term success. Adopting common software development principles and design patterns will prevent some costly maintenance in the future. One of these principles is the **Don't Repeat Yourself** (**DRY**) principle; the most basic idea behind the DRY principle is to reduce long-term maintenance costs by removing all unnecessary duplication.

TIP

There are a few times when it is okay to have a duplicate code, at least temporarily. As Donald Knuth so eloquently stated, "Premature optimization is the root of all evil (or at least most of it) in programming."

The DRY testing pattern embraces the DRY principle and expands on it. Not only do we remove the duplicate code and duplicate test implementations, but we also remove duplicate test goals.

NOTE

A test goal or test target is the main idea behind any given test. The rule of thumb is if you cannot describe what the test is supposed to accomplish in a sentence or two, then the test is too complicated or it does not understand what it is testing. A good example of a test goal would be "an anonymous user should be able to purchase product X on the website."

We are trying to avoid accidentally testing any functionality not related to the current test. For example, if the target of the current test is the registration flow, this test should not fail if a social media icon fails to load. Social media icons should have a test of their own that is not related to registration tests.

NOTE

David Thomas and Andrew Hunt formulated the DRY principle in their book, *The Pragmatic Programmer*, by Andrew Hunt and David Thomas, published by *Addison-Wesley Professional*. The DRY principle is sometimes referred to as **Single Source Of Truth** (**SSOT**) or **Single Point Of Truth** (**SPOT**) because it attempts to store every single piece of unique information in one place only.

Advantages of the DRY testing pattern

Writing tests using the DRY testing pattern has many advantages. Here are four advantages:

- **Modular tests**: Tests and test implementations are self-sufficient. Any test can run in any order. Also, test actions such as clicking or registering a new user are shared during the tests.
- **Reduced duplication**: All actions such as filling out a form are neatly kept in a single place instead of having multiple copies peppered all over the suite.
- **Fast updates**: Having unique actions in a single place makes it easy to update the tests to mimic new growth of the application.
- **No junk code**: Constant upkeep of the test suite, with deletion of duplicates, prevents the test suite from having code that is no longer used.

Disadvantages of the DRY testing pattern

There are some disadvantages of setting your tests according to the DRY testing pattern; it is a lot of work and requires a lot of buy in from the whole team. Here are some of the most common issues:

- **Complicated project structure**: Some test actions will be logically grouped with other similar actions. Filling out a login form and clicking the login button will probably happen in the same implementation file. However, some actions will inevitably end up in a different file, making it hard to find them.
- **Lack of a good IDE**: There aren't many good IDEs that will notify the test developer if a test action has already been implemented. Most test developers will reimplement the action they need instead of looking for it.

    ### TIP

    The best way around this problem is to have a lot of in-team communication. Asking whether anyone has already implemented an action in code will save you time and prevent duplication.

- **Constant upkeep**: Keeping the test suite clean and applying the DRY test pattern will need dedication from the team. Duplicate code needs to be pruned and deleted instead of being ignored.

    ### NOTE

    In statically typed languages such as Java, we can use static analysis tools that can be used to monitor code duplication.

- **Programming skills**: This needs to be improved by the whole team. One test developer who keeps duplicating logic and cargo-culting can spoil the elegant test suite in a matter of weeks.

    ### NOTE

> *Cargo cult* is a phrase commonly used to describe a programming style where a programmer uses a piece of code without understanding what the original intention of that code was. It can be described as "we do this because we always did this; I don't know why."

Let's start DRY-ing out our tests. The first and obvious choice is the setup and teardown methods.

MOVING CODE INTO A SETUP AND TEARDOWN

Most modern testing frameworks include the concept of a setup and teardown. Each framework can call them by a different name. For example, in Cucumber, the setup is called `Background`, while in Rspec, it is called `Before`. No matter what name the framework chooses, the idea behind these two methods remains the same. This setup is run before tests and is used to get the environment in a test-ready state. The teardown is used to clean up after the tests to put the environment back into a pristine state. Some frameworks allow the setup and teardown to be run before and after each individual test, while others only allow them to be executed before and after a group of tests; some even allow both.

Let's start cleaning up `product_review_test.rb` from Chapter 2, *The Spaghetti Pattern*, by adding the `setup` and `teardown` methods:

NOTE

> You can find the complete `product_review_test.rb` file at http://awful-valentine.com/code/chapter-2/.

1. The first thing we will do is add the `setup` and `teardown` methods at the top of our test; our code will look like this:

```
1  require 'rubygems'
2  require 'selenium-webdriver'
3  require 'test/unit'
4
5  class ProductReview < Test::Unit::TestCase
6
7     def setup
8
9     end
10
11    def teardown
12
13    end
14
```

2. Let's move the creation of the Firefox instance into `setup` and the quitting of the Firefox instance into `teardown`, as seen in the following screenshot:

```
require 'rubygems'
require '@selenium-webdriver'
require 'test/unit'

class ProductReview < Test::Unit::TestCase

  def setup
    @selenium = @selenium::WebDriver.for(:firefox)
  end

  def teardown
    @selenium.quit
  end

  def test_add_new_review
    @selenium.get("http://awful-valentine.com/")

    @selenium.find_element(:css, '.special-item a[href*="our-love-is-special"].more-info').click
    assert_equal("http://awful-valentine.com/our-love-is-special/", @selenium.current_url)
    # assert_equal("Our love is special!!", @selenium.find_element(:css, ".category-title").text
```

NOTE

We changed all instances of the selenium variable to `@selenium`; this makes our variable an instance variable for the current test class. Individual tests are now able to reference the `@selenium` variable instead of having to create their own.

Our tests are starting to look better right away; the setup method is helping us to create a new instance of Firefox before each test is started. The biggest advantage is that the `teardown` method will execute every single time the test finishes. This means that Firefox will be closed every time, even if the test fails before completion.

REMOVING DUPLICATION WITH METHODS

Let's keep refactoring our test's logic; the next item to refactor is the click on the home page for the product we desire to comment on. Let's create a new method called `select desired product on homepage`, and move the `click` code inside it, as seen here:

```
  def teardown
    @selenium.quit
  end

  def select_desired_product_on_homepage
    @selenium.find_element(:css, '.special-item a[href*="our-love-is-special"].more-info').
  end
```

After we move the `click` action to the new method, we need to invoke this method from our test, like this:

```
15    def select_desired_product_on_homepage
16      @selenium.find_element(:css, '.special-item a[href*="our-love-is-special"].more-info').click
17    end
18
19    def test_add_new_review
20      @selenium.get("http://awful-valentine.com/")
21
22      select_desired_product_on_homepage
23      assert_equal("http://awful-valentine.com/our-love-is-special/", @selenium.current_url)
24      assert_equal("Our love is special!!", @selenium.find_element(:css, ".category-title").text)
25
```

We need to perform the same refactoring in the `test_adding_a_duplicate_review`test. This way, both tests use the same method call to select a product on the home page.

REMOVING EXTERNAL TEST GOALS

In the previous chapter, we added two assertions; they were there to verify that clicking on the **MORE INFO** link on the home page takes us to the correct product. The assertions are shown here:

```
19    def test_add_new_review
20      @selenium.get("http://awful-valentine.com/")
21
22      select_desired_product_on_homepage
23      assert_equal("http://awful-valentine.com/our-love-is-special/", @selenium.current_url)
24      assert_equal("Our love is special!!", @selenium.find_element(:css, ".category-title").text)
25
```

These do not adhere to the DRY testing pattern, because the tests are no longer testing the product review functionalities; now they also test the product retrieval and display functionalities.

Besides performing duplicate assertions, they make the test suite unstable. Test that check product review functionality broke because of a minor editorial change to the page content. The most logical thing to do is delete this instability causing code and write a set of tests specifically to test product descriptions. We will delete the assertions now and add the new tests in Chapter 4, *Data-driven Testing*.

Let's take a look at the test after we delete the unnecessary assertions:

```
19  def test_add_new_review
20    @selenium.get("http://awful-valentine.com/")
21
22    select_desired_product_on_homepage
23
24    @selenium.find_element(:id, "author").send_keys("Dima")
25    @selenium.find_element(:id, "email").send_keys("dima@@selenium.com")
26    @selenium.find_element(:id, "url").send_keys("http://awful-valentine.com")
27    @selenium.find_element(:css, "a[title='5']").click
28    @selenium.find_element(:id, "comment").clear
```


USING A METHOD TO FILL OUT THE REVIEW FORM

Finally, the biggest chunk of duplication comes from filling out the review form. Since the values that we insert in the form fields are identical between the two tests, we should be able to easily pull out this duplication to `fill_out_comment_form`. Our new method will look like this:

```
18
19  def fill_out_comment_form
20    @selenium.find_element(:id, "author").send_keys("Dima")
21    @selenium.find_element(:id, "email").send_keys("dima@@selenium.com")
22    @selenium.find_element(:id, "url").send_keys("http://awful-valentine.com")
23    @selenium.find_element(:css, "a[title='5']").click
24    @selenium.find_element(:id, "comment").clear
25    @selenium.find_element(:id, "comment").send_keys("This is a comment for product #{ENV['USERNAME'] || ENV['U
26    @selenium.find_element(:id, "submit").click
27  end
28
```

REVIEWING THE REFACTORED CODE

Our tests are starting to look completely different from when we first began refactoring. In the book *Refactoring to Patterns*, the author *Joshua Kerievsky*, shows readers how to refactor code into a series of small, easy-to-understand actions. The idea is to avoid refactoring the whole class or file in one go. We might have an idea of how the code is supposed to look when we are finished with it, but rewriting everything into its final form right away tends to prove difficult. So, it is better to take many tiny steps than one giant step that will often leave us confused and frustrated.

NOTE

Refactoring to Patterns by *Joshua Kerievsky*, published by *Addison-Wesley Professional*.

Staying with the principle of small, easy-to-understand changes, let's review our test code so far before we make any more modifications to it. First, let's take a look at the four new methods we added.

```ruby
require 'rubygems'
require 'selenium-webdriver'
require 'test/unit'

class ProductReview < Test::Unit::TestCase

  def setup
    @selenium = Selenium::WebDriver.for(:firefox)
  end

  def teardown
    @selenium.quit
  end

  def select_desired_product_on_homepage
    @selenium.find_element(:css, '.special-item a[href*="our-love-is-special"].more-info').click
  end

  def fill_out_comment_form
     @selenium.find_element(:id, "author").send_keys("Dima")
     @selenium.find_element(:id, "email").send_keys("dima@@selenium.com")
     @selenium.find_element(:id, "url").send_keys("http://awful-valentine.com")
     @selenium.find_element(:css, "a[title='5']").click
     @selenium.find_element(:id, "comment").clear
     @selenium.find_element(:id, "comment").send_keys("This is a comment for product #{ENV['USERNAME'] || ENV['US
     @selenium.find_element(:id, "submit").click
  end

```

Everything should look familiar so far. The `setup` and `teardown` methods run before and after each test and manage the instances of Selenium WebDriver and Firefox.
The `select_desired_product_on_homepage` method can be invoked on the home page to click on the **MORE INFO** button for the product we want to review. Finally, the `fill_out_comment_form` method fills out the review form for the product we've selected.

Next, let's take a look at `test_add_new_review`, shown here:

```ruby

  def test_adding_a_duplicate_review
    @selenium.get("http://awful-valentine.com/")

    select_desired_product_on_homepage
    fill_out_comment_form

    error = @selenium.find_element(:id, "error-page").text
    assert_equal("Duplicate comment detected; it looks as though you\u2019ve already said that!", error)
  end
end

```

So far, we have dramatically improved the code quality of our tests. By moving out some of the duplication into individual methods, we made our test suite a lot easier to maintain in the long run.
The `select_desired_product_on_homepage` and `fill_out_comment_form` methods can

now be reused by any test in our test class. This means that if we ever need to update our test to adhere to the new functionalities, we only need to do it once in the appropriate method; all of the tests will automatically work.

Since we are extremely dedicated to having a good test suite, we will not stop refactoring just yet. Our next goal is to fix test instability caused by the Spaghetti pattern; we will break the test-on-test dependency in the next section.

TIP

Make sure you fully understand all of the actions performed so far before moving on to the next section.