

**Username:** Universidad de Alicante, SIBYD **Book:** Selenium Design Patterns and Best Practices. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

---

## Chapter 7. The Page Objects Pattern

*"There are two ways of constructing a software design: one way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies."*

--C. A. R. Hoare

**Object-oriented programming (OOP)** is not a new concept in computer science. It has been around since the early 1950s and has been integrated into almost every modern programming language. Selenium WebDriver is written using OOP and we have been interacting with individual objects this whole time though you might not have realized it. Even though OOP offers a lot of advantages for the code base, which we will discuss later in this chapter, a lot of tests written in Selenium do not take full advantage of it.

We are ready to take the principles and design patterns discussed throughout this book and create a fully functional Page Objects framework. To accomplish this task, we will be covering the following topics:

- Objects and OOP
- The Page Objects pattern
- The test tool independence pattern
- The YAGNI principle
- Making a test more or less intelligent

**Username:** Universidad de Alicante, SIBYD **Book:** Selenium Design Patterns and Best Practices. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

## Understanding objects

If you have attended any of the Selenium conferences or read any blogs on the topic, then the topic of Page Objects must have come up multiple times. Before we get into the nitty-gritty of the Page Objects pattern implementation, let's first talk about objects.

### Describing a literal object

A standard definition of an object is a material thing that can be touched, seen, and interacted with, such as a person, a car, or this book. We interact with objects on a daily basis without a second thought. Let's take a look at the cup of coffee sitting right there on your desk; tea or water if you are not a coffee drinker. Do you sit and ponder the meaning of the cup of coffee and its position within this universe? No, we just drink it, and if it's cold we reheat it or pour it into the sink as we get a new cup. By describing the temperature of the cup of coffee, we described the properties that it has. Similarly, the act of drinking out of the cup or pouring the coffee into the sink describes the actions it can perform.

### Object properties

Object properties (attributes) are things that describe the current state of the object. Our cup has several attributes that we can describe with this bit of pseudocode:

```
cup = CoffeeMug
cup.color = white
cup.hight = 5 inches
cup.contents = coffee
```

#### Note

Pseudocode is an informal high-level of describing something. It concentrates on describing a complex action or algorithm in programming-like language that is human-readable.

We can go on describing all of the attributes of the drinking utensil, such as its GPS, location, or elevation above sea level. This would become too time-consuming, so instead we will talk about the things our cup can do.

### Object actions

A *typical* cup, and I can't stress *typical* enough, has only one hole at the top. Through this hole, we can perform two actions with this cup; we can add liquids to the cup or we can remove them. Describing these actions with pseudocode will look like this:

```
liquid = coffee
cup.add(coffee) //Pouring fresh cup in the morning
cup.remove(1 sip) //This action would be in a loop until empty
```

### Objects within objects

One last item we should discuss before moving on is that objects can store other objects as a property. The coffee inside of our mug is not part of the cup itself. Instead, it is a value of the `contents` attribute. We can put other objects inside the cup, such as water, juice, or tea, which all have their own attributes and their own actions. When I filled my cup of coffee up, I followed this procedure:

```
liquid = FreshCoffee
liquid.add(Sugar)
liquid.add(Milk)

cup = CoffeMug
cup.add(liquid)
```

This little analogy is not a complete waste of time, because it helps us to better understand the concept of a programming object.

### Describing a programming object

In OOP, an object is an abstract representation of a data. Similar to the `cup` object in the preceding code, these abstract objects have properties and can

perform actions known as methods. When writing automated tests, we can use the same analogy to describe just about anything we do. For example, when filling out credit card information on the purchase form, we will be using this `CreditCard` object:

```
card = CreditCard
card.number = 4444 3333 2222 1111
card.expiration = 01/2050
```

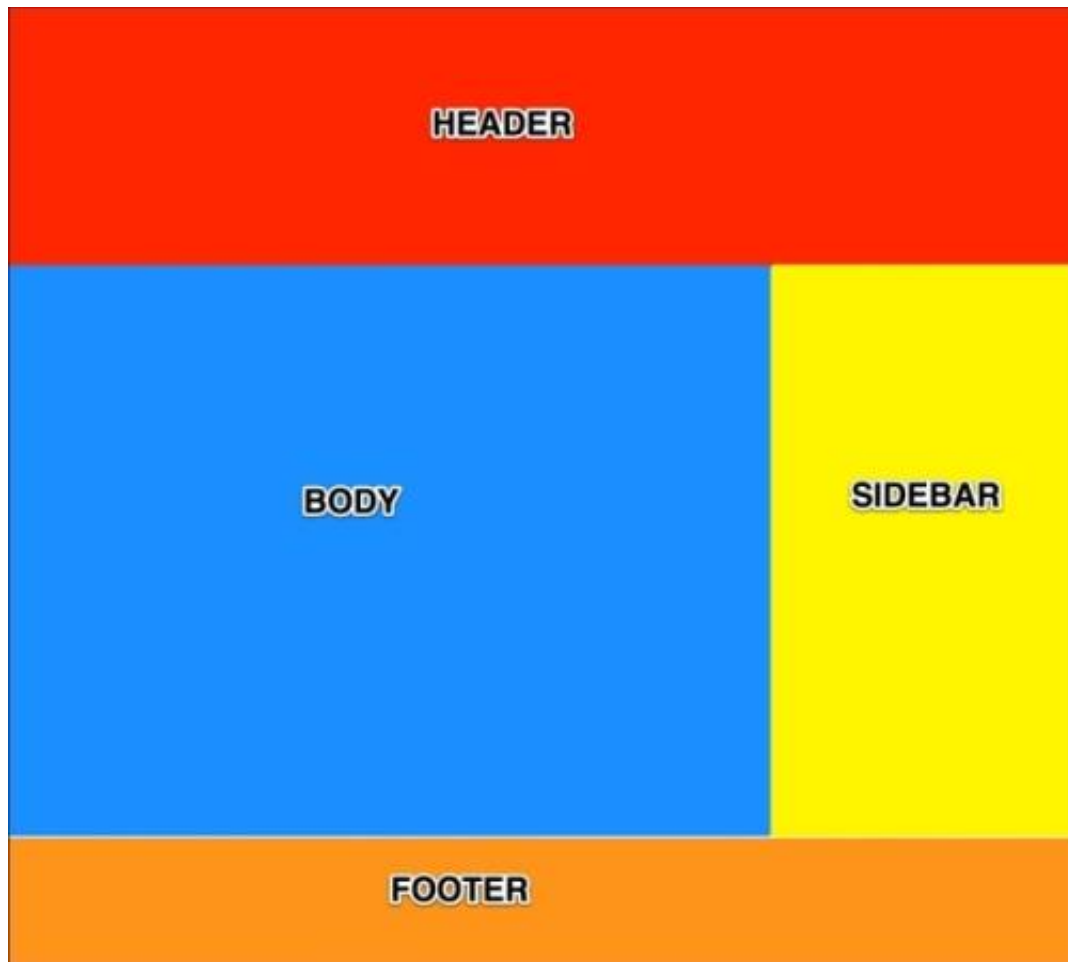
But why stop there? Why not use the similar analogy to describe every single page of the website we are testing?

## Describing a web page with objects

Earlier in this book, in [Chapter 2, The Spaghetti Pattern](#), we discussed different locator strategies to find elements on the page. By locating different elements on the page, we got a little glimpse of the hierarchy of any given page. We saw that some elements were located inside of DIVs, which were located inside of bigger DIVs, and so on. This hierarchical structuring of the web page separates different elements into groups. Let's take a look at the **Contact Us** page:

The screenshot shows the 'Contact Us' page of a website named 'Valentine'. The page layout includes a header with the logo and a navigation menu. The main content area is divided into a contact form and a sidebar. The contact form has fields for 'Your Name (required)', 'Your Email (required)', 'Subject', and 'Your Message', along with a 'Send' button. The sidebar contains a 'Cart' section showing 0 items and an 'Advertisement' section.

We can subdivide it into four clearly visible sections: the header, the body, the sidebar, and the footer. These sections are marked in the following image:



Now that the page is clearly sectioned into smaller objects, we can use a little pseudocode to describe the web page as objects:

```
page = ContactUsPage  
page.header = PageHeader  
page.body = PageBody  
page.sidebar = PageSidebar  
page.footer = PageFooter
```

Using an analogy similar to the coffee mug from earlier, we are able to describe any web page in terms of top-level objects that contain more and more granular and smaller objects within them. This style of describing a given web page is called the Page Objects pattern.

**Username:** Universidad de Alicante, SIBYD **Book:** Selenium Design Patterns and Best Practices. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

## The Page Objects pattern

The Page Objects pattern describes any web page in terms of a hierarchical **Domain Specific Language (DSL)**. The application specific DSL helps to hide the page implementation; the test is no longer allowed to directly interact with a given page, but instead uses a framework of classes and methods to accomplish the same goal. This pattern abstracts the implementation details, such as element IDs, into a framework specifically designed for the application being tested.

### Note

A DSL is a computer language that has been highly specialized for a specific application. It uses a general programming language such as Ruby or Java to implement classes and methods, which specifically apply to the application at hand.

## Advantages of the Page Objects pattern

There are many advantages of using this pattern of test development; let's take a look at a handful:

- **DSL framework:** After implementing the Page Objects pattern, we end up with a framework that describes the application from business point of view. Each action performed by a test using this framework should be easy to comprehend to anyone in the given field. That is to say, a test written for an accounting system that is heavy on the field's jargon might not be easy to comprehend to the laymen; however, anyone with basic knowledge of the field should understand the intentions of each action.

### Note

Referring to something as *business* is standard shorthand to describe the parts of the application that only the customer sees, that is, no code. The customer is anyone who uses the finished product, including people from within the company.

- **Testing behavior:** Similar to BDD, the Page Object pattern helps to test the desired behavior of the application using its DSL.

### Note

More information about BDD can be found in [Chapter 6, Testing the Behavior](#).

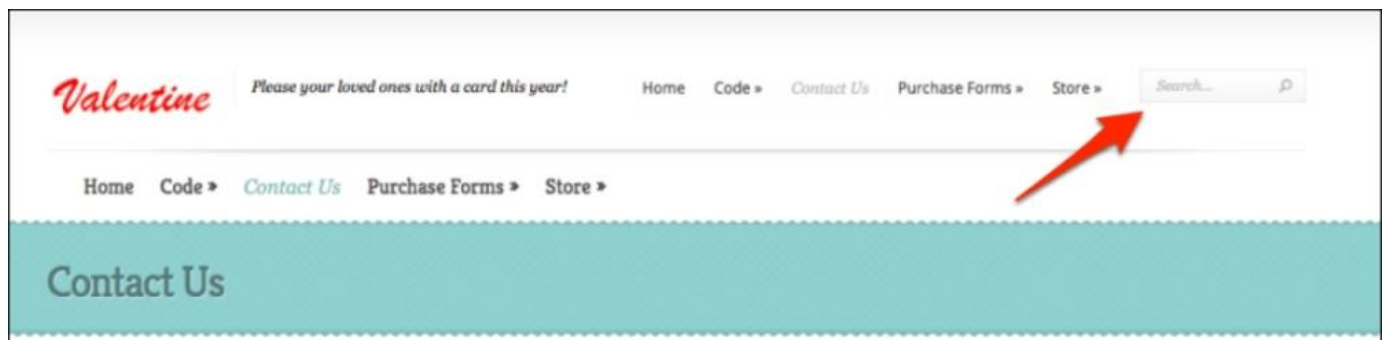
**DRY:** Unlike BDD, which has the disadvantage of phrasing a single action in multiple ways, a well-implemented and rigid Page Objects framework has one and only way to accomplish any action. This prevents duplicate implementation of the same `click` or `fill_out_form` methods.

### Note

In this context, rigidity refers to how well the rules of a framework are enforced. A flexible framework might have multiple classes or methods, which accomplish a given goal, whereas a rigid one would allow only one. Any new code that breaks this rule is not allowed.

**Modular and reusable:** Since each Page Object is made from multiple smaller objects, such as header section or login form, the smaller objects can be shared between multiple Page Objects.

**Clear Intentions:** Similar to BDD, the intended actions can be clearly represented in code. For example, a test that wishes to use the search field in the header, as shown in the following screenshot, it does not have to create a cryptic element locator search. Instead, a test that is attempting to search for `cheese` will perform an action similar to this `ContactUsPage.header.search("cheese")`. This is a lot simpler to understand than a cryptic XPATH query for the search input box.



## Disadvantages of the Page Objects pattern

There are some disadvantages to this approach. Let's take a look at them:

- Complexity is increased when using Page Objects framework. As the name implies, we can't just write a simple procedural test, we need to create a framework.
- Programming design patterns should be followed to make the code consistent and easy to understand. Otherwise, the framework quickly becomes muddled and complex to use and maintain.

### Note

A good introduction to design patterns can be found in *Design Patterns: Elements of Reusable Object-Oriented Software*, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, Addison-Wesley Professional.

- As with any new tool, it is tempting to get carried away and use it everywhere. It's tempting to implement a Page Objects framework on a test suite that only has 10 tests; this time could probably have been spent better improving existing code.

**Username:** Universidad de Alicante, SIBYD **Book:** Selenium Design Patterns and Best Practices. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

## Creating a Page Objects framework

Now that we have a theoretical knowledge of Page Objects, let's put it to use. When building a new `Page` class, we can take multiple approaches to implement. We can use any tool that our OOP language provides for us. For this example, we will be using the inheritance as a way to quickly create new `Page` classes.

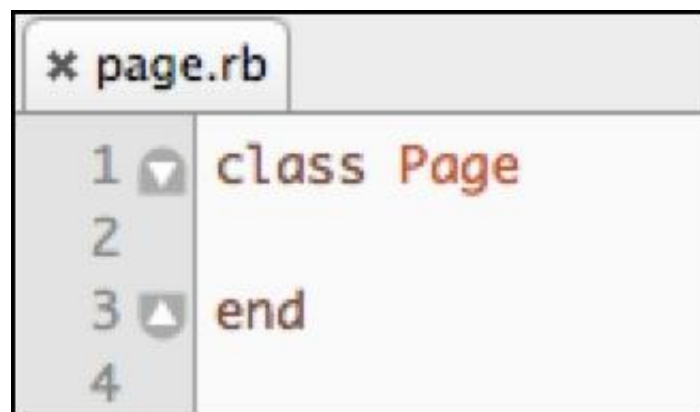
### Note

Inheritance is a feature of OOP languages, which allows new classes to be based on another class, creating a subclass. The newly created subclass inherits all of the functionality of the parent class.

The majority of the web pages on our site follow a similar pattern of display: header, body, sidebar, and footer. This means we can create a generic `Page` class that will provide us with access to different sections of the page.

### Creating a page super class

The first step of the implementation is to create a `page.rb` file that will host our class. The code inside will look like this:



```

x page.rb
1 class Page
2
3 end
4
  
```

This class will provide us with access to different parts of every page. When the test needs to check the content of the shopping cart in the sidebar, it will ask the current page for the `Sidebar` object; it will ask for the `ShoppingCart` object from that object, which will provide the desired information, such as the subtotal. The code described will look something like this:

```
current_page.sidebar.cart.subtotal
```

We can implement the getter methods for the sidebar and body inside the `Page` class.

### Note

The getter method is used to retrieve information from within an object. Since each object hides all of the properties from the rest of the world by design, it needs to have a method to retrieve the properties it wishes to share. Similarly, a setter method is used to update properties inside of the object.

I've seen multiple ways to implement the getters for different objects on the page. One approach is to break up each section into modules and have each individual page with the appropriate page section. For example, if the page containing the contact form has all four major sections, then the


`ContactUsPage` class will declare this in the following manner:

```

* contact_us_page.rb

1  include Footer
2  include Sidebar
3  include Body
4  include Footer
5
6  class ContactUsPage < Page
7
8  end
9

```



Since the error pages on our website only have a body section and no footers or headers, we would implement the [ErrorPage](#) class like this:

```

* error_page.rb

1  include Body
2
3  class ErrorPage < Page
4
5  end
6

```

This approach works well. However, to reduce the number of files created and referenced in this chapter, we will add the getters in the class itself:



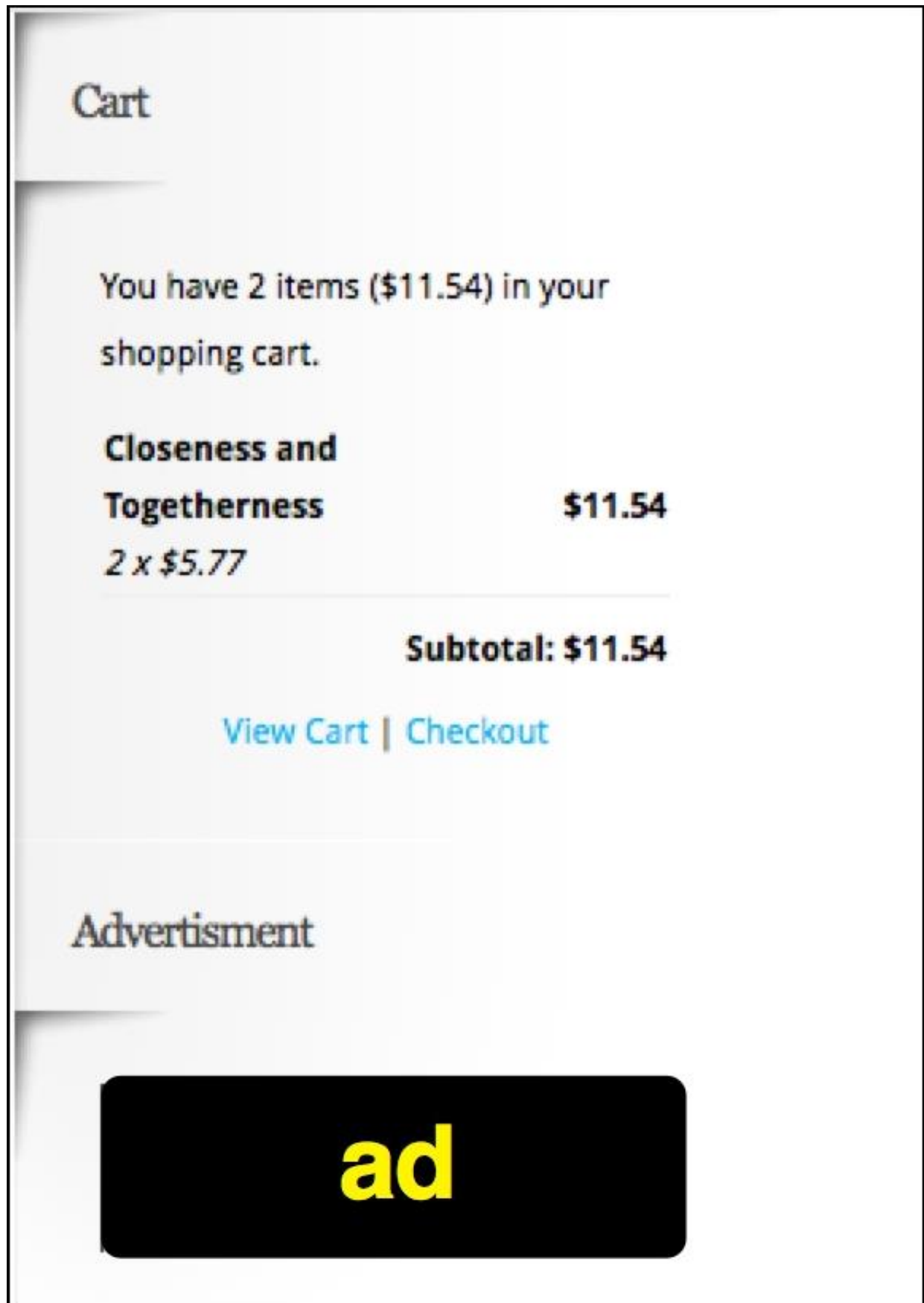
```
* page.rb

1 class Page
2   def initialize(selenium)
3     @selenium = selenium
4   end
5
6   def header
7     Header.new(@selenium)
8   end
9
10  def body
11    Body.new(@selenium)
12  end
13
14  def sidebar
15    Sidebar.new(@selenium)
16  end
17
18  def footer
19    Footer.new(@selenium)
20  end
21 end
22
```

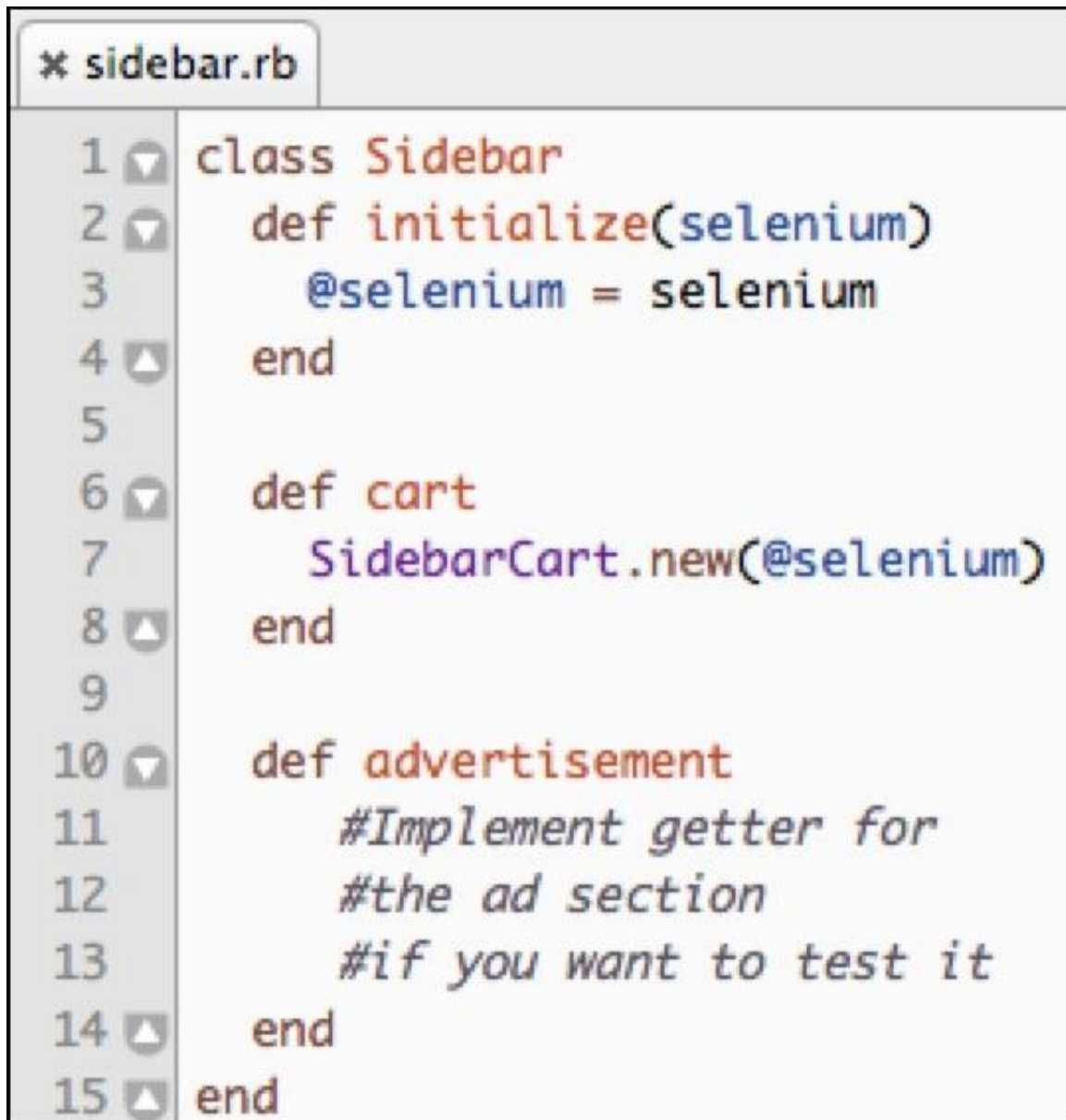
In this chapter, we will be writing a test that adds an item into the cart and checks that the sidebar displays the said item properly. For this reason, let's implement the code that deals with the sidebar next.

### Implementing sidebar objects

Before creating a sidebar class, let's take a look at the sidebar on the page and understand the two main sections it will break into. When we have an item in the shopping cart, the sidebar looks like this:



The sidebar separates into the **Cart** and **Advertisement** sections. This means that the sidebar class will have to have two getter methods, which return the appropriate object for each section. Let's implement this in `sidebar.rb` as follows:



```
* sidebar.rb
1 class Sidebar
2   def initialize(selenium)
3     @selenium = selenium
4   end
5
6   def cart
7     SidebarCart.new(@selenium)
8   end
9
10  def advertisement
11    #Implement getter for
12    #the ad section
13    #if you want to test it
14  end
15 end
```

Since we won't be testing the **Advertisement** section, the `advertisement` method is not implemented. We will move on to the `SidebarCart` class now.

### Implementing the SidebarCart class

Let's take a closer look at the sidebar shopping cart shown in the following screenshot:

You have 2 items (\$11.54) in your shopping cart.

**Closeness and Togetherness** **\$11.54**

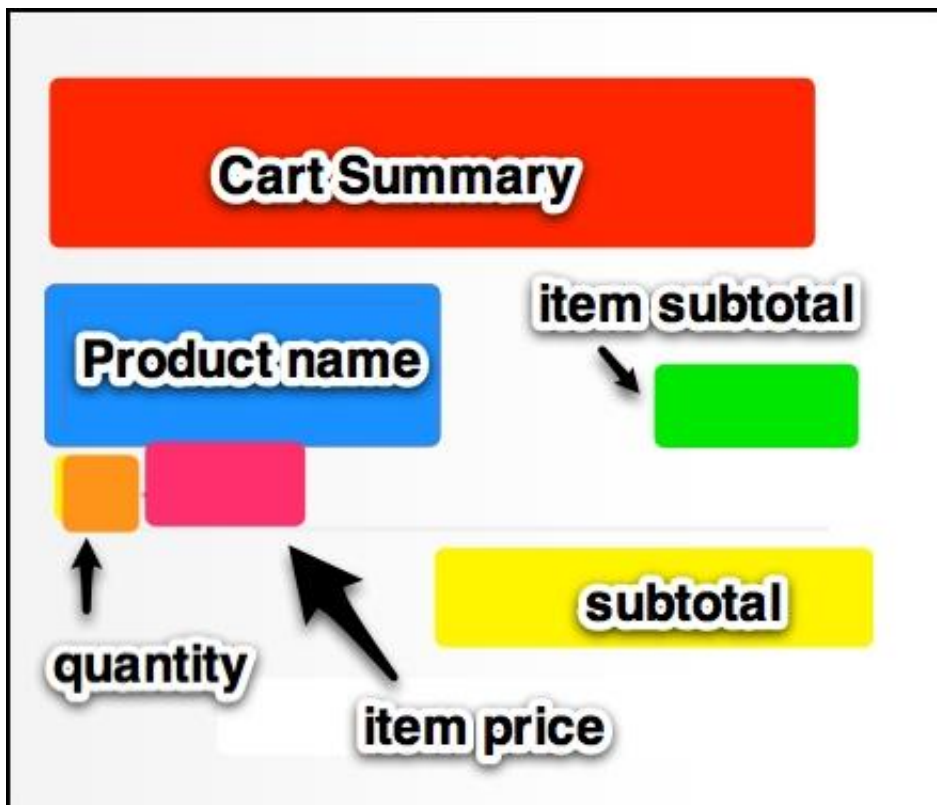
*2 x \$5.77*

---

**Subtotal: \$11.54**

[View Cart](#) | [Checkout](#)

There is a lot of information displayed in such a small place. Let's divide the whole cart into smaller sections in this breakdown:



In order to access these different pieces of information, we will need to implement a getter method for each item in the `SidebarCart` class. However, we will not implement them all because of the YAGNI principle. Since our test will only check the summary and subtotal, we will only implement those methods.

#### Note

The YAGNI principle says that if you do not need something, do not implement it. If we ever write a test that examines the product name and quantity, then we will implement the getter methods at that point.

For now, the `SidebarCart` class looks like this:

```
x sidebar_cart.rb
1 class SidebarCart
2   def initialize(selenium)
3     @selenium = selenium
4   end
5
6   def summary
7     @selenium.find_element(:id, "Cart66WidgetCartEmptyAdvanced").text
8   end
9
10  def subtotal
11    @selenium.find_element(:class, "Cart66Subtotal").text
12  end
13 end
```

Our tests are now able to interrogate the sidebar cart of any page that contains it. The test will simply follow the chain of objects until it finds the current summary or subtotal of the cart. Following this pattern, we can implement code to interact with other parts of the application. When implementing the code to interact with other parts of the application, we will keep the YAGNI principle. If we spend our time implementing a comprehensive framework instead of writing tests, we have wasted our time! The objects that were implemented in the Page Object framework but don't have a single test using them are useless. Furthermore, they quickly become obsolete when the application changes but no test failures occur to show us that the object we wrote is no longer relevant.

## Adding Self Verification to pages

Not all `ElementNotVisibleError` exceptions are the same. Sometimes, the button or DIV is not present on the page because of a defect. However, there are times when the test cannot find the element because the browser is on the completely wrong page. Let's take a look at a scenario that demonstrates the second situation.

We are testing the registration flow of the application. After filling out and submitting the registration form, the page should redirect us to the account page. On the account page, our test needs to check that the username is displayed before moving on. However, our test did not notice that registration form refreshed with duplicate username error. Our test now fails with `ElementNotVisibleError`, because our test assumed that it is on account page, but in fact still is on registration page.

This type of test failure is very common and is extremely misleading. In [Chapter 5, Stabilizing the Tests](#), we started to take screenshots every time any failure occurred. These screenshots will help us to understand the test failure, but what if our tests would detect that they are on the wrong page and fail with a much clearer error? Let's add a `verify` method to our `Page` class.

```
22
23 def verify(selenium)
24   if URI.parse(selenium.current_url).path != page_path
25     raise "Unexpected page. Expected #{page_path} but full path was #{selenium.current_url}"
26   end
```

This method gets the `current_url` of the browser from Selenium and parses it with the `URI` class. Once the URL is parsed, we grab the current `path` and compare it to the value of `page_path` method; all of this is seen on line 24. If the two paths do not match, we raise a `RuntimeError` with a helpful message that explains which page the test expected to be on, and the actual full URL in the browser. We print the full URL of the current page in case we got redirected away from our application to a new domain, such as a defect in which a link should open the target URL in a new browser window, but instead redirects in the current window.

### Tip

It might be a good idea to make the `verify` method do some other verification of the current page. The page title is another good item to verify on each page we visit.

All we have to do now is have the class initializer call the `verify` method:

```

1 class Page
2   def initialize(selenium)
3     @selenium = selenium
4     verify(@selenium)
5   end
6

```

One last thought before moving on to implementing individual page classes: the `verify` method will check the correctness of the current page by inheriting it from the `Page` super class. If we have a one off page that does not follow the verification pattern of other pages, we can overwrite the super method and create individualized verifications for each page that needs it.

### Implementing individual page classes

Now that we have a way to access different parts of individual page with the object framework and the ability to verify that we are on the correct page, it is time to start implementing individual page classes. Let's take a look at the `ContactUsPage` implementation:

```

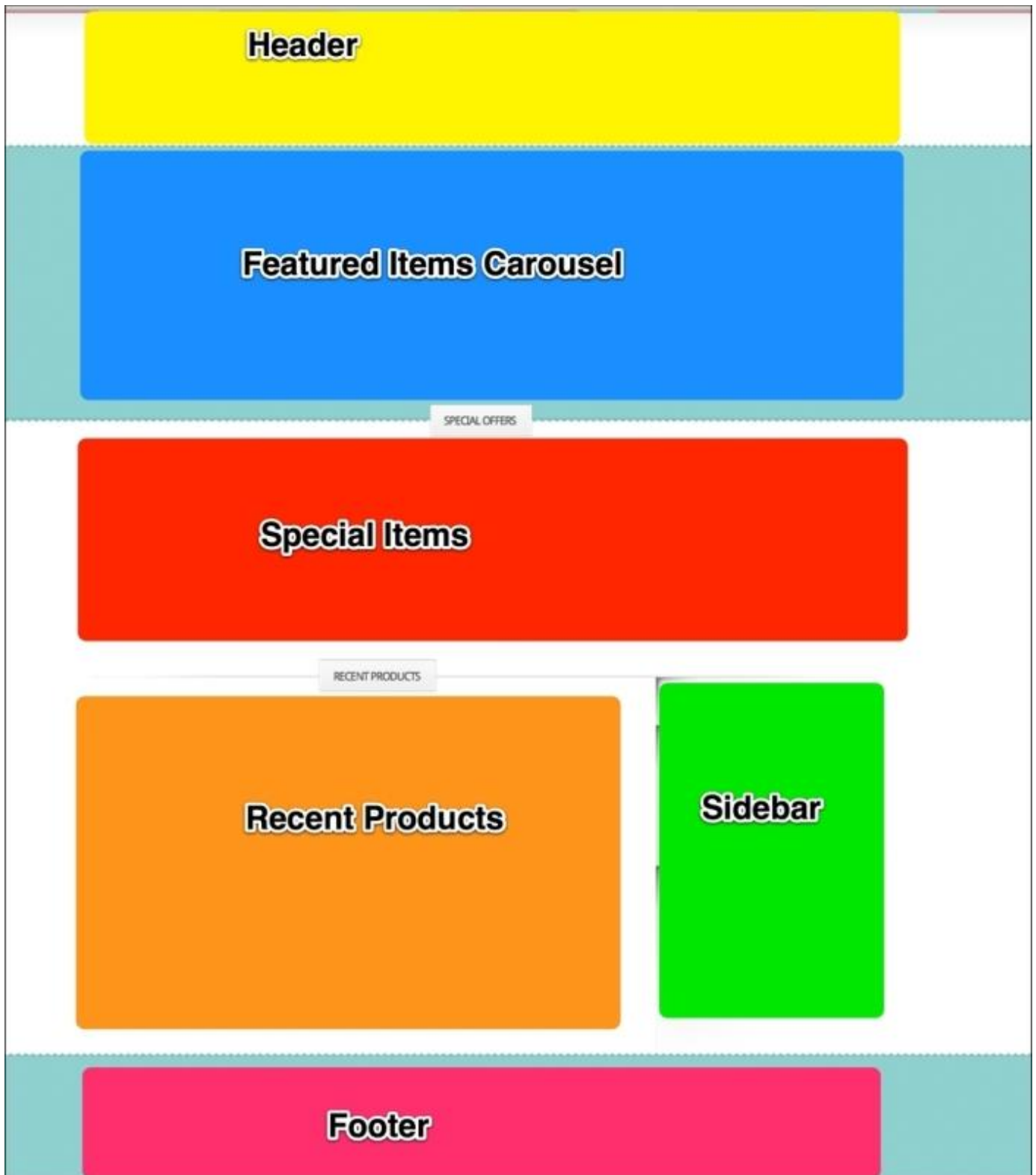
1 class ContactUsPage < Page
2   def page_path
3     "/contact-us/"
4   end
5 end
6

```

As you can see, the amount of code required to create new page classes that adhere to the standard page layout is minimal. Since there is nothing special about the `ContactUsPage` class, it can inherit all of the interactions from the `Page` super class. But what if we wanted to implement a `HomePage` class?

The majority of the pages on our website follow the same four section layout described in the *Describing a web page with objects* section of this chapter. However, the home page has six major sections, as shown in the following image:





The **body** section of the page is missing; it with **Featured Items Carousel**, the **Special Items** section, and the **Recent Products** section. The **Header**, **Sidebar**, and **Footer** sections remain the same. So the **HomePage** class needs to reflect this uniqueness. Let's take a look at the **class** definition:

```

* home_page.rb

1 class HomePage < Page
2   def page_path
3     "/"
4   end
5
6   def body
7     nil
8   end

```

We start the class by declaring the path of the existing page, so that the `verify` method can check that we are on the right page. Also, we overwrite the `body` method from the super class. Since the home page does not technically have a main body section, we will just return a `nil`. Next, we will implement the three methods needed to access the unique page sections found on the home page.

```

9
10 def special_items
11   @selenium.find_elements(:class, "special-item").collect do |element|
12     SpecialItem.new(element, @selenium)
13   end
14 end
15
16 def featured_item_carousel
17   #Implement me
18 end
19
20 def recent_products
21   #Implement me
22 end
23

```

On line 10, we have a method that searches for all instances of the `special-item` class and creates an array of the `SpecialItem` objects. Since we do not have a test that uses the `featured_item_carousel` or `recent_products` sections, we will not implement these methods yet. However, we will have a test that will add one of the **Special Offers** item to the cart, so let's take a quick look at the `SpecialItem` class:

```

* special_item.rb

1 class SpecialItem
2   def initialize(element, selenium)
3     @element = element
4     @selenium = selenium
5   end
6
7   def add_to_cart
8     @element.find_element(:class, "add-to-cart").click
9     @selenium.find_element(:id, "fancybox-outer").find_element(:class, "purAddToCart").click
10  end
11 end
12

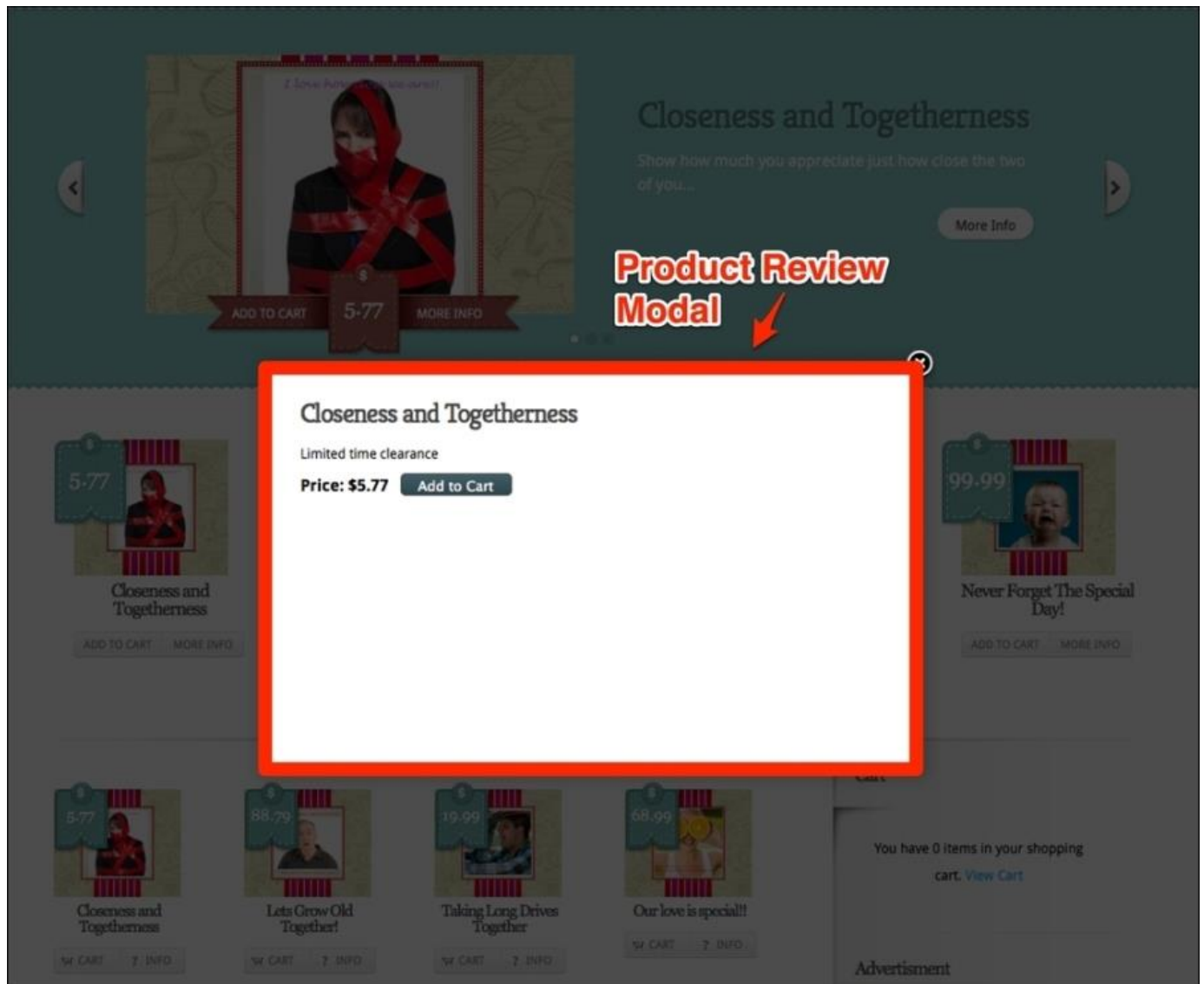
```



Each `SpecialItem` object initializes with the `element` that WebDriver found on the home page. This is done so that each `SpecialItem` instance has a reference only to itself, that is, it does not know about the existence of other special items on the home page.

Furthermore, each `SpecialItem` object implements an `add_to_cart` method as an action that it can perform.

Note that the reference to `selenium` is passed into the `SpecialItem` class. Typically, having the `element` reference alone is not only enough but is encouraged since we want the class to be as isolated as possible. However, due to peculiarities of our website's implementation, after clicking on the **Add To Cart** button for a product, a review modal opens up. This is shown in the following screenshot:



The modal does not reside within the scope of the `SpecialItem` object, so we need access to the whole scope of the page to add the item to the cart. This workaround is atypical.

### Tip

Whenever possible, make each Page Object element as *dumb and blind* of anything else happening on the page as possible. The less they know about the world outside of them, the easier it is to maintain them in the long run.

After adding the `SpecialItems` object to the `HomePage` class, our test should easily be able to add a product to the cart with this simple to understand line of code:

```
HomePage.new(@selenium).special_items.first.add_to_cart
```

The preceding method call will add the `first` product in the **Special Offers** section to the cart. We can add more functionality to the `SpecialItem` class as necessary. For example, instead of choosing the item to add by the position in the array, such as `first` or `third`, we can add a method to select the desired `SpecialItem` object by product `name` or by target URL. Our test might look like this:

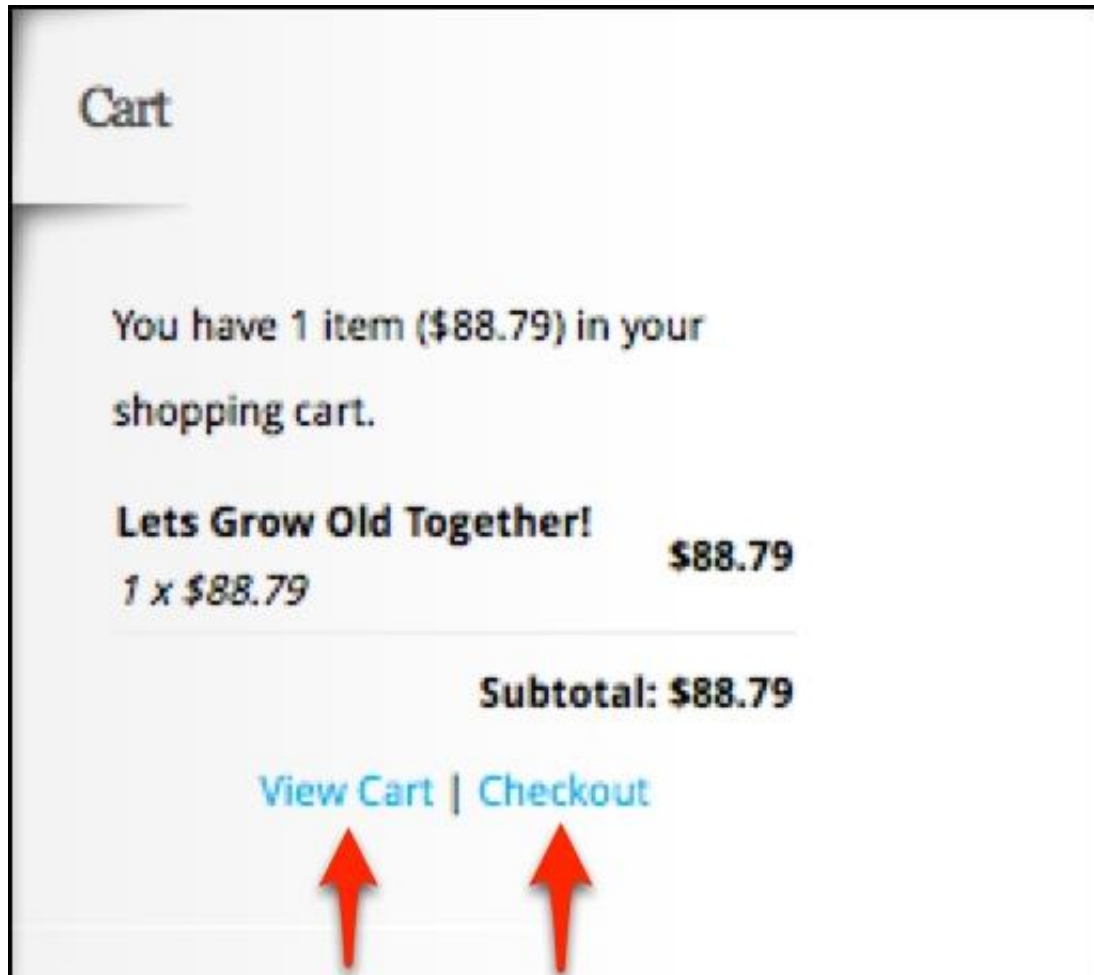
```
HomePage.new(@selenium).special_items.find("Our love is
special!!").add_to_cart
```

We will not go into the implementation details of this functionality, but it sounds like a worthy exercise for the reader to practice with. Last but not least, let's put together all of the code we just added into a test!

## Increasing the number of sidebar objects as the website grows

Before moving on to implementations of Page Objects in different testing frameworks, let's think about our `SidebarCart` class and how it will organically change as our website changes. Let's start by adding new methods to test the existing functionality.

When writing a test that checks individual items in our cart, for things such as quantity or unit price, all we have to do is add a couple of new methods to retrieve this information. We can also add a couple of methods that will perform an action of clicking on the **View Cart** and **Checkout** links:



These methods will allow us to interact with specific parts of the web page. If in the future our website adds functionality to modify the contents of the cart from the sidebar, such as changing quantity or deleting items, we can easily add two new methods to accomplish this as well. Once the initial framework is set up, the functionality added to it will grow organically as the test needs change.

**Username:** Universidad de Alicante, SIBYD **Book:** Selenium Design Patterns and Best Practices. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

## Running tests with the Page Objects framework

The largest advantage of the Page Object pattern is that it is not a zero-sum approach. That is, we do not need to convert the entire test suite to the Page Object framework to take advantage of it. Instead, we can slowly add new `Page` subclasses as they are needed and updating the existing tests to use the newly created classes as they become available. For a while, our test suite might look like a hybrid of direct Selenium `click` methods and the `add_to_cart` methods from the framework. This is perfectly acceptable as long as our code is continuously improving in the positive direction.

### Note

In the following test examples, we are not using the *The Action Wrapper pattern* section from [Chapter 5, Stabilizing the Tests](#). Thus, we have a mix of Selenium `get` and `click` method calls, and we are missing all of the stability improvements added in that chapter. This is done for both brevity and to demonstrate that the test suite can be improved in small portions.

## Using Page Objects in the Test::Unit framework

The Test::Unit framework that we have been using since [Chapter 1, Writing the First Test](#), is a good starting point to implement our Page Objects. The test, minus the `setup` and `teardown` methods, will look like this:

```

12
13 def test_cart_on_contact_page
14   @selenium.get "http://awful-valentine.com/"
15
16   page = HomePage.new(@selenium)
17   page.special_items.first.add_to_cart
18   @selenium.get "http://awful-valentine.com/contact-us"
19
20   page = ContactUsPage.new(@selenium)
21   assert_equal("You have 1 item ($5.77) in your shopping cart.",
22               page.sidebar.cart.summary)
23   assert_equal("$5.77", page.sidebar.cart.subtotal)
24 end
25

```

After navigating to the home page on line 14, we allow the `HomePage` object take over the test. Using this `Page` object, we add an item to the cart and then navigate to the **Contact Us** page on line 18. On line 20, the `ContactUsPage` object takes over and validates that we have landed on the appropriate web page, as all of the `Page` subclasses do. We then use the method chain to retrieve the `summary` and `subtotal` of the shopping cart.

As you can see, with this method, our test knows very little about the classes and IDs of different elements on the page. This may seem excessive and complicated at first; after all, when writing our test in Selenium, we want to be able to click on buttons from within the test. However, the possibilities that open up to us when using this approach are endless. Let's take a look at this piece of code in particular:

`ContactUsPage.new(@selenium).sidebar.cart.summary`

This code explains to us the behavior of our application in just a few simple method names. From this, we know the following facts about the current page:

- Our test should be on the **Contact Us** page, represented by the `ContactUsPage` object.
- The **Contact Us** page has a concept of `sidebar`, unlike some pages that do not. The `SideBar` object will allow us to interact with items within.
- We know that within the current page's sidebar, we can find a shopping cart, and use the `SidebarCart` object to interact with it.
- With the use of the `SidebarCart` object, we can retrieve `summary` of the cart or the subtotal.

We get all of this information by just looking at the method call, isn't that amazing? If we wanted to implement a method that retrieves the shopping cart

**summary** by using Selenium alone, our code would look like this:

```
@selenium.find_element(:id, "sidebar")
  .find_element(:class, "widget")
  .find_element(:id, "Cart66AdvancedSidebarAjax")
  .find_element(:id, "Cart66WidgetCartEmptyAdvanced").text
```

The test now has all of the IDs and classes hard coded in it. Furthermore, if we change the plugin we are using to display our sidebar cart, these IDs and classes will change. We will have to fix every instance it is used. With the Page Object pattern, the only change that we will need to make in our framework is how the **Sidebar** and **SidebarCart** classes locate the web elements on the page. Since the code is stored in a central place, all of the tests will automatically start using the new implementation of our website.

Let's run our Test::Unit test and make sure that it is passing with our test framework:

```
mbp-3:code dima$ ruby test_unit_example.rb
Run options:

# Running tests:
.

Finished tests in 11.802721s, 0.0847 tests/s, 0.1695 assertions/s.

1 tests, 2 assertions, 0 failures, 0 errors, 0 skips
```

The new test is working great! Let's take a look at how the test would be implemented in RSpec and Cucumber.

## Using Page Objects in different testing frameworks

Our framework is independent from different testing tools. This means that we can use it with Test::Unit, Cucumber, and RSpec tools, but it is not limited to just them. This is great for us, since it allows us to follow the test tool independence pattern, which we will discuss in greater detail in the next section. In the meantime, let's pretend that we have a large website with multiple teams working on different sections of the website. As always, some teams will want to test the code with the tools they are comfortable with. The good news is that everyone can test the entire website using the tool of their choosing, while at the same time, sharing the framework code.

### Looking at the Cucumber implementation

Since we have used Cucumber extensively in [Chapter 6, Testing the Behavior](#), let's start with it. The team responsible for implementing the sidebar widgets, such as the cart, loves to test with Cucumber. They have written this feature definition:

```
cucumber_example.feature
1  Feature: sidebar cart should display correct information on all pages.
2
3
4  Scenario: User adds checks sidebar cart on contact us page
5    Given I am on the home page
6    And I add first special offers item to the cart
7
8    And I navigate to Contact Us Page
9    Then the shopping cart should have correct information
10
11
```

To implement this test, we will have the following step definitions:

```

1
2 Given(/^I am on the home page$/) do
3   @selenium = Selenium::WebDriver.for(:firefox)
4   @selenium.get "http://awful-valentine.com/"
5 end
6
7 Given(/^I add first special offers item to the cart$/) do
8   page = HomePage.new(@selenium)
9   page.special_items.first.add_to_cart
10 end
11
12 Given(/^I navigate to Contact Us Page$/) do
13   @selenium.get "http://awful-valentine.com/contact-us"
14 end
15
16 Then(/^the shopping cart should have correct information$/) do
17   page = ContactUsPage.new(@selenium)
18   expect(page.sidebar.cart.summary).to eq("You have 1 item ($5.77) in your shopping cart.")
19   expect(page.sidebar.cart.subtotal).to eq("$5.77")
20   @selenium.quit
21 end

```

As we can see, aside from a slightly different way of declaring steps, the interaction with different pages remains the same. The consistency in how the tests interact with the web pages is very important, because it allows different teams understand the tests written by another team with another tool. Let's run our Cucumber test and make sure it passes:

```

mbp-3:code dima$ cucumber -r page_objects/cucumber_steps.rb cucumber_example.feature
Feature: sidebar cart should display correct information on all pages.

  Scenario: User adds checks sidebar cart on contact us page # cucumber_example.feature:4
    Given I am on the home page # page_objects/cucumber_steps.rb:5
    And I add first special offers item to the cart # page_objects/cucumber_steps.rb:10
    And I navigate to Contact Us Page # page_objects/cucumber_steps.rb:15
    Then the shopping cart should have correct information # page_objects/cucumber_steps.rb:19

1 scenario (1 passed)
4 steps (4 passed)
0m11.556s

```

## Looking at the RSpec implementation

RSpec is a BDD tool that follows a similar philosophy as Cucumber: *test the behavior of the application, not the implementation*. RSpec uses a different syntax to accomplish this task. Unlike Cucumber, which tries to describe the behavior in a human language, RSpec tries to use a much more rigid syntax that does not allow as much variation in how someone can describe functionality. Since RSpec definitions resemble a programming language instead of the human language, some developers prefer it to Cucumber.

### Note

Even though both RSpec and Cucumber are great tools for testing behavior, the minor difference between them cause a lot of strife between some developers. These debates of preference remind me of Mac versus Windows versus Linux arguments typically heard on any development team.

The team responsible for the shopping cart functionality loves to write tests using RSpec or Test::Unit. To test the shopping cart in the sidebar widget, we will write a test like this:



```

2
3 describe "Sidebar Shopping cart" do
4   context "contact us page" do
5
6     before(:all) do
7       @selenium = Selenium::WebDriver.for(:firefox)
8     end
9
10    after(:all) do
11      @selenium.quit
12    end
13
14    it "should have correct subtotal and summary" do
15      @selenium.get "http://awful-valentine.com/"
16
17      page = HomePage.new(@selenium)
18      page.special_items.first.add_to_cart
19      @selenium.get "http://awful-valentine.com/contact-us"
20
21      page = ContactUsPage.new(@selenium)
22      expect(page.sidebar.cart.summary).to eq("You have 1 item ($5.77) in your shopping cart.")
23      expect(page.sidebar.cart.subtotal).to eq("$5.77")
24    end
25  end

```

As we can see, some things are slightly different from everything we have written so far. The `before` and `after` methods take the place of `setup` and `teardown`, and the description of the functionality is made in short clipped `describe`, `context`, and `it` statements. The interactions with the page, however, remain the same between the three tools. Let's run the RSpec test to make sure everything is passing:

```

mbp-3:code dima$ rspec rspec_example.rb
.
Finished in 11.85 seconds (files took 0.4129 seconds to load)
1 example, 0 failures

```

Now that we have working examples of it, let's formally define the test tool independence pattern.

**Username:** Universidad de Alicante, SIBYD **Book:** Selenium Design Patterns and Best Practices. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

## The test tool independence pattern

Test tool independence occurs when the test suite is not heavily integrated with any given testing tool. For example, switching from Selenium 1 to Selenium WebDriver is difficult, because both tools use different methods to locate and click on page elements. If we wrote our framework in such a way as to hide these changing methods from the test, all we need to do to upgrade to Selenium WebDriver is update the `find_element` and `click` methods to use the new WebDriver API. This practice is referred to as the Adapter pattern. In the previous example of this chapter, the `SidebarCart` class acts as an adapter between the test and the instance of Selenium WebDriver by translating the `add_to_cart` method call into a WebDriver `click` method.

### Note

In software design, Adapter pattern is used to map functionality of different objects that have different interfaces. Adding an adapter object between the two objects that wish to communicate with each other does this; the adapter object acts as an interpreter between the two objects.

Just because our test suite is written with the Page Object pattern does not make it test tool independent. It is just as easy to lock into a given tool or a specific version of that tool inside of a `Page` class.

## Advantages of the test tool independence

There are many advantages of writing our test framework in such a way that it does not directly depend on any testing tool. Let's look at several of these here:

- **Easy upgrade:** Different tools, such as gems or libraries, change all the time. As new features are added, the public methods might change completely, becoming incompatible with previous versions. By hiding the method implementation from the test, our test does not need to know about the current version of any third-party library.
- **Easy tool switching:** With our framework, we can switch between the testing tools we want without chaining the test core much. This applies both to testing frameworks, such as Cucumber, but also applies to the testing tool, such as Selenium. After all, just because our tests are written in a regular Firefox browser today, it does not mean we will not want to add support for Chrome or a headless browser in the future.
- **Consistent descriptive API:** Since our framework describes behavior, such as adding item to the cart, instead of listing a series of clicks required to accomplish that task, it is easy to read and understand our test's intention.

## Disadvantages of the test tool independence

As always, every time we make our test more independent and resilient, we increase the amount of complexity and overhead. A simple test, described earlier in the chapter, can be written in a few lines of code if we allow the test to talk directly to Selenium. With the intention of making our test suite future-proof, we have to add many new wrapper and adapter classes today.

**Username:** Universidad de Alicante, SIBYD **Book:** Selenium Design Patterns and Best Practices. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

## The right way to implement Page Objects

I will not venture a claim that the implementation of the Page Objects in this chapter is the right and only way to do it. Just like there are many programming languages and different ways to write code, there are multiple ways to implement Page Objects. Choosing the right approach will be one of the first and most difficult tasks to figure out. I'd like to spend this section talking about different approaches we could have taken when writing our framework.

### Making pages smarter than tests


In the framework we implemented, the `@selenium` instance is passed between different `Page` objects as the test progresses. For example, after we have created an instance of Firefox browser with `WebDriver`, we pass it into each class like this:

```
@selenium.get "http://awful-valentine.com/"
page = HomePage.new(@selenium)
page.special_items.first.add_to_cart

ShoppingCartPage.new(@selenium)
```

This approach is good because it is clear to see the order of progression from page to page. It's clear to see that `@selenium` moves first to the home page and hands off itself to the `HomePage` object. Then, the test adds a product into the shopping cart, followed by test's expectation that the `ShoppingCartPage` class will be needed next. In this scenario, the test is smart and the pages are dumb. If we make our pages smarter and the test dumber, our code will look like this:

```
@selenium.get "http://awful-valentine.com/"
page = HomePage.new(@selenium)
page.special_items.first.add_to_cart
assert_equal("$5.77", page.cart.total)
```



In the preceding code, the `ShoppingCartPage` object is never explicitly declared, so the test is oblivious to the class it is asserting against. All it knows is that the next page will contain a shopping `cart` object that will return a `total` value.

The implementation of the `add_to_cart` method in the `SpecialItem` class will change to become smarter. This is the code from before the change:

```
def add_to_cart
  @element.find_element(:class, "add-to-cart").click
  @selenium.find_element(:id, "fancybox-outer").find_element(:class, "purAddToCart").click
end
```

Now, this method will know that `ShoppingCartPage` is the next in the application flow, and it will return that object back to the test:

```
def add_to_cart
  @element.find_element(:class, "add-to-cart").click
  @selenium.find_element(:id, "fancybox-outer").find_element(:class, "purAddToCart").click
  ShoppingCartPage.new(@selenium)
end
```





The instance of `@selenium` is passed between the objects. We can take this approach further and make the test not use `@selenium` at all. In the following code, the test no longer knows the URL of the `HomePage`, because the `HomePage` class has a new `navigate_to` method that takes care of navigation:

With this approach, the test dictates what needs to be done, while the test framework takes care of how things should be done. One thing to be careful about is that the framework will become too smart, that is, the behavior logic is stored in the framework and not in the test. See the *Placing logic in Page Objects* section for more information.

## Making tests smarter than pages

We can head in the opposite direction and make the tests know how the application should behave, while leaving as little logic in the `Page` classes as possible. One way to accomplish this is to make all of the page interactions into static class methods. Our test code will look something like this:

```
@selenium = Selenium::WebDriver.for(:firefox)
HomePage.special_items(@selenium).first.add_to_cart
ShoppingCartPage.cart(@selenium).total
```

Now our pages are stateless, that is, every time we want to perform an action, we have to pass in the current instance of `@selenium` to it. The test is the one that dictates the flow of the application, and the `Page` classes only perform the actions they are requested to only with the information provided by the test.

Which should be smarter, the test or the page? That's a tough question, probably the best solution is to have a compromise and make parts of the Page Object framework smart and other parts intentionally dumb, as the situation dictates.

## Using modules instead of inheritance

When I first learned about object inheritance, I wanted to use it everywhere. This is typical of any new skill or programming pattern we learn as developers. However, sometimes a module/mixin is a better and cleaner solution than creating an inheritance hierarchy. In the framework we created, we used inheritance to give different pages ability to access the header, footer, and other objects on the page. This setup works well if most of the web pages on our website are extremely similar. In case of the `HomePage` class, we had to overwrite the `body` method because the home page didn't have that section.

This approach will quickly become complicated if we have to overwrite nonexisting objects on every new `Page` subclass we create. Instead of inheriting everything from the `Page` superclass, a more practical solution would be to have each individual subpage import only the functionality it has and ignore everything that does not apply.

## Placing logic in Page Objects

One of the useful shortcuts we added in the `Page` class is the `verify` method. Each page automatically checks itself to make sure it is where it is supposed to be. We can add more logic to verify that the page is completely loaded. For example, if we have a certain image or form that needs to appear on the page every time, we could make that check happen automatically the page loads the page. If for whatever reason the element is not present, the test will fail, saying that the page was not completely loaded.

It is too easy to get carried away with verifying everything. Having some verification can be useful in debugging a test failure, but putting too much logic into the `Page` classes can be detrimental in the long term. Let's take a look at two scenarios where we have too much logic in our Page Object:

- Every page class contains detailed information about all the elements on the page. If the social network icons are missing or some content such as an image is missing, the Page Object framework throws an error to let us know that the page is not fully loaded. This is good practice when testing for page completeness every time; however, it might prevent a registration test from completing because an unrelated asset is missing.
- The login action on the Page Object checks whether the browser already has a logged in user. This check will log out the current user and login with the user the test desires. This useful check can prevent test failures due to data pollution from previous tests, where the previous test did not teardown properly. At the same time, this functionality can mask a poorly written test, which reduces the quality and usefulness of the test suite as a whole.

In conclusion, be wary about putting too much logic into the individual `Page` classes.

**Username:** Universidad de Alicante, SIBYD **Book:** Selenium Design Patterns and Best Practices. No part of any chapter or book may be reproduced or transmitted in any form by any means without the prior written permission for reprints and excerpts from the publisher of the book or chapter. Redistribution or other use that violates the fair use privilege under U.S. copyright laws (see 17 USC107) or that otherwise violates these Terms of Service is strictly prohibited. Violators will be prosecuted to the full extent of U.S. Federal and Massachusetts laws.

---

## Summary

In this chapter, you had a brief introduction to the Page Object pattern. Using this pattern, we are able to create a test suite framework, which is simple to understand and easy to maintain in the long term. We discussed different advantages of using the Page Objects as opposed to writing direct Selenium commands. These advantages are portability, upgradability, and reusability.

By using the test tool independence pattern, we demonstrated that our Page Object framework could be used with any testing tool the development wishes to use. We concluded the chapter by discussing several alternative implementations for the framework.

In the next chapter, we will talk about prioritizing the test growth in the test suite. We will also discuss different ways to manage our test environments.