The Screenplay Pattern

Better Interactions for Better Automation

[Slide]

Hello, everyone! Thanks for attending <INSERT EVENT NAME HERE>. I’d personally like to thank <INSERT HOST NAMES HERE> for inviting me to speak today.

[Slide]

My name is Andrew Knight, or “Pandy” for short. I’m the Automation Panda. (I build solutions to testing problems. I’m currently the Lead Software Engineer in Test at PrecisionLender, a Q2 Company. Be sure to read my blog and follow me on Twitter at “AutomationPanda”.)

[Slide]

I’m also the lead developer for *Boa Constrictor*, the .NET Screenplay Pattern.

Today, I’m going to introduce you to the Screenplay Pattern as a new way to automate interactions. The Screenplay Pattern isn’t exactly “new” - it’s been around for a number of years - but it is still new to a lot of folks in our industry. I strongly believe that Screenplay is a much better way to interact with Web pages than the Page Object Model.

[Slide]

In this talk, I’ll back up that claim in three parts:

1. First, I’ll cover problems with traditional ways of automating interactions.
2. Second, I’ll explain why the Screenplay Pattern is a better way.
3. Third, I’ll show how to use the Screenplay Pattern with a C# library named Boa Constrictor.

[Slide]

To start, let’s define that big “I” word I kept tossing around:

[Slide]

Interactions.

[Slide]

Simply put, interactions are how users operate software. For this talk, I’ll focus on Web UI interactions, like clicking buttons and scraping text.

[Slide]

Interactions are indispensable to testing. The simplest way to define “testing” is interaction plus verification. That’s it! You do something, and you make sure it works.

Think about any functional test case that you have ever written or executed. The test case was a step-by-step procedure, in which each step had interactions and verifications.

[Slide]

Here’s an example of a simple DuckDuckGo search test. DuckDuckGo is a search engine like Google or Yahoo. The steps here are very straightforward.

[Slide]

Opening the search engine requires navigation.

[Slide]

Searching for a phrase requires entering keystrokes and clicking the search button.

[Slide]

Verifying results requires scraping the page title and result links from the new page.

Interactions are everywhere!

[Slide]

Unfortunately, our industry struggles to handle automated Web UI interactions well. Even though most teams use Selenium WebDriver in their test automation code, every team seems to use it differently. There’s lots of duplicate code and flakiness, too. Let’s take a look at the way many teams evolve their WebDriver-based interactions. I will use C# for code examples, and I will continue to use DuckDuckGo for testing.

[Slide]

When teams first start writing test automation code using Selenium WebDriver, they frequently write raw calls. Anyone familiar with the WebDriver API should recognize these calls.

[Slide]

The WebDriver object is initialized using, say, ChromeDriver for the Chrome browser.

[Slide]

The first step to open the search engine calls “driver dot navigate dot go to URL” with the DuckDuckGo website address.

[Slide]

The second step performs the search by fetching Web elements using “driver dot find element” with locators and then calling methods like “send keys” and “click”.

[Slide]

The third step uses assertions to verify the contents of the page title and the existence of result links.

[Slide]

Finally, at the end of the test, the WebDriver quits the browser for cleanup.

Like I said, these are all common WebDriver calls. Unfortunately, there’s a big problem in this code.

[Slide]

*Race conditions*. There are three race conditions in this code in which the automation does NOT wait for the page to be ready before making interactions! WebDriver does not automatically wait for elements to load or titles to appear. Waiting is a huge challenge for Web UI automation, and it is one of the main reasons for “flaky” tests.

[Slide]

You could set an implicit wait that will make calls wait until target elements appear, but they don’t work for all cases, such as the title in race condition #2.

[Slide]

Explicit waits provide much more control over waiting timeout and conditions. They use a “WebDriverWait” object with a pre-set timeout value, and they must be placed explicitly throughout the code. Here, they are placed in the three spots where race conditions could happen. Each “wait dot until” call takes in a function that returns true when the condition is satisfied.

[Slide]

These waits are necessary, but they cause new problems. First, they cause duplicate code because Web element locators are used multiple times. Notice how “search form input homepage” is called twice.

[Slide]

Second, raw calls with explicit waits makes code less intuitive. If I remove the comments from each paragraph of code, what’s left is a wall of text. It is difficult to understand what this code does at a glance.

[Slide]

To remedy these problems, most teams use the Page Object Pattern. In the Page Object Pattern, each page is modeled as a class with locator variables and interaction methods. So, a “search page” class could look like this.

[Slide]

At the top, there could be a constant for the page URL and variables for the search input and search button locators. Notice how each has an intuitive name.

[Slide]

Next, there could be a variable to hold the WebDriver reference. This reference would come via dependency injection through the constructor.

[Slide]

The first method would be a “load” method that navigates the browser to the page’s URL.

[Slide]

And, the second method would be a “search” method that waits for the elements to appear, enters the phrase into the input field, and clicks the search button.

This page object class has a decent structure and a mild separation of concerns. Locators and interactions have meaningful names. Page objects require a few more lines of code than raw calls at first, but their parts can easily be reused.

[Slide]

The original test steps can be rewritten using this new SearchPage class. Notice how much cleaner this new code looks.

[Slide]

The other steps can be rewritten using page objects, too.

[Slide]

Unfortunately, page objects themselves suffer problems with duplication in their interaction methods.

[Slide]

Suppose a page object needs a method to click an element. We already know the logic: wait for the element to exist, and then click it.

But what about clicking another element? This method is essentially hard coded for one button.

[Slide]

A second “click” method is needed to click the other button.

[Slide]

Unfortunately, the code for both methods is the same. The code will be the same for any other click method, too. This is copy pasta, and it happens all the time in page objects. I’ve seen page objects grow to be thousands of lines long due to duplicative methods like this.

At this point, some teams will say, “Aha! More duplicate code? We can solve this problem with more Object-Oriented Programming!”

[Slide]

And they’ll create the infamous “base page”, a parent class for all other page object classes.

[Slide]

The base page will have variables for the WebDriver and the wait object.

[Slide]

It will also provide common interaction methods, such as this click method that can click on any element. Abstraction for the win!

[Slide]

Child pages will inherit everything from the base page. Child page interaction methods frequently just call base page methods.

I’ve seen many teams stop here and say, “This is good enough.” Unfortunately, this really isn’t very good at all!

[Slide]

The base page helps mitigate code duplication, but it doesn’t solve its root cause. Page objects inherently combine two separate concerns: page structure and interactions. Interactions are often generic enough to be used on any Web element. Coupling interaction code with specific locators or pages forces testers to add new page object methods for every type of interaction needed for an element. Every element could potentially need a click, a text, a “displayed”, or any other type of WebDriver interaction. That’s a lot of extra code that shouldn’t be necessary. The base page also becomes very top-heavy as testers add more and more code to share.

[Slide]

Most frustratingly, the page object code I showed here is merely one type of implementation. What do your page objects look like? I’d bet dollars to doughnuts that they look different than mine. Page objects are completely free form. Every team implements them differently. There’s no official version of the Page Object Pattern. There’s no conformity in its design. Even worse, within its design, there is almost no way for the pattern to enforce good practices. That’s why people argue whether page object locators should be public or private. Page objects would be better described as a “convention” rather than as a true design pattern.

[Slide]

There must be a better way to handle interactions. Thankfully, there is.

[Slide]

Let’s take a closer look at how interactions happen.

[Slide]

First, there is someone who initiates the interactions. Usually, this is a user. They are the ones making the clicks and taking the scrapes. Let’s call them the “Actor”.

[Slide]

Second, there is the product under test. For our examples in this talk, that’s a Web app. It has pages with elements. Web page structure is modeled using locators to access page elements from the DOM. Keep in mind, the thing under test could also be anything else, like a mobile app, a microservice, or even a command line.

[Slide]

Third, there are the interactions themselves. For Web apps, they could be simple clicks and keystrokes, or they could be more complex interactions like logging into the app or searching for a phrase. Each interaction will do the same type of operation on whatever target page or element it is given.

[Slide]

Finally, there are objects that enable Actors to perform certain types of Interactions. For example, browser interactions need a tool like Selenium WebDriver to make clicks and scrapes. Let’s call these things “Abilities”.

Actors, Abilities, and Interactions are each different types of concerns. We could summarize their relationship in one line.

[Slide]

**Actors use Abilities to perform Interactions.**

Actors use Abilities to perform Interactions.

[Slide]

This is the heart of the Screenplay Pattern. In the Page Object Convention, page objects become messy because concerns are all combined. The Screenplay Pattern separates concerns for maximal reusability and scalability.

[Slide]

So, let’s learn how to Screenplay, using Boa Constrictor.

[Slide]

“Boa Constrictor” is an open source C# implementation of the Screenplay Pattern my team and I developed at PrecisionLender. It is the cornerstone of PrecisionLender’s end-to-end test automation solution. It can be used with any .NET test framework, like SpecFlow or NUnit. The project has a rich doc site hosted using GitHub Pages. The GitHub repository name is q2ebanking/boa-constrictor, and the NuGet package name is Boa.Constrictor.

[Slide]

Let’s rewrite that DuckDuckGo search test from before using Boa Constrictor. As you watch this talk, I recommend just reading along with the code as it appears on screen to get the concepts. Trying to code along in real time might be challenging. After this talk, you can take the official Boa Constrictor tutorial to get hands-on with the code.

[Slide]

To use Boa Constrictor, you will need to install the Boa Constrictor, RestSharp, and Selenium WebDriver NuGet packages. My example code will also use Fluent Assertions and ChromeDriver.

[Slide]

The Actor is the entity that initiates Interactions. All Screenplay calls start with an Actor. Most test cases need only one Actor.

The Actor class optionally takes two arguments. The first argument is a name, which can help describe who the actor is. The name will appear in logged messages. The second argument is a logger, which will send log messages from Screenplay calls to a target destination. Loggers must implement Boa Constrictor’s ILogger interface. ConsoleLogger is a class that will log messages to the system console. You can define your own custom loggers by implementing ILogger.

[Slide]

Abilities enable Actors to initiate Interactions. For example, an Actor needs a Selenium WebDriver instance to click elements on a Web page.

Read this new line in plain English: "The actor can browse the Web with a new ChromeDriver." Boa Constrictor's fluent-like syntax makes its call chains very readable. “actor dot Can” adds an Ability to an Actor.

[Slide]

“BrowseTheWeb” is the Ability that enables Actors to perform Web UI Interactions. “BrowseTheWeb dot With” provides the WebDriver object that the Actor will use, which, in this case, is a new ChromeDriver object. Boa Constrictor supports all browser types.

All Abilities must implement the IAbility interface. Actors can be given any number of Abilities. “BrowseTheWeb” simply holds a reference to the WebDriver object. Web UI Interactions will retrieve this WebDriver object from the Actor.

[Slide]

Before the Actor can call any WebDriver-based Interactions, the Web pages under test need models. These models should be static classes that include locators for elements on the page and possibly page URLs. Page classes should only model structure - they should not include any interaction logic.

The Screenplay Pattern separates the concerns of page structure from interactions. That way, interactions can target any element, maximizing code reusability. Interactions like clicks and scrapes work the same regardless of the target elements.

The SearchPage class has two members. The first member is a URL string named Url. The second member is a locator for the search input element named SearchInput.

A locator has two parts. First, it has a plain-language Description that will be used for logging. Second, it has a Query that is used to find the element on the page. Boa Constrictor uses Selenium WebDriver's By queries. For convenience, locators can be constructed using the statically imported L method.

[Slide]

The Screenplay Pattern has two types of Interactions. The first type of Interaction is called a Task. A Task performs actions without returning a value. Examples of Tasks include clicking an element, refreshing the browser, and loading a page. These interactions all "do" something rather than "get" something.

Boa Constrictor provides a Task named Navigate for loading a Web page using a target URL. Read this line in plain English: "The actor attempts to navigate to the URL for the search page." Again, Boa Constrictor's fluent-like syntax is very readable. Clearly, this line will load the DuckDuckGo search page.

[Slide]

“Actor dot attempts to” calls a Task. All Tasks must implement the ITask interface. When the Actor calls “AttemptsTo” on a task, it calls the task’s “PerformAs” method.

[Slide]

“Navigate” is the name of the task, and “dot to URL” provides the target URL.

[Slide]

The Navigate Task’s “PerformAs” method fetches the WebDriver object from the Actor’s Ability and uses it to load the given URL.

[Slide]

“Search page dot URL” comes from the SearchPage class we previously wrote. Putting the URL in the page class makes it universally available.

[Slide]

The second type of Interaction is called a Question. A Question returns an answer after performing actions. Examples of Questions include getting an element's text, location, and appearance. Each of these interactions return some sort of value.

Boa Constrictor provides a Question named ValueAttribute that gets the "value" of the text currently inside an input field. Read this line in plain English: "The actor asking for the value attribute of the search page's search input element should be empty."

[Slide]

“Actor dot asking for” calls a Question. All Questions must implement the IQuestion interface. When the Actor calls “AskingFor” or the equivalent “AsksFor” method, it calls the question’s “RequestAs” method.

[Slide]

“ValueAttribute” is the name of the Question, and “dot Of” provides the target Web element’s locator.

[Slide]

The ValueAttribute’s “RequestAs” method fetches the WebDriver object, waits for the target element to exist on the page, and scrapes and returns its value attribute.

[Slide]

“Search page dot search input” is the locator for the search input field. It comes from the SearchPage class.

[Slide]

Finally, once the value is obtained, the test must make an assertion on it. “Should be empty” is a Fluent Assertion that verifies that the search input field is empty when the page is first loaded.

[Slide]

The test case's next step is to enter a search phrase. Doing this requires two interactions: typing the phrase into the search input and clicking the search button. However, since searching is such a common operation, we can create a custom interaction for search by composing the lower-level interactions together.

[Slide]

The “SearchDuckDuckGo” task takes in a search phrase.

[Slide]

In its “PerformAs” method, it calls two other interactions: “SendKeys” and “Click”.

[Slide]

Using one task to combine these lower-level interactions makes the test code more readable and understandable. It also improves automation reusability. Read this line in plain English now: “The actor attempts to search DuckDuckGo for ‘panda’.” That’s concise and intuitive!

[Slide]

The last test case step should verify that result links appear after entering a search phrase. Unfortunately, this step has a race condition: the result page takes a few seconds to display result links. Automation must wait for those links to appear. Checking too early will make the test case fail.

Boa Constrictor makes waiting easy. Read this line in plain English: "The actor waits until the appearance of result page result links is equal to true." In simpler terms, "Wait until the result links appear."

[Slide]

“WaitsUntil” is a special method. It will repeatedly call a Question until the answer meets a given condition.

[Slide]

For this step, the Question is the appearance of result links on the result page. Before links are loaded, this Question will return “false”. Once links appear, it will return “true”.

[Slide]

The Condition for waiting is for the answer value to become “true”. Boa Constrictor provides several conditions out of the box, such as equality, mathematical comparisons, and string matching. You can also implement custom conditions by implementing the “ICondition” interface.

[Slide]

Waiting is smart – it will repeatedly ask the question until the answer is met, and then it will move on. This makes waiting much more efficient than hard sleeps. If the answer does not meet the condition within the timeout, then the wait will raise an exception. The timeout defaults to 30 seconds, but it can be overridden.

Many of Boa Constrictor’s WebDriver-based interactions already handle waiting. Anything that uses a target element, such as “Click”, “SendKeys”, or “Text” will wait for the element to exist before attempting the operation. We saw this in some of the previous example code. However, there are times where explicit waits are needed. Interactions that query appearance or existence do not automatically wait.

[Slide]

The final step is to quit the browser. Boa Constrictor’s “QuitWebDriver” task does this. If you don’t quit the browser, then it will remain open and turn into a zombie. Always quit the browser. Furthermore, in whatever test framework you use, put the step to quit the browser in a cleanup or teardown routine so that it is called even when the test fails.

[Slide]

And there we have our completed test using Boa Constrictor’s Screenplay Pattern. All the separated concerns come together beautifully to handle interactions in a much better way.

[Slide]

The Screenplay Pattern can be used for more than Web UI interactions. Boa Constrictor has interactions for REST APIs using RestSharp, too. Let’s quickly step through a REST API test.

[Slide]

The Actor is the same as before.

[Slide]

To call REST APIs, the Actor needs an Ability named “CallRestApi”. It uses a RestSharp “RestClient” with a base URL. For this test, we will use a public API called Dog API, which returns random pictures of dogs. One Actor can have multiple Abilities, as long as each Ability has a different type.

[Slide]

Boa Constrictor uses RestSharp’s RestRequest objects directly, which specify resource path and HTTP method.

[Slide]

To call the REST API, Boa Constrictor uses this call: “The actor calls the REST request” using the given request object.

[Slide]

We can then check the response object for response code and other data.

[Slide]

This test is quite basic, but Boa Constrictor can do some advanced tricks, like downloading files, dumping responses, and automatically deserializing response bodies. REST API interactions may also be composed with Web UI interactions.

[Slide]

As we said before, the Screenplay Pattern can be summed up in one line:

[Slide]

Actors [Slide] use Abilities [Slide] to perform Interactions.

It’s that simple. Actors use Abilities to perform Interactions.

[Slide]

For those who like Object-Oriented Programming, the Screenplay Pattern is, in a sense, a SOLID refactoring of the Page Object Convention. SOLID refers to five design principles for maintainability and extensibility. I won’t go into detail about each principle here because the information is a bit dense, but if you’re interested, then snap a quick screenshot, and check out each of these principles later. Wikipedia is a good source. You’ll find that the Screenplay Pattern follows each one nicely.

[Slide]

So, why should you use the Screenplay Pattern over Page Object Convention or raw WebDriver calls? There are a few key reasons.

[Slide]

First, the Screenplay Pattern, and specifically the Boa Constrictor project, provide rich, reusable, reliable interactions out of the box. Boa Constrictor already has Tasks and Questions for every type of WebDriver-based interaction. Each one is battle-hardened and safe.

[Slide]

Second, Screenplay interactions are composable. Like we saw with searching for a phrase, you can easily combine interactions. This makes code easier to use and reuse, and it avoids lots of duplication.

[Slide]

Third, the Screenplay Pattern makes waiting easy using existing questions and conditions. Waiting is one of the toughest parts of black box automation.

[Slide]

Fourth, Screenplay calls are readable and understandable. They use a fluent-like syntax that reads more like prose than code.

[Slide]

Finally, the Screenplay Pattern, at its core, is a design pattern for any type of interaction. In this talk, I showed how to use it for Web UI interactions, but the Screenplay Pattern could also be used for mobile, REST API, and other platforms. You can make your own interactions, too!

[Slide]

Overall, the Screenplay Pattern [Slide] provides better interactions [Slide] for better automation.

That’s the point. It’s not just another Selenium WebDriver wrapper. It’s not just a new spin on page objects. Screenplay is a great way to exercise any feature behaviors under test.

And, as we saw before…

[Slide]

The Screenplay Pattern isn’t that complicated. Actors use Abilities to perform Interactions. That’s it. The programming behind it just has some nifty dependency injection.

[Slide]

If you’d like to start using the Screenplay Pattern for your test automation, there are a few ways to get started.

[Slide]

If you are programming in C#, you can use Boa Constrictor, the library I showed in the examples.

[Slide]

If you are programming in Java or JavaScript, you can use Serenity BDD – a mature, complete test automation framework that includes the Screenplay Pattern. Serenity BDD greatly influenced Boa Constrictor, but the two are entirely separate projects. Boa Constrictor is NOT Serenity BDD for .NET. Instead, Boa Constrictor aims to be a simpler, standalone implementation of the Screenplay Pattern.

[Slide]

If you are programming in Python, hold onto your seats! Python is my favorite programming language, and I think it’s one of the best languages for test automation. I’m currently developing a Screenplay implementation in Python in my spare time. It will be similar to Boa Constrictor, but the code will be even simpler because it’s Python!

[Slide]

If none of those options suit you, then you could create your own. The Screenplay Pattern does require a bit of boilerplate code, but it’s worthwhile in the end. You can always reference code from Boa Constrictor and Serenity BDD.

[Slide]

If you want to learn more about Boa Constrictor, please visit the doc site. It provides thorough information about the project and the Screenplay Pattern. I recommend taking the hands-on tutorial so you can develop a test automation project yourself with Boa Constrictor. It covers both Web UI and REST API interactions. Also, since Boa Constrictor is an open source project, I’d love for you to contribute!

[Slide]

If you want to see how to refactor an existing test automation project using Boa Constrictor’s Screenplay Pattern, check out this livestream I recently did with SabotageAndi, the lead developer of SpecFlow. We refactored all the page objects in an existing project into Screenplay calls. The amount of code reduced drastically. Even though SabotageAndi was new to Boa Constrictor, he was able to figure it out easily.

[Slide]

Thank you so much for taking the time to learn more about the Screenplay Pattern and Boa Constrictor. I’d like to give special thanks to everyone at PrecisionLender and Q2 who helped make Boa Constrictor’s open source release happen.

Again, my name is Andrew Knight. I’m the Automation Panda. Be sure to read my blog, follow me on Twitter, and reach out to me if you’d like to join the Boa Constrictor project! Thank you.