HELP YOU TO SOLVE DESIGN PROBLEMS

All about design patterns in automation testing.

WRITTEN BY
ANTON SMIRNOV

All about design patterns in automation testing.

Learn to create a framework for automation testing fundamentals fast.

This version was published on 2021-05-08

The views expressed in this book are those of the author.

Contact details:

- antony.s.smirnov@gmail.com

Related Websites:

- **All about design patterns in automation testing: https://test-engineer.site/**

- **Author’s Software Testing Blog: https://test-engineer.site/**

Table of Contents

Introduction.

Pretty often you can see test automation framework successfully running tests and reporting results but not doing what it's supposed to do: providing a reliable way for team members to build automated tests, and get reliable results.

This often happens when a test automation framework is built without planning in advance and understanding how it will be used.

At first, the team realizes that they need automated tests. One of the engineers decides to take care of it (or gets assigned) — using the tools they are familiar with; they automate the first bunch of tests.

Since initially, it's a proof of concept, some things are being implemented via the fastest and most obvious solution, which is not always utilizing the industry's best practices. Such solutions introduce technical debt. If not addressed early, the impact of technical debt grows once the framework is expanded.

As a result, few iterations later, the team gets a test automation framework that can pretty well-run tests that were in the mind of the author building it. But making a step aside, expanding coverage to additional features, or trying to get other engineers owning tests creation via such framework becomes a challenging task.

Have you ever wondered how to set up a test automation framework? Well, in this book you will learn about everything you'll need to successfully create such a framework. We're going to look at the pros and cons of preconfigured testing environments and those that are created dynamically.

This book is based on more than 5+ years of experience in the field of test automation. During this time, a huge collection of solved questions has accumulated, and the problems and difficulties characteristic of many beginners have become clearly visible. In the course of working in different places, I have repeatedly had to create a framework for testing automation from scratch. It was obvious and reasonable for me to summarize this material in the form of a book that will help novice testers quickly build an automation testing framework on a project and avoid many annoying mistakes.

This book does not aim to fully disclose the entire subject area with all its nuances, so do not take it as a textbook or Handbook — for decades of development testing has accumulated such a volume of data that its formal presentation is not enough, and a dozen books.

Also, reading just this one book is not enough to become a "senior automated testing engineer". Then why do we need this book?

First, this book is worth reading if you are determined to engage in automated testing – it will be useful as a "very beginner" and have some experience in automation.

Secondly, this book can and should be used as reference material.

Thirdly, this book — a kind of "map", which has links to many external sources of information (which can be useful even experienced automation engineer), as well as many examples with explanations.

This book is not intended for people with high experience in test automation. From time to time, I use a learning approach and try to “chew” all the approaches and build the stages step by step.

Some people more experienced in software test automation also having may find it slow, boring, and monotonous.

This book is intended for people who first approach the creation of an automation testing framework, especially if their goal is to add automation to their test approach.

First of all, I wrote this book for a tester with experience in the field of “manual” software testing, the purpose of which is to move to a higher level in the tester career.

Summary:

We can safely say that this book is a kind of guide for beginners in the field of automation software testing.

I have a huge knowledge of the field of test automation. I also have quite a lot of experience building automation on a project from scratch.

I have repeatedly had to develop and implement the framework of testing automation on projects.

The learning approach focuses on a huge chunk of theory on building the automation testing framework. The book also discusses the theory of test automation in detail.

However, the direction of automation to support testing is no longer limited to testing, so this book is suitable for anyone who wants to improve the use of automation: managers, business analysts, users, and, of course, testers.

Testers use different approaches for testing on projects. I remember when I first started doing testing, I was drawing information from traditional books and was unnecessarily confused by some concepts that I rarely had to use. And most of the books, to my great regret, did not address the aspects and approaches to test automation. Most books on testing begin by showing how you can test a software product with basic approaches. But I do not consider the approaches and implementations of test automation at the testing stage.

My main goal is to help you start building an automation testing framework using a strategy and have the basic knowledge you need to do so.

This book focuses on theory rather than a lot of additional libraries, because once you have the basics, building a library and learning how to use it becomes a matter of reading the documentation.

This book is not an "exhaustive" introduction. This is a guide to getting started in building an automation testing framework. I focused on the examples.

I argue that in order to start implementing an automation testing framework, you need a basic set of knowledge in testing and management to start adding value to automation projects.

In fact, when I started creating the automation testing framework first, I used only the initial level of knowledge in the field of testing and development.

I also want the book to be small and accessible so that people actually read it and apply the approaches described in it in practice.

Acknowledgments.

This book was created as a "work in progress" on **leanpub.com**. My thanks go to everyone who bought the book in its early stages, this provided the continued motivation to create something that added value, and then spends the extra time needed to add polish and readability.

I am also grateful to every QA engineer that I have worked with who took the time to explain their approach. You helped me observe what a good QA engineer does and how they work. The fact that you were good, forced me to 'up my game' and improve both my coding and testing skills. All mistakes in this book are my fault.

Chapter 1. Theory of design patterns.

Test automation has its own set of tasks, so there is a set of useful design patterns for this area.

Design patterns are a controversial topic. If you Google this question, you will find many other examples of design patterns in design automation that are used in various teams. In this article, I would like to collect all the patterns accumulated over several years of personal practice that I had to deal with. The article does not include patterns that I consider questionable, not useful, or that he has not encountered. There is no such thing as a “good design pattern” or a “bad design pattern”. The term “design pattern” was coined as a formulation of the problem and the proposed solution. It is very important not just to bring a design pattern to your project. It is important to understand their purpose, problems, how and how it can help you, and what problems it can solve.

There are many problems in design and development automation, and when faced with these problems, people formulated patterns. Initially, the classic Patterns were formulated a long time ago by the four who Patterns.

The book outlines all the patterns they encountered in the object-oriented world at that time. There is a problem — there is a solution, and for a long time, this concept of design patterns grew and developed, adding new patterns.

The main drivers of almost all patterns in test automation are the following factors: reliability, clarity, flexibility, maintainability, and stability.

I divided all the patterns into several groups:

- Structural patterns.
- Data patterns.
- Technical patterns.
- Business involvement patterns.

Chapter 2. Structural patterns.

Structural patterns, the main task of which is to structure the code of our tests — to simplify support, avoid duplicates, and problems with obfuscation. This makes it easier for test engineers working with the same issues to understand and change them, and easier to maintain.

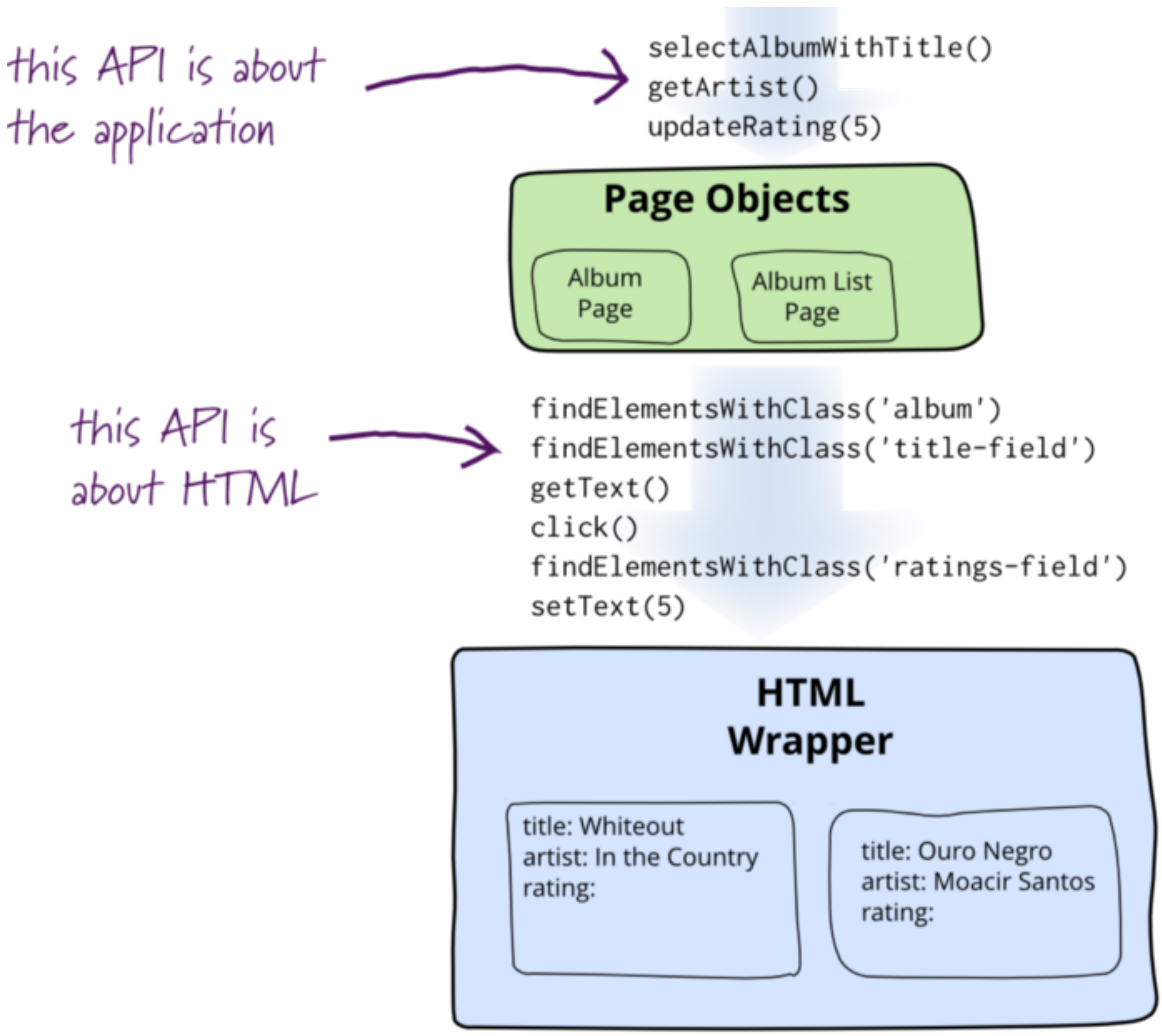
- Page Object
- Fluent/Chain of invocations
- Factory/Page Factory
- Loadable Component
- Strategy

The first and most famous pattern in test automation is the Page Object or Page Object Model. This pattern is representative of the structural patterns group.

The main goal of structural patterns is to structure test code to simplify maintenance, avoid duplication, and increase clarity. By doing that we will make it easier for other test engineers, not familiar with other codebases, to understand and to start working with the tests right away.

Page Object pattern.

Page Object is one of the most useful and used architectural solutions in automation. This design pattern helps to encapsulate the work with the individual elements of the page that allows you to reduce the amount of code and simplify its support. If, for example, the design of one of the pages is changed, we will only need to rewrite the corresponding class that describes this page.



When you write tests against a web page, you need to refer to elements within that web page in order to click links and determine what's displayed. However, if you write tests that manipulate the HTML elements directly your tests will be brittle to changes in the UI.

A page object wraps an HTML page, or fragment, with an application-specific API, allowing you to manipulate page elements without digging around in the HTML.

There are 3 problems that the Page Object pattern helps solve:

- First problem. There is a logical structure of the application when creating a test in the code, we do not understand exactly where we are now.
  When creating, we don’t see the UI directly with our test.
  Where are we after step 15? On which page?
  What actions can I do there? Can I, for example, call the login method again after step 15?
- Second problem. I want to separate the technical details (in this case, speaking about the web, these are elements in the browser, elements that perform certain actions), spread them out, and remove them from the logic of my tests, so that the test logic remains clean and transparent.
- Third problem. I want to reuse the code that I put in the pages later. Because if a lot of scripts go through the same pages, I will constantly re-use my code.

The Page Object pattern approach suggests creating one java class per application page (web page) and separate test class for each web page.

For this chapter, I will use the demo website: http://automationpractice.com/ Navigate to this website and try to get familiar with the basic functionality. The first page displayed on navigation is the landing page. Searching for any product should display the product search page.

Similarly, clicking on the Sign In button should display the Sign In page. Try to imagine the implementation of these pages in your head using the Page object pattern.

I divided it into several pages:

LandingPage.class

```
public class LandingPage {

    private static final By SIGN_IN = By.cssSelector("a[class='login']");

    public AuthorizationPage openAuthenticationPage() {
        WaitCondition waitCondition = new WaitCondition();
        waitCondition.waitForVisibilityOfElementLocatedBy(SIGN_IN).click();

        return new AuthorizationPage();
    }
}
```

AuthorizationPage.class

```
public class AuthorizationPage extends AbstractPages {

    private static final By LOGIN_FORM = By.cssSelector("form[id='login_form']");

    private static final By EMAIL_FIELD = By.cssSelector("input[id='email']");

    private static final By PASSWORD_FIELD = By.cssSelector("input[id='passwd']");

    private static final By SIGN_IN_BUTTON = By.cssSelector("button[id='SubmitLogin']");

    private static final By INPUT_EMAIL = By.cssSelector("input[id='email_create']");

    private static final By CREATE_ACCOUNT_BUTTON = By.cssSelector("button[id='SubmitCreate']");

@Step
public AccountPage enterCredentialUser() {
        checkThatLoginFormAvailable();
        enterUserEmail();
        enterPassword();
        clickSignInButton();
        return new AccountPage();
    }

@Step
public CreateAccountPage enterEmailForNewUser() {
        WaitCondition waitCondition = new WaitCondition();
        waitCondition.waitForVisibilityOfElementLocatedBy(INPUT_EMAIL).clear();

waitCondition.waitForVisibilityOfElementLocatedBy(INPUT_EMAIL).sendKeys(createEmailForNewUser());
waitCondition.waitForVisibilityOfElementLocatedBy(CREATE_ACCOUNT_BUTTON).click();

        return new CreateAccountPage();
    }

    private AuthorizationPage checkThatLoginFormAvailable() {
        WaitCondition waitCondition = new WaitCondition();
        waitCondition.waitForVisibilityOfElementLocatedBy(LOGIN_FORM).isDisplayed();

        return this;
    }
```

```
    private AuthorizationPage enterUserEmail() {
        WaitCondition waitCondition = new WaitCondition();
        User user = getJsonData("account", User.class, "account");

        waitCondition.waitForVisibilityOfElementLocatedBy(EMAIL_FIELD).clear();
        waitCondition.waitForVisibilityOfElementLocatedBy(EMAIL_FIELD).sendKeys(user.getEmail());

        return this;
    }

    private AuthorizationPage enterPassword() {
        WaitCondition waitCondition = new WaitCondition();
        User user = getJsonData("account", User.class, "account");

        waitCondition.waitForVisibilityOfElementLocatedBy(PASSWORD_FIELD).clear();

waitCondition.waitForVisibilityOfElementLocatedBy(PASSWORD_FIELD).sendKeys(user.getPassword();

        return this;
    }

    private AuthorizationPage clickSignInButton() {
        WaitCondition waitCondition = new WaitCondition();

        waitCondition.waitForVisibilityOfElementLocatedBy(SIGN_IN_BUTTON).isDisplayed();
        waitCondition.waitForVisibilityOfElementLocatedBy(SIGN_IN_BUTTON).click();

        return this;
    }
}
```

ShoppingPage.class

```
public class ShoppingPage {

    private static final By PROCEED_CHECKOUT = By.cssSelector("a[class*='standard-checkout button-
medium']");

    private static final By PROCEED_ADDRESS = By.cssSelector("button[name='processAddress']");

    private static final By PROCEED_CARRIER = By.cssSelector("button[name='processCarrier']");

    private static final By AGREE_CHECKED = By.cssSelector("input[id='cgv']");

    private static final By CONFIRM_BUTTON = By.cssSelector("p[id='cart_navigation']
button[type='submit']");

@Step
public ShoppingPage completeOrder() {
        final WaitCondition waitCondition = new WaitCondition();
        waitCondition.waitForVisibilityOfElementLocatedBy(PROCEED_CHECKOUT).isDisplayed();
        waitCondition.waitForVisibilityOfElementLocatedBy(PROCEED_CHECKOUT).click();

        waitCondition.waitForVisibilityOfElementLocatedBy(PROCEED_ADDRESS).isDisplayed();
        waitCondition.waitForVisibilityOfElementLocatedBy(PROCEED_ADDRESS).click();

        waitCondition.waitForVisibilityOfElementLocatedBy(PROCEED_CARRIER).isDisplayed();
        waitCondition.waitForVisibilityOfElementLocatedBy(PROCEED_CARRIER).click();

        selectAgree();

        return new ShoppingPage();
    }

    private ShoppingPage selectAgree() {
        final WaitCondition waitCondition = new WaitCondition();
        waitCondition.waitForVisibilityOfElementLocatedBy(AGREE_CHECKED).isDisplayed();
        waitCondition.waitForVisibilityOfElementLocatedBy(AGREE_CHECKED).click();
        return this;
    }
```

```
@Step
public ShoppingPage paymentPurchases(final PayMethod payMethod) {
        final WaitCondition waitCondition = new WaitCondition();

waitCondition.waitForVisibilityOfElementLocatedBy(By.cssSelector(payMethod.getPayMethod())).click();

        return this;
    }

@Step
public ShoppingPage confirmOrder() {
        final WaitCondition waitCondition = new WaitCondition();
        waitCondition.waitForVisibilityOfElementLocatedBy(CONFIRM_BUTTON).click();

        return this;
    }

@Step
public AccountPage checkThatOrderSuccess() {
        Assert.assertTrue(DriverHolder.getDriverThread().getCurrentUrl().contains("controller=order-
confirmation"));

        return new AccountPage();
    }
}
```

StorePage.class

```
public class StorePage {

    private static final By CATEGORY_LIST = By.cssSelector("div[id='center_column'] h3");

    private static final By PRODUCT_LIST = By.cssSelector("ul[class='product_list grid row'] img");

    private static final By ADD_CART = By.cssSelector("form[id='buy_block'] p[id='add_to_cart']");

    private static final By PROCEED_BUTTON = By.cssSelector("div[class='button-container'] a");

    private static final By TABLE_ORDERS = By.cssSelector("table[id='order-list'] tbody tr");

@Step
public AccountPage checkThatSiteMapFunctionalWorkDone() {
        final List<String> notAllEqualList = Arrays.asList("Our offers", "Your Account",
"Categories", "Pages");
        final List<WebElement> elementList =
DriverHolder.getDriverThread().findElements(CATEGORY_LIST);
        Assert.assertTrue(!elementList.isEmpty());

        Assert.assertTrue(elementList.size() == 4);

        for (int i = 0; i < elementList.toArray().length; i++) {
            final String textActual = elementList.get(i).getText();
            final String textExpected = notAllEqualList.get(i).toUpperCase();
            Assert.assertEquals(textActual, textExpected);
        }
        return new AccountPage();
    }

@Step
public StorePage selectFirstProductFromList() {
        final WaitCondition waitCondition = new WaitCondition();
        final List<WebElement> elementList =
DriverHolder.getDriverThread().findElements(PRODUCT_LIST);
        elementList.stream().findFirst().get().click();

        waitCondition.waitForVisibilityOfElementLocatedBy(ADD_CART).isDisplayed();
        waitCondition.waitForVisibilityOfElementLocatedBy(ADD_CART).click();

        return this;
    }

@Step
public ShoppingPage proceedToCheckoutStage() {
        final WaitCondition waitCondition = new WaitCondition();
        waitCondition.waitForVisibilityOfElementLocatedBy(PROCEED_BUTTON).isDisplayed();
```

```
        waitCondition.waitForVisibilityOfElementLocatedBy(PROCEED_BUTTON).click();

        return new ShoppingPage();
    }

@Step
public AccountPage checkThatOrdersHistoryNotEmpty() {
        final List<WebElement> webElementList =
DriverHolder.getDriverThread().findElements(TABLE_ORDERS);

        Assert.assertTrue(!webElementList.isEmpty());

        return new AccountPage();
    }

@Step
public StorePage selectSortBy(final SortBy sortBy) {
        final Select select = new
Select(DriverHolder.getDriverThread().findElement(By.cssSelector("select[id='selectProductSort']")))
;
        select.selectByValue(sortBy.getValue());

        return new StorePage();
    }

@Step
public AccountPage checkThatSelectLowerPriceSeller(final OrderWay orderWay) {

Assert.assertTrue(DriverHolder.getDriverThread().getCurrentUrl().contains(orderWay.getValue()));

        return new AccountPage();
    }
}
```

Best Practices.

Now when we know what Page Object is and how it can be utilized in our project let's dive into more advanced techniques and best practices of development with it.

- Test Logic. There's generally good advice to keep all your test logic (including assertions) away from Page Objects.
- Reusable elements. If several app views contain the same widget or menu it's always a good move to create separate objects for the common element or extend both page objects from one superclass with extracted common logic.

- Chain Methods. Chain methods are considered the industry standard in designing Page Objects since they allow you to write automated tests in the fashion you write your usual test cases.
- Waits. In the real-world apps usually have dynamic elements and complicated animations. Thus another best practice is to wait until your view is opened in the constructor of your Page Object class.

There are other design patterns that we will also look at in the following chapters. Some use the page factory to create instances of their page objects. A discussion of all this is beyond the scope of this chapter. In this chapter, I just wanted to introduce concepts to make the reader aware of the Page Object pattern.