

The background of the book cover features a low-angle, black and white photograph of a modern building's architecture, showing curved concrete structures and a series of small, upward-pointing light fixtures. A large, semi-transparent red rectangle is overlaid on the upper half of the image, serving as a backdrop for the title and subtitle.

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 2021-05-24

The right of Anton Smirnov to be identified as the author of this work has been asserted by him in accordance with the Copyright, Design and Patents Act 1988.



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

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

Contact details:

- antony.s.smirnov@gmail.com

Related Websites:

- This book is for sale at <https://leanpub.com/allaboutdesignpatternsinautomationtesting>

Every effort has been made to ensure that the information contained in this book is accurate at the time of going to press, and the publishers and author cannot accept any responsibility for any errors or omissions, however, caused. No responsibility for loss or damage occasioned by any person acting, or refraining from action, as a result of the material in this publication can be accepted by the editor, the publisher, or the author.

Table of Contents

All about design patterns in automation testing.	2
<i>Introduction.....</i>	5
<i>Chapter 1. Theory of design patterns.</i>	11
<i>Chapter 2. Structural patterns.....</i>	12
Page Object pattern.....	13
Fluent / Chain of Invocations pattern.....	20
Page Factory pattern.	23
Loadable Component pattern.....	26
Strategy pattern.	29
<i>Chapter 3. Creational patterns.</i>	35
Factory Method Pattern.....	36
Abstract Factory pattern.	44
Singleton pattern.....	52
<i>Chapter 4. Data Patterns.</i>	55
Value Object pattern.	56
Builder pattern.....	60
Assert Object/Matchers pattern.....	69
Data Registry pattern.	73
Object Pool / Flyweight pattern.....	74
Data Provider pattern.	79
<i>Chapter 5. Technical Patterns.....</i>	83
Decorator pattern.	84
Proxy pattern.	92
<i>Chapter 6. Business Involvement Patterns.</i>	97
Keyword Driven Testing pattern.	98
Behavior Specification pattern.	100
Behavior Driven Development pattern.....	101
Steps pattern.....	102
<i>Bonus. The mediator pattern.....</i>	104
<i>Conclusion.....</i>	108

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.



This work is licensed under a [Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License](https://creativecommons.org/licenses/by-nc-sa/4.0/)

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.

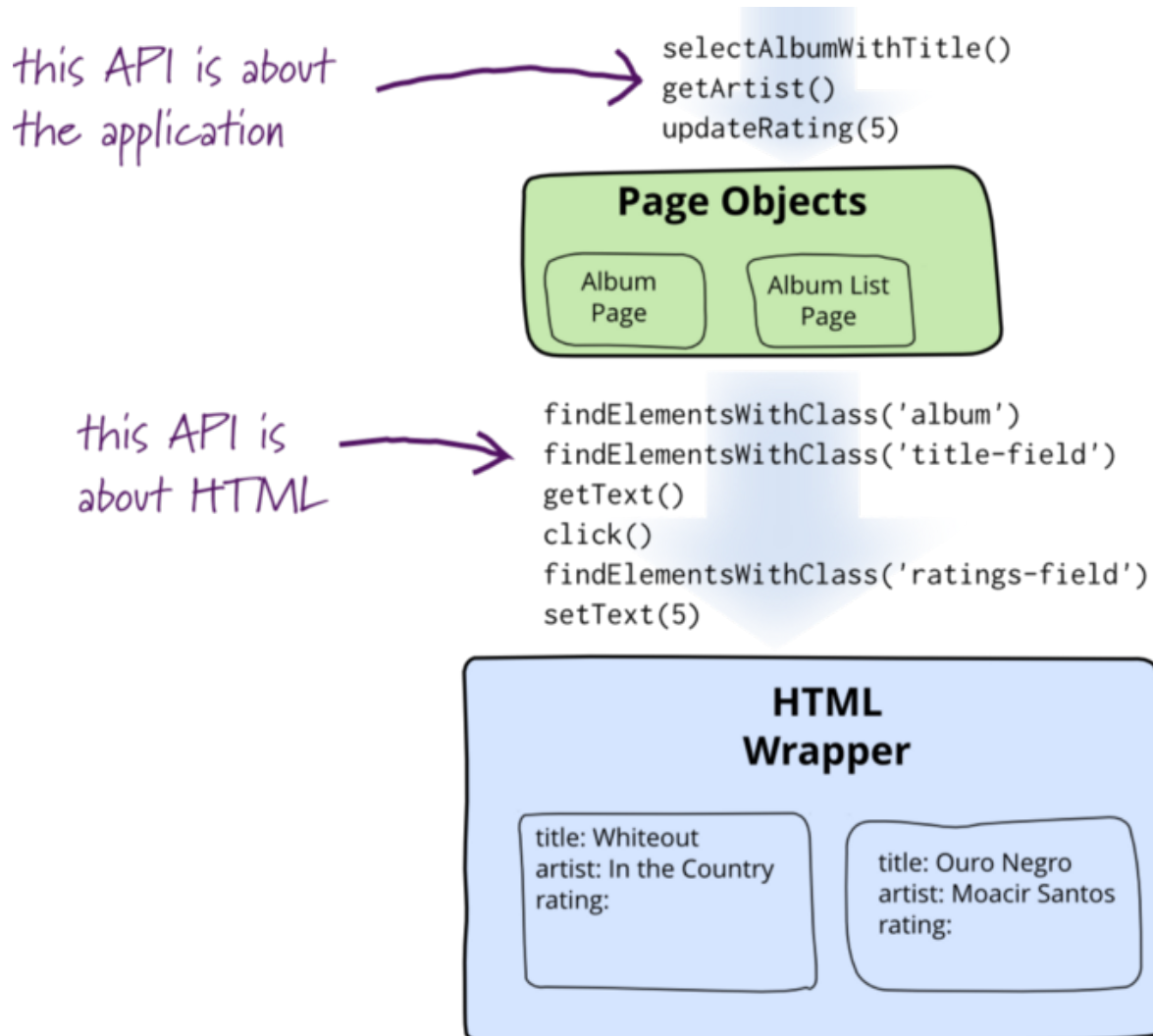
- Page Object
- Fluent/Chain of invocations
- Factory/Page Factory
- Loadable Component
- Strategy

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= 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("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("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'] img");

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

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

Fluent / Chain of Invocations pattern.

The second pattern we were going to talk about is Chain of Invocations. It's usually being used along with Page Object, so most of you might be familiar with it too. The problem it resolves is helping test developer to determine if she can use the object or she should switch to other one. For example, user is on the login page and he's pressing some button. By writing code in old fashion we wouldn't be sure if he's still on the login page or already on the home page.

And now just imagine if you had 50 similar methods in your test. You cannot be sure if you can invoke them right away, or they depend on some order, or even they can not be invoked after one of them was executed. For example, can I click on hint dialog if I haven't entered any character into input field? Probably not, because at the very moment, it doesn't exist yet!

```
@Test
public void simpleAuthorizationTest() {
    LandingPage landingPage = new LandingPage();
    AuthorizationPage authorizationPage = new AuthorizationPage();
    AccountPage accountPage = new AccountPage();
    landingPage.openAuthenticationPage();
    authorizationPage.enterPassword("admin");
    authorizationPage.enterPassword("admin");
    authorizationPage.clickSignInButton();
    accountPage.checkFullNameForNewUser();
    accountPage.clickSignOutButton();
}
```

After we apply the pattern to the code of the automatic script it will look like this:

```
@Test
public void simpleAuthorizationTest() {
    new LandingPage()
        .openAuthenticationPage()
        .enterCredentialUser()
        .checkFullNameUser()
        .clickSignOutButton();
}
```

```

public class LandingPage {

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

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

        return new AuthorizationPage();
    }
}

```

```

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']");

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

    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;
    }
}

```

```

public class AccountPage extends AbstractPages {

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

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

    private static final By CONTACT_US_LINK = By.cssSelector("div[id='contact-link']");

    @Step
    public AccountPage checkFullNameUser() {
        final WaitCondition waitCondition = new WaitCondition();

        final User user = getJsonData("account", User.class, "account");

        Assert.assertTrue(waitCondition.waitForVisibilityOfElementLocatedBy(ACCOUNT).getText().contains(user
            .getFullName()));

        return this;
    }
}

```

```

@Step
public AccountPage clickSignOutButton() {
    final WaitCondition waitCondition = new WaitCondition();
    waitCondition.waitForVisibilityOfElementLocatedBy(SIGN_OUT_BUTTON).isDisplayed();
    waitCondition.waitForVisibilityOfElementLocatedBy(SIGN_OUT_BUTTON).click();

    return this;
}

```

This pattern is important in case you're trying to scale your test automation framework. Large amount of pages, elements and methods available for usage may create a confusion, especially for someone not really familiar with your domain. But by implementing Fluent Invocations modern IDE will give you a hint every time you'll try to invoke some method on the object by autocompletion feature.

Usually if you want to interrupt method chain, all you have to do is to assign the return value (e.g. string, number etc.) to some variable. This way you'd show that this is the end of the sequence of invocations. At this point developer should stop and think about next method to be invoked in test script before doing that.

Chain of Invocations is easy to implement. All you have to do is to return the value in every method of Page Object. This might be this, some value or any other object, for example next Page Object after method invocation (transition to other page).

This pattern doesn't help to reduce code a lot (that's not its mission thought), but it allows you to not repeat yourself by putting the object again and again before invoking its methods. Also IMO it makes code a little bit prettier.

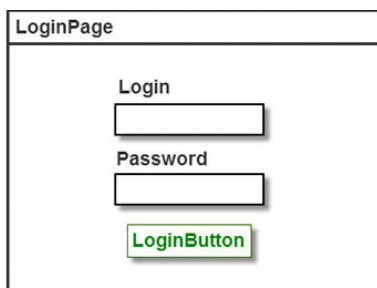
Page Factory pattern.

A Page Factory or Factory is an extension of the page object template. This helps to further encapsulate the page attributes and methods by providing **findBy** annotations. This pattern can be applied not only to pages. This pattern has arisen because sometimes, to initialize your page, you need to do more than just say "new page" or open. In fact, we hide you in more logic on the page and inside this page, there is still some additional logic, and we need to register this logic and initialize its elements. We need this information to be hidden from the person who creates this page, so that it is hidden – this is technical information that is not important to anyone.

This is where the Factory approach applies. We use this approach: I say "**new MainPage**", pass the driver there, and then say "page, open". If I wanted to do something extra on this opening, I would either need to put it in the open method, which would become the factory method, because it would open this page, initializing it and making it new, or I would need to put it in the constructor, which may not be very good either. So, there is an alternative approach-when you just specify your factory. You can simply implement **PageFactory** and **initElements**, and you will get an instance of the class of this page with all the initialized elements that are in this page.

An additional factor that will work here is the initialization of all elements. I can open any page, and I don't have to start from the main page.

For login form



The diagram shows a rectangular window titled "LoginPage". Inside the window, the text "Login" is positioned above a horizontal input field. Below this, the text "Password" is positioned above another horizontal input field. At the bottom of the form, there is a green rectangular button with the text "LoginButton" in white.

automation script looks like this:

```
public class LoginPage extends BaseClass {
    WebDriver driver;
    private static final By LOGIN_FIELD =
        By.cssSelector("input[class='login']");
    private static final By PASSWORD_FIELD =
        By.cssSelector("input[class='password']");
    private static final By SUBMIT_BUTTON =
        By.cssSelector("button[type='submit']");

    public LoginPage(WebDriver driver) {
        super(driver);
    }

    public HomePage login(User user) {
        driver.findElement(LOGIN_FIELD)
            .sendKeys(user.getUsername());
        driver.findElement(PASSWORD_FIELD)
            .sendKeys(user.getPassword());
        driver.findElement(SUBMIT_BUTTON)
            .click();
        return new HomePage(driver);
    }
}
```

After we apply the pattern to the code of the automatic script it will look like this:

```
class PageObject {
    WebDriver driver;

    PageObject(final WebDriver driver) {
        this.driver = driver;
        PageFactory.initElements(driver, this);
    }
}
```


And **LoginPage.class** look like this:

```
public class LoginPage extends PageObject {

    @FindBy(name = "input[class='login']")
    private WebElement login;

    @FindBy(name = "input[class='password']")
    private WebElement password;

    @FindBy(id = "button[type='submit']")
    private WebElement submitButton;

    LoginPage(WebDriver driver) {
        super(driver);
    }

    @Step
    public void login(User user) {
        this.login.clear();
        this.login.sendKeys(user.getUsername());

        this.password.clear();
        this.password.sendKeys(user.getPassword());
        this.submitButton.click();
    }
}
```

Pay attention to `PageFactory#initElements` invocation. This static helper initializes all fields with `FindBy` annotations on the page, which will be found on it on each call. The main advantage is the fact that now we work directly with fields, buttons, windows, etc., and do not worry about low-level driver's interactions exactly the same way our app users do.

Loadable Component pattern.

This pattern is used by most of the developers who work with user interface tests. The thing is, when the test makes the transition from one page to another, it doesn't know if the targeted page was loaded completely. This means that when you open a logical page and are going to work with any element of this page, you need to wait and make sure that the page is fully loaded and only then perform some actions. This approach requires that before each of your actions, you must wait for the element, process it, and wait for the next element to appear. Often, this wait is not done, because many expect that the framework will do it for you. For example, the web driver has the concept of implicit wait. The framework will implicitly be able to wait for the element from the side of the browser that you will work with. Sometimes that's not enough. There are cases when implicit wait is small and it does not work for your automation scripts, the so-called explicit wait appears when you are thoroughly waiting for something, and it turns out that now, after each such good action that changes the logical page, another wait is added to your test logic: something was done-wait, something was done – wait.

Such a template loads your test logic very much, because you don't have it in the test description itself, you just say "go to that page" and already include this expectation there.

To avoid this, you can inherit every page you make from the loadable component and overload the method from loaded, thereby hiding this expectation and specifying it for each page inside the page itself. In this case, you will have encapsulated logic in one place – if you call the same page in five places and you don't have to write this wait with your hands anywhere else. That's what this pattern is designed for and exists for. Again, there is no strict rule on how to do this wait. Personally, I like to move this waiting to Base Page and override the abstract method for waiting for a particular element. How to do that in your case, completely up to you. But before implementing such complicated logic you need to be sure that your app page transition could cause you some problem. Otherwise, just the usual implicit wait might be enough.

```

public class LoginPage {

    private final WebDriver driver;

    public LoginPage(WebDriver driver) {
        this.driver = driver;
    }

    public void login(User user) {
        setLogin(user.username);
        setPassword(user.password);
        clickLogin();
    }

    public void setLogin(String login) {
        WebElement field =
driver.findElement(By.cssSelector("input[class='login']"));
        clearAndType(field, login);
    }

    public void setPassword(String password) {
        WebElement field =
driver.findElement(By.cssSelector("input[class='password']"));
        clearAndType(field, password);
    }

    public void clickLogin() {
        WebElement field =
driver.findElement(By.cssSelector("button[class='submit']"));
        field.click();
    }

    private void clearAndType(WebElement field, String text) {
        field.clear();
        field.sendKeys(text);
    }
}

```

After we apply the pattern to the code of the automatic script it will look like this:

```

public interface LoadableComponent {

    void load();

    void isLoading();
}

```

```

public class LoginPage implements LoadableComponent {

    private final WebDriver driver;

    public LoginPage(WebDriver driver) {
        this.driver = driver;
    }

    public void login(User user) {
        setLogin(user.username);
        setPassword(user.password);
        clickLogin();
    }

    public void setLogin(String login) {
        WebElement field =
driver.findElement(By.cssSelector("input[class='login']"));
        clearAndType(field, login);
    }

    public void setPassword(String password) {
        WebElement field =
driver.findElement(By.cssSelector("input[class='password']"));
        clearAndType(field, password);
    }

    public void clickLogin() {
        WebElement field =
driver.findElement(By.cssSelector("button[class='submit']"));
        field.click();
    }

    private void clearAndType(WebElement field, String text) {
        field.clear();
        field.sendKeys(text);
    }

    @Override
    public void load() {
        driver.get("https://github.com/SeleniumHQ/login");
    }

    @Override
    public void isLoaded() {
        String url = driver.getCurrentUrl();
        assertTrue("Not on the login entry page: " + url, url.endsWith("/login"));
    }
}

```

Strategy pattern.

In Strategy pattern, we create objects which represent various strategies and a context object whose behavior varies as per its strategy object. The strategy object changes the executing algorithm of the context object.

Strategy pattern is used whenever we want to have more than one implementation of the same action/sequence of actions, which is done differently. Depending on the context we could choose the implementation. In the Strategy pattern, we create objects which represent various strategies and a context object whose behavior varies as per its strategy object. The strategy object changes the executing algorithm of the context object.

The main goal of the strategy pattern is the interchangeability of the class family. This template is used to define a class family. Encapsulating each of them and organizing interchangeability.

Let's take this pattern as an example. Let's imagine that we have a developer. This developer has several types of activity. Sleep, code writing, and rest. And the developer somehow changes his activity during the day.

First, let's create the **Activity** interface. The Activity interface has only one public method **justDoIt()**:

```
public interface Activity {  
    public void justDoIt();  
}
```

Next, we will create four implementations of this interface. The **Coding** class will implement the **justDoIt** method and return the string Coding. By analogy with the **Coding** class, we will create the other classes

Reading, Sleeping and Training. All these classes implement the **justDoIt()** method by analogy with their own name.

Coding class:

```
public class Coding implements Activity {  
  
    @Override  
    public void justDoIt() {  
        PrintStream printStream = System.out;  
        printStream.println("Coding.....");  
    }  
}
```

Reading class:

```
public class Reading implements Activity {  
  
    @Override  
    public void justDoIt() {  
        PrintStream printStream = System.out;  
        printStream.println("Reading.....");  
    }  
}
```

Sleeping class:

```
public class Sleeping implements Activity {  
  
    @Override  
    public void justDoIt() {  
        PrintStream printStream = System.out;  
        printStream.println("Sleeping.....");  
    }  
}
```

Training class:

```
public class Training implements Activity {  
  
    @Override  
    public void justDoIt() {  
        PrintStream printStream = System.out;  
        printStream.println("Training....");  
    }  
}
```

Developer class:

```
public class Developer {  
  
    Activity activity;  
  
    public void setActivity(Activity activity) {  
        this.activity = activity;  
    }  
  
    public void executeActivity() {  
        activity.justDoIt();  
    }  
}
```

The entire implementation on our side is ready and now we need to create a client class so that we can test the work of our template.

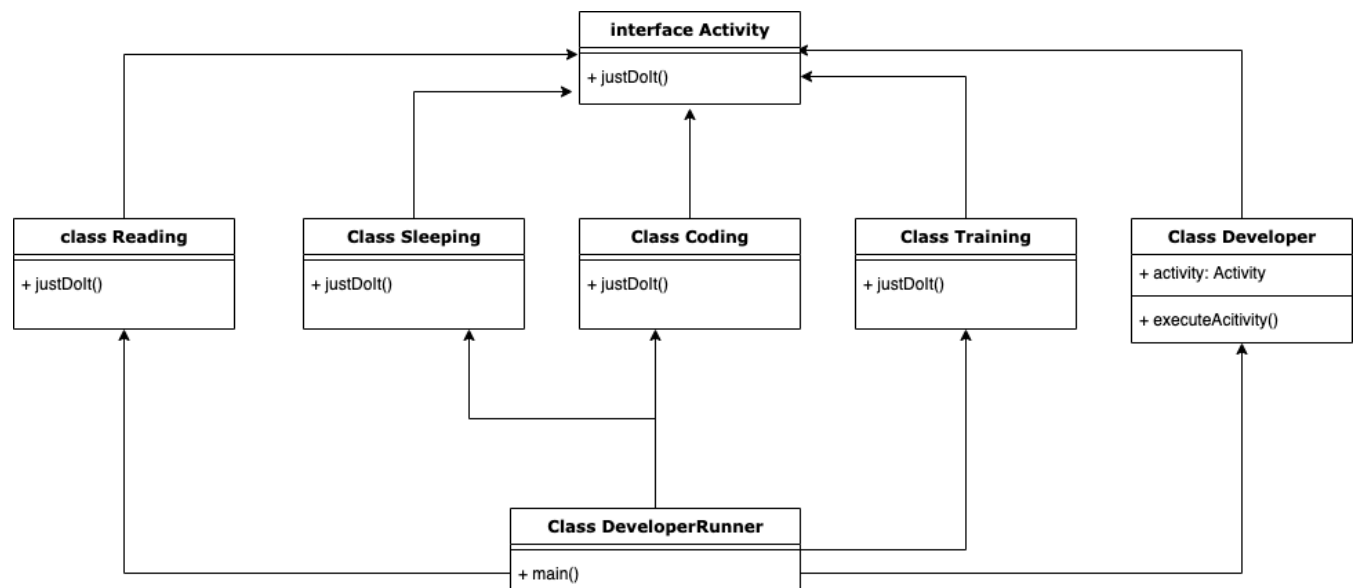
The entire implementation on our side is ready and now we need to create a client class so that we can test the work of our template. Let's call our class DeveloperRunner. In the DeveloperRunner class, we will create a developer. Next, we need to install and execute the activity of our developer:

```
public class DeveloperRunner {  
  
    public static void main(String[] args) {  
        Developer developer = new Developer();  
  
        developer.setActivity(new Training());  
        developer.executeActivity();  
  
        developer.setActivity(new Coding());  
        developer.executeActivity();  
  
        developer.setActivity(new Reading());  
        developer.executeActivity();  
  
        developer.setActivity(new Sleeping());  
        developer.executeActivity();  
    }  
}
```

The output result to the console will be as follows:

```
Training.....  
Coding.....  
Reading.....  
Sleeping.....
```

in the UML diagram, our implementation looks like this:



This pattern can be used, for example, in validation: you need to create an abstract validate method and directly apply this strategy to it. It can also be some kind of computational algorithm: you can use a complex or simple algorithm, but you do not want to directly fit it into your code and get the flexibility to substitute a specific implementation of a specific strategy.

Let's look at an example of implementing the strategy pattern for an example that is close to testing automation. The easiest example is user registration. You might want to have two different implementations of this particular action. The first one would be the actual flow of transitions through the pages for successful user registration. The other one would be a short API call which is invoked when a new user is needed for the test.

The implementation for the Web may look like this:

```
public interface RegistrationStrategy {
    User register(final String username, final String password);
}
```


WebRegister class:

```
public class WebRegister implements RegistrationStrategy {
    WebDriver driver;

    @Override
    public User register(String username, String password) {
        String userName = UserRegistry.getUserName();
        String userPassword = PasswordConstructor.generatePassword();
        new SignInPage()
            .openRegistrationPage()
            .setEmail(userName)
            .setPassword(userPassword)
            .clickLogin();

        return new User(username, userPassword);
    }
}
```

SignInUp class:

```
public class SignInPage {

    WebDriver driver;

    public SignInPage setEmail(final String email) {
        WebElement field =
driver.findElement(By.cssSelector("input[class='email']"));
        enter(field, email);

        return this;
    }

    public SignInPage setPassword(final String password) {
        WebElement field =
driver.findElement(By.cssSelector("input[class='password']"));
        enter(field, password);

        return this;
    }

    public SignInPage clickLogin() {
        WebElement field =
driver.findElement(By.cssSelector("button[class='submit']"));
        field.click();

        return this;
    }

    private void enter(WebElement field, String text) {
        field.clear();
        field.sendKeys(text);
    }

    public SignInPage openRegistrationPage() {
        driver.get("https://login.com/register");

        return this;
    }
}
```

The implementation for the API may look like this:

```
public class ApiRegister implements RegistrationStrategy {  
  
    @Override  
    public User register(String username, String password) {  
        User user = new User(UserRegistry.getUserName(),  
            PasswordConstructor.generatePassword());  
        Assert.assertEquals(user.toString(),  
            put("https://login.com/api/user").getBody().toString());  
  
        return user;  
    }  
}
```

You may not want to invoke “long” registration every time in your tests. But sometimes you’ll need it, for example when you validate actual registration through the web. And vice versa, we want new user creation for test needs to be fast and reliable. That’s why REST registration would be suitable here.

The strategy helps to make our test automation framework more flexible and easier in maintenance by using the separation of concepts. Again you need to be careful and not implement it in situations when you can do fine without it.

Chapter 3. Creational patterns.

In software engineering, creational design patterns are design patterns that deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. The basic form of object creation could result in design problems or added complexity to the design. Creational design patterns solve this problem by somehow controlling this object creation.

- **Factory Method**
Creates an instance of several derived classes
- **Abstract Factory**
Creates an instance of several families of classes
- **Singleton**
A class of which only a single instance can exist

Factory Method Pattern.

The factory design pattern says that define an interface (A java interface or an abstract class) and let the subclasses decide which object to instantiate. The factory method in the interface lets a class defers the instantiation to one or more concrete subclasses. Since these design patterns talk about the instantiation of an object and so it comes under the category of creational design pattern. If we notice the name Factory method, that means there is a method which is a factory, and in general, factories are involved with creational stuff and here with this, an object is being created. It is one of the best ways to create an object where object creation logic is hidden from the client. Now Let's look at the implementation.

- Define a factory method inside an interface.
- Let the subclass implements the above factory method and decides which object to create. In Java, constructors are not polymorphic, but by allowing a subclass to create an object, we are adding polymorphic behavior to the instantiation.

Let's take an example. We have a class **Program** this is a class of client:

```
public class Program {  
    public static void main(String[] args) {  
        JavaDeveloper javaDeveloper = new JavaDeveloper();  
        javaDeveloper.writeJavaCode();  
    }  
}
```

And create a JavaDeveloper class with one method:

```
public class JavaDeveloper {  
    public void writeJavaCode() {  
        System.out.printf("Java Developer write java code ..... ");  
    }  
}
```

After running the program, we will see the execution in the console:

```
Java Developer write java code .....
```

For example, next, we need to create a CppDeveloper

```
public class Program {  
    public static void main(String[] args) {  
        CppDeveloper cppDeveloper = new CppDeveloper();  
        cppDeveloper.writeCppCode();  
    }  
}
```

and

```
public class CppDeveloper {  
    void writeCppCode() {  
        System.out.printf("C++ developer write C++ code .... ");  
    }  
}
```

After running the program, we will see the execution in the console:

```
C++ developer write C++ code ....
```

As we can see our program is working. But if we need to change the programming language or add some methods, we will have to write a lot of the same type of code. We will also have to perform a lot of actions with the client class **Program**. This approach is inconvenient and wrong.

Below we will take a different approach. Creating the Developer interface.

```
public interface Developer {  
    void writeCode();  
}
```

Also, create two implementation classes JavaDeveloper and CppDeveloper classes.

```
public class JavaDeveloper implements Developer {

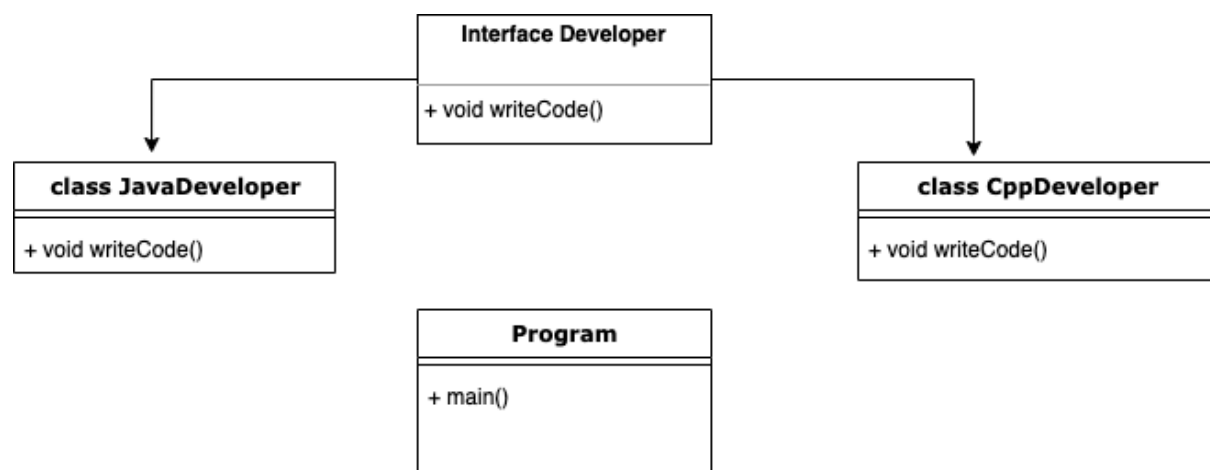
@Override
public void writeCode() {
    System.out.printf("Java Developer write java code ..... ");
}

}
```

and

```
public class CppDeveloper implements Developer {
    @Override    public void writeCode() {
        System.out.printf("C++ developer write C++ code .... ");
    }
}
```

In the form of a diagram, our classes look like this:



We can use this implementation as follows:

```
public class Program {
    public static void main(String[] args) {
        Developer developer = new JavaDeveloper();
        developer.writeCode();
    }
}
```

and

```
public class Program {  
    public static void main(String[] args) {  
        Developer developer = new CppDeveloper();  
        developer.writeCode();  
    }  
}
```

This implementation is much more convenient than the previous implementation. However, we still have to write a lot of duplicate code and create an instance of `JavaDeveloper` or `CppDeveloper`. This approach is not flexible enough. In order to improve this implementation, we will create the `Developer Factory` interface.

```
public interface DeveloperFactory {  
    Developer createDeveloper();  
}
```

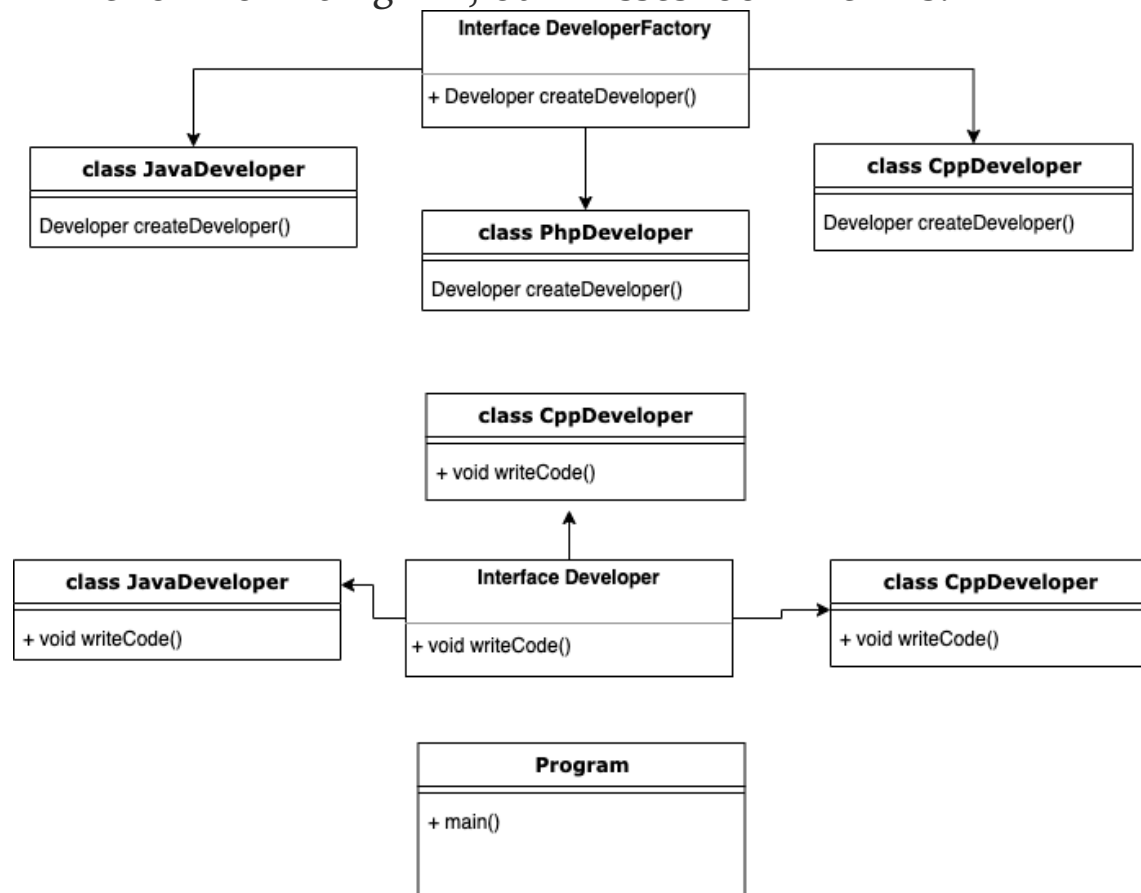
Let's create two implementations of the `Developer Factory` interface:

```
public class JavaDeveloperFactory implements DeveloperFactory {  
    @Override    public Developer createDeveloper() {  
        return new JavaDeveloper();  
    }  
}
```

and

```
public class CppDeveloperFactory implements DeveloperFactory {  
  
    @Override  
    public Developer createDeveloper() {  
        return new CppDeveloper();  
    }  
}
```

In the form of a diagram, our classes look like this:



Now in the client class, we need to register the following implementation.

```

public class Program {
    public static void main(String[] args) {
        DeveloperFactory developerFactory = new JavaDeveloperFactory();
        Developer developer = developerFactory.createDeveloper();
        developer.writeCode();
    }
}
  
```

Java Developer writes Java code

And

```

public class Program {
    public static void main(String[] args) {
        DeveloperFactory developerFactory = new CppDeveloperFactory();
        Developer developer = developerFactory.createDeveloper();
        developer.writeCode();
    }
}
  
```

Cpp Developer writes Cpp code

But we still have to create a developer instance. In order to avoid this, you need to create a more flexible implementation.

Let's create an implementation of a method whose task is to return a specific developer factory depending on the programming language.

```
static DeveloperFactory createDeveloperBySpecialty(final String specialty)
{
    if ("java".equalsIgnoreCase(specialty)) {
        return new JavaDeveloperFactory();
    } else if ("c++".equalsIgnoreCase(specialty)) {
        return new CppDeveloperFactory();
    } else {
        throw new RuntimeException(specialty + "is unknow specialty");
    }
}
```

The final implementation looks like this.

```
public class Program {
    public static void main(String[] args) {
        DeveloperFactory developerFactory = createDeveloperBySpecialty("java");
        Developer developer = developerFactory.createDeveloper();
        developer.writeCode();
    }

    static DeveloperFactory createDeveloperBySpecialty(final String specialty) {
        if ("java".equalsIgnoreCase(specialty)) {
            return new JavaDeveloperFactory();
        } else if ("c++".equalsIgnoreCase(specialty)) {
            return new CppDeveloperFactory();
        } else {
            throw new RuntimeException(specialty + "is unknow specialty");
        }
    }
}
```

Java Developer writes Java code

And

```
public class Program {
    public static void main(String[] args) {
        DeveloperFactory developerFactory =
createDeveloperBySpecialty("c++");
        Developer developer = developerFactory.createDeveloper();
        developer.writeCode();
    }

    static DeveloperFactory createDeveloperBySpecialty(final String
specialty) {
        if ("java".equalsIgnoreCase(specialty)) {
            return new JavaDeveloperFactory();
        } else if ("c++".equalsIgnoreCase(specialty)) {
            return new CppDeveloperFactory();
        } else {
            throw new RuntimeException(specialty + "is unknow specialty");
        }
    }
}
Cpp Developer writes Cpp code ....
```

For example, we need to create an implementation for a PHP developer.

```
public class PhpDeveloper implements Developer {
    @Override
    public void writeCode() {
        System.out.printf("Php Developer writes Php code....");
    }
}

public class PhpDeveloperFactory implements DeveloperFactory {
    @Override
    public Developer createDeveloper() {
        return new PhpDeveloper();
    }
}

public class Program {
    public static void main(String[] args) {
        DeveloperFactory developerFactory = createDeveloperBySpecialty("php");
        Developer developer = developerFactory.createDeveloper();
        developer.writeCode();
    }

    static DeveloperFactory createDeveloperBySpecialty(final String specialty) {
        if ("java".equalsIgnoreCase(specialty)) {
            return new JavaDeveloperFactory();
        } else if ("c++".equalsIgnoreCase(specialty)) {
            return new CppDeveloperFactory();
        } else if ("php".equalsIgnoreCase(specialty)) {
            return new PhpDeveloperFactory();
        } else {
            throw new RuntimeException(specialty + "is unknown specialty");
        }
    }
}
Php Developer writes Php code....
```

As you can see, the implementation has become quite simple and flexible.

But for more flexible and high-quality implementation, we can replace if-else conditions with Map.

```
public class NameOfDeveloper {  
  
    private static final HashMap<String, DeveloperFactory> CODE_COUNTRY = new HashMap<>();  
  
    static {  
        CODE_COUNTRY.put("java", new JavaDeveloperFactory());  
        CODE_COUNTRY.put("cpp", new CppDeveloperFactory());  
        CODE_COUNTRY.put("php", new PhpDeveloperFactory());  
    }  
  
    public static DeveloperFactory returnNameOfDeveloper(final String specialty) {  
        return CODE_COUNTRY.get(specialty);  
    }  
}
```

and after the changes, our code will look like this:

```
public class Program {  
    public static void main(String[] args) {  
        DeveloperFactory developerFactory = NameOfDeveloper.returnNameOfDeveloper("java");  
        Developer developer = developerFactory.createDeveloper();  
        developer.writeCode();  
    }  
}
```

Java Developer writes Java code

Factory Method is a widely used, creational design pattern that can be used in many situations where multiple concrete implementations of an interface exist.

The pattern removes complex logical code that is hard to maintain and replaces it with a design that is reusable and extensible. The pattern avoids modifying existing code to support new requirements.

Abstract Factory pattern.

Abstract Factory patterns work around a super-factory which creates other factories. This factory is also called a factory of factories. This type of design pattern comes under a creational pattern as this pattern provides one of the best ways to create an object.

In the Abstract Factory pattern, an interface is responsible for creating a factory of related objects without explicitly specifying their classes. Each generated factory can give the objects as per the Factory pattern.

The purpose of the abstract factory template is to create an interface that generates a family of interconnected or interdependent objects. At the same time, there is no strict binding to a specific class.

Typically, this template is used to create many interconnected objects. For example, if we do not want our system to depend on the method of creation, layout and representation of the object that is included in it. Or we need to provide a lot of objects but at the same time disclose only their interfaces and not the implementation.

Consider an example. We are creating a WebSite project. To create this project, we need Php Developer who will write code. Also, we need QA Engineer who will do it testing web site. Also, we need a project manager who will manage our project. In order for our project to be created, we need the developer to write the code. QA Engineer tested the site. And the manager managed the project. The implementation looks like this:

```
public class Program {  
  
    public static void main(String[] args) {  
        PrintStream printStream = System.out;  
  
        PhpDeveloper phpDeveloper = new PhpDeveloper();  
        QAEngineer qaEngineer = new QAEngineer();  
        ProjectManager projectManager = new ProjectManager();  
  
        printStream.println("The start create project....");  
        phpDeveloper.writeCode();  
        qaEngineer.testCode();  
        projectManager.manageProject();  
    }  
}
```

Php developer implementation:

```
public class PhpDeveloper {  
  
    public void writeCode() {  
        PrintStream printStream = System.out;  
        printStream.println("Php developer writes Php code..... ");  
    }  
}
```

QA engineer implementation:

```
public class QAEngineer {  
    public void testCode() {  
        PrintStream printStream = System.out;  
        printStream.println("QA engineer testing code .... ");  
    }  
}
```

Project manager implementation:

```
public class ProjectManager {  
    public void manageProject() {  
        PrintStream printStream = System.out;  
        printStream.println("PM manage website project.....");  
    }  
}
```

After launching our implementation, the console will display:

```
The start create project....  
Php developer writes Php code.....  
QA engineer testing code ....  
PM manage website project.....
```

Imagine that our company has a large number of projects and they are constantly changing. Applying the approach, we implemented above is not effective.

Let's try to apply a more effective approach. First, let's create the Developer interface with method writeCode. Second, let's create the QAEngineer interface with method testCode and also let's create the ProjectManager interface with method manageProject.

Developer:

```
public interface Developer {  
    void writeCode();  
}
```

QAEngineer:

```
public interface QAEngineer {  
    void testCode();  
}
```

ProjectManager:

```
public interface ProjectManager {  
    void manageProject();  
}
```

Now we need to create the ability to implement the entire team at once. Not individually for each developer, tester, or product manager. And the whole team. To do this, we need to create an entity **ProjectTeamFactory**. It is this entity that will create the team for our projects. This entity is convenient and effective. Depending on the requirement for an example, we can create several developers, qa engineers or project managers. And change the number of the team depending on the requirements.

```
public interface ProjectTeamFactory {  
  
    Developer getDeveloper();  
  
    QAEngineer getQAEngineer();  
  
    ProjectManager getProjectManager();  
}
```

For example, imagine that we need a ready-made team to create an Internet banking system.

To do this, we will create a **JavaDeveloper** class that will inherit from the **Developer** interface. Create a **QATester** class that will inherit from the **QAEngineer** interface and create a **BankingPM** class that will inherit from the **ProjectManager** interface.

JavaDeveloper:

```
public class JavaDeveloper implements Developer {  
  
    @Override  
    public void writeCode() {  
        PrintStream printStream = System.out;  
        printStream.println("Java developer writes Java code..... ");  
    }  
}
```

QATester:

```
public class QATester implements QAEngineer {  
  
    @Override  
    public void testCode() {  
        PrintStream printStream = System.out;  
        printStream.println("QA tester test banking code.....");  
    }  
}
```

BakingPM:

```
public class BankingPM implements ProjectManager {  
  
    @Override  
    public void manageProject() {  
        PrintStream printStream = System.out;  
        printStream.println("Banking PM manages banking project .....");  
    }  
}
```

Now we need to create a factory for our team so that our team is created immediately and as a whole. Now we need to create a factory for our team so that our team is created immediately and as a whole. One developer. One tester. One product manager. To do this, we create the **BankingTeamFactory** class. This class will implement the **ProjectTeamFactory** interface. And redefine the methods as follows. The **getDeveloper** method will return a new **JavaDeveloper**. The **getTester** method will return a new **QATester**. And the **getProjectManager** method will return the new **BankingPM**.

```

public class BankingTeamFactory implements ProjectTeamFactory {

    @Override
    public Developer getDeveloper() {
        return new JavaDeveloper();
    }

    @Override
    public QAEngineer getQAEngineer() {
        return new QATester();
    }

    @Override
    public ProjectManager getProjectManager() {
        return new BankingPM();
    }

}

```

Now let's say we needed to create a website project. Let's say we needed a ready made team that can easily develop web sites.

We need to create a **PhpDeveloper** class that will implement the **Developer** interface. We will also create a **ManualTester** class that will implement the **QAEngineer** interface and a **WebSitePM** class that will implement the **ProjectManager** interface

PhpDeveloper:

```

public class PhpDeveloper implements Developer {

    @Override
    public void writeCode() {
        PrintStream printStream = System.out;
        printStream.println("Php Developer writes Php code ....");
    }

}

```

ManualTester:

```

public class ManualTester implements QAEngineer {

    @Override
    public void testCode() {
        PrintStream printStream = System.out;
        printStream.println("Manual tester tests code..... ");
    }

}

```

WebSitePM:

```

public class WebSitePM implements ProjectManager {

    @Override
    public void manageProject() {
        PrintStream printStream = System.out;
        printStream.println("WebSite PM manages website project ..... ");
    }

}

```


Now we need to create the **WebSiteTeamFactory** class by analogy with the bank project. This class will implement **ProjectTeamFactory**. And the methods of this class will return the new **PhpDeveloper**. New **ManualTester** and new **WebSitePM**.

```
public class WebSiteTeamFactory implements ProjectTeamFactory {

    @Override
    public Developer getDeveloper() {
        return new PhpDeveloper();
    }

    @Override
    public QAEngineer getQAEngineer() {
        return new ManualTester();
    }

    @Override
    public ProjectManager getProjectManager() {
        return new WebSitePM();
    }
}
```

We have two teams. One command is for websites and the other is for the banking system. Suppose we need to implement two more projects. The first project is an auction site and let's call it **AuctionSiteProject**. The second project is for the super banking system **SuperBankSystem**.

To begin with, we will implement the banking system. To do this you must first create a **ProjectTeamFactory** factory that will return new **BankingTeamFactory**. Then we get it through the corresponding interfaces of the developer, QA engineer and product manager.

Now we need to check our implementation. The developer must call the **writeCode** method. You must call the **testCode** method for the **QAEngineer** and the **manageProject** method for the product manager.

```

public class SuperBankSystem {

    public static void main(String[] args) {
        PrintStream printStream = System.out;
        ProjectTeamFactory projectTeamFactory = new BankingTeamFactory();
        Developer developer = projectTeamFactory.getDeveloper();
        QAEngineer qaEngineer = projectTeamFactory.getQAEngineer();
        ProjectManager projectManager = projectTeamFactory.getProjectManager();

        printStream.println("Creating bank system ..... ");

        developer.writeCode();
        qaEngineer.testCode();
        projectManager.manageProject();
    }
}

```

In the console, our program will output:

```

Creating bank system .....
Java developer writes Java code.....
QA tester test banking code.....
Banking PM manages banking project .....

```

Now, by analogy with **SuperBankSystem**, we can easily write an implementation for **AuctionSiteProject**.

```

public class AuctionSiteProject {
    public static void main(String[] args) {
        PrintStream printStream = System.out;
        ProjectTeamFactory projectTeamFactory = new WebSiteTeamFactory();
        Developer developer = projectTeamFactory.getDeveloper();
        QAEngineer qaEngineer = projectTeamFactory.getQAEngineer();
        ProjectManager projectManager = projectTeamFactory.getProjectManager();

        printStream.println("Creating auction system ..... ");

        developer.writeCode();
        qaEngineer.testCode();
        projectManager.manageProject();
    }
}

```

In the console, our program will output:

```

Creating auction system .....
Php Developer writes Php code ....
Manual tester tests code.....
WebSite PM manages website project .....

```

Our implementation is fully ready, working, and flexible enough. Now, with this implementation, we can add any developer to any team. And a team can create a specific project.

I want to draw your attention to the fact that an abstract factory can be not only an abstract class, but also an interface and a concrete class. Specific factories most often implement the Singleton pattern.

Use the Abstract Factory pattern in such cases:

- The system should not depend on the way objects are created.
- The system works with one of several object families.
- Objects within a family are interconnected.

Singleton pattern.

Singleton pattern is one of the simplest design patterns in Java. This type of design pattern comes under a creational pattern as this pattern provides one of the best ways to create an object.

This pattern involves a single class that is responsible to create an object while making sure that only a single object gets created. This class provides a way to access its only object which can be accessed directly without the need to instantiate the object of the class.

To understand the pattern, let's look at its implementation using an example. Let's say we have some program and we would like all actions in this program to be logged. We need to create a primitive logger and call it **ProgramLogger**.

First, we need to create an instance of the **ProgramLogger** class. And create a string with the name **logFile**. In this line we will record all the actions performed in our program.

Now we need to create a public static method that will return us an instance of **programLogger**. In this method, we will check the **programLogger** instance for null and create a new instance. We will create a private empty constructor and two methods. One for adding information to our **addLogInfo** log. And the second one is for displaying information from our **showLogInfo** log.

The code for implementing the Singleton pattern looks like this:

```
public class ProgramLogger {

    private static ProgramLogger programLogger;

    private static String logFile = "Now, the log file is empty.\n \n";

    public static ProgramLogger getProgramLogger() {
        if (programLogger == null) {
            programLogger = new ProgramLogger();
        }
        return programLogger;
    }

    private ProgramLogger() {

    }

    public void addLogInfo(String logInfo) {
        logFile += logInfo + "\n";
    }

    public void showLogInfo() {
        PrintStream printStream = System.out;
        printStream.println(logFile);
    }

}
```

The implementation example above is only suitable for a single-threaded environment. But if, for example, another thread wants to access our **getProgramLogger** method. The result can be fatal. To avoid problems with threads we need to make the **getProgramLogger** is a synchronized method. In other words, if one thread accesses our **getProgramLogger** method, it will need to wait until the first thread finishes working with it.

To check that our implementation is working correctly. We need to create a **ProgramRunner** class and call our method several times and make sure that regardless of the number of calls, it will be the same file. That is, the hash code of our file will be the same everywhere.

```
public class ProgramRunner {

    public static void main(String[] args) {
        System.out.println(ProgramLogger.getProgramLogger().toString());
        System.out.println(ProgramLogger.getProgramLogger().toString());
        System.out.println(ProgramLogger.getProgramLogger().toString());
        System.out.println(ProgramLogger.getProgramLogger().toString());
        System.out.println(ProgramLogger.getProgramLogger().toString());
    }

}
```

The output result to the console will be as follows:

```
singleton.ProgramLogger@1f32e575  
singleton.ProgramLogger@1f32e575  
singleton.ProgramLogger@1f32e575  
singleton.ProgramLogger@1f32e575  
singleton.ProgramLogger@1f32e575
```

Now I will give an example of how you can apply the pattern Singleton in the field of test automation.

Let's say we are creating a framework for automating an application. And we need to always have one instance of our **WebDriver**. In this case, the implementation of our task using the Singleton pattern will look like this.

```
public class SeleniumWebDriver {  
    private static WebDriver driver;  
  
    private SeleniumWebDriver() {  
    }  
  
    public static WebDriver getInstance() {  
        if (driver == null) {  
            synchronized (SeleniumWebDriver.class) {  
                if (driver == null) {  
                    driver = new ChromeDriver();  
                }  
            }  
        }  
  
        return driver;  
    }  
}
```

but personally, I would try not to get carried away with this implementation and would use:

```
private static final ThreadLocal<WebDriver> DRIVER_THREAD = new ThreadLocal<>();
```

Chapter 4. Data Patterns.

We would like to pull data and data management away from the test logic as far as possible, in order to remove the amount of so-called boilerplate code from our tests. This will make the logic even more transparent and easier to maintain for those who write these automation scripts.

- Value Object
- Builder
- Assert Object/Matchers
- Data Registry
- Object Pool/Flyweight
- Data Provider

The main goal of data patterns is to split data and test logic as well as to reduce boilerplate and code duplication in our tests. It should make them more understandable and easier in maintenance for anyone who works with them.

Value Object pattern.

This pattern very simple, I think you all used it, but to my great regret, I have seen many projects where this is neglected, and in the end, it turns out to be very difficult. That's really sad because from a design perspective Value Object can make your code more readable and significantly reduce the number of repeatable constructions. Why do we need to use the pattern of the Value Object? It is Immutable: once it is created, it cannot be changed, because this is its task, it serves to transfer data from point A to point B, and not to be modifiable or carry third-party effects.

```
public class ValueObject {  
  
    public void createUser(String firstName,  
                           String lastName,  
                           int age,  
                           boolean isMarried,  
                           List<String> accomplishments) {  
  
        enter(firstName, into("name"));  
        enter(lastName, into("lastName"));  
        enter(age, into("age"));  
        enterMaritalStatus(isMarried);  
        accomplishments.forEach(this::addAccomplishment);  
    }  
}
```


We will convert this code:

```
public class User {

    private String firstName;

    private String lastName;

    private String age;

    private boolean isMarried;
    private List<String> accomplishments;

    public User(String firstName,
                String lastName,
                String age,
                boolean isMarried,
                List<String> accomplishments) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
        this.isMarried = isMarried;
        this.accomplishments = accomplishments;
    }

    public String getFirstName() {
        return firstName;
    }

    public String getLastName() {
        return lastName;
    }

    public String getAge() {
        return age;
    }

    public boolean isMarried() {
        return isMarried;
    }

    public List<String> getAccomplishments() {
        return accomplishments;
    }

}
```

And use:

```
public class ValueObject {

    public void createUser(User user) {
        enter(user.firstName, into("name"));
        enter(user.lastName, into("lastName"));
        enter(user.age, into("age"));
        enterMaritalStatus(user.isMarried);
        user.accomplishments.forEach(this::addAccomplishment);
    }

}
```

I'd explain the pattern this way. If we have multiple objects which have some common logic (in the case above **createUser** accepts five parameters — first name, last name, age, marital status, and accomplishments), it's better to merge them into one entity. In this case, the User will be our Value Object, which aggregates all needed information about the actual users into it.

To ease the process of creation of such objects you could use additional libraries, like **Lombok** in case you're working with Java, this tool will create needed constructors, generate getters for all fields, and finalize them afterward without any manual work from your side.

Lombok converts annotations in source code to Java statements before the compiler processes them:

```
@Getter
public class User {

    private String firstName;

    private String lastName;

    private String age;

    private boolean isMarried;

    private List<String> accomplishments;

    public User(String firstName,
                String lastName,
                String age,
                boolean isMarried,
                List<String> accomplishments) {
        this.firstName = firstName;
        this.lastName = lastName;
        this.age = age;
        this.isMarried = isMarried;
        this.accomplishments = accomplishments;
    }
}
```

By adding the **@Getter** annotation we told Lombok to, well, generate these for all the fields of the class. Also, you may be adding the **@Data** annotation. And this annotation told Lombok to, well, generate Getter/Setter and Constructor for all the fields of the class.

```
@Data
public class User {

    private String firstName;

    private String lastName;

    private String age;

    private boolean isMarried;

    private List<String> accomplishments;
}
```

The Value Objects pattern transforms values in our projects into real objects, giving us more type safety, hiding implementation, and giving a home to all related logic. That being said, we should always evaluate if the mentioned benefits outweigh the drawbacks of creating extra classes, which, in Java, implies extra source files and a rapidly growing size of the project.

Builder pattern.

Builder pattern builds a complex object using simple objects and using a step-by-step approach. This type of design pattern comes under a creational pattern as this pattern provides one of the best ways to create an object.

A Builder class builds the final object step by step. This builder is independent of other objects.

Let's imagine we have an extremely complex object and they can be configured in many different ways. The first option which comes to mind is to create as many types of constructors as we have parameter variations in this object and every time, we meet a new one we'd need to create a new constructor for that. As a result, we'll get a crazy number of constructors which will definitely confuse the end-user of the framework. The builder pattern is used to make the process of building such objects easier and with the help of modern IDE hints more intuitive.

The purpose of the Builder pattern is to separate the construction of a complex object from its representation so that we can get different representations as a result of the same construction.

Let's take an example of how the Builder pattern works. Suppose we want to create a **WebsiteBuilder**.

WebSiteBuilder will contain such fields as the site name, CMS system type, and price. We also need to be able to display our website data.

WebSiteBuilder class:

```
public class WebSiteBuilder {
    public static void main(String[] args) {
        PrintStream printStream = System.out;
        WebSite webSite = new WebSite();

        webSite.setName("Visit card");
        webSite.setCms("WordPress");
        webSite.setPrice(500);

        printStream.println(webSite);
    }
}
```

WebSite class:

```
public class WebSite {
    private String name;
    private String cms;
    private int price;

    public void setName(String name) {
        this.name = name;
    }

    public void setCms(String cms) {
        this.cms = cms;
    }

    public void setPrice(int price) {
        this.price = price;
    }

    @Override
    public String toString() {
        return "WebSite{" +
            "name='" + name + '\'' +
            ", cms='" + cms + '\'' +
            ", price=" + price +
            '}';
    }
}
```

After running our implementation in the console, we will get:

```
WebSite{name='Visit card', cms='WordPress', price=500}
```

Now our task becomes more complicated. Let's assume that we need to create various web sites. It is not possible to use our implementation in the future. If we start using our implementation, it will generate a lot of duplicate code and overhead. Therefore, it is worth taking a different approach.

Creating a new **WebSite** class. In which we will add our old fields name, cms, price. The cms field will be implemented as an Enum.

WebSite class:

```
public class WebSite {
    private String name;
    private CMS cms;
    private int price;

    public void setName(String name) {
        this.name = name;
    }

    public void setCms(CMS cms) {
        this.cms = cms;
    }

    public void setPrice(int price) {
        this.price = price;
    }

    @Override
    public String toString() {
        return "WebSite{" +
            "name='" + name + '\'' +
            ", cms=" + cms +
            ", price=" + price +
            '}';
    }
}
```

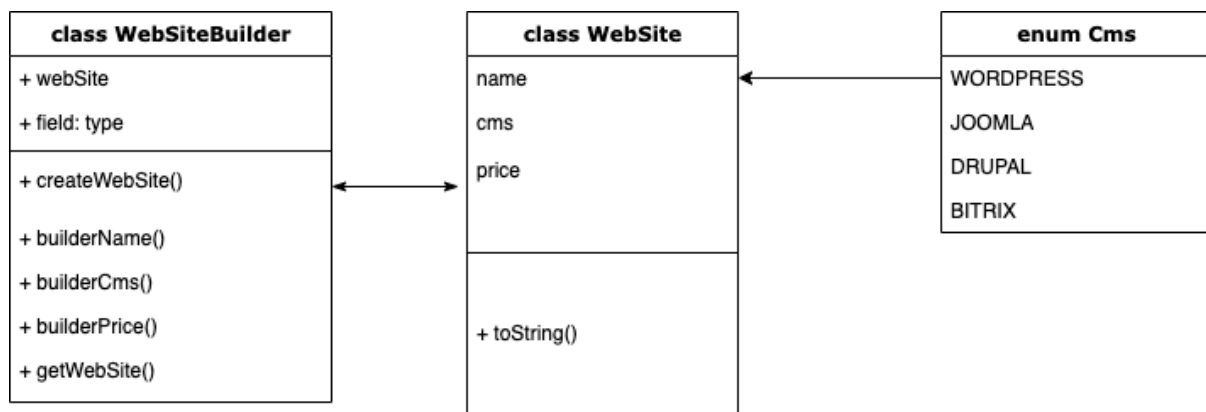
CMS enum:

```
public enum CMS {
    WORDPRESS, JOOMLA, DRUPAL, BITRIX;
}
```

Now we need to create a certain entity with which we will control our sites. To do this, we need to create an abstract class. And let's call it **WebSiteBuilder**. The main task of this class will be to create or design sites for the needs of the client program. In the abstract **WebSiteBuilder** class, we implement the website field. Creating the **createWebSite()** method We also implement the abstract methods **buildName**, **buildCms**, **buildPrice** inside the class. In order to display the information, we need to add the **getWebSite** method. The implementation of this abstract **WebSiteBuilder** class will look like this:

```
public abstract class WebSiteBuilder {  
  
    WebSite webSite = new WebSite();  
  
    public void createWebSite() {  
        webSite = new WebSite();  
    }  
  
    abstract void buildName();  
  
    abstract void buildCms();  
  
    abstract void buildPrice();  
  
    WebSite getWebSite() {  
        return webSite;  
    }  
}
```

Let's see how our implementation looks on the UML diagram:



Now we need to create an implementation of the abstract **WebSiteBuilder** class. To do this, we need to create two additional **VisitCardWebSiteBuilder** classes that will inherit from **WebSiteBuilder**:

```
public class VisitCardWebSiteBuilder extends WebSiteBuilder {

    @Override
    void buildName() {
        webSite.setName("VisitCard");
    }

    @Override
    void buildCms() {
        webSite.setCms(CMS.WORDPRESS);
    }

    @Override
    void buildPrice() {
        webSite.setPrice(500);
    }
}
```

By analogy with **VisitCardWebSiteBuilder**, we will create another class **EnterpriseWebSiteBuilder**, which will serve as an implementation for corporate sites:

```
public class EnterpriseWebSiteBuilder extends WebSiteBuilder {

    @Override
    void buildName() {
        webSite.setName("Enterprise web site");
    }

    @Override
    void buildCms() {
        webSite.setCms(CMS.BITRIX);
    }

    @Override
    void buildPrice() {
        webSite.setPrice(1000);
    }
}
```


Now we need to create the Director class. The main purpose of this class is to be a kind of link and reduce the amount of work in our final client class. It is the Director class that will determine which WebSite we will create. Inside the Director class, we will create the webSiteBuilder field. Setter for the webSiteBuilder field and create the buildWebSite() method. The task of the buildWebSite() method is to return the finished website. The implementation of this class looks like this:

```
public class Director {  
  
    WebSiteBuilder webSiteBuilder;  
  
    public void setWebSiteBuilder(WebSiteBuilder webSiteBuilder) {  
        this.webSiteBuilder = webSiteBuilder;  
    }  
  
    WebSite buildWebSite() {  
        webSiteBuilder.createWebSite();  
        webSiteBuilder.buildName();  
        webSiteBuilder.buildCms();  
        webSiteBuilder.buildPrice();  
  
        WebSite webSite = webSiteBuilder.getWebSite();  
  
        return webSite;  
    }  
}
```

Now we need to create the client class. Let's call this class **BuildWebSiteRunner** for visit card web site:

```
public class BuildWebSiteRunner {  
  
    public static void main(String[] args) {  
        PrintStream printStream = System.out;  
  
        Director director = new Director();  
        director.setWebSiteBuilder(new VisitCardWebSiteBuilder());  
        WebSite webSite = director.buildWebSite();  
  
        printStream.println(webSite);  
    }  
}
```

The following information will be displayed in the console:

```
WebSite{name='VisitCard', cms=WORDPRESS, price=500}
```

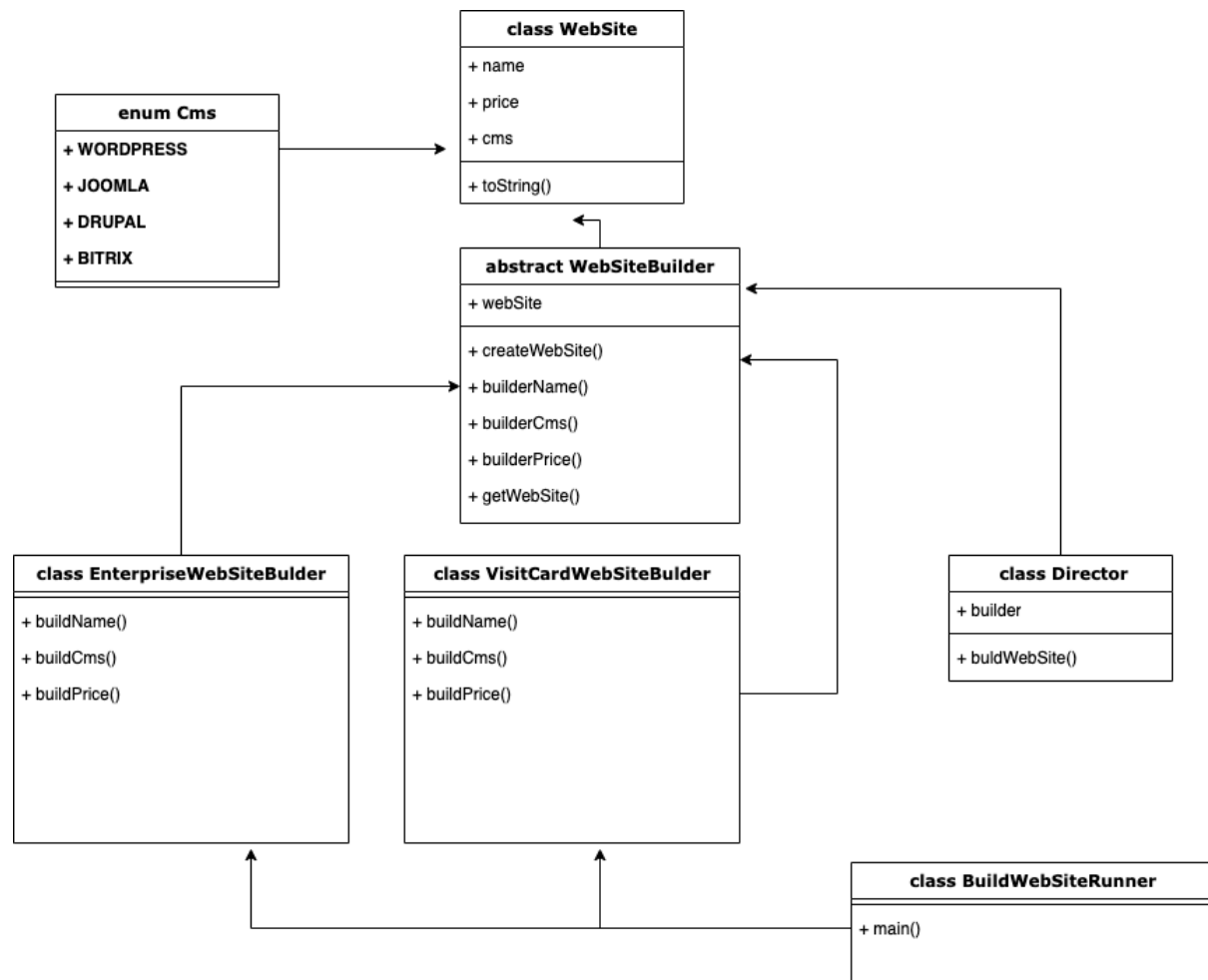
BuildWebSiteRunner for enterprise web site:

```
public class BuildWebSiteRunner {  
  
    public static void main(String[] args) {  
        PrintStream printStream = System.out;  
  
        Director director = new Director();  
        director.setWebSiteBuilder(new EnterpriseWebSiteBuilder());  
        WebSite webSite = director.buildWebSite();  
  
        printStream.println(webSite);  
    }  
}
```

The following information will be displayed in the console:

```
WebSite{name='Enterprise web site', cms=BITRIX, price=1000}
```

Let's see how our implementation looks on the UML diagram:



The Builder pattern in test automation is well used to describe authorization to the server. Below is an example of such an implementation:

```
public class Builder {  
  
    public void createBuilder() {  
        Server server = Server.runtimeBuilder()  
            .withUrl("http://localhost")  
            .withPort(1234)  
            .withoutLogging()  
            .withResponsesEnqueued(new Response())  
            .build();  
    }  
}
```

Under the hood it would look something like this:

```
public class Server {  
    private final String url;  
    private final int port;  
    private final List<Response> responseQueue;  
    private final boolean isLoggingEnabled;  
  
    private Server(String url, int port, List<Response> responseQueue,  
boolean isLoggingEnabled) {  
        this.url = url;  
        this.port = port;  
        this.responseQueue = responseQueue;  
        this.isLoggingEnabled = isLoggingEnabled;  
    }  
  
    public static ServerBuilder runtimeBuilder() {  
        return new ServerBuilder();  
    }  
  
    public static class ServerBuilder {  
        private String url = "http://10.0.0.0";  
        private int port = 6767;  
        private final List<Response> responseQueue = new ArrayList<>();  
        private boolean isLoggingEnabled = true;  
  
        public Server build() {  
            return new Server(url, port, responseQueue, isLoggingEnabled);  
        }  
  
        public ServerBuilder withUrl(String url) {  
            this.url = url;  
            return this;  
        }  
  
        public ServerBuilder withPort(int port) {  
            this.port = port;  
            return this;  
        }  
    }  
}
```

```
public ServerBuilder withResponsesEnqueued(Response... responses) {
    this.responseQueue.clear();
    for (Response response : responses) {
        this.responseQueue.add(response);
    }
    return this;
}

public ServerBuilder withoutLogging() {
    this.isLoggingEnabled = false;
    return this;
}
}
```

In the example above all the parameters have default values when ServerBuilder is created, which means the user can create the Server object without having to set up anything. This might be changed as well if we want our object to have mandatory configurable fields. In that case, I'd throw an exception saying that the object is not fully configured.

Assert Object/Matchers pattern.

The majority of the people heard about the next pattern but I've seen only a few actually use it. Its name is "Assert Object" or simply Matcher. Usually, it might be used whenever we need to do domain-specific assertions on some object. Let's take a look at the example below:

```
@Test
public void onlyOneResponseWithErrorCodeShouldBeReturned() {
    List<Response> responses = server.getResponses();
    assertEquals(1, responses.size());
    Response response = responses.get(0);
    assertEquals(ResponseCode.ERROR_403, response.getCode());
}
```

In the original example, we do not see the obvious logic of the checks in the test. It might be not clear to the user what exactly is going to be verified. At first, we're validating that server returns only one response and then that the response has a **403-error** code. We have to split these checks into two separate assertions because otherwise, it would be difficult to understand what went wrong. But for someone not familiar with our domain it would be still difficult to understand tests like that.

Matcher pattern helps us to create asserts as reusable constructions, which reduce overall code duplication. For instance, if we need to verify that server returns one response, but with 200 response codes.

```
@Test
public void onlyOneResponseWithErrorCodeShouldBeReturned() {
    assertThat(server).hadSingleResponseWithCode(
        ResponseCode.ERROR_403);
}

public static ServerAssert assertThat(Server server) {
    return new ServerAssert(server);
}

public class ServerAssert {
    private final Server server;

    public ServerAssert(Server server) {
        this.server = server;
    }

    public void hadSingleResponseWithCode(ResponseCode
responseCode) {
        List<Response> responses = server.getResponses();
        assertEquals(1, responses.size());
        Response response = responses.get(0);
        assertEquals(responseCode, response.getCode());
    }
}
```

Besides that this approach creates domain logic in our tests. That's why I prefer to implement it the way shown above when we create a group of asserts and put them together in a separate class (e.g. `ServerAssert`), which is responsible for all possible checks on the `Server` object. Then all we need to do is to create a static method `assertThat` accepting the `Server` object and returning `ServerAssert` instead. It looks great and can be read much easier than before and the assertion code underneath stays the same.

*Another option for creating matchers is to build static methods for each of them. There are multiple libraries out there that already have a lot of ready-to-use matchers bundled in and provide easy-to-use API for creating your own ones. The most popular of them are **Hamcrest** and **AssertJ**. If you haven't used them before I suggest at least pay attention to those ones and think about building them into your framework. Again, they won't have your domain-specific Matchers, but they might significantly simplify creating ones.*

For the Java programming language, there are two most popular libraries:

- **Hamcrest** is a framework for writing matcher objects allowing 'match' rules to be defined declaratively. There are a number of situations where matchers are invaluable, such as UI validation or data filtering, but it is in the area of writing flexible tests that matchers are most commonly used.
- **AssertJ** is a java library providing a rich set of assertions, truly helpful error messages, improves test code readability, and is designed to be super easy to use within your favorite IDE.

```
public class HamcrestExample {

    public void createSomeHamcrestExample() {
        assertThat(theBiscuit, equalTo(myBiscuit));
        assertThat("chocolate chips", theBiscuit.getChocolateChipCount(),
equalTo(10));
        assertThat("hazelnuts", theBiscuit.getHazelnutCount(), equalTo(3));
    }
}
```

```
public class AssertJExample {

    public void createSomeAssertJExample() {
        assertThat(frodo.getName()).isEqualTo("Frodo");
        assertThat(frodo).isNotEqualTo(sauron)
            .isin(fellowshipOfTheRing);

        assertThat(fellowshipOfTheRing).hasSize(9)
            .contains(frodo, sam)
            .doesNotContain(sauron);

        assertThat(ringBearers).hasSize(4)
            .contains(entry(oneRing, frodo), entry(nenya, galadriel))
            .doesNotContainEntry(oneRing, aragorn);
    }
}
```

Assertj Fluent Assertions – this library makes several significant improvements.

```
assertThat(list).contains("element");

assertThat(list).hasSize(9);

assertThat(list).containsExactlyInAnyOrder("one", "two", "three");

assertThat(list)
    .contains("element")
    .hasSize(9)
    .containsExactlyInAnyOrder("one", "two", "three");
```

First, the order changes before there were an **assertThat** expected value and then a parameter, then using this library you say **assertThat** parameter and expected value. Secondly, you save space thanks to the library syntax, you write **assertThat** only once and can-do **n-th** number of checks on the same variable.

This library also allows you to create your own checks.

- Simple and readable.
- To give clear error messages.
- Expandable.

Data Registry pattern.

The Data registry pattern is an interesting one. The main approach is as follows: we want our tests to be independent and try to split our test data across them, but as a result, we get completely opposite. For example, test A uses user1, user2, and user3 and they are hardcoded as test data. This might be the problem since we want completely independent tests, right? But we force another test to be aware that user1, user2, and user3 are already occupied by test A. Another concern is that developers not familiar with this could use those users in other tests and this may cause issues. Data registry allows us to generate unique data and avoid duplications. In the example below I'm using the simplest approach possible: on every **getUser** invocation, the static thread-safe counter will be incremented by 1, guaranteeing that each time the unique user is created.

```
public class ResponseTest {  
  
    public static class UserRegistry {  
        private static AtomicInteger COUNTER = new AtomicInteger(0);  
  
        public UserRegistry() {  
            int index = COUNTER.incrementAndGet();  
            return new User("User_" + index, index);  
        }  
    }  
}
```

In your case, the pattern logic might be much more complex, e.g. the registry could take the user from a database, file, predefined data set, etc. But the outcome will be the same: your tests will be truly independent, since each time they use UserRegistry they get exclusive users avoiding test interception issues.

Object Pool / Flyweight pattern.

The **Object Pool** or **Flyweight** pattern is used by even fewer developers. Flyweight is the classic pattern from the **GangOfFour** book, which solves the problem of retaining and operating with the heavy in terms of resources objects or set of objects. Instead of creating them every time we need them, we take them, use them and return them to the so-called Pool for future uses.

The goal of the Flyweight pattern is to support many small objects. A good example of using this template is when a large number of objects are used. And most of the state of objects can be taken out, or applications do not depend on the identity of the object.

Let's look at an example. Let's say we have a developer. We implement it through the Developer interface. This interface has only one method, **writeCode()**

```
public interface Developer {  
  
    void writeCode();  
}
```

And we will have two implementations of this interface. **JavaDeveloper** class and **CppDeveloper** class. The implementation of these classes looks like this.

JavaDeveloper class:

```
public class JavaDeveloper implements Developer {  
  
    @Override  
    public void writeCode() {  
        PrintStream printStream = System.out;  
        printStream.println("Java developer writes Java code.....");  
    }  
}
```

CppDeveloper class:

```
public class CppDeveloper implements Developer {  
  
    @Override  
    public void writeCode() {  
        PrintStream printStream = System.out;  
        printStream.println("Cpp developer writes code....");  
    }  
}
```

Let's imagine a situation where we need to use a large number of developers many times. To implement this approach, we need to create the ProjectRunner client class. We will also create the Flyweight class. And let's call it DeveloperFactory. The DeveloperFactory class will contain a Map by line and by the Developer interface:

Map<String, Developer>

Also in this class, we implement the **getDeveloperBySpecialty** method. This method will return us the developer by their specialization.

The implementation of this class looks like this:

```
public class DeveloperFactory {  
    PrintStream printStream = System.out;  
    public static final Map<String, Developer> developers = new HashMap<>();  
  
    public Developer getDeveloperBySpecialty(final String speciality) {  
        Developer developer = developers.get(speciality);  
  
        if (developer == null) {  
            switch (speciality) {  
                case "java":  
                    printStream.println("Hiring Java developer.....");  
                    developer = new JavaDeveloper();  
                    break;  
                case "c++":  
                    printStream.println("Hiring C++ developer .....");  
                    developer = new CppDeveloper();  
                    break;  
            }  
            developers.put(speciality, developer);  
        }  
        return developer;  
    }  
}
```

Next, we will write an implementation of the client code that will look like this:

```
public class ProjectRunner {
    public static void main(String[] args) {
        DeveloperFactory developerFactory = new DeveloperFactory();
        List<Developer> developers = new ArrayList<>();

        developers.add(developerFactory.getDeveloperBySpecialty("java"));
        developers.add(developerFactory.getDeveloperBySpecialty("java"));
        developers.add(developerFactory.getDeveloperBySpecialty("java"));

        developers.add(developerFactory.getDeveloperBySpecialty("c++"));
        developers.add(developerFactory.getDeveloperBySpecialty("c++"));
        developers.add(developerFactory.getDeveloperBySpecialty("c++"));

        for (Developer developer : developers) {
            developer.writeCode();
        }
    }
}
```

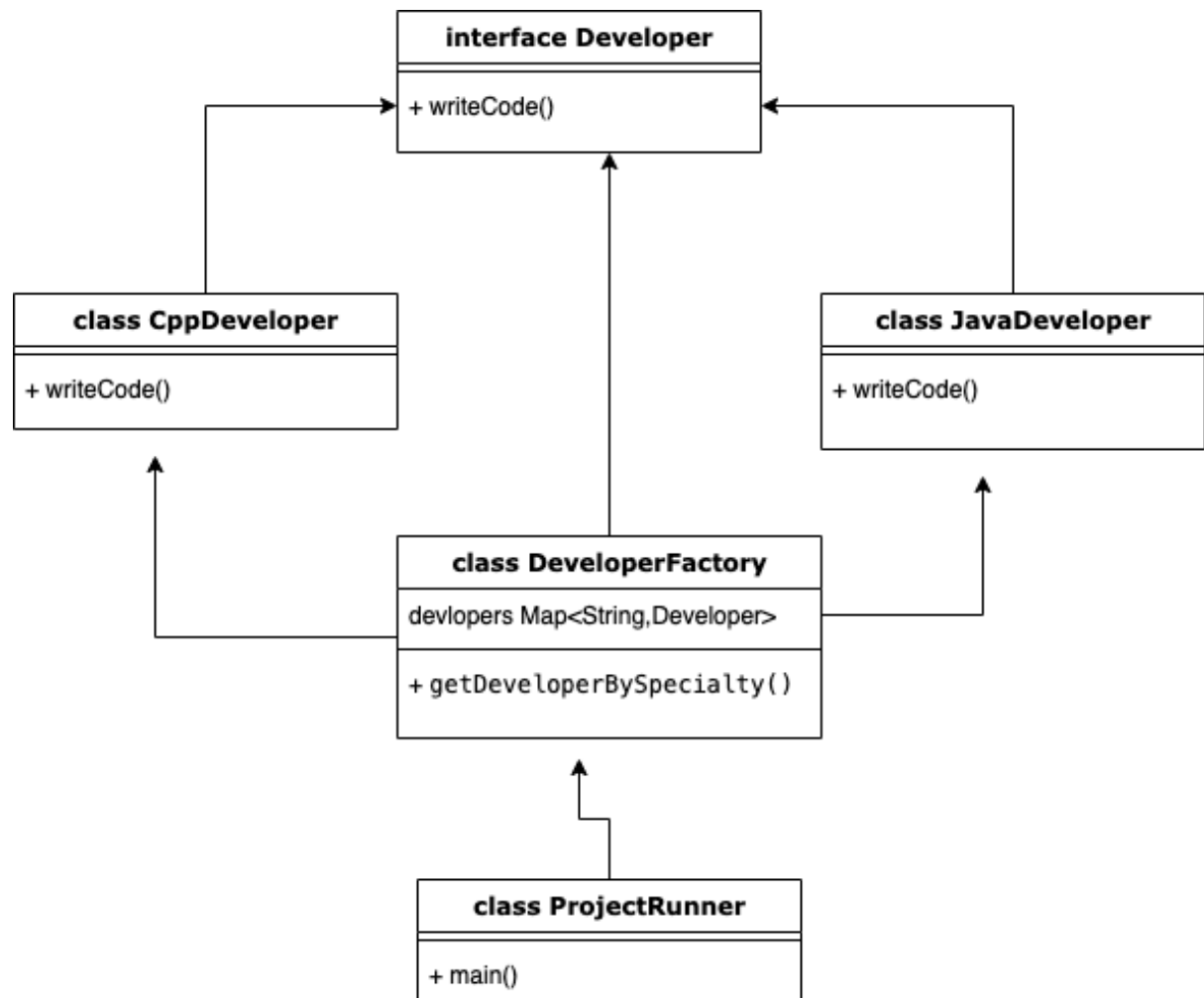
The following information will be displayed in the console:

```
Hiring Java developer.....
Hiring C++ developer .....
Java developer writes Java code.....
Java developer writes Java code.....
Java developer writes Java code.....
C++ developer writes code....
C++ developer writes code....
C++ developer writes code....
```

As we can see in the console. We use many times the developers. But we only hire them twice:

```
Hiring Java developer.....
Hiring C++ developer .....
```

Let's see what our implementation looks like on a UML diagram:



Using this pattern we could implement lots of interesting things, for instance, browser pool. I heard quite a few complaints from different people that web tests take crazy amounts of time because they require to start the browser, load the first page, import user profiles, etc. But it's not necessary to create a browser in the actual test, instead, we could use Background Pool, which set up to retain the needed amount of "hot" browsers. After we're done with the browser we just return it to the pool and clear its data. And this might be done in the background, in parallel with the actual test threads. And only after the browser is ready to be used again it can be given back to the next test as a fresh instance.

That said, this pool configuration and browser setup parts might be taken out outside of the test, significantly minimizing the time and resources spent on it.

The other example is page usage. You don't have to wait until the needed page is opened if all of the tests start from the same page. You could have the Page Pool as well and request it from there. This means it's gonna be opened in one of the browser instances in the background beforehand and will be waiting for the test to pick it up in a ready-to-use state.

Another well-known option for using this approach would be Database Pool. Instead of working with the real database, we could start the needed number of database containers on the different ports (it can be done with Docker or another virtualization tool) and "kill" it after we do not need it anymore. That way we'd always have a clean database without the need of tearing it down, cleaning it up, collecting and uploading the data, etc.

Data Provider pattern.

Data Provider is one of the most widely used data patterns among test engineers. If you want to implement Data-Driven tests and are willing to run the same test logic on multiple sets of data, you could load the data from outer sources (like Excel or CVS table), remote services, or hardcode them in-place. Also, it allows you to write data-driven tests, which means you can run the same test methods multiple times with different data sets.

A very important function of the **TestNG** framework is **DataProvider**. To use the **DataProvider** function in your tests, you must declare a method with the **@DataProvider** annotation and then use this method in the test method using the “**dataProvider**” attribute in the Test annotation.

```
public class SimpleDataProvider {

    @DataProvider(name = "verifyCountry")
    public static Object[][] verifyCountry() {
        return new Object[][]{
            {"Estonia", "EE", "EST"},
            {"Germany", "DE", "DEU"},
            {"Afghanistan", "AF", "AFG"}
        };
    }

    @Test(dataProvider = "verifyCountry")
    @Story("GET countries from https://restcountries.eu/rest/v2/aplha")
    public void testRequestWithSeveralCountries(String name, String
alpha2Code, String alpha3Code) {
    }
}
```

To be able to read data from third-party sources, you can use:

```
@DataProvider
private static Object[][] testDataProvider() {
    try {
        return ReadExcelSheet.getTableArray(
            "src/main/resources/TestData.xls");
    } catch (Exception e) {
        return null;
    }
}
```

| But I do not recommend using this method because it is a bad practice.

This could be done in the way I showed above by reading from the source and returning untyped data (simple array of arrays or strings). But the modern approach would be to utilize the **Value Object pattern**, we were talking about previously, and provide data in terms of entities.

For more advanced use of Data Provider in TestNG there is an excellent library Test Data Supplier:

Common DataProvider forces using a quite old and ugly syntax which expects one of the following types to be returned from DP method's body:

Object[][]

Iterator<Object[]>

That's weird, as developers tend to use Stream and Collection API for data manipulation in the modern Java world.

Just imaging if you could use the following syntax to supply some filtered and sorted data into the test method's signature:

```
public class SimpleDataProvider {

    @DataSupplier
    public Stream<User> getData() {
        return Stream.of(
            new User("Tom Anderson", "password2"),
            new User("Sam Smith", "password3"),
            new User("Mark Wahlberg", "password1"))
            .filter(u -> !u.getName().contains("Virus"))
            .sorted(comparing(User::getPassword));
    }

    @Test(dataProvider = "getData")
    public void shouldSupplyStreamData(final User user) {
        // ...
    }
}
```

If you work with the JUnit framework you can use parameterization.

The **@RunWith** and **@Parameter** annotations are used to pass parameter values to the test. **@Parameters** returns **List[]**. These values are passed to the constructor as an argument.

```
@RunWith(value = Parameterized.class)
public class JunitTest {

    private String country;

    public JunitTest(String country) {
        this.country = country;
    }

    @Parameters
    public static Collection<Object[]> verifyCountry() {
        Object[][] data = new Object[][]{{"Estonia"}, {"EE"}, {"EST"},
{"Germany"}};
        return Arrays.asList(data);
    }

    @Test
    public void pushTest() {
        System.out.println("Parameterized Number is : " + country);
    }
}
```

Unfortunately, there are quite a few restrictions in JUnit:

- We must follow the JUnit signature to declare the parameter;
- The parameter is passed to the class constructor to initialize the class field that will contain the parameter value;
- The return type of the method with the **@Parameters** annotation must be List []
- Data must be of a primitive type (String, int)

But there is an alternative. If you are used to the TestNG syntax and would like to use it in a project with JUnit you can use:

JUnit-dataprovider. A TestNG like dataprovider (see [here](#)) runner for JUnit having a simplified syntax compared to all the existing JUnit4 features.

```

@RunWith(DataProviderRunner.class)
public class DataProviderTest {

    @DataProvider
    public static Object[][] dataProviderAdd() {
        // @formatter:off
        return new Object[][]{
            {"Estonia", "EE", "EST"},
            {"Germany", "DE", "DEU"},
            {"Afghanistan", "AF", "AFG"}
            /* ... */
        };
        // @formatter:off
    }

    @Test
    @UseDataProvider("dataProviderAdd")
    public void testAdd() {
        //some code here
    }
}

```

I love using Data Patterns in my automation, they help me to keep my code healthy and handle resources in the most optimized way possible.

It's important that we used both patterns (Data Provider and Value Object) in one approach since it helped us to avoid the passing of multiple parameters to a method and to make code cleaner and more readable.

Chapter 5. Technical Patterns.

Technical patterns serve to make the technical aspects separate from the test logic, and often to provide additional low-level control over the technical part.

- Decorator
- Proxy

The main goal of Technical Patterns is to encapsulate technical details from test logic, providing extra low-level control over them.

Decorator pattern.

A decorator is a technical design pattern that lets you attach new behaviors to objects by placing these objects inside special wrapper objects that contain the behaviors.

Decorator is a very well-known pattern since it was mentioned in the GoF list and discussed in many other programming books and articles. The example of this one is simple.

To understand how to apply a decorator template, let's look at an abstract example. Let's assume that we have a certain **interface Developer** and this interface has only one **makeJob** method:

```
public interface Developer {  
    public String makeJob();  
}
```

Let's create one implementation of this interface and call it **JavaDeveloper**. Our JavaDeveloper class will implement the Developer interface. And the makeJob method will return the string write java code.

```
public class JavaDeveloper implements Developer {  
  
    @Override  
    public String makeJob() {  
        return "write Java code.";  
    }  
}
```

Let's create a client program and call it Task. Our task creates a developer instance. And outputs to the console the result of executing the makeJob developer method.

```
public class Task {  
  
    public static void main(String[] args) {  
        Developer developer = new JavaDeveloper();  
  
        System.out.println(developer.makeJob());  
    }  
}
```

write Java code.Process finished with exit code 0

Let's assume that now we need the makeJob method to execute Senior Java Developer. Objectively this Java developer's makeJob method differs from the Senior Java Developer method. Let's assume that now we need the makeJob method to execute Senior Java Developer. Objectively, this Java developer's makeJob method differs from the Senior Java Developer method. Senior Java Developer has additional responsibilities. For example, execute Code review.

To make this implementation convenient and functional, we need to use a certain template. We will use the decorator template rather than making implementations from the Developer interface.

Let's create a decorator and call it **DeveloperDecorator**. The **DeveloperDecorator** class is an implementation of the Developer interface. It has a developer instance. It also has a constructor. And the makeJob method calls the developer method.

```
public class DeveloperDecorator implements Developer {  
  
    Developer developer;  
  
    public DeveloperDecorator(Developer developer) {  
        this.developer = developer;  
    }  
  
    @Override  
    public String makeJob() {  
  
        return developer.makeJob();  
    }  
}
```

Now let's create a Senior Developer implementation. Creating the Senior Developer class and extending it with the Developer Decorator. Let's create the **makeCodeReview** method, and this method will return the string make code review. After that, we redefine the makeJob

method and add the **makeCodeReview** method to it. And it will look like this:

```
public class SeniorJavaDeveloper extends DeveloperDecorator {
    public SeniorJavaDeveloper(Developer developer) {
        super(developer);
    }

    public String makeCodeReview() {
        return " Make code review.";
    }

    @Override
    public String makeJob() {
        return super.makeJob() + makeCodeReview();
    }
}
```

Now we will change our Task program as follows. Now the implementation will be conducted around the SeniorJavaDeveloper which accepts the Java developer as a constructor:

```
public class Task {

    public static void main(String[] args) {
        Developer developer = new SeniorJavaDeveloper(new JavaDeveloper());

        System.out.println(developer.makeJob());
    }
}
```

write Java code. Make code review.Process finished with exit code 0

By analogy with Senior Java Developer, we will create a new implementation of Java Team Lead.

```
public class JavaTeamLead extends DeveloperDecorator {

    public JavaTeamLead(Developer developer) {
        super(developer);
    }

    public String sendWeekReport() {
        return " Send week report to customer.";
    }

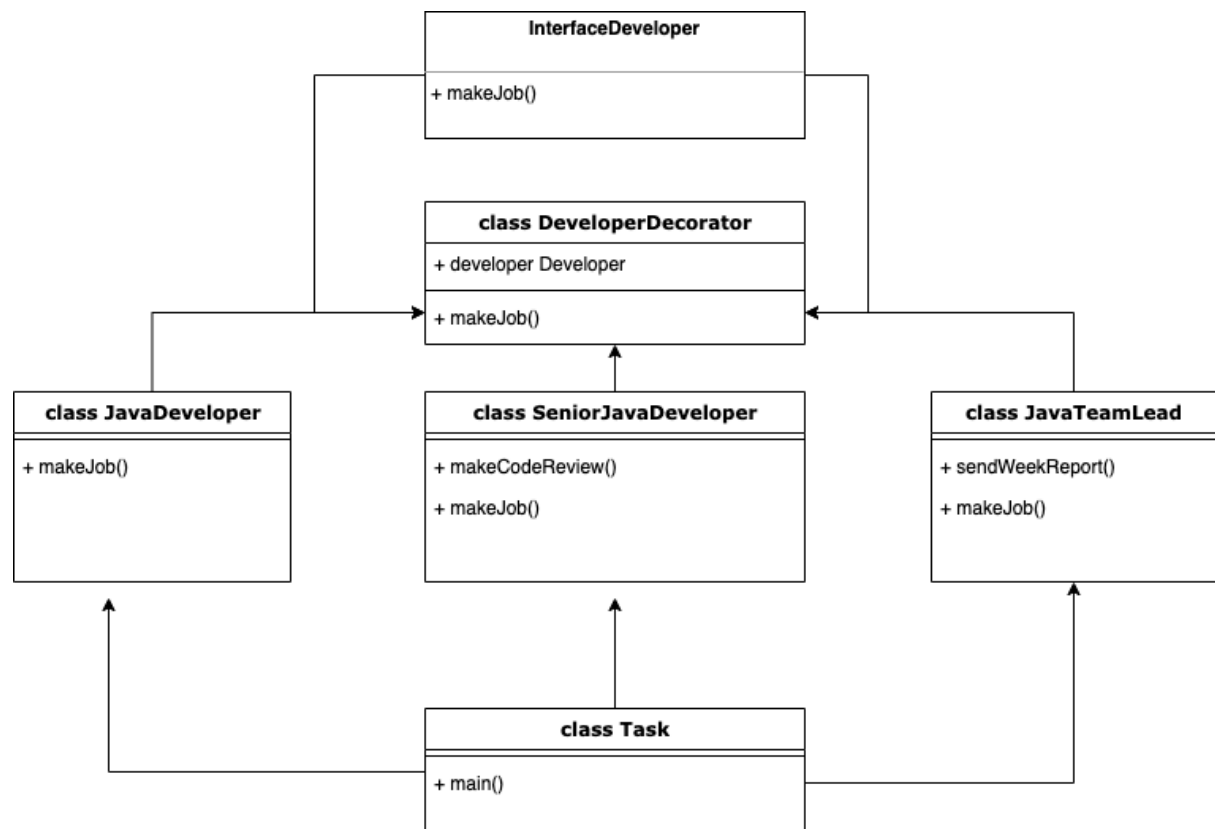
    @Override
    public String makeJob() {
        return super.makeJob() + sendWeekReport();
    }
}
```

And we make changes in our class client:

```
public class Task {  
    public static void main(String[] args) {  
        Developer developer = new JavaTeamLead(new SeniorJavaDeveloper(new JavaDeveloper()));  
        System.out.println(developer.makeJob());  
    }  
}
```

write Java code. Make code review. Send week report to customer.Process finished with exit code 0

Let's present our implementation in the form of a diagram.



In order to fix the material about the Decorator template, let's look at a more approximate example. This example is not abstract and may occur to you while working on the project. And so let's start.

Often there is a situation when the framework for automation, must support different versions of the components of the website. As a rule, this is because the web or mobile application is constantly changing and being updated. The most common example of this approach is if a company uses A / B testing and it is performed at the component level. If you have similar requirements and your case is very similar to the one described above, the decorator pattern may be the perfect approach for you. The template allows you to package components in so-called additional “envelopes” that overwrite or supplement only certain functionality. You don’t need to write a new class for each new feature of the component: only the changes should be implemented. The decorator template is similarly ideal if the web components of your application change depending on the size of the browser or the type of device.

Below we will look at a more real-world example of using the decorator template. In our example, two components or forms are given for authorization and log in. The second form has an additional “cancel” button.

In the usual approach, you can create classes for each component to solve this problem. However, the problem is that the second class will have a complete duplicate of the first class and will only have an implementation of the “cancel” method. However, creating more than one class results in code duplication and disconnects each new variation of the component from its peers. This becomes even more complex if you have multiple login components, such as for desktop and mobile devices.

Let’s try to solve this problem using the decorator template. We already know the theoretical part of the decorator template from the first part of this article. And now we will put this knowledge into practice.

The **LoginStructure** is the interface for every login component. It states that each component needs to have a defined login method.

```
public interface LoginStructure {  
    public void login(String email, String password);  
}
```

The **BasicLogin** has a concrete implementation of a login method. In this example, it just outputs “Basic login” on the command line.

```
public class BaseLogin implements LoginStructure {  
  
    @Override  
    public void login(String email, String password) {  
        System.out.println("Base login:" + email + "," + password);  
    }  
}
```

This class is the heart of the pattern. A **LoginDecorator** can take any **LoginStructure** and wrap some functionality around it. After this, the result remains a LoginStructure.

```
public class LoginDecorator implements LoginStructure {  
  
    private final LoginStructure loginStructure;  
  
    public LoginDecorator(LoginStructure loginStructure) {  
        this.loginStructure = loginStructure;  
    }  
  
    @Override  
    public void login(String email, String password) {  
        loginStructure.login(email, password);  
    }  
}
```

This **MobileLoginDecorator** overrides the basic login functionality with a new class that is specific to mobile. Again, it simply outputs “Mobile login” to keep this example short.

```

public class MobileLogin extends LoginDecorator {

    public MobileLogin(LoginStructure loginStructure) {
        super(loginStructure);
    }

    @Override
    public void login(String email, String password) {
        super.login(email, password);
        System.out.println("email" + email + "password" + password);
    }
}

```

This **CancelButton** can add the cancel functionality to any login component.

```

public class CancelButton extends LoginDecorator {

    public CancelButton(LoginStructure loginStructure) {
        super(loginStructure);
    }

    public void cancel() {
        System.out.println("Click the cancel button.");
    }
}

```

Now we can test how it all behaves.

```

public class Main {

    public static void main(String[] args) {
        System.out.println("The Design pattern Decorator.");
        LoginStructure loginStructure = new BaseLogin();
        loginStructure.login("admin", "admin");

        loginStructure = new MobileLogin(loginStructure);
        loginStructure.login("test", "test");

        loginStructure = new CancelButton(loginStructure);
        ((CancelButton) loginStructure).cancel();
    }
}

```

```
The output of it all is:The Design pattern Decorator.  
Basic login:admin,admin  
Basic login:test,test
```

```
Click the cancel button.Process finished with exit code 0
```

Even though this seems to be a lot of overhead at first, it is less work in the long run. You need only to implement the changed or added functionality in a decorator and add it to the component you want to change. This keeps your test code concise and modular.

The decorator design pattern is a good choice in the following cases:

- When we wish to add, enhance or even remove the behavior or state of objects.
- When we just want to modify a single object's functionality and leave others unchanged.

Proxy pattern.

In proxy pattern, a class represents the functionality of another class.

In proxy pattern, we create an object having an original object to interface its functionality to the outer world.

Proxy is the pattern that allows intervening into the process running between you and another user, introducing new logic in-between without affecting either of the sides.

The purpose of the Proxy pattern is to replace the object and control access to it.

Let's take an example of how the Proxy pattern works. Let's say we have a project on github. We have full access to this project and we can download and run it.

Let's create the project interface. This interface will have one **run()** method.

The implementation looks like this:

```
public interface Project {  
    public void run();  
}
```

Next, we need to create an implementation of this project. Creating the **RealProject** class. The **RealProject** class will implement the **Project** interface. Since our project is a repository on **github**, it has a link that will be expressed as a **url** field in the **RealProject** class. We will also add the **load()** method. This method will respond to the ability to download the project.

The implementation of this class will look like this:

```
public class RealProject implements Project {
    PrintStream printStream = System.out;
    private String url;

    public RealProject(String url) {
        this.url = url;
        load();
    }

    public void load() {
        printStream.println("Loading project from " + url + " .....");
    }

    @Override
    public void run() {
        printStream.println("Running project " + url + " .....");
    }
}
```

To test the work of our implementation, we will create a client and call it **ProjectRunner**.

The implementation of this class will look like this:

```
public class ProjectRunner {

    public static void main(String[] args) {
        Project project = new
RealProject("https://github.com/exampleproject/realproject");

        project.run();
    }
}
```

The console will display:

```
Loading project from https://github.com/exampleproject/realproject .....
Running project https://github.com/exampleproject/realproject .....
```

As we can see in the console, before starting the project, this project is downloaded. This happens because in the constructor we assign a link and start downloading:

```
public RealProject(String url) {
    this.url = url;
    load();
}
```

Now we will try to change our implementation and apply the Proxy pattern. We will not load the objects when creating the project but only when starting the project. In other words, we will have a lazy link. This is exactly what the Proxy pattern is for.

To implement the Proxy pattern, create the **ProxyProject** class. The **ProxyProject** class will implement the Project interface. The class will contain the **url** and **realProject** fields inside. The implementation of this class will look like this:

```
public class ProxyProject implements Project {

    private String url;
    private RealProject realProject;

    public ProxyProject(String url) {
        this.url = url;
    }

    @Override
    public void run() {
        if (realProject == null) {
            realProject = new RealProject(url);
        }

        realProject.run();
    }
}
```

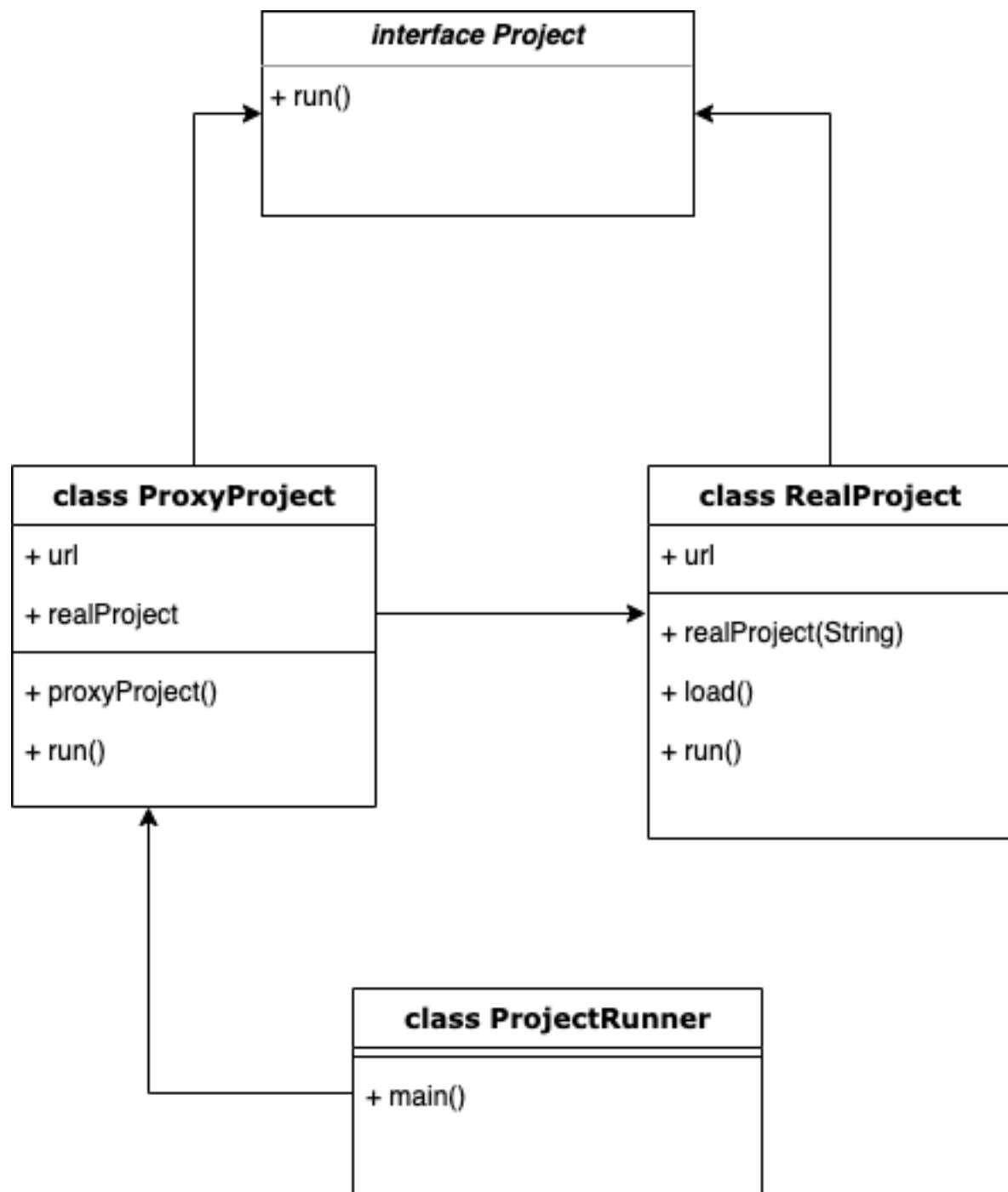
You need to make changes in the client class:

```
public class ProjectRunner {

    public static void main(String[] args) {
        Project project = new ProxyProject("https://github.com/exampleproject/realproject");

        project.run();
    }
}
```

Let's see what our implementation looks like on a UML diagram:



This pattern might be useful when you, for example, want to add logging, to enable or to disable something, to have control over some additional recourses, etc. The most popular method of using it in tests is to set up the HTTP proxy. It allows to dynamically enable and disable host blacklists, excluding or stubbing the third-party sites like Facebook or Twitter in your tests.

Sometimes it's the only way you can check some exceptional scenarios for the external services like those ones.

Another two examples I could come up with are caching of nonfunctional assets (like images or CSS), which do not affect the functionality of the app, and collecting HTTP traffic for further analysis while running the tests (by analysis I mean verifying if some images were missing or if any recourses took a lot of time to load, etc).

Use HTTP Proxy for tests to:

- Blacklist external resources (Facebook, Twitter, Ads, etc)
- Cache images and other non-functional assets.
- Collect HTTP traffic for analysis (loading time, 404 errors, redirects, etc)
- Speed up page loading.

Chapter 6. Business Involvement Patterns.

We try to bring product owners, business analysts, and other people responsible for requirements as close to test automation as possible so that they see the benefits of it, and they invest in it as much as we do, because we are a single team.

- Keyword Driven Testing
- Behavior Specification
- Behavior Driven Development
- Steps

The main goal of these patterns is to bring Product Owners, Business Analytics, and other people, responsible for requirements, as close to test automation as possible. This way its benefits might become crystal clear to them and they would get involved in it as much as we are. It's the perfect scenario for all the members of the team, isn't it?

Keyword Driven Testing pattern.

To make the latter possible we could use the popular pattern, which allows to distance tests from the code and helps to make it possible to write them in human-readable language, understood by a regular person.

Keyword Driven Tests are written using keywords - domain commands, clear to everyone in the team. Hence, they might contain some data, they should be implemented by people, who are familiar with the technical details and are able to design them in code. But at the same time, the framework should allow anyone to write the tests - it might be a manager, business analytic, or test engineer without any coding experience.

```
*** Settings ***
Documentation    A test suite with a single test for valid login.
This test has a workflow that is created
using keywords in the imported resource file.

Resource        resource.csv

*** Test Cases ***
Valid login
[Set Up]
Open Browser
Open Login Page
Input Username    admin
Input Password    admin
Submit Credentials
Welcome Page should be Opened
[Teardown]
Close Browser
```

In the example above, you can see how this pattern is implemented with the help of the Robot Framework. There are lots of free and paid frameworks out there, which help to adopt the KDT approach and make Business life easier.

The main idea behind this is that there is a finite number of operations in any app. This means that if all of them are implemented as keywords, we are able to build an infinite number of test scenarios by combining them in one or another way.

By doing this you could provide the working tool to someone who wants to be involved in the building of the automation on the project.

This pattern faced a lot of critics from people who implemented it in their automation but weren't able to involve anyone from the business side in writing the tests. That's why before implementing Key Driven Testing on your project you should discuss its usage with all the stakeholders. Otherwise, it might be overhead to build something complex like this without business awareness and support.

Behavior Specification pattern.

The Behavior Specification this pattern suggests transforming the writing of the tests into defining the expected behavior. This way the feature can be described in the form of behavior scenarios. Here is a simple example of this pattern:

```
Feature: Addition
  In order to avoid silly mistakes
  As a math error
  I want to be told the sum of two numbers

Scenario: Add two numbers
  Given I entered 5 into the calculator
  And I entered 7 into the calculator
  When I pressed Add button
  Then The result should be 12 on the screen
```

The scenario describes the simple workflow of the addition of two numbers using calculator buttons and verification of the result on its screen. This test is not a “test” in common sense, it’s more a behavior specification. It’s easy to read and can be understood by anyone. But again, this pattern will be super useful only if the business wants to be involved in the testing process (e.g., by creating the acceptance criteria based on those specs and tracking the results of tests).

If the business on your project is not ready for this kind of approach, you don’t have a problem that needs the given solution, and by implementing it you’d create another unnecessary layer of abstraction above your test logic.

Behavior Driven Development pattern.

Usually Behavior Specification is used along with concept of Behavior Driven Development, which is saying that:

- at first we need to describe the behavior of functionality as a spec
- next we want to write a test for that
- then we need to implement that functionality
- verify that test passes
- verify that behavior scenario passes

As a result we'll get working low-level tests and comprehensive high-level scenarios, making developers and business happy, right? But I'm afraid its not that simple. It would be complete waste of time if this approach is used by engineers (technical people) exclusively without involving any business to writing scenarios.

Steps pattern.

The step pattern's idea might be pretty interesting regardless of what approach you're using: Behavior Driven, Keyword Driven Development, or Behavior Specification. When we're talking about logic scenarios, we think about them as if they consist of steps. But when you implement them in your code, these steps often get lost among all the technical details, data manipulations, and other things we're doing in our tests. It would be great if we could isolate those steps from other code showing what this particular test does and easing the hurdle of its maintenance. Step's pattern will help us to do that.

Your test steps might be divided into groups by functionality and gathered into the special classes or methods. If you don't have the step you need in your test, you can simply create one. You can use the Steps Pattern either with some technical framework (e.g., Page Object) or without any. It doesn't matter as long as you divide your logic into the Steps:

```
public void bookShouldBeSearchedByName() {  
    loginToBookStore("admin", "admin");  
    openBookSearchPage();  
    enterSearchCriteria("name", "All about design patterns in automation testing.");  
    assertFalse(isEmptySearchResult());  
    openBookDetailsByIndex(1);  
  
    assertEquals("All about design patterns in automation testing.", getBookTitle());  
    assertEquals("Anton Smirnov", getBookAuthor());  
}
```

The Steps example above can be easily transformed to a Test Scenario:

Steps:

1. Login to book store as user 'admin' and password 'admin'
2. Open search books pages
3. Try to search by word 'All about design patterns in automation testing.'
4. Results should not be empty
5. Open the first found book details

Expected results:

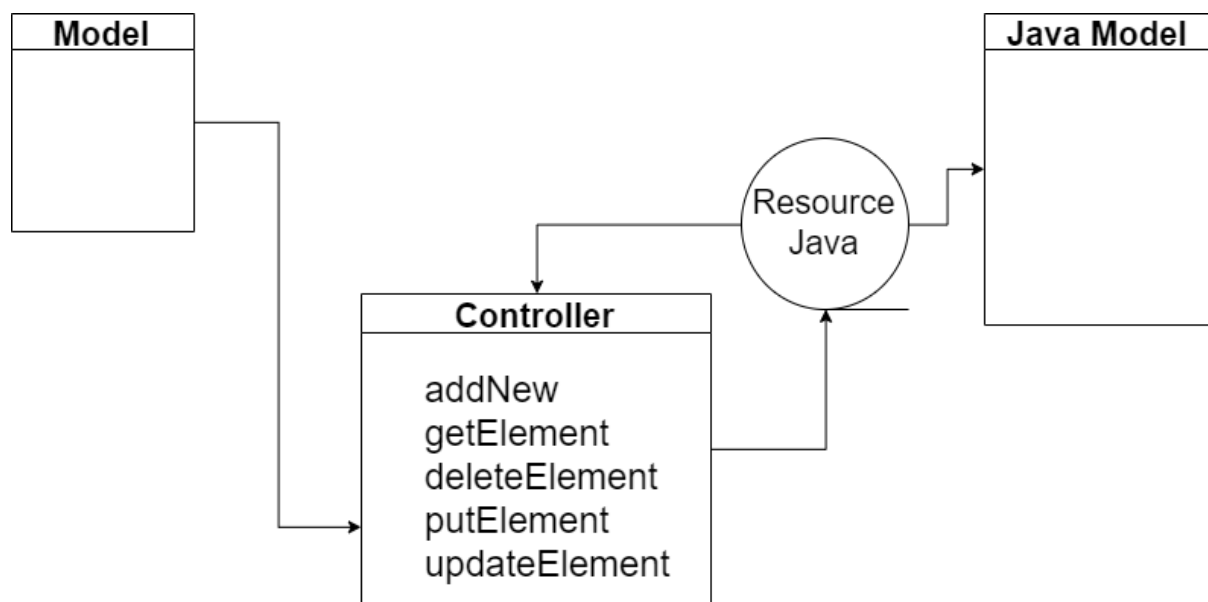
1. Book title is 'All about design patterns in automation testing.'
2. Author is 'Anton Smirnov'

Writing test scenarios in this way may help to avoid using any extra Test Case Management system, since all of the Test Steps, either automated or manual ones, are stored in one place, which is your framework. As a bonus now you can easily track, how many steps are automated, how many of them are broken, who was responsible for their creation etc.

Bonus. The mediator pattern.

A controller approach is essentially an approach with the implementation of the intermediary pattern. It is also called a mediator.

The mediator pattern is a behavioral design pattern that allows you to reduce the connectivity of many classes to each other by moving these relationships into a single intermediary class. The Mediation pattern causes objects to communicate not directly with each other, but through a separate mediation object that knows who to redirect a particular request to. Because of this, the components of the system will depend only on the intermediary, and not on dozens of other components.



In other words, the main purpose control resource and transfer of any facilities or fields.

Thus, the mediator hides all the complex relationships and dependencies between classes of individual components of the program. And the fewer relationships classes have, the easier it is to modify, extend, and reuse them.

Example:

Pilots of landing or departing aircraft do not communicate directly with other pilots. Instead, they communicate with the dispatcher, who coordinates the actions of several aircraft at the same time. Without the controller, the pilots would have to be on the alert all the time and monitor all the surrounding aircraft on their own, and this would lead to frequent disasters in the sky.

It is important to understand that the controller is not needed during the entire flight. It is involved only in the area of the airport when you need to coordinate the interaction of many aircraft.

Pros:

- Eliminates dependencies between components, allowing them to be reused.
- Simplifies interaction between components.
- Centralizes management in one place.

Cons:

- The middleman can swell up a lot.

Next, we consider the implementation of the mediator pattern in conjunction with Rest Assured.

Initially, we need to create a Controller for the Model.

In this controller, there will be methods that will carry out some manipulations. In essence, they will perform manipulations to the resource, if we talk about a normally standardized resource that has implemented methods **GET**, **POST**, **PUT**, **UPDATE**, **OPTIONS**, **DELETE** and etc. In one class there will be a convenient manipulation with these methods.

In other words, we create an intermediary that will work with the model and resource.

Create the controller class :

```
public class Controller {
    public Model addNew(Model newModel) {
        return given().body().when().post().as(Model.class);
    }
}
```

The type it will have is Model because we want to get a response in response and test it.

This example implements a method of deserializing a method to a Model.

With the participation of the controller, we can also reconfigure the basic settings for our test.

As a rule, such settings are made either in the controller itself or in the abstract controller class.

Example controller:

```
public class Controller {
    private RequestSpecification requestSpecification;
    private Model model;

    public Controller(Model testModel) {
        requestSpecification = new RequestSpecBuilder()
            .addHeader("Content-type", "application/json; charset=UTF-8")
            .setBaseUri("some base url")
            .setBasePath("some base path")
            .setContentType(ContentType.JSON)
            .log(LogDetail.ALL).build();
        RestAssured.responseSpecification = new ResponseSpecBuilder()
            .expectContentType(ContentType.JSON)
            .expectResponseTime(Matchers.lessThan(15000L))
            .build();
        RestAssured.defaultParser = Parser.JSON;
        this.model = testModel;
    }
}
```

Example of an abstract controller:

```
public abstract class AbstractController {
    static {
        RestAssured.requestSpecification = new RequestSpecBuilder()
            .addHeader("Content-type", "application/json; charset=UTF-8")
            .setBaseUri("some base url")
            .setContentType(ContentType.JSON)
            .log(LogDetail.ALL).build();
        RestAssured.responseSpecification = new ResponseSpecBuilder()
            .expectContentType(ContentType.JSON)
            .expectResponseTime(Matchers.lessThan(15000L))
            .build();
        RestAssured.defaultParser = Parser.JSON;
    }
}
```

The automation script looks like this:

```
@Test(description = "The automation script for added new model")
public void testAddNew() {
    Model model = new Model();
    Controller controller = new Controller(model);
    controller.addNewElement(model);
}
```

Conclusion.

Automation testing has established its trend in testing, and more tools will be developed to meet the growing needs of software development teams. Finding the perfect tool is still difficult, but we have good news: you have much more choice than before. After carefully considering your requirements, the pros and cons of each solution-try not to be too ambitious at an early stage and try out the top 3 relevant candidates. By creating a POC for these solutions, you will better know the critical factors of your project and fine-tune your shortlist. This approach gives you a good chance to determine the appropriate tool for the current state and information of the next choice when your project is more mature.

Each application is different from the others, as is the process used to test them. As more teams move to a flexible model, creating a flexible framework for automated testing is critical.

Choose a tool (or framework) that can quickly and easily adapt to your processes. When choosing a automated testing tool, you should look for one that is perfectly suited to your needs. Initially, I recommend that you think about and decide which tool you will use native or cross-platform. In cases of a cross-platform tool, I recommend choosing a more flexible one. It is important that the tool can support a wide range of programming languages. This will allow your team and organization, regardless of experience and skill set, to contribute to your testing efforts.

I hope that if you have read this book to the end, you have an idea about the basics mocks of automation testing. This chapter offers books and websites to help you continue to learn. Of course, you've seen a lot of code snippets. Some of them you can use in your practice. Some may need to be modified before use. Most of the code you saw was written as annotated `@Test` methods with operators. Pretty much what you'll be writing in the real world.

Recommended Reading

I don't recommend a lot of Java books because they are a very personal thing. There are books that people rave about that I couldn't get my head around. And there are those that I love that other people hate. But since I haven't provided massive coverage of the Java language. I've pretty much given you "just enough" to get going and understand the code you read. I'm going to list the Java books that I gained most from, and still refer to:

"Effective Java" by Joshua Bloch

"Implementation Patterns" by Kent Beck

"Growing Object-Oriented Software, Guided by Tests" by Steve Freeman and Nat Pryce

"Core Java: Volume 1 - Fundamentals" by Cay S. Horstmann and Garry Cornell

"Covert Java : Techniques for Decompiling, Patching and Reverse Engineering" by Alex Kalinovsky

"Java Concurrency in Practice" by Brian Goetz

"Mastering Regular Expressions" by Jeffrey Friedl Now, to justify my selections...

Effective Java

"Effective Java" by Joshua Bloch, at the time of writing in its 2nd Edition. This book works for beginners and advanced programmers. Java developers build up a lot of knowledge about their language from other developers. "Effective Java" helps short cut that process. It has 78 chapters. Each, fairly short, but dense in their coverage and presentation. When I first read it, I found it heavy going, because I didn't have enough experience or knowledge to understand it all. But I re-read it, and have continued to re-read it over the time I have

developed my Java experience. And each time I read it, I find a new nuance, or a deeper understanding of the concepts. Because each chapter is short, I return to this book to refresh my memory of certain topics. This was also the book that helped me understand enum well enough to use them and helped me understand concurrency well enough to then read, and understand, “Java Concurrency in Practice”. I recommend that you buy and read this book early in your learning. Even if you don’t understand it all, read it all. Then come back to it again and again. It concentrates on very practical aspects of the Java language and can boost your real-world effectiveness tremendously.

Implementation Patterns

Another book that benefits from repeated reading. You will take different information from it with each reading, depending on your experience level at the time. “Implementation Patterns” by Kent Beck explains some of the thought processes involved in writing professional code.

This book was one of the books that helped me:

concentrate on keeping my code simple,
decide to learn the basics of Java (and know how to find information when I needed it),
try to use in built features of the language, before bringing in a new library to my code.

The book is thin and, again dense. Most complaints I see on-line seem to stem from the length of the book and the terseness of the coverage. I found that beneficial, it meant very little padding and waste. I have learned, or re-learned, something from this book every time I read it.

Other books that cover similar topics include “Clean Code” by Robert C. Martin, and “The Pragmatic Programmer” by Andrew Hunt and David Thomas. But I found “Implementation Patterns” far more useful and applicable to my work.

Growing Object-Oriented Software

Another book I benefited from reading when I wasn't ready for it. I was able to re-read it and learn more. I still gain value from re-reading it.

“Growing Object-Oriented Software, Guided by Tests”, by Steve Freeman and Nat Pryce

Heavily focused on using @Test method code to write and understand your code. It also covers mock objects very well. This book helped change my coding style, and how I approach the building of abstraction layers.

Covert Java

“Covert Java : Techniques for Decompiling, Patching and Reverse Engineering”, by Alex Kalinovsky starts to show its age now as it was written in 2004. But highlights some of the ways of working with Java libraries that you really wouldn't use if you were a programmer. But sometimes as a tester we have to work with pre-compiled libraries, without source code, and use parts of the code base out of context. I found this a very useful book for learning about reflection and other practices related to taking apart Java applications. You can usually pick this up quite cheaply second hand. There are other books the cover decompiling, reverse engineering and reflection. But this one got me started, and I still find it clear and simple.

Java Concurrency in Practice

Concurrency is not something I recommend trying to work with when you are starting out with Java. But at some point you will probably want to run your code in parallel, or create some threads to make your code perform faster. And you will probably fail, and not really understand why. I used “Effective Java” to help me get started. But “Java Concurrency in Practice” by Brian Goetz, was the book I read when I

really had to make my automation abstraction layer work with concurrent code.

Core Java: Volume 1

The Core Java books are massive, over 1000 pages. And if you really want to understand Java in depth then these are the books to read. I find them to be hard work and don't read them often. I tend to use the JavaDoc for the Java libraries and methods themselves. But, periodically, I want to have an overview of the language and understand the scope of the built in libraries, because there are lots of in-built features that I don't use, that I would otherwise turn to an external library for. Every time I've flicked through "Core Java", I have discovered a nuance and a new set of features, but I don't do it often.

Mastering Regular Expressions

We didn't cover the full power of Regular Expressions in this book. I tend to try and keep my code simple and readable so I'll use simple string manipulation to start with. But over time, I often find that I can replace a series of if blocks and string transformations with a regular expression. Since I don't use regular expressions often I find that each time, I have to re-learn them and I still turn to "Mastering Regular Expressions" by Jeffrey E.F. Friedl. As an alternative to consider: "Regular Expressions Cookbook" by Jan Goyvaerts, which is also very good.

I sometimes use the tool RegexMagic regexmagic.com, written by Jan Goyvaerts when writing regular expressions, it lets me test out the regular expression across a range of example data, and generate sample code for a lot of different languages. Jan also maintains the web site regular-expressions.info with a lot of tutorial information on it

Recommended Web Sites:

For general Java news, and up to date conference videos, I recommend the following web sites.

theserverside.com

infoq.com/java

Make sure you subscribe to the RSS feeds for the above sites. I will remind you that I have a web site MySite and I plan to add more information there, and links to other resources over time. I will also add additional exercises and examples to that site rather than continue to expand this book. Remember, all the code used in this book, and the answers to the exercises is available to download from GitHub my

Next Steps

This has been a beginner's book. You can see from the “Advancing Concepts” chapter that there are a lot of features in Java that I didn't cover. Many of them I don't use a lot and I didn't want to pad out the book with extensive coverage that you can find in other books or videos.

I wanted this book to walk you through the Java language in an order that I think makes sense to people who are writing code, but not necessarily writing systems. Your next step? Keep learning. I recommend you start with the books and videos recommended here, but also ask your team mates. You will be working on projects, and the type of libraries you are using, and the technical domain that you are working on, may require different approaches than those mentioned in this book. I hope you have learned that you can get a lot done quite easily, and you should now understand the fundamental classes and language constructs that you need to get started. Now: start writing @Test methods which exercise your production code investigate how much of your repeated manual effort can be automated Thank you for your time spent with this book. I wish you well for the future. This is just the start of your work with Java. I hope you'll continue to learn more and put it to use on your projects. My ability to use automation to support my testing and add value on projects continues to increase, the more I learn how to improve my coding skills. I hope yours does too.

References Java For Testers

github.com/eviltester/javaForTestersCode
JavaForTesters.com

Joshua Bloch en.wikipedia.org/wiki/Joshua_Bloch
youtu.be/pi_I7oD_uGI Kent Beck twitter.com/kentbeck

“Three Rivers Institute” threeriversinstitute.org Growing Object
Oriented Software, Guided by Tests

growing-object-oriented-software.com

Steve Freeman’s Blog higherorderlogic.com

natpryce.com

Core Java Book

horstmann.com/corejava.html

Java Concurrency

In Practice jcip.net/s3-website-us-east-1.amazonaws.com/

Regular Expressions

Mastering Regular Expressions home page regex.info

regular-expressions.info/

regexmagic.com

regexpal.com

www.regexr.com