Лабораторная работа №7

Тестирование пользовательского интерфейса

Цель: Познакомится с основами написания UI тестов, их спецификой, а также фреймворками, используемыми для UI тестирования.

План занятия:

1. Изучить теоретические сведения.
2. Выполнить практическое задание.

1. Теоретические сведения

Перед началом работы необходимо установить следующее:

- Open JDK (https://openjdk.java.net)
- Java IDE (рекомендуемыя IDE: IntelliJ IDEA Community)
- Git (https://git-scm.com/downloads)

Опционально (в случае если используется НЕ IntelliJ IDEA):

- Maven (https://maven.apache.org)
- Git клиент (например, SourceTree https://www.sourcetreeapp.com)

1.1. Тестирование пользовательского интерфейса

Интерфейс пользователя (user interface, UI) - разновидность интерфейса обеспечивающее взаимодействие через графические элементы (меню, кнопки, значки, списки и т. п.)

Функциональное тестирование пользовательского интерфейса подразумевает эмуляцию действий пользователя, то есть, сценарии тестов приближены к реальным сценариям взаимодействия конечного пользователя с системой и проверяют работоспособность системы в целом.

Поскольку тестовые сценарии имитируют действия пользователя они являются более сложными по сравнению с тестами для API или модульными

тестами, но их количество обычно ощутимо меньше чем количество других тестов. Несмотре на меньшее количество каждый такой тест может покрывать гораздо больший кусок функциональности приложения по сравнению с другими типами тестов за счёт косвенного зедействования всех компонентов системы, вовлечённых в бизнес логику, запущенную действиями пользователя.

Существует целый ряд фреймворков, помогающих реализовать взаимодействие с пользовательским интерфейсом. Один из самых популярных инструментов такого рода, предназначенный для автоматизации работы с веб приложениями - Selenium.

1.2. Selenium

Начнем с базового определения. По сути, Selenium - это целый комплекс решений, содержимое которого можно объединить в один пакет для работы с различными сервисами. Стоит отметить, что все решения находятся в свободном доступе вместе с открытым кодом. Наиболее популярными являются следующие продукты:

- Selenium WebDriver
- Selenium RC
- Selenium Server
- Selenium Grid
- Selenium IDE

Из-за разнообразной направленности ПО, многие путаются с определением Selenium, принимая его за конкретный продукт для одного сервиса.

Selenium WebDriver

Selenium WebDriver – это программная библиотека для управления браузерами. Часто употребляется также более короткое название WebDriver. Эти библиотека используется для отправки HTTP запросов драйверу (отсюда и название WebDriver), с помощью протокола JsonWireProtocol, в которых указано действие, которое должен совершить браузер в рамках текущей сессии. Примерами таких команд могут быть команды нахождения

элементов, парсинг текста страницы/элемента, нажатие кнопок или переход по ссылкам на странице веб-сайта.

Иногда говорят, что это «драйвер браузера», но на самом деле это целое семейство драйверов для различных браузеров, а также набор клиентских библиотек на разных языках, позволяющих работать с этими драйверами.

Это основной продукт, разрабатываемый в рамках проекта Selenium. (официальная документация: https://www.selenium.dev/documentation)

Как уже было сказано, WebDriver представляет собой семейство драйверов для различных браузеров плюс набор клиентских библиотек для этих драйверов на разных языках программирования:

Примеры поддерживаемых браузеров: Chrome, Firefox, IE/Edge, Opera, Safari, PhantomJS

Примеры поддерживаемых языков программирования: Java, Python, C#, Ruby, JavaScript, Kotlin

Основными понятиями в Selenium Webdriver являются:

- **Webdriver** - самая важная сущность, ответственная за управление браузером. Основной ход скрипта/теста строится именно вокруг экземпляра этой сущности.
- **Webelement** - вторая важная сущность, представляющая собой абстракцию над веб элементом (кнопки, ссылки, инпута и др.). Webelement инкапсулирует методы для взаимодействия пользователя с элементами и получения их текущего статуса.
- **By** - абстракция над локатором веб элемента. Этот класс инкапсулирует информацию о селекторе(например, CSS), а также сам локатор элемента, то есть всю информацию, необходимую для нахождения нужного элемента на странице.

Сам процесс взаимодействия с браузером через Webdriver API довольно прост: Нужно создать Webdriver:

```
WebDriver driver = new ChromeDriver();
```

При выполнении этой команды будет запущен Chrome браузер, при условии, что браузер установлен в директорию по умолчанию и путь к ChromeDriver сохранен в системной переменной PATH или путь до драйвера задан в опциях Webdriver в коде.

Драйвер браузера необходимо предварительно скачать, например отсюда: https://chromedriver.chromium.org/downloads Версия браузера, установленного на машине на которой будут запускаться тесты должны быть совместима с версией драйвера!

Далее необходимо открыть тестируемое приложение, перейдя по url:

```
driver.get("http://mycompany.site.com");
```

Далее следует серия действий по нахождению элементов на странице и взаимодействию с ними:

```
 WebElement element = driver.findElement(By.id("#element_id")));
```

После нахождения элемента, кликнем по нему:

```
element.click();
```

Далее следует совокупность похожих действий, как того требует сценарий. В конце теста должна быть какая-то проверка, которая и определит в конечном счете результат выполнения теста:

```
assertEquals("Webpage expected title", driver.getTitle());
```

После теста надо закрыть браузер:

```
 driver.quit();
```

Следует отметить, что для поиска элементов доступно два метода:

Первый - найдет только первый элемент, удовлетворяющий локатору

```
WebElement element = driver.findElement(By.id("#element_id")));
```

Второй - вернет весь список элементов, удовлетворяющих запросу:

```
List<WebElement> elements =
 driver.findElements(By.name("elements_name"))
```

DOM

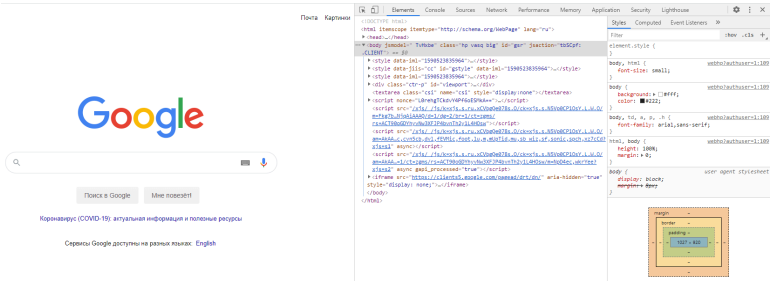
Объектная Модель Документа (DOM) – это программный интерфейс (API) для HTML и XML документов. DOM предоставляет структурированное представление документа и определяет то, как эта структура может быть доступна из программ, которые могут изменять содержимое, стиль и структуру документа. Представление DOM состоит из структурированной группы узлов и объектов, которые имеют свойства и методы. По существу, DOM соединяет веб-страницу с языками описания сценариев либо языками программирования.

Веб-страница – это документ. Документ может быть представлен как в окне браузера, так и в самом HTML-коде. В любом случае, это один и тот же документ. DOM предоставляет другой способ представления, хранения и управления этого документа. DOM полностью поддерживает объектно-ориентированнное представление веб-страницы, делая возможным её изменение при помощи языка описания сценариев наподобие JavaScript.

Стандарты W3C DOM и WHATWG DOM формируют основы DOM, реализованные в большинстве современных браузеров. Многие браузеры предлагают расширения за пределами данного стандарта, поэтому необходимо проверять работоспособность тех или иных возможностей DOM для каждого конкретного браузера.

Для просмотра DOM какой-либо страницы в вашем браузере можно использовать такой инструмент разработчика как инспектор DOM. Инспектор обычно открывается по нажатию клавиши F12 или через контекстное меню страницы ("Просмотреть код" для Chrome или "Исследовать элемент" для Firefox)

Пример DOM инспектора в Chrome браузере:



Используя инспектор можно получить подробную информацию о элементах на странице и использовать её для построения локатора элементов.

Локаторы

Поскольку Webdriver - это инструмент для автоматизации веб приложений, то большая часть работы с ним это работа с веб элементами(WebElements). WebElements - ни что иное, как DOM объекты, находящиеся на веб странице. А для того, чтобы осуществлять какие-то действия над DOM объектами / веб елементами необходимо их точным образом определить(найти).

```
WebElement element = driver.findElement(By.<locator>);
```

Таким образом в Webdriver определяется нужный элемент. By - класс, содержащий статические методы для идентификации элементов:

By.id

Пример:

```
<div id="element">
     <p>some content</p>
</div>
```

Поиск элемента:

```
WebElement element = driver.findElement(By.id("element_id"));
```

By.name

Пример:

```
<div name="element">
     <p>some content</p>
</div>
```

Поиск элемента:

```
WebElement element = driver.findElement(By.name("element_name"));
```

By.className

Пример:

```
<img class="logo">
```

Поиск элемента:

```
WebElement element = driver.findElement(By.className("element_class"));
```

By.TagName

Пример:

```
<div>
     <a class="logo" ref="...">...</a>
     <a class="support" ref="...">...</a>
</div>
```

Поиск элемента:

```
List<WebElement> elements = driver.findElements(By.tagName("a"));
```

By.LinkText

Пример:

```
<div>
     <a ref="...">text</a>
     <a ref="...">Another text</a>
</div>
```

Поиск элемента:

```
WebElement element = driver.findElement(By.linkText("text"));
```

By.cssSelector

Пример:

```
<div class='main'>
     <p>text</p>
     <p>Another text</p>
</div>
```

Поиск элемента:

```
WebElement element=driver.FindElement(By.cssSelector("div.main"));
```

By.XPath

Пример:

```
<div class='main'>
     <p>text</p>
     <p>Another text</p>
</div>
```

Поиск элемента:

```
WebElement element = driver.findElement(By.xpath("//
div[@class='main']"));
```

Ожидания

Ожидания - непременный атрибут любых UI тестов для динамических приложений. Они нужны для синхронизации работы приложения и тестового скрипта. Скрипт выполняется намного быстрее реакции приложения на команды, поэтому часто в скриптах необходимо дожидаться определенного состояния приложения для дальнейшего с ним взамодействия.

Самый просто пример - переход по ссылке и нажатие кнопки:

```
driver.get("http://google.com");
driver.findElement(By.id("element_id")).click();
```

В данном случае необходимо дождаться пока не появится кнопка с id = element_id и только потом совершать действия над ней. Для этого и существуют ожидания.

Ожидания бывают:

Неявные ожидания (implicit waits) - конфигурируют экземпляр WebDriver делать многократные попытки найти элемент (элементы) на странице в течении заданного периода времени, если элемент не найден сразу. Только по истечении этого времени WebDriver бросит ElementNotFoundException.

```
WebDriver driver = new FirefoxDriver();
driver.manage().timeouts().implicitlyWait(10, TimeUnit.SECONDS);
driver.get("http://some_url");
WebElement dynamicElement =
driver.findElement(By.id("dynamicElement_id"));
```

Неявные ожидания обычно настраиваются сразу после создания экземпляра WebDriver и действуют в течении всей жизни этого экземпляра, хотя переопределить их можно в любой момент. К этой группе ожиданий также можно отнести неявное ожидание загрузки страницы:

```
driver.manage().timeouts().pageLoadTimeout(10, TimeUnit.SECONDS);
```

А также неявное ожидание отработки скриптов:

```
driver.manage().timeouts().setScriptTimeout(10, TimeUnit.SECONDS);
```

Явные ожидания (explicit waits) - это код, который ждет наступления какого-то события, прежде чем продолжит выполнение команд скрипта. Такое ожидание срабатывает один раз в указанном месте. Самым худшим вариантом является использование Thread.sleep(1000), в случае с которым скрипт просто будет ждать определенное количество времени. Это не гарантирует наступление нужного события либо будет слишком избыточным и увеличит время выполнения теста.

Более предпочтительно использовать WebDriverWait и ExpectedCondition:

```
WebDriver driver = new FirefoxDriver();
driver.get("http://some_url");

WebElement dynamicElement = (new WebDriverWait(driver, 10))
.until(ExpectedConditions.presenceOfElementLocated(By.id("dynamicElement_id")));
```

В данном случае скрипт будет ждать элемента с id = dynamicElement_id в течении 10 секунд, но продолжит выполнение, как только элемент будет найден. При этом WebDriverWait класс выступает в роли конфигурации ожидания - задаем, как долго ждать события и как часто проверять его наступление. ExpectedConditions - статический класс, содержащий часто используемые условия для ожидания.

1.3. Page Object

Page Object — это паттерн для реализации автоматизированных проверок, который помогает в создании гибких страниц с объектами для тестирования браузерных приложений. Суть в том, чтобы создавать уровни абстракции для отделения тестов от предметов тестирования (страниц), и обеспечить простой интерфейс для элементов на странице, чтобы инкапсулировать работу с отдельными элементами страницы и, как следствие, уменьшить количество кода и его поддержку. Если, к примеру, дизайн одной из страниц изменён, то нам нужно будет переписать только соответствующий класс, описывающий эту страницу.

Основные преимущества:

- Разделение кода тестов и описания страниц
- Объединение всех действий по работе с веб-странией в одном месте

Для примера возьмём страницу логина, которая есть практически на всех сайтах. Создадим класс с описание этой страницы, используя шаблон Page Object:

```
public class LoginPage {
    By usernameLocator = By.id("username");
```

```
    By passwordLocator = By.id("passwd");
    By loginButtonLocator = By.id("login");

    private final WebDriver driver;

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

        if (!"Login".equals(driver.getTitle())) {
            throw new IllegalStateException("This is not the login
page");
        }
    }

    public LoginPage typeUsername(String username) {
        driver.findElement(usernameLocator).sendKeys(username);
        return this;
    }

    public LoginPage typePassword(String password) {
        driver.findElement(passwordLocator).sendKeys(password);
        return this;
    }

    public HomePage submitLogin() {
        driver.findElement(loginButtonLocator).submit();
        return new HomePage(driver);
    }

    public LoginPage submitLoginExpectingFailure() {
        driver.findElement(loginButtonLocator).submit();
        return new LoginPage(driver);
    }

    public HomePage loginAs(String username, String password) {
        typeUsername(username);
        typePassword(password);
        return submitLogin();
    }
```

Page Factory

Еще одним вариантом применения Page Object шаблона является использование класса Page Factory из библиотеки Selenium. Давайте

разберёмся, как с ним работать. Сначала нам нужно создать простой Page Object:

```
public class GoogleSearchPage {
    private WebElement q;
    public void searchFor(String text) {
        q.sendKeys(text);
        q.submit();
    }
}
```

Теперь, чтобы всё работало корректно, нам нужно инициализировать наш page object. Это выглядит так:

```
public class UsingGoogleSearchPage {
    public static void main(String[] args) {
        WebDriver driver = new HtmlUnitDriver();

        driver.get("http://www.google.com/");

        GoogleSearchPage page = PageFactory.initElements(driver,
 GoogleSearchPage.class);

        page.searchFor("Cheese");
    }
}
```

Основным недостатком использования Page Factory является то что при таком подходе к инициализации элементов страницы возникают трудности в работе со страницами, содержащими динамические секции и\или элементы которые могут появлятся во время работы со страницей (например, после определённых действий пользователя), а не сразу же после её загрузки. Поскольку в современных веб приложениях такой сценарий достаточно распространён Page Factory используется редко и считается не самым удачным подходом.

1.4. Data Driven Testing

Data Driven Testing (DDT) – подход к созданию/архитектуре автоматизированных тестов (юнит, интеграционных, чаще всего применимо

к backend тестированию), при котором тест умеет принимать набор входных параметров, и эталонный результат или эталонное состояние, с которым он должен сравнить результат, полученный в ходе прогонки входных параметров. Такое сравнение и есть assert такого теста. Притом как часть входных параметров, могут передаваться опции выполнения теста, или флаги, которые влияют на его логику.

Особым плюсом хорошо-спроектированного DDT является возможность ввода входных значений и эталонного результата в виде, удобном для всех ролей на проекте – начиная от мануального тестировщика и заканчивая менеджером (тест менеджером) проекта, и даже, product owner-а (на практике автора такие случаи случались). Соответственно, когда Вы способны загрузить мануальных тестировщиков увеличением покрытия и увеличением наборов данных – это удешевляет тестирование. А также в целом, удобный и понятный формат позволяет более наглядно видеть, что покрыто, а что нет, это – по сути, и есть документация тестирования. К примеру, это может быть XLS файл с понятной структурой (хотя чаще всего properties файла достаточно).

В контексте данной лабораторной работы будет достаточно отделить тестовые данные от самих тестов посредством использования TestNG data provider-ов.

1.5. Allure

Allure framework — это инструмент для построения понятных отчётов автотестов. Это гибкий и легкий инструмент, который позволяет получить не только краткую информацию о ходе выполнения тестов, но и предоставляет всем участникам производственного процесса максимум полезной информации из повседневного выполнения автоматизированных тестов.

Полная документация для Allure: https://docs.qameta.io/allure

Пример Allure отчёта:

Для более тесной привязки Allure к реализации тестов, а также для отображения более подробной информации в отчёте о результатах выполнения тестов используются Allure аннотации.

Аннотации

Allure предоставляет набор аннотаций для отображения боле детальной информации в финальном отчёте. Описание для некоторых поддерживаемых аннотаций можно найти ниже.

@Description — аннотация, размещаемая над тестом или шагом. Позволяет прикрепить описание к тесту или шагу теста. Данная аннотация может принимать параметры «value» — текст описания и «useJavaDoc». При установке значения «true» последнему параметру, в качестве описания будет браться джавадок, размещенный над методом.

@Feature — аннотация, размещаемая над тестом. Позволяет группировать тесты по проверяемому функционалу. Данная аннотация принимает параметр «value» — наименование функционала.

@Story — аннотация, размещаемая над тестом. Позволяет группировать тесты по User story. Данная аннотация принимает параметр «value» — наименование User story.

```
@Feature(value = "Simple math operations")
```

```
@Story(value = "Addition of numbers")
@Test
public void sumTest() {
    Steps.checkSummationStep(5, 4, 9);
}
```

@Flaky — аннотация, размещаемая над тестом. Позволяет пометить автотест как нестабильный. Если автотест падает хотя бы в одном перезапуске (например, папка «target» между прогонами одного и того же теста не очищается) — в отчете на данном тесте вы увидите знак бомбы. Кроме того, данной аннотацией можно помечать классы с нестабильными тестами.

@Severity — аннотация, размещаемая над тестом. Позволяет указать уровень критичности функционала, проверяемого автотестом. Данная аннотация принимает параметр «value», который может быть одним из элементов enum SeverityLevel: BLOCKER, CRITICAL, NORMAL, MINOR или TRIVIAL.

@Issue — аннотация, размещаемая над тестом. Позволяет линковать автотесты с заведенными Issue. Данная аннотация принимает параметр «value», в котором указывается ID дефекта в баг-треккинговой системе.

@TmsLink — аннотация, размещаемая над тестом. Позволяет линковать автотесты с тестовыми кейсами, шаги которых описаны в системах управления тестированием. Данная аннотация принимает параметр «value», в котором указывается ID теста в системе управления тестированием.

@Attachment - применяется к методу, возвращающему данные которые необходимо прикрепить к отчёту. При использовании данного метода считанная информация будет добавлена в отчет в виде файла с соответствующим расширением. Часто применение это аннотации может быть удобно для прикрепления скриншотов экрана в случае падения теста.

Пример:

```
 @Attachment(value = "Вложение", type = "application/json",
 fileExtension = ".txt")
public static byte[] getBytesAnnotationWithArgs(String resourceName)
 throws IOException {
```

```
    return Files.readAllBytes(Paths.get("src/main/resources",
 resourceName));
}
```

Сборка отчёта

Сначала необходимо добавить в pom.xml вашего проекта allure-maven плагин. Для этого модифицируем секцию build следующим образом:

```
 <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-surefire-plugin</artifactId>
                <version>2.20</version>
                <configuration>
                    <argLine>
                        -javaagent:"${settings.localRepository}/
org/aspectj/aspectjweaver/${aspectj.version}/aspectjweaver-
${aspectj.version}.jar"
                    </argLine>
                </configuration>
                <dependencies>
                    <dependency>
                        <groupId>org.aspectj</groupId>
                        <artifactId>aspectjweaver</artifactId>
                        <version>${aspectj.version}</version>
                    </dependency>
                </dependencies>
```

```
            </plugin>
            <plugin>
                <groupId>io.qameta.allure</groupId>
                <artifactId>allure-maven</artifactId>
                <configuration>
                    <reportVersion>2.6.0</reportVersion>
                </configuration>
            </plugin>
        </plugins>
    </build>
```

Затем надо убедиться, что автотесты, по которым вы хотите получить отчет, уже выполнены (прогнать тесты можно командой mvn clean test) и в папке «target» вашего проекта присутствует «allure-results» с исходными файлами для построения отчета.

Выполнить команду

```
mvn allure:serve
```

Пример POM для подключения Allure можно посмотреть здесь: https://github.com/pavelzhogolev/testng-sample-project/blob/ui_tests_enhanced_with_spring)

2. Задания

Требования к заданиям: Задания 1 и 2 должны соответствовать следующим требованиям:

- не допускается использование статических ожиданий (sleep)
- обязательно использование Page Object паттерна в реализации;
- тестовые данные должны быть отделены от тестового сценария в соответствии с концепцией DDT подхода (достаточно будет использовать передачу тестовых данных в тест из TestNG data provider-a)

опционально:

- использовать отдельный класс для хранения параметров пользователя, необходимых для регистрации (например, класс Account) и использовать её в тестах;

- вынести все общие параметры, широко используемые в тестах в property файлы.

Пример реализации тестов для пользовательского интерфейса можно посмотреть здесь: https://github.com/pavelzhogolev/testng-sample-project/tree/ui_tests_enhanced

Задание 1

Напишите тест, проверяющий успешную регистрацию нового пользователя в тестовом онлайн магазине: http://automationpractice.com Необходимо проверить создание пользователя с заполнением всех полей профиля пользователя. Примерный сценарий:

1. Перейти на главную страницу сайта;
2. Нажать "Sign in" на верхней панели сайта;
3. Ввести корректный email в форму для создания нового пользователя и подтвердить создание пользователя;
4. Заполнить полную форму данными пользователя;
5. Нажать кнопку "Register";
6. Проверить что пользователь был успешно создан.

Представленный сценарий можно разбить на необходимое количество тестовых шагов на ваше усмотрение.

Задание 2

Напишите негативный тест, проверяющий сценарий с попыткой регистрации пользователя без указаняи одного или всех обязательных полей для формы с полными данными о пользователе. Примерный сценарий:

1. Перейти на главную страницу сайта;
2. Нажать "Sign in" на верхней панели сайта;
3. Ввести корректный email в форму для создания нового пользователя и подтвердить создание пользователя;
4. Заполнить полную форму данными пользователя пропустив одно или все поля обязательные к заполнению;

5. Нажать кнопку "Register";
6. Проверить что наверху страницы отобразился список ошибок соответствующий незаполненным полям.

Примечание: проверить надо не только наличие каких-либо ошибок, но и их соответствие тем полям которые были незаполнены.

Задание 3 (по желанию) Реализуйте поддержку генерации автоматических отчётов о результатах выполнения ваших тестов с использованием Allure фреймворка. Постарайтесь использовать такие Allure аннотации как:

- @Severity
- @Description
- @DisplayName
- @Attachment