ЛАБОРАТОРНА РОБОТА №1: АВТОМАТИЗАЦІЯ ТЕСТУВАННЯ З SELENIUM WEBDRIVER

1.1 Мета роботи: створення та налаштування тестового середовища з нуля, а також написання та запуск першого автоматизованого тесту, що демонструє роботу з основними типами локаторів, використовуючи принципи повторного використання коду.

1.2 Теоретичні відомості.

У сфері тестування програмного забезпечення **мова програмування Java** займає особливе місце завдяки своїй універсальності, об'єктно-орієнтованій природі та широкій підтримці з боку спільноти розробників. Її застосування в процесах валідації й верифікації програмного коду стало невід’ємною частиною забезпечення якості програмних продуктів, особливо в умовах зростаючих вимог до надійності та масштабованості систем.

Java є однією з найпопулярніших мов серед автоматизованого тестування. Це зумовлено не лише її сумісністю з численними фреймворками та бібліотеками, орієнтованими на тестування, але й здатністю інтегруватися з різноманітними середовищами розробки та інструментами безперервної інтеграції. Тестувальники, які обирають Java як основну мову для автоматизації, мають змогу створювати гнучкі, розширювані та підтримувані тестові рішення, що особливо важливо в умовах тривалого життєвого циклу програмного забезпечення.

Крім того, Java вирізняється високим рівнем читабельності та структурованості коду, що полегшує процес розробки тестових сценаріїв і підвищує ефективність колективної роботи. Завдяки статичній типізації ця мова сприяє ранньому виявленню помилок ще на етапі компіляції, що знижує ризики виникнення критичних збоїв під час виконання тестів. Також не можна оминути увагою кросплатформеність Java, яка дозволяє забезпечити стабільне функціонування тестових рішень на різних операційних системах і

апаратних платформах.

У контексті розробки тестів для веб-додатків або розподілених систем Java демонструє високу ефективність завдяки своїй екосистемі, що включає численні утиліти для створення, запуску та моніторингу тестів. Це, зокрема, полегшує реалізацію комплексного підходу до тестування, охоплюючи як модульний, так і інтеграційний рівні перевірки функціональності.

Роль Java у тестуванні програмного забезпечення визначається не лише її технічними характеристиками, але й здатністю забезпечити високу якість та надійність кінцевого продукту. З урахуванням динаміки розвитку IT-галузі та вимог до гнучкості розробки, застосування Java в тестуванні є стратегічно доцільним і виправданим вибором.

У контексті тестування програмного забезпечення **Maven** виступає не просто як інструмент керування проєктами, а як фундаментальний компонент, що забезпечує структурованість, повторюваність та стандартизацію процесів збірки й автоматизації. Його застосування суттєво спрощує управління залежностями, конфігурацією середовища та інтеграцією інструментів тестування, що має особливе значення в умовах складних та масштабних проєктів.

Однією з ключових переваг Maven у тестуванні є можливість централізованого управління бібліотеками та фреймворками, які використовуються в автоматизованих тестах. Завдяки декларативному підходу до опису конфігурації через файл pom.xml, забезпечується прозорість і передбачуваність структури проєкту, що дозволяє уникнути конфліктів версій залежностей та сприяє швидкому налаштуванню середовища новими учасниками команди.

У системах автоматизації тестування Maven виконує роль посередника між кодом і тестовою інфраструктурою. Його інтеграція з популярними фреймворками, такими як JUnit або TestNG, дозволяє виконувати тести як окремий етап збірки або в рамках CI/CD-процесу. Це особливо важливо для підтримання якості коду в реальному часі, коли зміни оперативно

перевіряються автоматизованими тестами ще до їхнього потрапляння в основну гілку розробки.

Ще одним важливим аспектом використання Maven у тестуванні є стандартизація структури проєкту. Такий підхід не лише полегшує навігацію та супровід тестового коду, але й робить можливою повторну використання компонентів, що значно підвищує ефективність командної роботи. Крім того, підтримка плагінів, орієнтованих на аналіз якості коду, покриття тестами та генерацію звітності, сприяє побудові цілісного процесу контролю якості.

Maven у сфері тестування програмного забезпечення є не просто інструментом автоматизації, а стратегічним засобом, що дозволяє забезпечити системний, відтворюваний та масштабований підхід до управління тестовим процесом. Його використання не тільки покращує якість продукту, але й оптимізує ресурси команди, знижуючи витрати на обслуговування та супровід автоматизованої інфраструктури.

У процесі тестування програмного забезпечення інтегроване середовище розробки **IntelliJ IDEA** посідає важливе місце завдяки своїй функціональності, зручності та високому рівню інтеграції з інструментами, необхідними для автоматизації та аналізу якості коду. Це середовище забезпечує не лише ефективну розробку програмного забезпечення, але й сприяє побудові цілісного циклу тестування в межах одного інтерфейсу, що значно підвищує продуктивність і скорочує час на виконання рутинних операцій.

Однією з головних переваг IntelliJ IDEA для тестувальників є її глибока інтеграція з популярними фреймворками тестування, зокрема JUnit і TestNG. Завдяки цьому користувач може легко створювати, запускати та налагоджувати тестові сценарії безпосередньо зі свого робочого середовища. Автоматичне розпізнавання тестових класів, можливість параметризації запуску, відображення результатів у структурованому форматі та підтримка відлагодження тестів у реальному часі створюють умови для гнучкого й контрольованого тестового процесу.

Крім того, IntelliJ IDEA підтримує інтеграцію з системами керування проєктами, такими як Maven і Gradle, що забезпечує автоматичне підключення залежностей, конфігурацію середовища та полегшене управління зовнішніми бібліотеками. Це особливо важливо для тестування складних систем, де велика кількість модулів потребує узгодженого підходу до управління їх взаємозв'язками.

Ще одним аспектом, що заслуговує на увагу, є функції інтелектуального доповнення коду, рефакторингу, навігації та аналізу, які суттєво спрощують написання й підтримку тестів. Система підказок, перевірка правильності синтаксису в режимі реального часу та миттєве виявлення помилок дозволяють підвищити якість тестового коду та уникнути багатьох типових помилок ще на етапі розробки.

IntelliJ IDEA виступає не лише як інструмент для написання коду, а як повноцінне середовище для супроводу всього життєвого циклу тестування. Його використання дозволяє досягти високої ефективності, забезпечити глибокий контроль над тестовим процесом і створити сприятливі умови для підтримки якості програмного продукту на всіх етапах розробки.

Поняття **залежностей** у тестуванні програмного забезпечення має критичне значення для побудови стійкої, керованої та масштабованої тестової інфраструктури. Йдеться передусім про зовнішні компоненти, бібліотеки, фреймворки та інструменти, які використовуються для створення, виконання та аналізу тестів. Залежності формують основу середовища, в якому функціонують автоматизовані тести, і водночас можуть істотно впливати як на стабільність, так і на відтворюваність результатів тестування.

Використання залежностей дозволяє уникнути дублювання коду та пришвидшити процес розробки тестів, оскільки велика кількість повторюваних або стандартних функцій уже реалізована в готових бібліотеках. Зокрема, завдяки таким залежностям тестувальники мають змогу застосовувати готові рішення для написання assertions, генерації тестових даних, емуляції поведінки зовнішніх сервісів або реалізації звітності. Проте

кожна нова залежність водночас створює потенційне джерело нестабільності, особливо коли вона має власні версійні обмеження або залежить від інших бібліотек, що конфліктують між собою.

Інструменти на кшталт Maven відіграють ключову роль у керуванні залежностями, автоматизуючи процес їх завантаження, оновлення та інтеграції в проєкт. За допомогою декларативного опису конфігурації залежностей користувачі можуть точно вказувати, які версії бібліотек необхідні, що дозволяє досягти передбачуваності під час виконання тестів і мінімізувати ризик "несумісних оновлень".

Крім зовнішніх, у тестуванні існує також поняття внутрішніх залежностей – таких, що виникають між самими модулями, класами чи методами всередині тестової системи. Надмірна взаємозалежність між тестами або відсутність ізоляції від коду тестування може призводити до каскадних збоїв, коли зміна в одному компоненті тягне за собою порушення роботи кількох тестових сценаріїв. Тому сучасні підходи до побудови тестів акцентують увагу на ізоляції, використанні mock-об'єктів та принципі інверсії залежностей. По суті ж, в першій лабораторній роботі, залежності – це бібліотеки, які наш проєкт використовуватиме. Кожна з них відповідає за певну функцію, яку ми не будемо реалізовувати самостійно:

- `selenium-java`: основна бібліотека `Selenium WebDriver`. Вона надає необхідні класи та методи для взаємодії з браузером, наприклад, `WebDriver`, `WebElement`, а також методи для пошуку елементів (`findElement`) та виконання дій (наприклад, `click`, `sendKeys`).

- `webdrivermanager`: ця бібліотека є дуже важливою, оскільки вона автоматично завантажує та налаштовує драйвер браузера (наприклад, `chromedriver`). Без неї вам довелося б вручну завантажувати драйвер і вказувати шлях до нього, що є громіздким процесом.

- `junit-jupiter-api`: це фреймворк для написання та запуску тестів. Він дозволяє вам позначати методи як тести за допомогою анотації `@Test` і надає класи для перевірки результатів, такі як

`Assertions.assertEquals`.

- `selenide`: це фреймворк, що спрощує роботу з Selenium. **Selenide** надає чистий і більш читабельний API, а також має вбудовані "розумні" очікування, що значно прискорює написання стабільних тестів.

Selenide є надбудовою над Selenium, що надає простий та інтуїтивно зрозумілий API. Основна відмінність полягає в тому, що Selenide приховує багато рутинних деталей, роблячи код значно чистішим.

У сфері автоматизованого тестування веб-застосунків бібліотека Selenide посідає окреме місце як інструмент, що поєднує простоту використання з високою функціональністю. Її поява стала відповіддю на потребу в ефективнішій та стабільнішій альтернативі класичному підходу на основі Selenium WebDriver, який, хоча й потужний, часто вимагає значних зусиль для обробки очікувань, стабілізації тестів і взаємодії з веб-елементами.

Selenide базується на тих самих принципах, що й Selenium, однак значно спрощує щоденні завдання тестувальника. Зокрема, вона автоматично реалізує очікування елементів DOM, що знижує кількість помилок, пов’язаних із асинхронною природою веб-додатків. Завдяки цьому зменшується обсяг ручної роботи, необхідної для реалізації надійного тесту, а код стає більш лаконічним і читабельним. Саме читабельність є однією з головних переваг Selenide: тести, написані з її використанням, максимально наближені до природної мови, що полегшує їх супровід та верифікацію навіть для фахівців, які не займаються програмуванням на постійній основі.

Інтеграція Selenide з сучасними інструментами тестування, такими як JUnit, TestNG, Allure, а також її сумісність із Maven, забезпечує повноцінну автоматизацію всього процесу: від запуску тестів до генерації звітності. Бібліотека також підтримує роботу в різних браузерах, легко конфігурується й дозволяє здійснювати паралельне тестування, що є ключовим у великих проєктах із високими вимогами до продуктивності.

Крім того, Selenide має низку вбудованих механізмів для роботи з

нестандартними веб-елементами, pop-up вікнами, алертами та iframes, що значно розширює її функціональні можливості у порівнянні з базовим WebDriver API. Особливу увагу також варто звернути на її стабільність: бібліотека спроєктована так, щоб автоматично обробляти типові збої, спричинені непередбачуваною поведінкою інтерфейсу користувача.

Виклик драйверу: Selenide автоматично керує драйвером. Вам не потрібно писати код для ініціалізації `WebDriverManager` або `ChromeDriver`. Просто викличте метод `open("URL")` – і Selenide зробить усе за вас.

Локатори $() та $$(): Selenide використовує синтаксис, схожий на `jQuery`, для пошуку елементів.

`$(selector)`: цей метод знаходить один елемент. У дужках ви можете використовувати CSS-селектори. Наприклад, `$("#username")` знайде елемент за `ID`, а `$(".btn-primary")` за класом. Selenide також дозволяє використовувати стандартні локатори `By` з Selenium, наприклад, `$(By.name("q"))`.

`$$(selector)`: цей метод знаходить список елементів. Він повертає колекцію, з якою ви можете працювати далі, наприклад, перевіряти її розмір.

Перевірки та очікування: Selenide має вбудовані "розумні" очікування. Не потрібно додавати `Thread.sleep()` або писати явні очікування. Selenide сам чекає, поки елемент з'явиться, стане видимим або буде доступний для взаємодії. Це робить тести стабільнішими. Перевірки виконуються за допомогою методів `shouldHave()`, `shouldBe()`, `shouldNotHave()`, які роблять асерти більш читабельними.

Ключові поняття:

asserts – це твердження, які використовуються для перевірки, чи фактичний результат виконання тесту відповідає очікуваному. Без них тест не має сенсу.

assertEquals(expected, actual) – найбільш поширений тип assert, що порівнює два значення. Якщо вони не збігаються, тест вважається провaленим.

@BeforeEach – метод, який виконується перед кожним тестом. Ідеальне місце для ініціалізації драйвера.

@AfterEach – метод, який виконується після кожного тесту. Використовується для очищення, наприклад, для закриття браузера.

1.3 Хід роботи.

1) Налаштування середовища. Встановлення всіх необхідних інструментів, які є фундаментом для автоматизації.

Крок 1: Встановлення Java Development Kit (JDK). Перейдіть на офіційний сайт Oracle за посиланням https://www.oracle.com/java/technologies/downloads/ або на сайт OpenJDK https://openjdk.org/, щоб завантажити останню стабільну версію JDK. Виберіть інсталятор, який відповідає вашій операційній системі (Windows, macOS, Linux). Запустіть інсталятор і слідуйте інструкціям на екрані. Зазвичай достатньо натискати «Далі». Після встановлення необхідно налаштувати системну змінну `JAVA_HOME`. Це важливо, оскільки інші програми (наприклад, `Maven`) будуть використовувати цю змінну для пошуку Java.

На Windows: відкрийте меню «Пуск», знайдіть «Змінити змінні середовища для поточного облікового запису» або «Система» → «Додаткові параметри системи» → «Змінні середовища». Створіть нову змінну користувача з назвою JAVA_HOME і вкажіть шлях до встановленої папки JDK (наприклад, C:\Program Files\Java\jdk-17). Після цього змініть змінну Path, додавши до неї %JAVA_HOME%\bin. .

Перевірка: відкрийте командний рядок (`cmd`) або термінал і виконайте команду: `java -version` Ви повинні побачити номер версії Java, що підтверджує успішну установку.

Крок 2: Встановлення Apache Maven.

Завантаження Maven: перейдіть на офіційний сайт https://maven.apache.org/download.cgi і завантажте архів (зазвичай це ZIP-файл). Розпакуйте його в будь-яку зручну директорію на вашому комп'ютері

(наприклад, C:\maven). Налаштуйте системну змінну M2_HOME, вказавши шлях до розпакованої папки Maven (наприклад, C:\maven\apache-maven-3.8.4). Додайте %M2_HOME%\bin до змінної Path після шляху до Java.

Відкрийте новий командний рядок і виконайте: `mvn -v` . З'явиться інформація про версію Maven, що свідчить про успішне налаштування.

Крок 3: Встановлення IntelliJ IDEA.

Завантажте безкоштовну **Community Edition** з офіційного сайту https://www.jetbrains.com/idea/download/. Встановіть IDE: запустіть інсталятор і дотримуйтесь інструкцій.

Створіть Maven-проєкт**:** запустіть IntelliJ, виберіть «New Project», далі оберіть «Maven». Дайте проєкту назву та збережіть його.

Крок 4: Додавання залежностей Selenium та JUnit. У IntelliJ відкрийте файл pom.xml. Додайте залежності: вставте наступний код у блок <project>.

```
XML
<dependencies>
    <dependency>
        <groupId>org.seleniumhq.selenium</groupId>
        <artifactId>selenium-java</artifactId>
        <version>4.1.2</version>
    </dependency>
    <dependency>
        <groupId>io.github.bonigarcia</groupId>
        <artifactId>webdrivermanager</artifactId>
        <version>5.0.3</version>
    </dependency>
    <dependency>
        <groupId>org.junit.jupiter</groupId>
        <artifactId>junit-jupiter-api</artifactId>
        <version>5.8.1</version>
        <scope>test</scope>
    </dependency>
    <!-- Додаткова залежність для Selenide -->
    <dependency>
        <groupId>com.codeborne</groupId>
        <artifactId>selenide</artifactId>
        <version>5.25.0</version>
        <scope>test</scope>
    </dependency>
</dependencies>
```

Збережіть файл, і `IntelliJ` автоматично завантажить всі необхідні

бібліотеки.

2) **Завдання 1: створення та виконання автотесту.**

Потрібно написати автоматизований тест, який перевіряє коректність відображення ключових елементів на головній сторінці Google.

Крок 1: Відкрити браузер та перейти на сторінку https://www.google.com.

Крок 2: Перевірити, що заголовок сторінки відповідає очікуваному.

Крок 3: За допомогою різних локаторів (ім'я, клас, тег, CSS, XPath) знайти ідентифікатори основних елементів, таких як поле пошуку, кнопки та посилання.

Крок 4: Перевірити, що кожен знайдений елемент є видимим і має очікуване значення.

Крок 5: Вивести результати пошуку в консоль.

Крок 6: Закрити браузер.

Створення базового класу для тестування.

На цьому етапі ми створимо базовий клас, який відповідатиме за ініціалізацію та закриття `WebDriver`. Це є найкращою практикою, оскільки дозволяє уникнути дублювання коду в кожному тестовому класі.

У директорії src/test/java створіть новий Java-клас BaseTest.java.

Напишіть код для BaseTest. Цей клас буде містити логіку для налаштування (setup) і очищення (teardown) середовища, використовуючи анотації JUnit.

```
Java
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import io.github.bonigarcia.wdm.WebDriverManager;

public abstract class BaseTest {

    protected WebDriver driver;

    @BeforeEach
    public void setUp() {
        // Налаштування WebDriverManager для автоматичного
```

```
завантаження драйвера
        WebDriverManager.chromedriver().setup();

        // Ініціалізація нового екземпляра ChromeDriver
        driver = new ChromeDriver();
    }

    @AfterEach
    public void tearDown() {
        if (driver != null) {
            // Затримка на 5 секунд для демонстрації результату
            try {
                Thread.sleep(5000);
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            // Закриття браузера
            driver.quit();
        }
    }
}
```

У тій же директорії створіть новий Java-клас з назвою `GoogleLocatorsTest.java`.

Скопіюйте наступний код у створений файл. Зверніть увагу, що клас `GoogleLocatorsTest` тепер успадковує функціональність `BaseTest` за допомогою ключового слова `extends`.

```
Java
import org.junit.jupiter.api.Test;
import org.openqa.selenium.By;
import org.openqa.selenium.WebElement;
import static org.junit.jupiter.api.Assertions.assertEquals;
import java.util.List;

public class GoogleLocatorsTest extends BaseTest {

    @Test
    public void demonstrateLocatorsWithAssertions() {
        // Відкриття веб-сайту
        driver.get("https://www.google.com");

        // Перевірка, що заголовок сторінки відповідає
очікуваному
        String expectedTitle = "Google";
        String actualTitle = driver.getTitle();
        assertEquals(expectedTitle, actualTitle, "Заголовок
сторінки не відповідає очікуваному!");

        System.out.println("Демонстрація локаторів на сторінці
```

```
Google:");
        System.out.println("-----------------------------------
----------");

        // 1. Пошук за ім'ям (By.name)
        WebElement searchBoxByName =
driver.findElement(By.name("q"));
        System.out.println("1. Знайшли елемент за 'name': " +
searchBoxByName.getAttribute("outerHTML"));

        // 2. Пошук за ім'ям класу (By.className)
        WebElement searchButtonByClass =
driver.findElement(By.className("gNO89b"));
        System.out.println("2. Знайшли елемент за 'className': "
+ searchButtonByClass.getAttribute("value"));

        // Додаткова перевірка: перевіряємо, чи текст на кнопці
вірний
        assertEquals("Пошук Google",
searchButtonByClass.getAttribute("value"), "Текст на кнопці
пошуку не відповідає очікуваному!");

        // 3. Пошук за тегом (By.tagName)
        List<WebElement> allLinks =
driver.findElements(By.tagName("a"));
        System.out.println("3. Знайшли " + allLinks.size() + "
посилань за 'tagName'.");

        // 4. Пошук за CSS-селектором (By.cssSelector)
        WebElement searchButtonByCss =
driver.findElement(By.cssSelector("input[name='btnK']"));
        System.out.println("4. Знайшли елемент за 'cssSelector':
" + searchButtonByCss.getAttribute("value"));

        // 5. Пошук за XPath (By.xpath)
        WebElement searchButtonByXPath =
driver.findElement(By.xpath("//input[@name='btnK']"));
        System.out.println("5. Знайшли елемент за 'xpath': " +
searchButtonByXPath.getAttribute("value"));

        System.out.println("-----------------------------------
----------");
        System.out.println("Всі локатори успішно
продемонстровані!");
    }
}
```

Запустіть тест: у `IntelliJ` клацніть правою кнопкою миші на класі `GoogleLocatorsTest` і виберіть `"Run..."`.

Завдання 2: автоматизація з Selenide.

Опис роботи з Selenide: у директорії `src/test/java` створіть новий Java-клас `GoogleSelenideTest.java`.

Скопіюйте цей код у новий файл.

```
Java
import com.codeborne.selenide.CollectionCondition;
import com.codeborne.selenide.Condition;
import com.codeborne.selenide.SelenideElement;
import org.junit.jupiter.api.Test;

import static com.codeborne.selenide.Selenide.*;

public class GoogleSelenideTest {

    @Test
    public void demonstrateLocatorsWithSelenide() {
        // Відкриття веб-сайту. Selenide автоматично завантажує
драйвер.
        open("https://www.google.com");

        System.out.println("Демонстрація локаторів з
Selenide:");
        System.out.println("-----------------------------------
----------");

        // Перевірка заголовка сторінки за допомогою вбудованих
асертів Selenide
        webdriver().shouldHave(Condition.title("Google"));

        // 1. Пошук за ім'ям (By.name)
        SelenideElement searchBoxByName = $("[name='q']");
        System.out.println("1. Знайшли елемент за 'name' та
перевірили, що він видимий.");

        // 2. Пошук за ім'ям класу (By.className)
        SelenideElement searchButtonByClass = $(".gNO89b");
        System.out.println("2. Знайшли елемент за
'className'.");

        // 3. Пошук за тегом (By.tagName)

$$("a").shouldBe(CollectionCondition.sizeGreaterThan(10));
        System.out.println("3. Знайшли посилання за 'tagName' та
перевірили, що їх більше 10.");

        // 4. Пошук за CSS-селектором (By.cssSelector)
        SelenideElement searchButtonByCss =
$("input[name='btnK']");
        System.out.println("4. Знайшли елемент за
'cssSelector'.");
```

```
        // 5. Пошук за XPath (By.xpath)
        SelenideElement searchButtonByXPath =
$x("//input[@name='btnK']");
        System.out.println("5. Знайшли елемент за 'xpath'.");

        // Приклад дії та асерту
        searchBoxByName.setValue("Selenide");
        searchButtonByCss.click();

        webdriver().shouldHave(Condition.title("Selenide - Пошук
Google"));

        // Selenide автоматично закриває драйвер в кінці тесту.

        System.out.println("-----------------------------------
----------");
        System.out.println("Всі локатори успішно
продемонстровані!");
    }
}
```

Запустіть тест: клацніть правою кнопкою миші на класі `GoogleSelenideTest` і виберіть "Run...". Результати виконання та висновки стосовно створення архітектури для керування драйвером у будь-якому тестовому класі відобразити в звіті.

Контрольні питання:

1. Навіщо потрібна змінна середовища JAVA_HOME?
2. Яку роль виконує Maven у проєкті автоматизації?
3. Чим відрізняється залежність selenium-java від webdrivermanager?
4. Яке призначення анотації @Test у JUnit?
5. Чому варто використовувати @BeforeEach і @AfterEach?
6. Що таке локатор і навіщо він потрібен?
7. Який локатор вважається найбільш стабільним і чому?
8. Який локатор найбільш гнучкий, але може бути "крихким"?
9. Чому By.id() краще, ніж By.className() для пошуку унікального елемента?
10. Як Selenide автоматично керує драйвером?
11. Чим відрізняються методи Selenide $ та $$?

12. Навіщо потрібні asserts у тесті?

13. Що таке assertEquals() і коли його використовують?

14. Як Selenide замінює Thread.sleep()?

15. Наведіть приклад assert з JUnit для перевірки заголовка сторінки.

16. Який рядок коду використовується для відкриття сторінки в Selenide?

17. Як створити базовий клас, від якого будуть успадковувати всі тести?

18. Чому важливо використовувати базовий клас у фреймворку автоматизації?

19. Чи можна використовувати стандартні локатори By із Selenium у Selenide?

20. Які переваги Selenide порівняно з "чистим" Selenium?

Інструкція для захисту:

1) Уважно ознайомитися з даними методичними вказівками та теоретичними відомостями.

2) Встановити необхідне програмне забезпечення та виконати 2 завдання.

3) Відповісти на контрольні питання.

4) Оформити звіт з лабораторної роботи (у кожного студента свій), в якому мають бути титульний лист (МОН, назва університету, кафедри, лабораторна робота №1, назва, група та ПІБ студента, місто та рік), в середині: мета роботи, хід роботи, результати виконання завдань (зі скріншотами та коротким описом), висновки (що саме було зроблено в роботі, пропозиції щодо покращення).

5) Захистити звіт у викладача, отримати оцінку.