Automated Tests Using Selenium Framework

Josef Horalek, Patrik Urbanik, and Vladimir Sobeslav(✉)

Faculty of Informatics and Management, University of Hradec Kralove, Hradec Kralove, Czech Republic
{josef.horalek,patrik.urbanik,vladimir.sobeslav}@uhk.cz

Abstract. Today, software testing is already an integral part of every software development cycle. However, even this does not guarantee that the final product will be free of software bugs. Testing needs to be implemented from the lowest possible stage of development so that the bugs can be detected as early as possible to reduce the cost of fixing them at later stages. However, this is hampered by the ever-increasing demands for software from customers, increasing testing requirements to the point where it is impossible to meet them all. This is why automated software testing is becoming increasingly popular. Automated test scripts can partially replace manual user testing and help significantly improve the quality of the software. Automated testing using the Selenium framework is the focus of this paper. The functionality and process of testing in Selenium are demonstrated with five testing scripts in a sample scenario.

Keywords: Automation · Automated testing · Selenium framework · Software quality

1 Introduction

The increasingly modern technologies and practices involved in the application development cycle are no guarantee that the software being developed will be bug-free. Therefore, it is very important to implement a testing process early in the development cycle to prevent bugs from being transferred to later phases or even to the final deployment. The very meaning of testing is often distorted. According to [1], testing aims not to prove that the software works correctly, but to increase its value by improving its quality by removing bugs. As a suitable definition, the author states: Testing is the process of executing a program with the intent of finding errors. Thus, the goal of software testing is to find bugs as early as possible, or in other words, to ensure a certain level of quality of the application in development. It is the so-called Quality Assurance (QA) that is of great importance in the corporate competitive environment [2]. According to [3], the meaning of quality, or software quality, can be expressed as: The degree to which the system, process, or component meets the specified requirements.

Software bugs can take many forms. They range from grammatical errors that do not affect the functionality of the software, to logical errors in the code that can result in crashes. It is not always true that an error leads to failure. Therefore, the terms software

C. V. Phan and T. D. Nguyen (Eds.): ICTCC 2022, LNICST 473, pp. 23–40, 2023.
https://doi.org/10.1007/978-3-031-28790-9_3

bug and software failure need to be distinguished. Failure only occurs when certain conditions (and/or errors) are activated. Thus, a failure never has to occur because the errors are never activated [4].

The cost to fix bugs increases in proportion to the stage of development at which the bug is discovered. Examples of well-known and very costly incidents related to development errors are the failed launch of the civilian rocket called Ariane 5 in 1996 [5], and the so-called Y2K Bug [6], which nowadays looks quite ridiculous. Among the more recent ones, the Log4j bug from December 2021, which affects millions of web servers and is described as one of the biggest bugs of recent years, is definitely worth mentioning [7].

The testing process itself can be divided into several categories depending on the focus. For example, based on the layer of the application into frontend and backend testing, then based on the knowledge level of the code into black-box, white-box, and grey-box, and last but not least based on the type of the execution into manual and automated testing [8].

The principle of automated testing is the creation of test scripts that run without human intervention. The tester in this case is responsible for creating and maintaining these scripts and for checking the test results [9]. The main advantages of automated tests are considered to be:

- Reusability of test scripts,
- higher coverage in the application under test,
- increased software quality,
- faster compared to manual testing,
- reduced cost and time of testing,
- much better error detection.

On the other hand, users see disadvantages such as:

- The actual scripting and maintenance,
- the need for even more skilled personnel,
- cannot fully replace manual testing,
- initial high cost of implementation [10].

A large number of aspects influence the effectiveness and sustainability of automated testing. These include the choice of appropriate test automation tools, the test framework used, the planning of automation activities, integration into DevOps at the organizational and technical level, and more.

The growing popularity of automated tests is mainly due to the fact that customers are placing ever greater demands on systems, thus increasing the complexity of the programs being developed. This increases the requirements for testing to the point where it cannot be done manually in terms of human resources, time, and money.

2 Methods and Technologies

The capabilities and effectiveness of automated tests depend on the technologies and methods that are used to run automated tests of the GUI of a corporate web application.

Test development tools can be divided into two categories according to their focus - complex development environments, so-called IDEs (Integrated development environments), and simpler editors. The main differences are in price, system requirements, performance, and the options offered by each environment. As part of the solution, the Visual Studio development environment was used as the main tool for creating automated tests, which has the advantage of a wide range of available extensions and tools. It is possible to download so-called NuGet packages into the project in Visual Studio, where these are basically files that contain compiled code created by another developer and in this way made available for use by other users, which were also used in the development of automatic Selenium tests. Specifically, the following have been used:

- `DotNetSeleniumExtras.WaitHelpers`,
- `Selenium.Support`,
- `Selenium.WebDriver`,
- `Selenium.WebDriver.ChromeDriver`,
- `Xunit`,
- `Xunit.assert`,
- `Xunit.runner.visualstudio`,
- `XunitXml.TestLogger`.

The automated tests use the *XUnit* tool [11], which is an open-source testing framework created by Jim Newkirk, the creator of another testing framework. At the time of the pilot design, the current XUnit version was 2.4.1. This framework tries to take a minimalist approach by scanning all public classes when running tests for methods with the `[Fact]` or `[Theory]` attribute. These attributes identify the method as a test. The `[Fact]` attribute denotes a method that has no arguments on the input. This is a classical test where everything must be specified in the method. In contrast, the `[Theory]` attribute is used for methods when we want to insert specific data into the test. This attribute alone is not enough and another attribute `[InlineData]` must be used with it. This is used to specify the subset of data on which the parameterized tests will be executed.

GitLab was chosen as the pilot testbed for automated tests. Unit tests were run in GitLab pipelines to separate the testing and production environments. All automated tests were run here, both for the frontend and the backend. In total, four environments are used in the development of the application:

1. *Develop* - An environment used mainly for programmers to develop new functionality. Sometimes manual testing is also done here.
2. *Test* - Here automatic tests are run for frontend and backend.
3. *Stag* - This environment is used for user acceptance testing.
4. *Prod* - The environment in which the application runs for live use. This environment, thanks to all the previous tests, should not get any bugs, or as few as possible.

Which environment to run the tests on is set using the `.gitlab-ci.yml` file. This file is used to set up individual pipelines within GitLab. An example of this particular file, for setting up Selenium tests to run automatically, is shown below.

```
stage: qa-test
  image: mcr.microsoft.com/dotnet/sdk
rules:
  - if: $CI_COMMIT_REF_NAME == "master"
      when: on_success
  - when: never
script:
  - dotnet test tests/uitest/Selenium_tests.csproj
  --logger html
artifacts:
  when: always
  paths:
      - ./tests/uitest/TestResults
  expire_in: 1 week
```

This file guarantees to run automatic tests if the name of the branch on which the new version of the application is being deployed is the master. The test environment is located in this branch. For other branches, this job will not run at all. It is also possible to see the artifact settings from this job. An artifact is a file that is created from the output of a job. In this case, it is the HTML report of the test result shown at the end of the article (Fig. 3).

A general flowchart of the solution (Fig. 1), where automated UI tests are run, is shown below. As mentioned, the tests are run automatically when a new version is deployed to the master branch, where the test environment is also set up. If the tests reveal a bug, a hotfix is created by the developer and then the tests are run again. If all is well, a new tag is created to ensure automatic deployment to the stag environment for customer testing. Customers have a certain amount of time, on the order of a few days, to test it. If they confirm that everything works, the version is manually deployed to the production environment for official use.

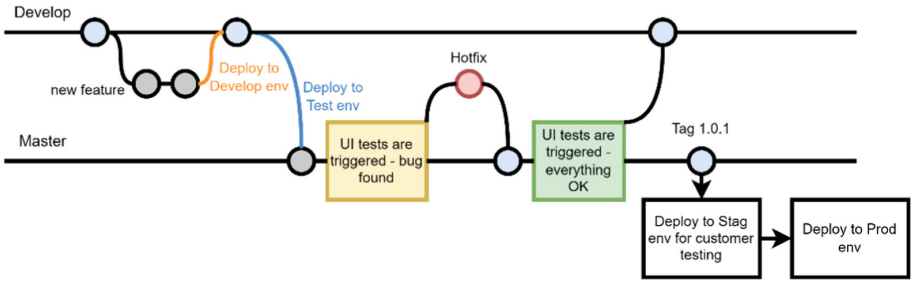

Fig. 1. Running automated UI Tests in GitLab

The *DevTool* [12] is also essential for developing automated tests. It is a set of tools integrated directly into Google Chrome. The most important tab, when creating automated UI tests, is the Elements tab, which displays all the elements of the web page. Here all the needed locators for the elements are retrieved, which are then included in the test script. The other tabs that the DevTool interface offers are:

- *Elements* - displays CSS and page elements,
- *Console* - displays JavaScript messages,
- *Network* - displays network activity,
- *Performance* - displays the loading speed of the web page,
- *Application* - displays all loaded resources, local storage, session storage, cookies, and application cache.

2.1 Selenium Tools

Selenium does not refer to only one particular tool but refers to three different tools, namely *Selenium IDE*, *Selenium WebDriver*, and *Selenium Grid* [13].

Selenium IDE
It is a tool or development environment that is primarily used to develop test scenarios for Selenium tests. It is a simple web browser extension, so it does not require any complex configuration. The Selenium IDE is a graphical environment that records the user's actions in the browser, using pre-existing Selenium commands with the parameters of the web elements that the user clicked on while recording the test. This makes it ideal for users with no experience in developing test cases for web applications [14].

Selenium WebDriver
Using the APIs provided by individual browsers, allows you to automate the functioning of the browser. For web browsers, it defines behavior control interfaces to simulate the behavior as if controlled by a real user [15].

Selenium Grid
The main purpose of Selenium Grid is to run tests in parallel on multiple machines. The grid allows you to run tests created in WebDriver remotely. It can run tests on both physical and virtual machines, and even in parallel for different browsers. So if a test suite contains, for example, fifty tests, using Grid it is possible to reduce the testing time by running a set of ten tests on five different machines in parallel [16].

3 Test Design and Implementation

Now the implementation of specific test cases will be shown. The case scenario is to make sure that the application for ordering and managing physical material for production is working properly and safely. For the purpose of this paper, the arguments of each test case are anonymized with more general variants. Some web elements are also changed in the same way. All modified data is shown in italics in the code samples, and specific values need to be entered instead for the tests to work properly.

3.1 Defining the Test Case

At the beginning of each test, it is absolutely necessary to think about what functionality of the web application we want to test and then design which steps the test will contain to achieve the desired level of testing. For this purpose, it is possible to use the already mentioned Selenium IDE tool or rely on the tester's knowledge of the tested application and let him develop the test case independently. For the needs of the implemented testing, the Selenium IDE tool was used to create a test case, which was comprehensively converted into executable code for use in the automated test. The main points of the test scenario are shown in Table 1.

Table 1. The test case/scenario

Step#	Step detail	Expected results	Test data
1	Open the login page	Display the login form	–
2	Enter a valid username and password	Relevant data entered	*username, password*
3	Click on the "Sign in" button	Successful login of a user	–
4	Search for desired materials	Display details of the entered material	*material*
5	Click on the "Order material" button	View the material order form	–
6	Enter the quantity of material in the "qty" field	The field contains the required data	*quantity*
7	Enter the order identifier in the "Order ID" field	The field contains the required data	*identifier*
8	Enter the desired replenishment date in the "Delivery Date" field	The field contains the required data	*deliveryDate*
9	Click on the "Create Order" button	The field contains the required data	–

3.2 Test Script

The test case project contains three separate classes – `SeleniumConfig.cs`, `Steps.cs`, and `Tests.cs`, which are available at https://github.com/horalekjos/Selenium_test. The `Tests.cs` class contains all the tests that will be run. These tests are composed using individual steps contained in the `Steps.cs` class. This makes it easier to manage the tests in smaller chunks and to make changes. It should always be sufficient to change only part of the code in a single step and not the entire test. Changes in individual steps do not affect other steps of the test.

The `SeleniumConfig` class is described below. All tests are closed in a `try/finally` block. The `try` block contains the test itself. The `finally` block calls the `Quit()` method which closes the browser window and ends the test session. This ensures that the container is always free for the next test and not blocked by the previous one. Each test consists of several smaller steps. The most important parts of the tests are shown below. Configuration file for test job setup in GitLab pipeline is available at `\gitlab\job-config-file\.gitlab-ci.yml`.

Basic Test Configuration

The basic class that is needed for the tests is `SeleniumConfig.cs`. This class is where all the basic settings related to the browser used for testing are made. This is done by creating an instance of the `ChromeOptions` class, which contains the appropriate methods for Chrome-specific settings. For these settings to take effect, a `ChromeOptions` object needs to be added to the WebDriver constructor.

In the code below you can see the two parameters for the settings. The first is the `headless` parameter. This makes the web browser run without a graphical interface. It is useful to use it if the tests are run on a server where this GUI is not visible, so it is not needed. This also saves limited server resources such as RAM or disk space. If new tests are being developed locally, it is preferable not to use this setting, as no video of the test run is available. This can make it difficult to find a possible reason why the test is broken, and the test developer has to go by the logs only.

The next parameter is the `start maximized`, where it is clear from the name that the browser will automatically start in the maximum window size, not just in a reduced form. In total, there are around 140 of these parameters/settings [17]. There is also a variable `hubAddress` in the script that contains the address to which the tests should connect. For local development and testing, they are redirected to port 4444, which exposes a running local docker container containing the Selenium grid. To run tests on a server, for example in Kubernetes clusters, this address must be changed to a specific corresponding IP address.

```
public class SeleniumConfig
{
  String hubAdress = "http://localhost:4444";
  ChromeOptions options = new ChromeOptions();
  public Zalenium()
  {}
  public void PrepareDriver(string name ="SeleniumTest")
  {
    options.AddArguments("--headless");
    options.AddArguments("--start-maximized");
    remoteDriver = new RemoteWebDriver(new Uri
    (hubAdress),options);
  }
}
```

User Login

In the `Steps.cs` class, the `Login` method is created to test the successful login. It is also used in all other tests since each instance of the web browser requires a new login to the application. For this reason, the login name and password are not provided as parameters, as it is not expected that different values will be used in each test.

```
public void Login(RemoteWebDriver driver)
{
  var wait = new WebDriverWait(driver, new TimeSpan(0, 0, 10));
  wait.Until(SeleniumExtras.WaitHelpers.ExpectedConditions.
  ElementToBeClickableBy.XPath("//button/span[text()=
  'SignIn']"))).Click();
  driver.FindElement(By.XPath("//span[text()=
  'CompanyUsers']")).Click();
driver.FindElement(By.Id("userNameInput")).SendKeys("username");
driver.FindElement(By.Id("passwordInput")).SendKeys("password");
driver.FindElement(By.Id("submitButton")).Click();
}
```

If we would like to verify that the user login was successful, we can add, for example, the code below, which checks the condition that the correct username is displayed in the required element. This element is only displayed after a successful login.

```
wait.Until(SeleniumExtras.WaitHelpers.ExpectedConditions.TextTo
BePresentInElementLocated(By.XPath("//button[3]/span[1]"),
"username"));
```

Ordering Materials for Production

The "*MaterialOrder_RequiredFields*" test checks the correct entry of data into the individual fields of the form for ordering physical material that is used in production after delivery. The form contains multiple fields. This test checks only the required fields to order the material. The test accepts the parameters `material`, `supplier`, `numberOfWeeks`, `view`, `quantity`, `identifier`, `deliveryDate`. The XUnit testing framework provides an InlineData attribute that passes specific values, provided by us, to our test. The order of values in this attribute corresponds to the order in which the parameters are declared in the function or test.

Next, the instance from the `Steps.cs` class is declared. This allows to access the methods of this class that make up the steps of the test. As you can see from the sample below, there are a total of five steps (the name of the steps should make it clear at a glance what each one does):

1. Login,
2. Dashboard_SearchMaterial,
3. EnterMaterialOrder_RequiredFields,
4. SearchForMaterialOrder,
5. DeleteMaterialOrder.

```
[Theory(DisplayName = "Material order (Required fields)")]
[InlineData("material", "supplier", "numberOfWeeks", "view",
"quantity", "identifier", "deliveryDate")]

public void MaterialOrder_RequiredFields(string partNumber,
string vendorCode, string numberOfWeeks, string viewName, string
quantity, string invoiceNO, string ETAdate){
  try
  {
    Steps step = new Steps();
    fix.remoteDriver.Navigate().GoToUrl(url);
    step.Login(fix.remoteDriver);
    step.Dashboard_SearchMaterial(fix.remoteDriver,
    material, supplier, numberOfWeeks, view);
    step.EnterMaterialOrder_RequiredFields
   (fix.remoteDriver, quantity, identifier, deliveryDate);
    step.SearchForMaterialOrder(fix.remoteDriver, identifier);
    step.DeleteMaterialOrder(fix.remoteDriver);
  }
finally{
    fix.remoteDriver.Quit();
  }
}
```

The example below shows an example of one step of the test, namely the step "*EnterMaterialOrder_RequiredFields*". In the first stage of the test, an instance of the `WebDriverWait` class is declared. The constructor of this class requires two parameters - which driver to wait for and how long to wait for each element. In this case, the wait will be a maximum of 10 s. If a specific condition is not met by this time, the test will exit with an error. Here, the condition must be met to be able to click on the selected element during this time. Successively, values are sent to each element of the form. After the values have been entered, the button to create the material order is clicked. At the end of this step, the successful creation of the order is checked by the displayed text, which must be "Material order created".

```
public void EnterMaterialOrder_RequiredFields
(RemoteWebDriver driver, string quantity, string identifier,
string deliveryDate){
(RemoteWebDriver driver, string quantity, string identifier,
string deliveryDate)
{
var wait = new WebDriverWait(driver, newTimeSpan(0, 0, 10));
wait.Until(SeleniumExtras.WaitHelpers.ExpectedConditions.Element
ToBeClickable(By.CssSelector("mat-icon[aria-label='Open new
order form']"))).Click();
wait.Until(SeleniumExtras.WaitHelpers.ExpectedConditions.Element
ToBeClickable(By.XPath("//input[@placeholder='Qty']"))).Click();
wait.Until(SeleniumExtras.WaitHelpers.ExpectedConditions.Element
ToBeClickable(By.XPath("//input[@placeholder='Qty']"))).SendKeys
(quantity);
wait.Until(SeleniumExtras.WaitHelpers.ExpectedConditions.Element
ToBeClickable(By.XPath("//input[@placeholder=OrderId']"))).
SendKeys(identifier);
wait.Until(SeleniumExtras.WaitHelpers.ExpectedConditions.Element
ToBeClickable(By.XPath("//input[@placeholder='DeliveryDate']")))
.Click();
Actions action = new Actions(driver);
action.SendKeys(Keys.Escape).Perform();
wait.Until(SeleniumExtras.WaitHelpers.ExpectedConditions.Element
ToBeClickable(By.XPath("//input[@placeholder='DeliveryDate']")))
.SendKeys(deliveryDate);
wait.Until(SeleniumExtras.WaitHelpers.ExpectedConditions.Element
ToBeClickable(By.Id("submit"))).Click();
// checking the displayed message after creating a new order
wait.Until(SeleniumExtras.WaitHelpers.ExpectedConditions.Element
IsVisible(By.XPath("//span[contains(text(),'Material order
created.')]")));
}
```

The Formula for Calculating the Number of Pieces of Material

The "*CreateCalculationFormula*" test verifies the correctness of the formula for calculating the number of pieces of material in stock in each week. This formula is then used in multiple places in the application. However, this is not the focus of this test, only its basic creation and subsequent deletion are tested here.

```
[Theory(DisplayName = "Calculation formula - Create, Search, De-
lete")]
[InlineData("view", "Selenium - Calculation formula")]
public void CreateCalculationFormula(string view, string
nameOfCalculationFormula){
  try{
    Steps step = new Steps();
    fix.remoteDriver.Navigate().GoToUrl(url);
    step.Login(fix.remoteDriver);
    step.Dashboard_SelectViewModule(fix.remoteDriver);
    step.OpenConcreteView(fix.remoteDriver,viewName);
    step.View_CreateCalculationFormula
     (fix.remoteDriver, nameOfCalculationLine);
    step.View_SearchForSpecificCalculationFormula
     (fix.remoteDriver, nameOfCalculationLine);
    step.View_DeleteCalculationFormula
     (fix.remoteDriver, nameOfCalculationLine);
  }finally{
    fix.remoteDriver.Quit();
  }
}
```

This test consists of six steps. The first is the “Login” step, which is the same for all tests. It ensures that the user logs in to the application. The second step “Dashboard_SelectViewModule” ensures that the appropriate application module is selected, which is used to set up the formulas. The third step “OpenConcreteView” opens the desired view. It needs to be specified which view to open by entering its name into the function parameter. The fourth step establishes the specific formula for the calculation. The fifth step searches the based record by the specified name, this step is given as an example below. The last step deletes this record and checks for successful deletion.

```
public void View_SearchForSpecificCalculationFormula
(RemoteWebDriver driver, string name)

{
  var wait = new WebDriverWait(driver, newTimeSpan(0, 0, 10));

  // search
  wait.Until(SeleniumExtras.WaitHelpers.ExpectedConditions.
  ElementToBeClickable(By.Id("Grid_searchbar"))).SendKeys(name);

  // solves the problem with the occasional disappearance of
  inserted text in the search bar

  for (int i = 0; i < 10; i++)
  {
    var enteredTextSearch=wait.Until(SeleniumExtras.WaitHelpers.
    ExpectedConditions.ElementIsVisible
   (By.Id("Grid_searchbar"))).GetAttribute("value");

    if (enteredTextSearch != name)
    {
      wait.Until(SeleniumExtras.WaitHelpers.ExpectedConditions.
      ElementToBeClickable(By.Id("Grid_searchbar"))).Clear();

      wait.Until(SeleniumExtras.WaitHelpers.ExpectedConditions.
      ElementToBeClickable
      (By.Id("Grid_searchbar"))).SendKeys(name);
      continue;
    }
     else{break;}
  }

   wait.Until(SeleniumExtras.WaitHelpers.ExpectedConditions.
   ElementIsVisible(By.CssSelector("td[aria-label= Selenium
   Calculation formula]")));
 }
```

This loop solves the occasional problem with the text disappearing into the search box, caused by the test script inserting the text too quickly. The `GetAttribute()` function gets the text that is inserted into the search bar. If this text does not match the name, that is being searched for, the script sends the text to the search box again and rechecks if the texts now match. If it does, it continues the test, if not, it tries to send the text to the array again. If the search text fails to be inserted within ten attempts, the test ends with an error. The last line checks whether the formula has indeed been searched for by name. If the name matches, the next step deletes it.

Virtual Material Group

The "*VirtualGroup*" test is designed to verify the correct creation and deletion of the so-called virtual group. This group can then be used to add other individual specific materials. The first step is to log in to the application, the second is to select the appropriate module to create the virtual group, the third is to create this group and the last step is to delete it.

```
[Theory(DisplayName = "Virtual group - Create,Delete")]
[InlineData("Selenium Virtual Group")]
public void VirtualGroup(string VirtualGroupName)
{
  try{
    Steps step = new Steps();
    fix.remoteDriver.Navigate().GoToUrl(url);
    step.Login(fix.remoteDriver);
    step.Dashboard_SelectVirtualGroupModule(fix.remoteDriver);
    step.CreateVirtualGroup(fix.remoteDriver,
    VirtualPNGroupName);
    step.DeleteVirtualGroup(fix.remoteDriver)
  }

  finally{
    fix.remoteDriver.Quit();
  }
}
```

Below is an example of the "*CreateVirtualGroup*" step. Firstly, the script clicks the button to add a virtual group and then enters the name of the group in the appropriate field. Secondly, it will compare whether the group name matches the one that was entered during creation. If it does, the test continues with the next step.

```
public void CreateVirtualGroup(RemoteWebDriver driver, string
nameOfGroup){
  var wait = new WebDriverWait(driver, new TimeSpan(0, 0, 10));

  // clicks the add button
  wait.Until(SeleniumExtras.WaitHelpers.ExpectedConditions.
  ElementToBeClickable(By.XPath("//mat-icon[text()='add']"))).
  Click();

  // virtual group name
  wait.Until(SeleniumExtras.WaitHelpers.ExpectedConditions.
  ElementToBeClickable(By.XPath("//input[@placeholder='Virtual
  Group add']"))).Click();

  wait.Until(SeleniumExtras.WaitHelpers.ExpectedConditions.
  ElementToBeClickable(By.XPath("//input[@placeholder='Insert
  Name']"))).SendKeys(nameOfGroup);

  wait.Until(SeleniumExtras.WaitHelpers.ExpectedConditions.
  ElementToBeClickable(By.XPath("/html/body/div[2]/div[2]/div/
  mat-dialog-container/app-platform-add-group/form/mat-dialog
  -actions/button[2]"))).Click();

  // Compare
  wait.Until(SeleniumExtras.WaitHelpers.ExpectedConditions.
  TextToBePresentInElementLocated(By.XPath("//div[contains
  (text(),'Selenium Virtual Group')]"), nameOfGroup));
}
```

Assigning User Rights

The last test "*UserMaterialsRights*" verifies the correct assignment of user rights. A user can be assigned rights to what material they have access to. A special module is used for this purpose, which is tested here. The test consists of four steps. As with the other tests, the first step is "Login", i.e., logging the user into the application. The second step is to select the module for assigning rights. The third step is to set rights for the user on the corresponding material and the last, fourth step, is to delete them.

```
[Theory(DisplayName = "User material rights - Create, Delete")]
[InlineData("userEmail", "material")]
public void UserMaterialsRights(string userEmail, string
material)
{
  try
  {
    Steps step = new Steps();
    fix.remoteDriver.Navigate().GoToUrl(url);
    step.Login(fix.remoteDriver);
    step.Dashboard_SelectUserMaterialRightsModule
    (fix.remoteDriver);
    step.UserMaterialRights_AddRights(fix.remoteDriver,
    userEmail, material);
    step.UserMaterialRights_DeleteRights(fix.remoteDriver);
  }finally{
    fix.remoteDriver.Quit();
}
```

The "*UserMaterialRights_AddRights*" step adds rights between the user and the required material. First, the script clicks the button to add the record. When the window opens, it enters the user's required email and the designation of the material they have rights to. It then presses the save button and proceeds to the next step of the test.

```
public void UserMaterialRights_AddRights(RemoteWebDriver driver,
string UserEmail, string VendorCodes)
{
  var wait = new WebDriverWait(driver, new TimeSpan(0, 0, 10));
  // clicks the add button
  wait.Until(SeleniumExtras.WaitHelpers.ExpectedConditions.
  ElementToBeClickable(By.XPath("//maticon[text()='add']"))).
  Click();

  // entering email and allowed records to view
  wait.Until(SeleniumExtras.WaitHelpers.ExpectedConditions.
  ElementToBeClickable(By.XPath("//input[@placeholder='User
  Email']"))).Click();
  wait.Until(SeleniumExtras.WaitHelpers.ExpectedConditions.

  ElementToBeClickable(By.XPath("//input[@placeholder=
  'User Email']"))).SendKeys(UserEmail);
  wait.Until(SeleniumExtras.WaitHelpers.ExpectedConditions.
  ElementToBeClickable(By.XPath("//textarea[@placeholder=
  'Material']"))).Click();
  wait.Until(SeleniumExtras.WaitHelpers.ExpectedConditions.
  ElementToBeClickable(By.XPath("//textarea[@placeholder=
  'Material']"))).SendKeys(material);
  // save button
  wait.Until(SeleniumExtras.WaitHelpers.ExpectedConditions.
  ElementToBeClickable(By.Id("submit"))).Click();
}
```

4 Assessment

Displaying the results is also important for the tests themselves. Here it depends on whether we want to see the results for locally running tests or for tests running only on the server.

For local development, it is advisable to use Text Explorer directly inside Visual Studio (Fig. 2). This will provide a clear summary of all the tests and additional information about them, such as the length of each test run, or the arguments used. If an error occurs during the test, the program will show the specific exception that caused the test to fail and, if possible, will also show the exact line in the code where the validation or error occurs.

If the automated tests are then run on the server, you must use other means to display the results. The `logger` tool provided by the `dotnet test` driver is used for this purpose. This command is used to execute tests in a given project. The tests are executed using the selected test framework, in this case, XUnit. Use the – `logger` parameter to specify the logging tool for the test results. It can be seen that in our case the resulting file should be in HTML format. Figure 3 below shows the generated report after the completion of five tests. In the upper left corner, there is basic information such as the total number of tests taken, the number of passes, failures, and skips. Also, the percentage of success rate and the total running time of all tests. In addition, the tests that failed are described in more detail. Thanks to this description, in which part of the code the error occurred, it is possible to check the tests manually, run them again or report the discovered error in the application for correction.

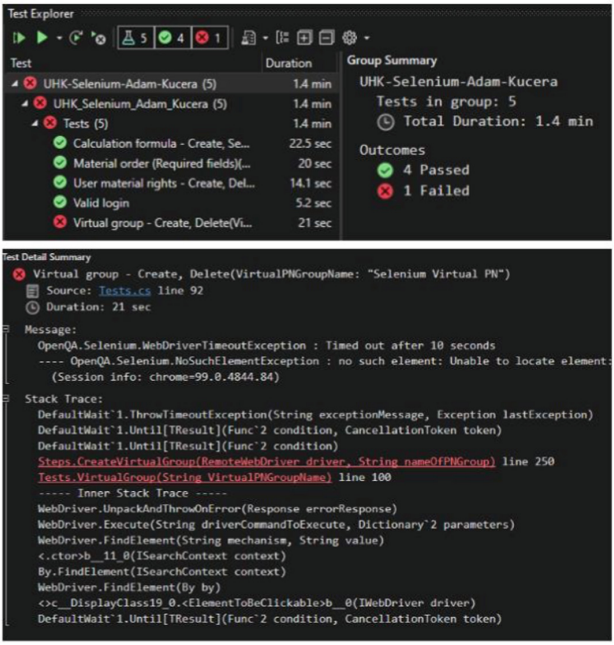

Fig. 2. Microsoft Visual Studio – Test Explorer

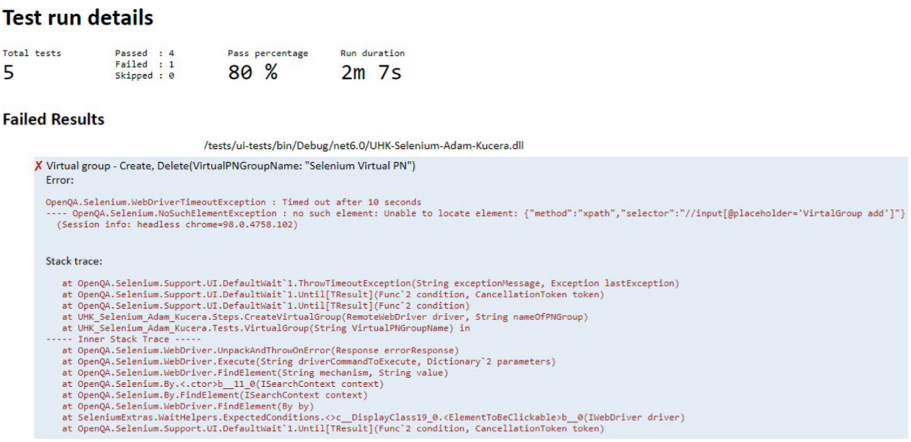

Fig. 3. HTML logger

5 Conclusion

The aim of the paper was to present the possibilities of creating tests for automatic testing of the graphical user interface of an application in a manufacturing company. The article shows a specific sample test suite written in C# using Selenium WebDriver. The tests cover the basic functionality of the application, such as logging in to the

application, ordering material, creating a formula for calculation, a virtual material group, and assigning rights to the user as to what material they can see. In addition, two possible report types for the results of these tests were shown. The first used Visual Studio's tool directly, which is very useful for local test creation. The second is an HTML report that is generated after the tests have finished running in the GitLab environment. The tests were created for the current version of the company application.

From the results of the pilot deployment of the presented automated tests, it is not possible to tell, whether it is more advantageous to use automated or manual testing for software development. Each type has its pros and cons, especially in its reliability and finding covered bugs. Thus, the optimal solution seems to be to use mainly automatic testing, suitably supplemented by manual testing, especially in the case of specific projects. However, automated testing has undoubtedly brought, in the pilot deployment, a great time saving, especially in regression testing, when its need is practically eliminated and gradually fully implemented into the company's processes.

Acknowledgment. The research has been supported by the Faculty of Informatics and Management UHK specific research project 2107 Integration of Departmental Research Activities and Students' Research Activities Support II. Special thanks to our colleague Adam Kučera for his significant help in the practical verification of the tests.

References

1. Myers, G.J., Badgett, T., Sandler, C.: The Art of Software Testing: Now Covers Testing for Usability, Smartphone Apps, and Agile Development Environments, 3rd edn. Wiley, Hoboken (2012)
2. Jamil, M.A., Arif, M., Abubakar, N.S.A., Ahmad, A.: Software testing techniques: a literature review. In: 2016 6th International Conference on Information and Communication Technology for The Muslim World (ICT4M), Jakarta, Indonesia, pp. 177–182, November 2016. https://doi.org/10.1109/ICT4M.2016.045
3. IEEE Standard Glossary of Software Engineering Terminology. IEEE. https://doi.org/10.1109/IEEESTD.1990.101064
4. Galin, D.: Software Quality Assurance. Pearson Education Limited, Harlow; New York (2004)
5. Lynch, J.: The Worst Computer Bugs in History: The Ariane 5 Disaster. BugSnag, 07 September 2017. https://www.bugsnag.com/blog/bug-day-ariane-5-disaster
6. The Editors of Encyclopedia Britannica: "Y2K Bug," Britannica Encyclopedia, 21 April 2021. https://www.britannica.com/technology/Y2K-bug
7. Stojmanovska, M.: 10 Biggest Software Bugs and Tech Fails of 2021. TestDevLab, 27 December 2021. https://www.testdevlab.com/blog/2021/12/27/10-biggest-software-bugs-and-tech-fails-of-2021/
8. Patton, R.: Software Testing, 2nd edn. Sams Pub, Indianapolis (2006)
9. Henry, P.: The Testing Network: An Integral Approach to Test Activities in Large Software Projects. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-78504-0
10. Rafi, D.M., Moses, K.R.K., Petersen, K., Mantyla, M.V.: Benefits and limitations of automated software testing: systematic literature review and practitioner survey. In: 2012 7th International Workshop on Automation of Software Test (AST), Zurich, Switzerland, pp. 36–42, June 2012. https://doi.org/10.1109/IWAST.2012.6228988
11. Xunit. In: NuGet [online]. [cit. 2022-05-05]. https://www.nuget.org/packages/xunit

12. ChromeDevTools: Overview. In: Chrome Developers [online]. [cit. 2022-05-05]. https://developer.chrome.com/docs/devtools/overview/List of
13. Selenium Overview: Selenium (2021). https://www.selenium.dev/documentation/overview/
14. Selenium IDE: Getting Started: Selenium IDE (2019). https://www.selenium.dev/selenium-ide/docs/en/introduction/getting-started
15. Selenium WebDriver: Selenium (2021). https://www.selenium.dev/documentation/webdriver/
16. When to Use a Grid: Selenium (2021). https://www.selenium.dev/documentation/grid/applicability/
17. Chromium Command Line Switches. In: Peter Beverloo [online]. [cit. 2022-05-05]. https://peter.sh/experiments/chromium-command-line-switches/