Is NLP-based Test Automation Cheaper than Programmable and Capture&Replay?

Maurizio Leotta[1 [0000−0001−5267−0602]], Filippo Ricca[1 [0000−0002−3928−5408]], Simone Stoppa[1], and Alessandro Marchetto[2 [0000−0002−6833−896X]]

[1] Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi (DIBRIS)
Università di Genova, Genova, Italy
maurizio.leotta@unige.it, filippo.ricca@unige.it,
4251721@studenti.unige.it
[2] University of Trento, Trento, Italy
alessandro.marchetto@unitn.it

Abstract. Nowadays, there is a growing interest in the use of Natural-Language Processing (NLP) for supporting software test automation. This paper investigates the adoption of NLP in web testing. To this aim, a case study has been conducted to compare the cost of the adoption of a NLP testing approach, with respect to more consolidated approaches, i.e., programmable testing and capture and replay testing, in two testing tasks: test cases development and test case evolution/maintenance. Even if preliminary, results show that NLP testing is quite competitive with respect to the more consolidated approaches since the cumulative testing effort of a NLP testing approach, computed considering both development and evolution efforts, is almost always lower than the one of programmable testing and capture&replay testing.

Keywords: Test Automation · Web Testing · NLP · Artificial Intelligence.

1 Introduction

End-to-End testing frameworks for web applications, such as e.g., Selenium WebDriver and Selenium IDE are nowadays consolidated solutions because they have proven their value in practice by reducing the cost of manual testing and improving the quality of released applications [6]. For this reason, they are used in combination with continuous integration to carry out continuous testing in DevOps processes [4].

However, the cost of test cases development, the cost of maintaining test cases, and the need for experienced developers to develop test suites are limiting their adoption and thus the benefits to the applications under test [9].

Recently, new tools and frameworks called code-less and based on Artificial intelligence (AI) [17] — and more specifically on Natural Language Processing (NLP) — have appeared on the market with the aim of reducing development and maintenance costs. The novelty of NLP-based test automation tools/frameworks is that the test cases are written in natural language and therefore even software testers with limited programming skills can produce executable test cases.

Many vendors have understood the enormous potential of AI in the context of testing and thus have proposed several different NLP-based test automation frameworks/tools (e.g., TestSigma[3], TestRigor[4] and TestProject[5]) capable of interpreting and executing test cases written in natural language. However, the benefits of this new category of approaches, in terms of costs reduction, have not yet been demonstrated in the field and thus, these are currently only promises.

The goal of our research is precisely to test NLP-based test automation tools/frameworks by means of a case study and comparing them on different aspects — e.g., test suite development and maintenance time — with more mature and consolidated solutions: i.e., with tools/frameworks belonging to programmable and capture&replay approaches.

Even if, at the moment, we are still a long way off, the contribution of this work is to start laying the foundations towards an *empirical knowledge base* that is able to guide project managers in choosing the most suitable category of testing frameworks/tools for their purposes. At the moment, thanks to this case study, we have found that this new generation of testing frameworks is very promising.

This paper is organized as follows: Section 2 sketches related works while Section 3 describes the three compared testing approaches used to implement E2E test suites (i.e., programmable, capture&replay, and NLP). Section 4 describes the main aspects of the empirical study we carried out to compare the approaches, while Section 5 reports the results of the study. Finally, Section 6 concludes the paper.

2 Related Work

In the literature, there is a growing interest in the adoption of techniques based on Natural Language Processing (NLP) for supporting the software testing automation.

Garousi et al. [7] survey the state-of-the-art. Most existing works investigate approaches to conduct and automate NLP-based analysis (i.e., morphologic, syntactic, and semantic NLP approaches) for assisting software testing in: (i) clustering related test cases, e.g., [18], [12]; (ii) generating test cases and defining input values from requirement specifications, written in natural language (NL), e.g., [19], [16]; and (iii) identifying test oracles aiming at verifying exceptional software behaviors, e.g., [14]. Some approaches adopt an intermediate representation, e.g., behavioral models represented as state machines, between the natural language specifications and the generated test cases and test artifacts (e.g., [1], [5]). Gupta et al. [8] pointed-out relevant issues related to the adoption of NLP in software testing: (i) requirement specifications are often constrained to a specific structure that limits their expressiveness; (ii) intermediate behavioral models are often large and complex since they need to be precise and comprehensive; (iii) manual rectification of models is often required; and (iv) additional intervention is often needed to obtain executable test cases.

The development of executable test cases is, in fact, a complex task when NLP techniques are adopted. For instance, in the programmable testing approach, executable test cases are developed according to the API/interfaces of the application under test. In model-based testing, (quasi-executable) test cases are developed from an abstract representation of the application under test (e.g., a UML model), thus transformation

[3] https://testsigma.com/ [4] https://testrigor.com/ [5] https://testproject.io/

approaches are required to obtain executable test cases. Similarly, when NLP-based approaches are used to support testing, adequate transformation approaches are required to transform abstract test cases into executable test cases.

Requirement specifications are often written in natural language. Gherkin[6] is a structured quasi-natural language that lets testers specify test cases by using a natural language structured around a set of predefined keywords. Colombo et al. [3] convert Gherkin specifications into models used within a web testing tool. In the work of Cauchi et al. [2], Gherkin is adopted for improving the communication gap, about safety-critical system properties, between developers and non-technical experts. Fitnesse is another structured quasi-natural language adopted for specifying NL-based acceptance test cases. Both Marchetto et al. [15] and Longo et al. [13] evaluate the adoption of Fitnesse. While Marchetto et al. [15] compare Fitnesse with programmable acceptance test cases, Longo et al. [13] compares the adoption of Fitnesse and Gherkin for writing acceptance test cases.

Differently from the literature, this work conducts a preliminary evaluation about the adoption of NLP for test case development and evolution. To the best of our knowledge, in fact, there is a lack in the literature of objective and comparative evaluations of the proposed NLP methods with respect to more traditional approaches, e.g., programming and capture&replay procedures, and others, in test case development and evolution [9–11]. We start filling the gap by reporting a cases study conducted in the Web testing domain.

3 Background

Gherkin is a test specification language that aims at providing a unique language for specifying test cases. The Gherkin language is a structured language composed of a set of keywords including the following ones:

- Feature: provide a high-level description of the test
- Example/Scenario: show an example of the test
- Given: represent the initial context of the test
- When: describe an action occurred
- And: another action occurred
- Then: describe the result

The code in Listing 1.1 shows a small example in which Gherkin is used to specify a test case for an online e-commerce application. The test aims at verifying the correct price of a product when it is added to the shopping cart.

Several testing approaches can be adopted for the functional testing of web applications. The choice among them could depend on different aspects including, e.g., the technology used in the implementation of the application, the available tools (e.g., Selenium WebDriver and Selenium IDE[7]), and the expertise of the involved testers [9]. In this work, we consider three testing approaches: programmable testing (PT), capture&replay testing (CRT), and NLP-based testing (NLT).

Programmable web testing (PT) is based on the manual implementation of test scripts (test cases) using ad-hoc programming languages, e.g., Java, PHP. A test case is a script composed of a set of instructions and programming commands written by developers and

[6] https://cucumber.io/docs/gherkin [7] https://www.selenium.dev

executed to exercise the application functionality. Often, testers can use libraries that expose APIs for interacting with web applications and providing the use of commands, e.g., click a button, fill fields and submit a form. Then, the test script is completed by developers with input values and assertions to check the obtained execution results.

Listing 1.1. Example of test case specified with Gherkin

```
1. // Gherkin TC 1: TestVerifyPriceOfaSingleProduct
2. Feature: Add a product to cart and verify the price
3. Scenario: A Customer wants to add a product to the cart
4. Given the user views the homepage
5. When the user adds an item to the cart
6. And clicks to cart
7. Then the page shows the cost for the product on the cart
```

Figure 1 shows a fragment of a programmable test script written adopting the design pattern Page Object[8] that implements the Gherkin code 1.1 in Listing 1.1. `@BeforeEach` and `@AfterEach` are constructs defining commands to be executed before and after the execution of the test case body. In the body, methods provided by the Page Objects, such as `addFirstProductToCart` that contribute to the logic of the test cases, i.e., add a product to the shopping cart in our example, are provided. Assertions (`assertEquals` condition) are used to verify the price of the product added to the cart. Selenium WebDriver is an example of tool supporting programmable web testing. The advantage of programmable testing is its flexibility and the reusability of the test cases. In fact, working with programming languages allows developers to directly handle in the scripts conditional statements, loops, logging, exceptions, as well as to create parametric (i.e., data-driven) test cases. The drawbacks, however, are that: (i) developers need to be skilled; (ii) to be effective, test development has to be subject to the programming guidelines and best practices typically used for software development; and (iii) a remarkable initial effort is required to develop test cases.

```
package test;

import static org.junit.Assert.assertEquals;
import static org.junit.Assert.assertTrue;
import org.junit.jupiter.api.AfterEach;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Test;
import org.openqa.selenium.WebDriver;
import org.openqa.selenium.chrome.ChromeDriver;
import pageobject.CustomerInfoPO;
import pageobject.HomepagePO;
import pageobject.PaymentPO;
import pageobject.ProductPO;
import pageobject.ShippingPO;

public class UserTest {

    WebDriver driver;

    @BeforeEach
    void setUp() throws Exception {
        System.setProperty("webdriver.chrome.driver", "src/tool/chromedriver");
        driver = new ChromeDriver();
        driver.get("http://localhost:1111/");
    }

    @AfterEach
    void tearDown() throws Exception {
        driver.close();
    }

    @Test
    public void testVerifyPriceSingleProduct() {
        HomepagePO home=new HomepagePO(driver);
        home.addFirstProductToCart();
        assertEquals("€20.00",home.getFirstProductPrice());
        }
}
```

Fig. 1. Example of PT test script

	Command	Target	Value
1	*open*	http://localhost:1111/	
2	*set window size*	1792x1008	
3	*click*	linkText=Add to cart	
4	*click*	linkText=Cart 1	
5	*click*	css=.card-body	
6	*verify text*	css=.my-auto	€20.00
7	*click*	id=empty-cart	
8	*click*	css=.pushy-link	
9	*close*		

Fig. 2. Example of CRT test script

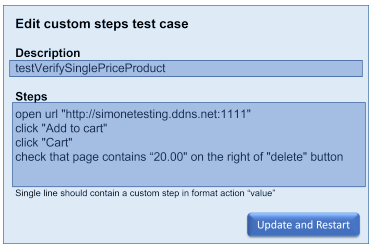

Fig. 3. Example of NLP test script

[8] https://martinfowler.com/bliki/PageObject.html

Capture&replay web testing (CRT) is usually used for regression testing. This testing approach is based on a first manual execution in which the tester manually exercises a web application by using a tool that records the whole execution session, thus all user events and interactions with the application elements, as well as all key pressed, mouse movements, link clicks, scripts' execution, are recorded. Test cases are scripts that are automatically composed by the tool and that can be used to replay the recorded testing sessions. Test cases are hence executed by re-executing the whole recorded sessions that can be also enriched with assertions for checking the result of the re-execution. Testers can also customize each re-execution by slightly changing input values and assertions, that can also be parametric to make the test scripts more flexible. Figure 2 shows a fragment of a test script recorded with a capture&replay tool that implements the Gherkin code in Listing 1.1, as example. We see that the starting web page to test is defined at the beginning of the test, then a set of `click` operations have been performed by the tester to add a product to the shopping cart, then the text content of a page element is checked with an assertion (`verify text`). Selenium IDE is an example of tool supporting capture&replay web testing. The advantage of capture&replay tools is that they are relatively simple to use. Hence, even testers without programming skills are able to build complex and complete test suites. The drawbacks, however, are that the resulting test scripts (1) have a lot of duplicated code, (2) are difficult to read in case of complex scenarios, and (3) contain hard-coded values (e.g., data inputs and page references and objects) that make the test scripts strongly coupled with the web application under test and as a consequence difficult to modify, e.g., for test maintenance and evolution purpose.

NLP-based testing uses NLP techniques to let testers write test scripts by using the natural language. NLP, in fact, is the part of artificial intelligence that allows machines to interpret natural language. The use of NLP techniques for testing purposes is still at the infancy and its effectiveness has to be empirically investigated. Figure 3 shows a fragment of a test script that implements the Gherkin code in Listing 1.1, as example. We can see that the test script is written as a sequence of simple natural language sentences in which verbs such as `open`, `click`, `check`, and their synonyms, are used to describe the actions to be executed on the application under test. TestSigma, TestRigor, and TestProject are three examples of commercial tools supporting natural language web testing. The advantage of NLP testing is that it can help in reducing the effort required to human testers for producing test cases. Furthermore, specific programming skills are not required as for the programmable approach. NLP can be used to write test cases in natural language, to transform such descriptions in executable test cases and to run them. The drawbacks, however, are that an adequate transformation approach based on NLP techniques is required to transform test cases written in natural language into executable test cases able to exercise the application under test.

4 Case Study Design

This section details planning and design of the case study we conducted to compare three web testing approaches: programmable testing (PT), capture&replay testing (CRT), and NLP-based testing (NLT). In terms of tools supporting these three testing approaches, we selected Selenium WebDriver (PT) and Selenium IDE (CRT) because they are well known

and used. As representative of the NLT available tools, we selected a commercial tool, according to a preliminary analysis we conducted [9]. The rest of this section presents the design of the case study.

4.1 Study Design

The goal of this study is to compare three web testing approaches, ***PT***, ***CRT*** and ***NLT*** with the purpose of assessing both short-term and long-term (i.e., across multiple releases) effort required in two main testing scenarios: (1) test case development and (2) test case evolution. In fact, we are mainly interested in comparing the effort required for the implementation of the initial test suites from scratch, and the effort required for the evolution of the test suites across subsequent releases of the applications. The results of the study can be useful for (i) practitioners (developers and managers), interested in understanding the usual costs and the possible returns of their investment associated with the adoption of the different web testing approaches; and (ii) researchers, interested in collecting empirical evidence about the usage of the different testing approaches.

The context of the study is defined as follows. The involved human subjects are: one of the authors, who defined the test specifications, and a junior professional web developer, who developed the test cases with the three approaches. The objects of the study are three open-source web applications.

4.2 Software Objects

To perform our experiment, we took into account three web applications (experimental objects) named: ExpressCart, Shopizer and OIM. These applications have been selected since they are: (1) medium-size applications; (2) quite representative of usual web applications in terms of functionality they provide and technology they use, i.e., programming languages, databases, libraries and frameworks; and (3) there are at least two major releases available (minor releases have not been considered since small changes between the applications releases lead to a large reuse of test cases, thus limiting the amount of empirical data for our study). This last point is relevant for the estimation of the test case evolution effort, i.e., the effort required to evolve and reuse the test cases for more than one software release. For each application, hence, we consider two subsequent major releases extracted from the application code repository that expose both *logical* and *structural* changes. While a logical change is a change in a system functionality that foresees the modification of the process underlying the specific functionality, a structural change is instead a change on the application structure that implies only some changes to the elements, e.g., of the application GUI layout/structure.

ExpressCart[10] is an e-commerce application that implements functionalities such as: shopping carts, payment methods, and administrative functions. The application is very rich and dynamic: it is mainly written in Javascript, by using frameworks such as Node.js and Express.js. Shopizer[11] is another e-commerce application, mainly written in Java,

[9] Being a commercial tool, we think it is better not to disclose its identity. In all cases, the other NLP tools considered were also very similar to the one chosen.

[10] https://expresscart.markmoffat.com/documentation.html (last access: February 2022)

[11] https://shopizer-ecommerce .github.io/documentation/#/starting (last access: February 2022)

that implements functionalities such as: catalog management, shopping carts, marketing components, smart pricing management, ordering, payment and shipping management. OIM[12] is an inventory management that implements transactions management, raw material management, batch, supplier, items, categories, and storage management. The application has been mainly developed in PHP by using AppGini[13], a web-database framework for applications building.

4.3 Research Questions and Metrics

The research questions of our study are the following ones:

- **RQ1: Developing Time**. What is the initial development effort required for creating test suites by adopting NLT with respect to more traditional approaches such as PT and CRT?
- **RQ2: Reuse**. How much of the test suites generated by adopting NLT can be reused 'as-is' with respect to more traditional approaches such as PT and CRT, when a new release of the application needs to be tested?
- **RQ3: Evolution Time**. What is the effort required for the evolution of test suites generated by adopting NLT with respect to the effort required to evolve test suites developed with traditional approaches such as PT and CRT, when a new release of the application needs to be tested?
- **RQ4: Trend in Releases.** How the cumulative effort (i.e., combining development and evolution effort) required by NLT varies in the time, with respect to the one required for applying traditional approaches such as PT and CRT, by considering several different application releases?

The first research question deals with the development cost in terms of time required to develop test suites from test specifications. We aim at verifying whether the adoption of NLT is costly in terms of required time, with respect to the time required to apply the more traditional approaches, PT and CRT. This could give practitioners an idea of the initial investment to be made to adopt the testing approaches. To answer *RQ1*, we measured the test suite development effort in terms of time (minutes) needed by a tester to develop the executable test cases. We compared the different efforts and estimated the ratio between NLT and the more traditional approaches.

The second research question deals with the resilience to changes of the developed test suites. We aim at verifying the capability of the testing approaches in developing test suites that can be reused to test new major releases of the application under test. This could give practitioners an idea of the capability of the testing approach to implement reusable test suites, i.e., suites that can be reused (as-is) to test a new software release. To answer *RQ2*, we considered the next major release of each web application under test ($_{v2}$) and counted the number of test cases reusable (as-is) for testing this new release, i.e., for which the execution does not fail in the new application release.

The third research question deals with the *evolution* cost required to evolve test suites by making them working for testing the new major release of the application under test.

[12] https://bigprof.com/appgini/applications/online-inventory-manager (last access: February 2022)

[13] https://appgini.en.softonic.com/

We aim at verifying whether a testing approach requires additional evolution costs, with respect to others, and we aim at estimating the ratio between the different costs. This could give practitioners an idea of the effort to be provided to make test suites usable for more than one software release. To answer *RQ3*, we considered the next major release of each web application under test and we evolved the initially developed test suites so as to make them usable also for testing this new release. We, hence, measured the test suite evolution effort, in terms of time (minutes) needed by a tester to fix the test cases that cannot be executed directly with the new application release ($_{v2}$).

The last research question is about the return on investment conducted for the adoption of the testing approaches. We aim at verifying how the cumulative testing effort (computed combining development and evolution effort) required to apply the NLT approach varies over the time and the application releases, with respect to the one required to apply the more traditional approaches PT and CRT. This could give practitioners an idea of the overall effort needed. To answer *RQ4*, we computed the cumulative testing effort for each approach as proposed in [9] and estimated the number of application releases after which the cumulative effort trend changes. For instance, let C_0 and N_0 the effort required for the initial development of CRT and NLT test cases, respectively, and let C_1, C_2,... and N_1, N_2 the test case evolution effort associated with the successive application releases. We are seeking the lowest value *n* such that: $\sum_{n=0}^{n} C_i \geq \sum_{n=0}^{n} N_i$. That value corresponds to the release number after which NLT test cases start to be cumulatively more convenient than CRT ones. Under the assumption that $C_i = C \; \forall i > 0$ and that $N_i = N \; \forall i > 0$ i.e., the same evolution effort is required for the software releases, we can find the following solution to the equation above: $\frac{N_0 - C_0}{C - N}$. Hence, after *n* releases, the cumulative effort of the initial development and evolution of NLT test cases is lower than the one of CRT test cases. It is worth to notice that a negative value obtained for *n* means that the cumulative cost of the NLT is always lower than the one of CRT. Similarly we can estimate the value of *n* for NLT vs PT and CRT vs PT. By estimating *n*, we could give practitioners an idea about when the investment in the adoption of a given testing approach could become of interest with respect to the other testing approaches, considering both development and evolution effort.

4.4 Procedure

In the study, programmable testing, capture&replay testing, and NLP-based testing have been adopted in two different testing tasks: (i) test case development and (ii) test case evolution. Two sub-sequent application releases of the objects of the study have been considered: ExpressCart$_{v1}$ / ExpressCart$_{v2}$, Shopizer$_{v1}$ / Shopizer$_{v2}$, and OIM$_{v1}$ / OIM$_{v2}$. In detail, the following procedure has been applied.

1. A preliminary training phase has been organized by asking the junior developer to test the PetClinic[14] application with the three different testing approaches, by starting from Gherkin test specifications, thus to practice them and their corresponding tools.
2. Each application and artifact (e.g., code and documentation) has been analyzed by the junior developer and by one of the authors to acquire knowledge about them, their functionalities, and the technology used to implement them.

[14] https://projects.spring.io/spring-petclinic

3. A test suite specification has been defined by one of the authors by describing a set of end-to-end functional test cases for each application object of the study: ExpressCart, Shopizer, and OIM. The main functionalities provided by each applications' version $_{v1}$ considered in the study have been covered at least once (mainly covered only normal cases, and not many corner cases). The test cases have been specified using the Gherkin language.
4. *Test cases development*: PT, CRT and NLT have been used, by the involved developer, for implementing the previously created test cases specifications for ExpressCart$_{v1}$, Shopizer$_{v1}$, and OIM$_{v1}$. In other terms, three executable test suites have been developed for testing the first release of the applications under test by using the three different tools considered in this case study. The three developed test suites are equivalent from the functional point of view, since they test exactly the same functionalities and have been developed by trying to adhere to the defined test specifications.
5. *Test case evolution:*
   - The executable test suites built at the previous point have been executed, by the developer, on the second application releases (i.e., ExpressCart$_{v2}$, Shopizer$_{v2}$, and OIM$_{v2}$) and identified the failing test cases, i.e., those test cases that, due to application changes between the first and the second application release, report a failure or an error.
   - Both structural and logical changes implemented in ExpressCart$_{v2}$, Shopizer$_{v2}$, and OIM$_{v2}$, with respect to the previous release of the same application, have been identified and considered.
   - The failed test cases have been repaired, by the junior developer, so that the full test suites can be executed without problems also in the second release ($_{v2}$) of the applications under test.

During the whole process, the development effort required for the development of the three test suites, as well as the evolution effort required for the evolution of the test suites, have been measured by the junior developer noting down the times. To balance as much as possible the learning effects in the experiment, the order of test suite development and evolution has been alternated. Finally, metrics (i.e., test cases development and evolution time, number of failed test cases, and cumulative effort trend) needed to answer the four research questions, have been analyzed.

4.5 Threats to Validity

Internal validity threats concern factors that may affect a dependent variable that are not considered in the study. The most relevant threat to the internal validity concerns the subjectivity and variability of the test cases implementation task, e.g., selection of the application functionalities to test, definition of test steps and input data. We tried to limit this threat by involving two persons, one for the definition of the test specifications and another one (the junior developer) for the test development, and by applying well-known testing criteria. Another (possible) impacting threat is related to the learning effect during the test case development and evolution tasks. As explained, we tried to consider it in the experiment design by altering the order of test suite development and evolution. *Construct*

validity threats concern the relationship between theory and observation. The most relevant threat to the construct validity concerns the use of time (development and evolution time) as measure of the testing effort. Even if we are conscious that it is questionable since several different aspects could impact the testing effort, we consider time as a reasonably proxy for estimating the testing effort since it is a widely adopted practice in the empirical software engineering. Another threat concern the fact that test cases have been specified in Gherkin: such specifications can be considered quite similar to the one used for NLT. On the one side, however, Gherkin test cases are abstract while NLT test cases are concrete test scripts characterized by executable steps, specific input values and assertions to check the output. Moreover, on the other side, this mimics what normally happens in the industry where E2E test cases are often specified in natural language.

Conclusion validity concerns the relationship between the treatment and the outcome. To analyze the data and answer the research questions of interest we chose to use non-parametric tests (i.e., Wilcoxon paired test), due to the size of the sample and because we could not safely assume normal distributions. Moreover, we applied corrections (specifically, Holm correction) to the statistical tests due to multiple re-executions.

External validity threats are related to the generalization of the results. The most relevant threat to external validity concerns the involvement of only one junior developer. Concerning this point, the involved developer has some industrial experience in the Web domain and testing with Selenium WebDriver and thus is a good representative of junior web developers, in general. Moreover, it is important to underline that the case study is challenging and time-consuming and therefore finding candidates to re-execute it is not easy. Another threat could be related to the applications adopted in the study. The applications are medium-size, realistic and representative of their domain, and based on modern technologies and languages. Other potentially impacting threats are related to the developed test suites and the used tools. Test suites have been developed as much as possible by following a systematic approach and by constructing, at least, one test case for each functionality provided by the applications. In terms of adopted tools, we used third-party frameworks/tools, well-known and available on the Internet, thus avoiding any bias of the authors.

5 Analysis of Results

5.1 RQ1: Developing Time

Table 1 reports general information about the developed test suites in terms of number and characteristics (e.g., lines of code) of test cases developed. To compare the CRT and PT code, we exported the native Selenese code (column “Sel”) — the language used by Selenium IDE — in Java using the export feature provided by Selenium IDE. In two cases, as expected, the Java test code (excluded the page objects - POs) is shorten than the CRT one, while in the case of Shopizer, this does not happen since several manual waits has been added by the junior developer in the PT code (on the contrary, CRT manages automatically such cases). Moreover, it is interesting to note that the number of NLT test case lines are always less than the number of Selenese lines: this is reasonable since Selenium records every interaction with the web application (e.g., click on a “name” field + type “John”: i.e., 2 lines) while NLT provides a higher level view (e.g., write “John” in

the "name" field: i.e., 1 line). The last column of the table shows the average number of steps for the NLT test cases.

Application	#Test cases	Code							
		PT				CRT		NLT	
		test LOCs	PO LOCs	Total LOCs	#PO	Sel lines	Java LOCs	Lines	AVG Lines
ExpressCart	40	842	932	1774	18	635	934	361	9.0
Shopizer	28	506	483	989	7	273	417	150	5.3
OIM	32	462	1065	1527	18	552	765	351	10.9

Table 1. Test suites code details

Table 2 reports the total test suite and average test case development effort (expressed in minutes) and the statistical difference observed (if any) between the distributions of the test development effort, to compare PT and CRT with NLT, computed using the Wilcoxon paired test with Holm correction. The last two columns report the effort ratio measured between PT and CRT with NLT. For instance, a value higher than 1 in the ratio between PT and NLT means that the PT test suite required more development effort (time) than the corresponding NLT test suite.

Application	Total Time (min)			Average Time (min)			p-value		Ratio	
	PT	CRT	NLT	PT	CRT	NLT	PT-NLT	CRT-NLT	PT/NLT	CRT/NLT
ExpressCart	315.7	45.8	156.6	9.6	1.4	4.7	**<0.01**	**<0.01**	2.02	0.29
Shopizer	225.1	47.7	75.4	8.0	1.7	2.7	**<0.01**	**<0.01**	2.98	0.63
OIM	310.4	86.4	93.2	10.0	2.8	3.0	**<0.01**	0.07	3.33	0.93

Table 2. Test suite development time (minutes)

That Table shows that the development effort for PT is always higher than for NLT (p-value < 0.01), while there is also a trend, statistically relevant for two out of three applications, for which the development effort for NLT is higher than the one of CRT. This is confirmed by the ratio (last columns of Table 2), indeed PT required more effort than NLT (PT/NLT ratio value is higher than 1) and CRT required less effort than NLT (CRF/NLT ratio is lower than 1). The observed result shows that PT requires more development time than NLT, since the former requires to develop the testing code (e.g., in Java) and the latter requires only to describe the test scenarios using a step-by-step natural language description (e.g., derived from the Gherkin descriptions). At the same time, the result of our case study shows also that CRT allows to produce test cases faster than NLT. In fact, the NLT approach requires, unlike CRT, the analysis of the description of test scenarios (written in Gherkin), their conversion in step-by-step actions/steps that exercise the application under test, and the definition of the needed input values. By analyzing the developed NLT test cases, we noticed that the junior developer tried to describe the test actions/steps by using a simple natural language, avoiding complex linguistic constructs; this was done to simplify the task and to avoid problems of understanding by the NLT tool.

RQ1. Summarizing, with respect to the research question RQ1, we can observe that: (i) the programmable test suites (PT) required the largest initial development effort; and (ii) there is a trend for which natural language test suites (NLT) require more effort compared to that required for capture&replay (CRT) suites.

5.2 RQ2: Reuse

Table 3 reports some information about the fixed/repaired test cases, i.e., those test cases developed for testing the application release $_{v1}$ and that failed in exercising the application release $_{v2}$, thus requiring some effort to be fixed. In particular, Table 3 reports, for each testing approach, the number of fixed test cases (column "Fixed") and also the statistical difference (if any) between PT and CRT with NLT distributions, computed by using the Wilcoxon paired test with the Holm correction.

Application	PT # Test Fixed	CRT #Test Fixed	NLT #Test Fixed	p-value PT-NLT	p-value CRT-NLT
ExpressCart	19	23	19	0.33	1
Shopizer	17	16	17	1	1
OIM	26	28	15	**0.01**	**0.03**
total	62	67	51	-	-
average	20.7	22.7	17	-	-

Table 3. Test suites evolution: changes

About the fixed test cases, we mainly observe trends that are not statistically relevant in most of the cases, apart for OIM. While for OIM, we observed that tests to be repaired differ significantly between PT/CRT and NL, for the other two applications no relevant difference has been observed. In general, we can observe that a large amount of test cases needs to be fixed (in the range between 48% and 90%). CRT has the largest number of test cases to be fixed, on average 73%, and variability for application under test 17% with respect to PT (respectively 67% and 14%) and NLT (respectively 56% and 6%).

As we have already said, the changes between the two selected versions of the web applications $_{v1}$ and $_{v2}$ considered in the experimentation were of two types: structural and logical. The number of structural changes were the following: ExpressCart 14, Shopizer 9 and OIM 9. While the logical changes were: ExpressCart 6, Shopizer 8 and OIM 14. It is possible to note that the number of changes is well distributed both between applications and types.

As expected, PT and CRT show overall a similar levels of reusability (see Table 3) since they are based on the same DOM-based interaction paradigm. More interesting is the result of NLT that appears to be able, on average more often than the other approaches, to compensate for the change and thus finding a working solution in the novel version of the app. This is due to the fact that the NLT actions are more abstract (e.g., Enter "John" into "name" field) than the one required in the PT and CRT approaches (e.g., driver.findElement(By.xpath("//*[@id='user-name']")).sendKeys("John"); where the web element is localized using a XPath expression) that suffer more from changes to the DOM.
RQ2. Summarizing, with respect to the research question RQ2, we can observe that: (i) the capture&replay suites (CRT) show the lowest reusability, while (ii) the natural language suites (NLT) show the highest test case reusability.

5.3 RQ3: Evolution Time

Table 4 reports (i) general information about the evolution test suites effort in terms of time (expressed in minutes) required to fix the failed test cases and (ii) the statistical difference observed between the distributions of the test evolution time to compare PT

and CRT with NLT, which is computed using the Wilcoxon paired test with the Holm correction. The Table also reports the evolution effort ratio measured between PT and CRT with NLT. For instance, a value higher than 1 in the ratio between PT and NLT means that the PT test suite required more evolution time than the corresponding NLT test suite.

Application	Total Time (min)			Average Time (min)			p-value		Ratio	
	PT	CRT	NLT	PT	CRT	NLT	PT-NLT	CRT-NLT	PT/NLT	CRT/NLT
ExpressCart	88.1	95.6	44.7	2.7	2.9	1.3	**0.01**	**0.04**	1.97	2.14
Shopizer	62.0	30.1	42.5	2.2	1.1	1.5	**0.03**	0.66	1.46	0.71
OIM	62.6	60.9	38.5	2.1	2.0	1.2	0.08	0.06	1.63	1.58

Table 4. Test suite evolution time (expressed in minutes)

From Table 4 it is apparent that: (i) PT required a higher evolution effort of NLT in all the applications; and (ii) CRT required a higher evolution effort than NLT in two out of three applications. Indeed, the penultimate column of Table 4 shows that PT has a ratio greater than 1 with respect to NLT. While CRT shows, with respect to NLT (last column), a ratio greater than 1 in two out of three applications. The fact that NLT requires less time to evolve the failed test cases with respect to PT is reasonable since in such a case no programming activities are required and to complete the maintenance task it is enough to edit the test description text. On the other hand, NLT is also faster than CRT for two applications out of three: also in this case edit the test description text seems to be simpler than directly editing the Selenese code, or re-recording the entire scenario. In the case of Shopizer, we can observe that the evolution time of NLT is higher of about 12 min with respect to CRT. This is explainable why the novel version of the application introduced a banner for the user-management of the cookies not straightforward to be managed using the NLT tool. The banner requested, in NLT, a few attempts before finding the correct interaction solution while in the case of CRT a simple recording of the interaction with the approve button was sufficient to solve the problem.

RQ3. Summarizing, concerning the research question RQ3, we can observe that: (i) the programmable test suites (PT) required a higher evolution effort compared to NLT; and (ii) the evolution effort required by the capture&replay suites (CRT) shows a high variability (but in two cases out of three is higher than the one required for NLT).

5.4 RQ4: Cumulative Effort

Table 5 reports the estimated application release n in which we foresee a significant change of the cumulative testing effort trend. Concerning the adoption of NLT, Table 5 shows that the cumulative testing effort of NLT is almost always lower than the one of PT and CRT, apart the case of Shopizer for CRT. The three negative values for n in column PT-NLT confirm what reported in the previous tables: NLT cost less during the initial development and also the cost of each evolution step is lower. Thus, the straight lines representing the cumulative costs never intersect for any positive value of n. Moreover, the two positive values of n in column CRT-NLT means that NLT have an initial higher cost w.r.t. CRT but just after a few releases the cumulative costs of NLT are lower since it requires lower maintenance costs. The only exception is the case of Shopizer, where both the development and evolution costs are lower for CRT, meaning that CRT show a lower cumulative cost for any positive value of n. Also in this case the explanation could be attributable to the introduction of the banner (see the answer to RQ3).

Application	Application releases: n	
	PT-NLT	CRT-NLT
ExpressCart	NLT costs less for $n > -3.6$	NLT costs less for $n > +2.2$
Shopizer	NLT costs less for $n > -7.7$	CRT costs less for $n > -2.2$
OIM	NLT costs less for $n > -9.0$	NLT costs less for $n > +0.3$

Table 5. Evolution cost: an approach that costs less starting from a release n<0 means that it costs less for both the initial development and the evolution costs.

RQ4. Summarizing, with respect to the research question RQ4, we can observe that overall the natural language suites (NLT) required the lowest cumulative testing effort with respect to the other approaches (i.e., PT and CRT) with only one exception (Shopizer that costs less when adopting CRT).

6 Conclusions

This paper reports a study conducted to compare NL-based web testing (NLT) and two more traditional testing approaches, i.e., programmable testing (PT) and capture&replay testing (CRT). The comparison is based on: the effort required for developing test suites; the resilience to changes and the effort required to evolve test suites; and the overall effort needed to apply each testing approach over multiple application releases.

Results show that: (i) NLT requires less development effort than PT but more effort than CRT; (ii) NLT shows the highest test case reusability, as well as (iii) the lowest evolution effort in most of the cases, with respect to traditional approaches; and (iv) NLT tends to require the lowest cumulative testing effort over the time, with respect to other approaches (we observed only an exception in one of the considered web application using CRT).

For the future, we are planning to: (i) conduct a larger study by extending the set of the considered web applications and involving others developers, currently we have involved only one participant, aiming at consolidating the obtained results; (ii) consider different tools than Selenium IDE/WebDriver and the one for NLT to support the obtained results, and (iii) conduct a study to estimate the expressiveness of the natural language used to develop test cases with the NLT approach and to exploit the potentiality of the engine underlying the NLT approach (i.e., the engine used to transform natural language based test cases into executable test cases).

References

1. Carvalho, G., Falcão, D., Barros, F., Sampaio, A., Mota, A., Motta, L., Blackburnc, M.: Nat2testscr: Test case generation from natural language requirements based on scr specifications. Science of Computer Programming **95**, 275–297 (2014). https://doi.org/10.1016/j.scico.2014.06.007
2. Cauchi, A., Colombo, C., Francalanza, A., Micallef, M., Pace, G.: Using gherkin to extract tests and monitors for safer medical device interaction design. In: 8th Symposium on Engineering Interactive Computing Systems (SIGCHI). ACM (jun 2016). https://doi.org/10.1145/2933242.2935868
3. Colombo, C., Micallef, M., Scerri, M.: Verifying web applications: From business level specifications to automated model-based testing. Electronic Proceedings in Theoretical Computer Science **141**, 14–28 (mar 2014). https://doi.org/10.4204/eptcs.141.2

4. Ebert, C., Gallardo, G., Hernantes, J., Serrano, N.: DevOps. IEEE Software **33**(3), 94–100 (May 2016). https://doi.org/10.1109/ms.2016.68
5. Fischbach, J., Vogelsang, A., Spies, D., Wehrle, A., Junker, M., Freudenstein, D.: SPECMATE: Automated creation of test cases from acceptance criteria. In: 13th International Conference on Software Testing, Validation and Verification (ICST). IEEE (oct 2020). https://doi.org/10.1109/icst46399.2020.00040
6. García, B., Gallego, M., Gortázar, F., Organero, M.: A survey of the selenium ecosystem. Electronics **9**, 1067 (06 2020). https://doi.org/10.3390/electronics9071067
7. Garousi, V., Bauer, S., Felderer, M.: NLP-assisted software testing: A systematic mapping of the literature. Information and Software Technology **126**, 106321 (oct 2020). https://doi.org/10.1016/j.infsof.2020.106321
8. Gupta, A., Mahapatra, R.P.: A circumstantial methodological analysis of recent studies on NLP-driven test automation approaches. In: Intelligent Systems, pp. 155–167. Springer Singapore (2021). https://doi.org/10.1007/978-981-33-6081-5_14
9. Leotta, M., Clerissi, D., Ricca, F., Tonella, P.: Capture-Replay vs. Programmable Web Testing: An Empirical Assessment during Test Case Evolution. In: Proceedings of 20th Working Conference on Reverse Engineering (WCRE 2013). pp. 272–281. IEEE (2013). https://doi.org/10.1109/WCRE.2013.6671302
10. Leotta, M., Clerissi, D., Ricca, F., Tonella, P.: Visual vs. dom-based web locators: An empirical study. In: Sven Casteleyn, Gustavo Rossi, M.W. (ed.) Proceedings of 14th International Conference on Web Engineering (ICWE 2014), LNCS, vol. 8541, pp. 322–340. Springer (2014). https://doi.org/10.1007/978-3-319-08245-5_19
11. Leotta, M., Clerissi, D., Ricca, F., Tonella, P.: Approaches and tools for automated end-to-end web testing. Advances in Computers **101**, 193–237 (2016). https://doi.org/10.1016/bs.adcom.2015.11.007
12. Li, L., Li, Z., Zhang, W., Zhou, J., Wang, P., Wu, J., He, G., Zeng, X., Deng, Y., Xie, T.: Clustering test steps in natural language toward automating test automation. In: 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering. ACM (nov 2020). https://doi.org/10.1145/3368089.3417067
13. Longo, D.H., Vilain, P., da Silva, L.P.: Measuring test data uniformity in acceptance tests for the FitNesse and gherkin notations. Journal of Computer Science **17**(2), 135–155 (feb 2021). https://doi.org/10.3844/jcssp.2021.135.155
14. Malik, M., Sindhu, M., Abbasi, R.: Test oracle using semantic analysis from natural language requirements. In: 22nd International Conference on Enterprise Information Systems. SCITEPRESS (2020). https://doi.org/10.5220/0009471903450352
15. Marchetto, A., Ricca, F., Torchiano, M.: Comparing "traditional" and web specific fit tables in maintenance tasks: A preliminary empirical study. In: 12th European Conference on Software Maintenance and Reengineering. IEEE (apr 2008). https://doi.org/10.1109/csmr.2008.4493327
16. Pribisalic, M.: Automatic generation of test cases from use-case specification using natural language processing. In: 33rd Bled eConference - Enabling Technology for a Sustainable Society. pp. 725–734 (2020). https://doi.org/doi.org/10.18690/978-961-286-362-3.52
17. Ricca, F., Marchetto, A., Stocco, A.: Ai-based test automation: A grey literature analysis. In: IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW). pp. 263–270 (2021). https://doi.org/10.1109/ICSTW52544.2021.00051
18. Tahvili, S., Hatvani, L., Ramentol, E., Pimentel, R., Afzal, W., Herrera, F.: A novel methodology to classify test cases using natural language processing and imbalanced learning. Engineering Applications of Artificial Intelligence **95**, 103878 (oct 2020). https://doi.org/10.1016/j.engappai.2020.103878
19. Wang, C., Pastore, F., Goknil, A., Briand, L.C.: Automatic generation of acceptance test cases from use case specifications: An NLP-based approach. IEEE Transactions on Software Engineering **48**(2), 585–616 (feb 2022). https://doi.org/10.1109/tse.2020.2998503