Received: Added at production | Revised: Added at production | Accepted: Added at production

DOI: xxx/xxxx

ARTICLE TYPE

An Empirical Study to Compare Three Web Test Automation Approaches: NLP-based, Programmable and Capture&Replay

Maurizio Leotta[1*], Filippo Ricca[1], Alessandro Marchetto[2], Dario Olianas[1]

[1]Dipartimento di Informatica, Bioingegneria, Robotica e Ingegneria dei Sistemi (DIBRIS), Università di Genova, Italy
[2]Dipartimento di Ingegneria e Scienza dell'Informazione (DISI), Università di Trento, Italy

Correspondence
*Maurizio Leotta, Email: maurizio.leotta@unige.it

Abstract

A new advancement in Test Automation is the use of Natural Language Processing (NLP) to generate test cases (or test scripts) from natural language text. NLP is innovative in this context and promises of reducing test cases creation time and simplifying understanding for "non-developer" Software Testers as well. Recently, many vendors have launched on the market many proposals of NLP-based tools and testing frameworks but their superiority has never been empirically validated. This paper investigates the adoption of NLP-based Test Automation in the web context with a series of case studies conducted to compare the costs of the NLP testing approach — measured in terms of test cases development and test cases evolution — with respect to more consolidated approaches, i.e., programmable (or script-based) testing, and capture and replay testing. The results of our study show that NLP-based Test Automation appears to be competitive for small to medium sized test suites such as those considered in our empirical study. It minimizes the total cumulative cost (development and evolution) and does not require Software Testers with programming skills.

KEYWORDS:
Test Automation, Web Testing, Natural Language Processing, Selenium, Page Object Pattern, Empirical Study.

1 | INTRODUCTION

Web applications are one type of application requiring fast time to market and excellent quality assurance. In fact, it is recognized that putting into production web applications that are late compared to expectations, or that users don't like, or even worse, that contain bugs can lead to the loss of customers [1, 2, 3]. A bug in a web application can cause both direct losses in corporate revenues, but also indirect losses due to the decline in customer loyalty and brand reputation.

Modern web applications are heterogeneous, distributed, complex, and they continuously run for providing everyday services. Therefore the quality assurance of the developed applications is a crucial aspect. Before the application release, the quality assurance process aims at ensuring that the application is free of bugs and meets high quality standards. Nowadays, standards and guidelines exist that support the quality assurance process [4, 5], however, it is fairly evident that software testing plays a critical role in the application quality improvement; it aims at discovering the presence of bugs and, thanks to the automatic execution of test cases [6], it can be effective and efficient. Automated test cases, in fact, can be run often and quickly with each application release, thus increasing the confidence of the released applications [7]. Test automation, hence, contributes to increase the application quality and makes the assurance process effective and efficient. Additionally, software testing and automated testing is also a crucial aspect for Continuous Integration (CI) processes, which form the foundation of DevOps [8] and agile methodologies since it supports the common goal of delivering quality and speed through a flexible process. The development process of web applications is characterized by frequent iterations, rapid changes in requirements, as well as rapid releases. However, such modern applications are developed and maintained by potentially large teams of developers working on the same application repository. Hence, building a CI pipeline can be a non-trivial and challenge. Test automation is an enabler of CI since it strongly contributes in automating the CI pipeline, making CI effective in practice.

Testing frameworks, such as Selenium WebDriver, Selenium IDE, and JUnit, are nowadays consolidated solutions to automate the test process [9, 10]. They have proven their value in practice by reducing the cost of manual testing and thus improving the quality of released applications [7, 9]. However, while useful, these frameworks also present challenges in their use [11]. In fact, developing executable test cases and then maintaining them is very expensive for companies. Furthermore, the development of test suites requires specialized personnel who must be technically able to produce good quality code and know how to use the testing frameworks. All of these reasons risk limiting the adoption of testing frameworks and thus the benefits to the applications under test [6].

Recently, a new category of tools and frameworks, often provided as SaaS (Software as a Service), named code-less and based on Artificial intelligence (AI) [12, 13] and Natural Language Processing (NLP) have appeared on the market. The major novelty of NLP-based test automation tools/frameworks is that the test cases are written in pure natural language, thus simplifying their production and understanding. The huge advantage is that with these tools even software testers with limited programming skills can produce executable test cases quickly.

Many vendors have understood the enormous potential of NLP based solutions in the context of web applications testing and thus have proposed several different testing automation frameworks and tools — e.g., TestSigma[a], TestRigor[b], Functionize[c], and TestProject[d] — able to interpret and execute test cases written in natural language. However, although very promising and interesting, the benefits of this new category of frameworks and tools, in terms of reduction in development and evolution times and thus in terms of costs, have not yet been proved in the field.

The purpose of this paper, which extends a previous one from a conference [14], is to present a series of case studies where this new NLP-based testing approach is compared with more established solutions. In particular, we chose a representative of the available NLP testing frameworks and compared it, considering test suite development and evolution time, with Selenium WebDriver and Selenium IDE, representatives of the categories programmable (or script-based) and capture&replay.

Our empirical work, started with our previous paper [14] then extended in this manuscript, makes the following main contributions:

- this is the first empirical study in the web context that compares the category of NLP based testing frameworks, able to simplify the writing and understanding of test cases, with more consolidated testing frameworks (i.e., Selenium WebDriver and Selenium IDE);
- this work extends the *empirical knowledge base* related to Web testing frameworks. This knowledge base is very useful because it can guide Company Managers to choose the most suitable category of testing frameworks/tools for their specific purposes;
- this empirical study shows with real data that this new category of NLP-based testing frameworks is really promising and that therefore software companies are right to propose/produce frameworks/tools of this type (or integrate NLP into already existing solutions).

The results of this empirical work may be of interest to both industrial and academic settings. Professionals can better understand and estimate the possible costs and returns of investments associated with the adoption of different web testing frameworks, while researchers, in addition to increase the empirical evidence on the use of different testing frameworks, can also be inspired to conduct new experiments or replications on the topic.

Compared to the published conference paper [14], all data of the empirical study were tripled, adding new applications belonging to different domains. The results obtained and discussed in the paper are now based on the development and evolution of nine test suites (instead of three in [14]) for different Web applications, built by three different software testers (only one

[a] https://testsigma.com/
[b] https://testrigor.com/
[c] https://www.functionize.com/
[d] https://testproject.io/

in [14]), thus reinforcing the generalizability of the results. Furthermore, additional diagrams, analysis, and statistical tests, with respect to the ones of the conference paper, have been carried out to further analyze relevant aspects, and present the outcomes, of this larger study.

This paper is organized as follows. Section 2 describes the three different testing approaches (i.e., programmable, capture&replay, and NLP-based) and the tools we selected to execute the empirical study. Section 3 describes the main aspects of the empirical study we carried out to compare the three approaches, i.e., design, experimental objects (i.e., the web application chosen), subjects (i.e., the software testers implementing the test suites), research questions, and procedure. Section 4 reports the results of the study and ends with a discussion on the pros and cons of the various approaches. Finally, Sections 5 illustrates related works and Section 6 concludes the paper.

2 | BACKGROUND

End-to-End (E2E) testing of web applications is a type of black box testing based on the concept of test scenario [15]. Test scenarios are sequences of steps/actions performed on the web application mimicking the actual actions performed by the web application users (or a manual Tester). For example, testing a login functionality could require to execute the following actions: insert username, insert password, click the login button, and verify that the user is authenticated in the web application. Starting from each test scenario, one or more test cases can be derived by specifying the actual data to use in each step (e.g., *username=John.Doe*) and the expected results (i.e., defining the assertions). The execution of each test case can be automated by implementing a corresponding test script following any of the existing approaches. The choice among the various approaches could depend on different criteria including, e.g., the technology used in the web application implementation, the available tools (e.g., Selenium WebDriver and Selenium IDE[e]), and the proficiency in coding of the involved Testers [6]. In this work, we consider and compare three different testing approaches: programmable web testing (PT) (also called script-based) and capture&replay web testing (CRT), that both represent two relevant choices for implementing web test scripts, and the novel NLP-based web testing (NLT) approach. In the following of the section we briefly introduce them.

2.1 | Gherkin

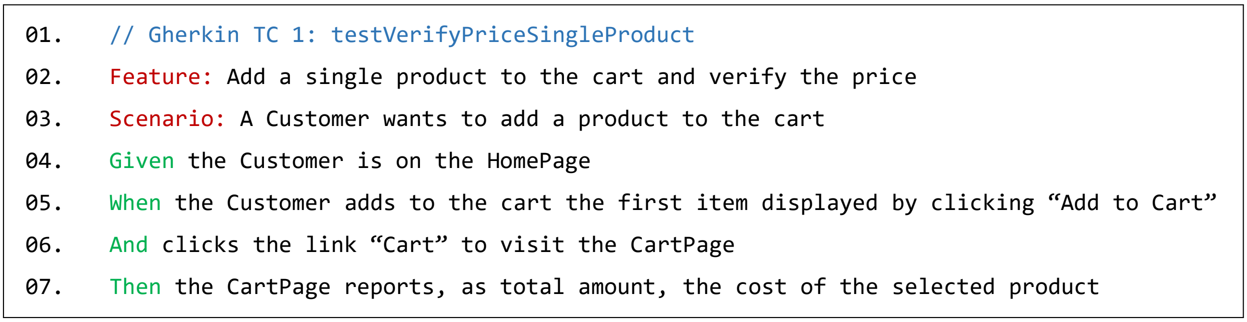
Before developing the test scripts relying on one specific testing approaches it is useful to describe/define the test cases to implement using a specification language. Gherkin[f] is a language widely used in this context [10]. More in detail, Gherkin is a DSL language that allows to describe the behavior of the software without specifying implementation details: therefore, it is suitable for sharing the description of usage scenarios within Software Teams and thus often employed as a test specification language. 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 shown in Figure 1 describes a simple example in which Gherkin is used to specify a test case for an online e-commerce application. The test's goal is to verify a product's correct price when it is added to the shopping cart.

[e] https://www.selenium.dev

[f] https://cucumber.io/docs/gherkin/reference/

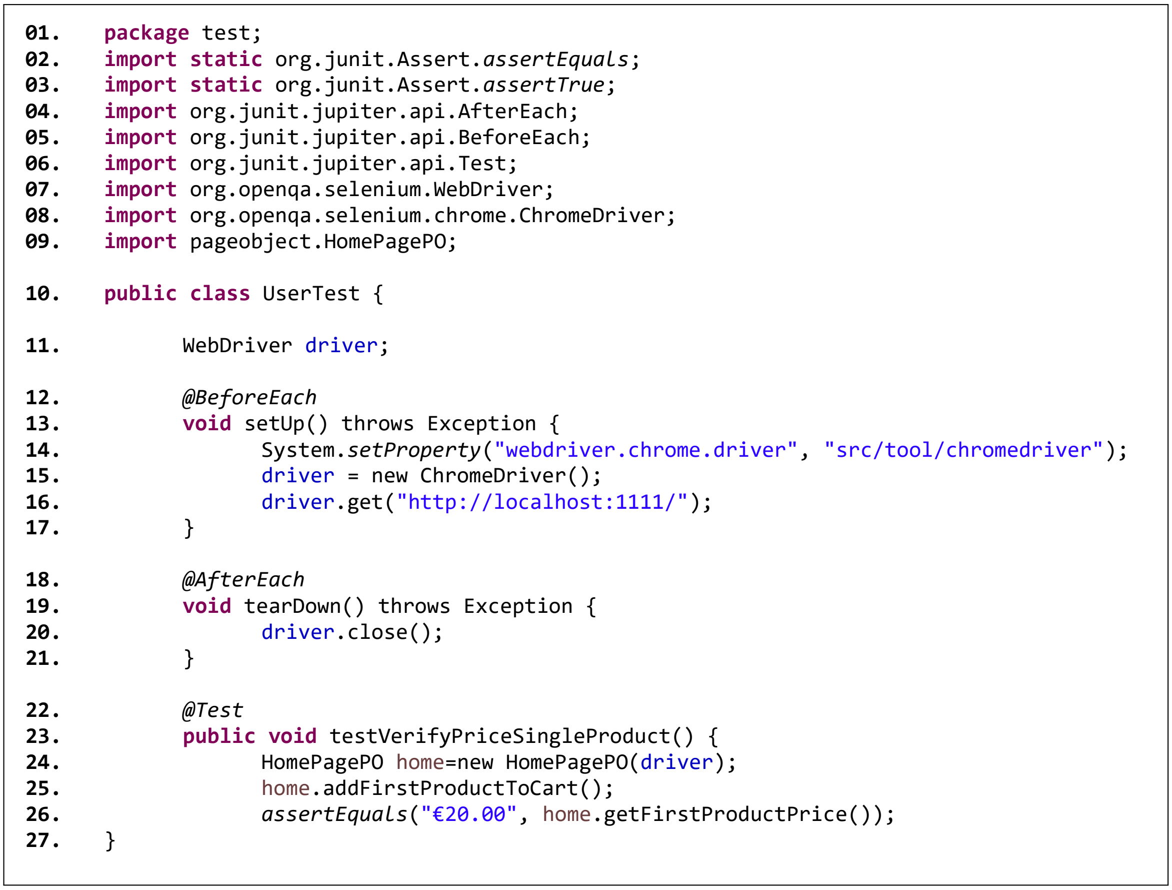


```
// Gherkin TC 1: testVerifyPriceSingleProduct
Feature: Add a single product to the cart and verify the price
Scenario: A Customer wants to add a product to the cart
Given the Customer is on the HomePage
When the Customer adds to the cart the first item displayed by clicking “Add to Cart”
And clicks the link “Cart” to visit the CartPage
Then the CartPage reports, as total amount, the cost of the selected product
```

FIGURE 1 Example of test case specified with Gherkin

2.2 | Programmable Web Test Automation (PT)

Programmable Web Test Automation is based on the development of web test scripts using programming languages (such as Java, Python, or Ruby) with the aid of specific libraries able to manage the browser execution. Usually, these libraries extend the programming language with powerful and user-friendly APIs, providing commands to interact with the web application such as click a button, fill a field, or submit a form. Test scripts are completed with assertions implemented using common libraries such as xUnit (e.g., JUnit, if the language chosen is Java) to check the obtained execution results.



```
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.HomePagePO;

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());
}
```

FIGURE 2 Example of PT test script

An example of programmable test script is reported in Figure 2; it represents a possible implementation for the Gherkin code 1 in Listing 1 relying on the state-of-the-practice tool Selenium WebDriver[g] [10]. The test script reported in the figure adopts the design pattern *Page Object*[h,i]. It is a popular web test design pattern, which aims at improving the test case maintainability and at reducing the duplication of code [15, 11] with possible benefits also during the initial development when the test suites are large [16]. Basically, each Page Object (PO) is a class that represents the web page elements as a series of objects and that encapsulates the functionalities provided by the web page into methods. Adopting the Page Object pattern in the test scripts implementation allows Testers to follow the *Separation of Concerns* design principle, since the test scenarios are decoupled from many implementations details. Indeed, such details (e.g., the locator used to specify the position in the DOM of a button to click) are moved into the Page Objects, a bridge between web pages and test cases, with the latter only containing the test logics. Thus, all the functionalities to interact with or to make assertions about a web page are offered in a single place, the PO, and can be easily called and reused within any test case.

Looking at Figure 2, we can see that in the test script the `@BeforeEach` and `@AfterEach` annotations are employed. They are constructs defining commands to be executed before and after the execution of the test script main body, respectively to set-up and reset the test script execution (e.g., open the web application and then close the browser). In the `@Test` method body, the steps are implemented by calling various methods provided by the POs (in this example HomePagePO), such as `addFirstProductToCart`, which contributes to the logic of the test cases, i.e., adding a product to the shopping cart. Assertions (`assertEquals` condition) are used to verify the price of the product added to the cart.

The advantage of programmable testing is its flexibility and the reusability of the test scripts. 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 scripts that can be executed even on different browsers [17, 18]. These benefits are amplified in particular when adopting specific patterns such as the *PO* pattern (as in the example reported here), allowing to reduce the coupling between web pages and test scripts, promote reusability, readability and maintainability of the test suites [15]. The drawbacks of programmable testing [15, 16], however, are the following: (i) Testers must have non trivial programming skills to adopt it; (ii) to be effective, programming guidelines and best practices typically used for software development must be followed; and (iii) a substantial initial effort is required to develop test scripts.

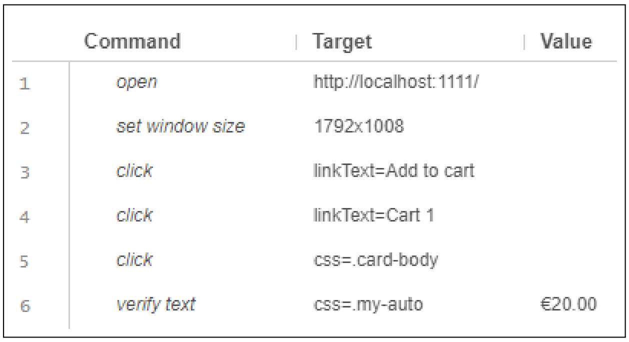
2.3 | Capture&Replay Web Test Automation (CRT)

Capture&Replay Web Test Automation (CRT) is often adopted 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, at the same time, records the whole execution session, including all user events and interactions with the web elements, as well as all keys pressed. Test scripts are automatically generated by the tool and can be used to replay the recorded testing sessions. Test scripts are hence executed by replaying the whole recorded sessions that are usually also enriched with assertions (e.g., at the end of the test scripts) for checking the result of the re-execution. Testers can also customize each re-execution by changing input values and assertions, that can also be parametric to make the test scripts more flexible.

As an example, Figure 3 shows a test script recorded with a capture&replay tool that implements the Gherkin code in Listing 1. In particular, we used Selenium IDE[j], a well-know and used tool supporting capture&replay web testing [10]. The Selenium IDE test script begins by opening the main page of the web application and then several `click` operations are performed to add a product to the shopping cart; finally, the textual content of a specific web element of the current page (the one having css=.my-auto) is checked with an assertion (`verify text`).

The advantage of capture&replay tools [15, 6] is that this kind of test scripts are relatively simple to produce. Hence, even Software Testers without programming skills are able to build complex and advanced 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, consequently, difficult to modify, e.g., during evolution.

[g] https://www.selenium.dev/documentation/webdriver/
[h] http://martinfowler.com/bliki/PageObject.html
[i] https://code.google.com/p/selenium/wiki/PageObjects
[j] https://www.selenium.dev/selenium-ide/



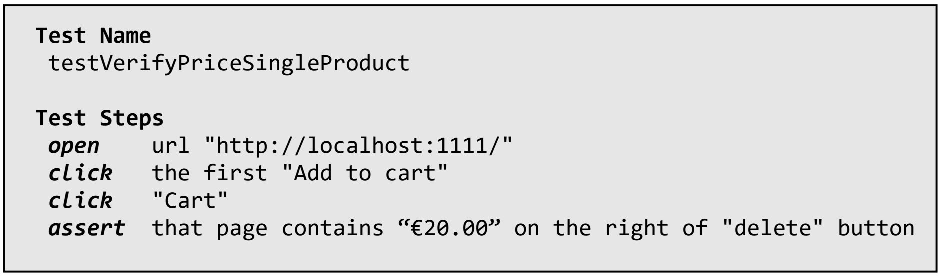
	Command	Target	Value
1	*open*	http://localhost:1111/	
2	*set window size*	1792x1008	
3	*click*	linkText=Add to cart	
4	*click*	linkText=Cart 1	
5	*click*	css=.card-body	
6	*verify text*	css=.my-auto	€20.00

FIGURE 3 Example of CRT test script

2.4 | NLP-based Web Test Automation (NLT)

NLP-based Web Test Automation (NLT) uses NLP techniques to let Software 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.

As an example, Figure 4 shows a fragment of a test script that implements the Gherkin code in Listing 1. We can see that the test script is written as a sequence of simple natural language sentences in which verbs such as `open`, `click`, `assert`, and their synonyms, are used to describe the actions to be executed on the application under test. Many examples of commercial tools supporting NLT appeared recently on the market such as TestSigma, TestRigor, Functionize, and TestProject.



```
Test Name
 testVerifyPriceSingleProduct

Test Steps
 open    url "http://localhost:1111/"
 click   the first "Add to cart"
 click   "Cart"
 assert  that page contains “€20.00” on the right of "delete" button
```

FIGURE 4 Example of NLP test script

The use of NLP in the web testing context emerged recently. For this reason, to the best of our knowledge, there are no studies that analyze the pros and cons of its adoption, including comparison with other existing approaches. The expected benefit of NLP testing is the reduction of effort required for human Testers to develop (and understand) test scripts. Furthermore, specific programming skills are not required as for the programmable approach. NLP is used to write test cases in natural language, and using specific tools, to transform such descriptions in executable test scripts and to run them. A possible drawback, however, could be that a powerful NLP engine, able to interpret the natural language, is necessary to transform natural language test cases into effective executable test scripts able to exercise the web application under test. In fact, it is known that understanding natural language is very complex for a machine (in particular the phases of semantic analysis and disambiguation) [19].

3 | CASE STUDY DESIGN

In this section, we detail the planning and the design of the series of case studies we carried out to compare three considered web testing approaches: programmable testing (PT), capture&replay testing (CRT), and NLP-based testing (NLT) following the template and the guidelines by Wohlin et al. [20]. In the following of this section, we describe the design of the case studies.

3.1 | Study Design

The goal of this work is to analyze and compare three web testing approaches, ***PT***, ***CRT***, and ***NLT*** with the purpose of quantifying both short-term and long-term (i.e., across multiple versions) effort required in two main testing scenarios: (1) test script development and (2) test script evolution. Indeed, we are mainly interested in comparing the effort required for the initial test suite implementation and the effort needed for the test suite evolution across subsequent application versions.

The results of this work can be of interest to both (i) practitioners (developers and managers) who, in their work activity, need to understand and estimate the possible costs and returns of their investment associated with the adoption of the different web testing approaches; and (ii) researchers that look for empirical evidence about the usage of the different testing approaches.

The context of the study is defined as follows. Four participants were involved in the study: one of the authors, who defined the test specifications, and three junior professional web Testers/Developers, who implemented the test scripts with the three approaches. The experimental objects of the study are nine open-source web applications.

3.2 | Web Testing Tools Representative of the Three Approaches

As representative tools supporting these three testing approaches, we chose Selenium WebDriver (PT) and Selenium IDE (CRT) since they are well-known and widely used in industry [10, 11]. Among the NLT available tools, we chose a commercial tool, according to a thorough analysis we conducted. We basically analyzed (both through studying the available documentation and testing them with trial versions) many commercial tools stating NLP web testing capabilities. Among the considered NLP tools we selected one that: (1) fully supports the NLP on the test case descriptions (i.e., it is an actual NLP-based tool), (2) can be used to implement test suites for the case studies without paying a fee (i.e., a free trial license is available without limitations in the functionalities offered). We cannot disclose the name of the NLT commercial tool used due to proprietary reasons.

3.3 | Software Objects

To perform our experiment, we took into account nine web applications (experimental objects) named: expressCart, Shopizer, OIM, PrestaShop, Kanboard, Bludit, MantisBT, Joomla!, and Simple Machine Forums (SMF). We selected these applications since they: (1) are medium-size applications; (2) are good representatives of typical web applications in terms of functionalities they provide and technologies they use, such as programming languages, databases, libraries and frameworks; and (3) provide at least two major versions available. The last aspect is particularly relevant in our study for the estimation of the test script evolution effort, i.e., the effort required to evolve and reuse the test scripts across software versions. As suggested in the Semantic Versioning 2.0.0 specification[k], each software version in a code repository like Github[l] is labeled with a three-digit schema: *MAJOR.MINOR.PATCH* (e.g., 1.02.001), where MAJOR should be incremented when incompatible API changes are made, MINOR when (backward-compatible) functionalities are added, and PATCH when (backward-compatible) bugs are fixed. This schema indicates that breaking changes are expected to be present only in major releases, even if studies [21] showed that in practice, also some minor versions can introduce at least one breaking change. In our study, we opted for considering only major versions, small changes between the minor applications versions can lead to a large reuse of test scripts, thus limiting the amount of evolution empirical data for our study. We applied a two-step process to select the versions to be considered in the study: (1) we looked at the code version system of the application, searching for two consecutive major versions; (2) we verified if code changes of such versions were also related to major breaking changes (e.g., incompatible API changes). For each application, we consider two subsequent major versions available in the application code repository that expose major *logical* and/or *structural* changes [15]. A logical change is a change to a system functionality that foresees the modification of the process underlying the specific functionality, while a structural change is a change to the application structure that leads only to some changes to the elements, e.g., of the application GUI layout/structure. As an example, imagine to have a logical change in the test case reported in Figure 1. The change is the following: when the user adds an element to the cart, the system shows directly the cart page. Thus the step *06. And clicks the link "Cart" to visit the CartPage* should be removed. This kind of change impacts all the three test scripts (PT, CRT and NLT, see respectively Figure 2, 3, and 4). On the contrary, a structural change on the page where the click is performed, i.e., *06. And clicks the link "Cart" to visit the CartPage*, could affect the various test implementation differently: (1)

[k] https://semver.org
[l] https://github.com/

if the link text changes (from Cart to MyCart), the NLT test should be modified; (2) if the css=.card_body locator changes the CRT (and the PT in case the same locator is used) implementation should be modified.

In the following, we provide a short description of each web application. expressCart[m] 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, Express.js and MongoDB. Shopizer[n] is another e-commerce application, mainly written in Java, that implements functionalities such as: catalog management, shopping carts, marketing components, smart pricing management, ordering, payment and shipping management. OIM[o] 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[p], a web-database framework for applications building. PrestaShop[q] is an open-source e-commerce web application written in PHP. It provides many functionalities to create and manage an online store. It is possible to set up a store with some demo data at the installation time. The admin panel provide many settings: for example, from the products' management to modify store configurations. Kanboard[r] is a project management web application that focuses on the Kanban methodology. It is a web-based solution that provides project management through a drag-and-drop interface and also automation, tagging, and scheduling functionalities. Bludit[s] is a content management system (CMS) to create Websites and Blogs. It is a Flat-File system, meaning that Bludit uses JSON files to store the content. Users don't need to install or configure a database, and they only need a web server with PHP support. Joomla![t] is a free and open-source content management system (CMS) for publishing web content on websites. Web content applications include discussion forums, photo galleries, e-Commerce and user communities and numerous other web-based applications. Mantis Bug Tracker (BT)[u] is a free and open-source, web-based bug-tracking system. It is usually employed to track software defects. However, users often configure MantisBT to serve as a more generic issue-tracking system and project management tool. Simple Machines Forum (SMF)[v] is an open-source web application that provides Internet forum and message board services.

Table 1 summarizes some important information of the Web applications that we have chosen as experimental objects of our study.

Application	Domain	Technology	Frameworks	1st release	Last commit	#Star	#Contr.
expressCart	E-commerce	Javascript	Node.js, Express.js, MongoDB	2017	Jan 2023	2.2k	23
Shopizer	E-commerce	Java	Spring framework	2015	Apr 2023	2.7k	5
OIM	Inventory management	PHP	AppGini app	2018	Mar 2022	5	1
PrestaShop	E-commerce	PHP	Symfony	2007	May 2023	7.2k	794
Kanboard	Project management	PHP	Symfony, PHPUnit	2014	May 2023	1.7k	331
Bludit	CMS	Javascript, PHP	no frameworks	2015	Feb 2022	1.1k	77
Joomla!	CMS	PHP	Symfony	2005	May 2021	4.5k	772
MantisBT	Bugs tracker	PHP	no frameworks	2000	May 2023	1.5k	132
SMF	Forum and messaging	PHP	no frameworks	2003	May 2023	504	94

TABLE 1 Experimental Objects of the study. Last commit, number of stars (stars are a metric related to trust and quality in a project) and number of contributors were obtained from GitHub projects of the related applications.

[m] https://github.com/mrvautin/expressCart
[n] https://shopizer-ecommerce .github.io/documentation/#/starting
[o] https://bigprof.com/appgini/applications/online-inventory-manager
[p] https://appgini.en.softonic.com/
[q] https://www.prestashop.com/
[r] https://kanboard.org/
[s] https://www.bludit.com/
[t] https://www.joomla.org/
[u] https://www.mantisbt.org/
[v] https://www.simplemachines.org/

3.4 | Participants

The test development and evolution tasks on the nine web applications has been carried out by three junior Testers/Developers. They conducted the experiment under the supervision of the researchers. They are web Testers/Developers with about 2/3 years of experience in the E2E web testing field. They have good knowledge of Selenium IDE and WebDriver test suites. Moreover, before performing the experiment, they practiced with the tool we selected for implementing the NLT approach by following tutorials and creating several sample test suites for various web applications of their choice. Finally, to complete the training, as part of the experimental procedure (see Section 3.6 for additional details), they created a test suite for a web application we provided with all the three different testing approaches.

3.5 | Research Questions and Metrics

To explain the research questions and the metrics we used to answer them, we introduce the following notation.

Let V_1^{app} be the first Version of a generic web application *app* under test (e.g., one of the applications we selected for our empirical study), while V_2^{app} be the next major version of the web application *app*. Instead, let TS_1^{app} be the Test Suite that a Software Tester developed for V_1^{app}, while let TS_2^{app} be the test suite associated to V_2^{app}.

Our study investigates the following research questions:

- **RQ1: Developing Time**. What is the development effort required for producing TS_1^{app} by adopting NLT with respect to more traditional approaches, such as PT and CRT?
- **RQ2: Reuse**. How much of TS_1^{app} produced with a NLT approach can be reused "as-is" with respect to more traditional approaches such as PT and CRT, when a new version of the application (i.e., V_2^{app}) needs to be tested?
- **RQ3: Evolution Time**. What is the effort required for evolving TS_1^{app} in TS_2^{app} with the three testing approaches considered, when a new version of the application (i.e., V_2^{app}) needs to be tested?
- **RQ4: Trend in Versions.** How the cumulative effort, computed 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 versions?

The first research question concerns the development cost in terms of the time required to create the test suites starting from test specifications (i.e., Gherkin in our case). In this way, we can precisely compare the cost of adopting NLT in terms of production time required with respect to the time required when adopting more traditional approaches, i.e., PT and CRT. The results of this comparison provide the practitioners with the initial investment needed to adopt the various testing approaches. To answer *RQ1*, we measured the development effort of TS_1^{app} in terms of time (minutes) needed by the Testers to create/produce the executable test scripts. We compared the different development efforts and estimated the ratio between NLT and the other two traditional approaches.

The second research question concerns the analysis of the resilience to changes of the developed test suites. In particular, we aim to evaluate the capability of the test suites, produced with the three considered testing approaches, to be reused to test new major versions of the application under test. This could provide practitioners with an estimate of the capability of the testing approach to implement reusable test suites, i.e., test suites that can be reused "as-is" to test a new software version. To answer *RQ2*, we executed the test suites TS_1^{app} (for each app and each testing approach) against V_2^{app} and counted the number of test scripts contained that passed without modifications. This number corresponds to the number of test scripts contained in TS_1^{app} reusable "as-is" without modifications for testing V_2^{app}.

The third research question concerns the cost of evolving TS_1^{app} in TS_2^{app} (i.e., *evolution cost*) in order to make TS_2^{app} working on V_2^{app}. We aim to understand whether a testing approach requires additional evolution costs, with respect to others, and to estimate the ratios between those costs. This could give practitioners an idea of the effort needed to make test suites usable across subsequent software versions. To answer *RQ3*, we considered V_2^{app}. In detail, we evolved TS_1^{app} in TS_2^{app} to make them usable also for testing V_2^{app}. For each test suite, the evolution effort was measured in terms of time (minutes) the Software Tester spent to fix the test scripts that cannot be executed directly against V_2^{app}.

The last research question concerns the return on investment (ROI) that can be achieved when adopting the three considered testing approaches. In detail, this RQ aims to understand how the cumulative testing effort (considering both the development and evolution efforts) required to apply the NLT approach varies across application versions. Such effort is compared with that of the more traditional approaches PT and CRT. In this way, practitioners are provided with an estimation of the overall effort needed by the various testing approach. To answer *RQ4*, we computed the cumulative testing effort for each approach as proposed in [6] and estimated the number of application versions after which the cumulative effort trend changes. For example, given C_0 and N_0 the effort required for the initial development of CRT and NLT test suite, respectively, and given C_1, C_2,... and N_1, N_2,... the test suite evolution effort associated with the successive application versions. We are seeking the lowest value n such that: $\sum_{i=0}^{n} C_i \geq \sum_{i=0}^{n} N_i$. That value corresponds to the version number after which NLT test scripts start to be cumulatively more convenient than CRT ones.

Figure 5 shows a graphical representation of an example where NLT costs are shown in green (dotted line) while CRT costs are in violet (dashed line). From the Figure it is clear that the initial development cost of NLT is greater than the one of CRT but its lower evolution cost allows to obtain a lower cumulative cost in the long term.

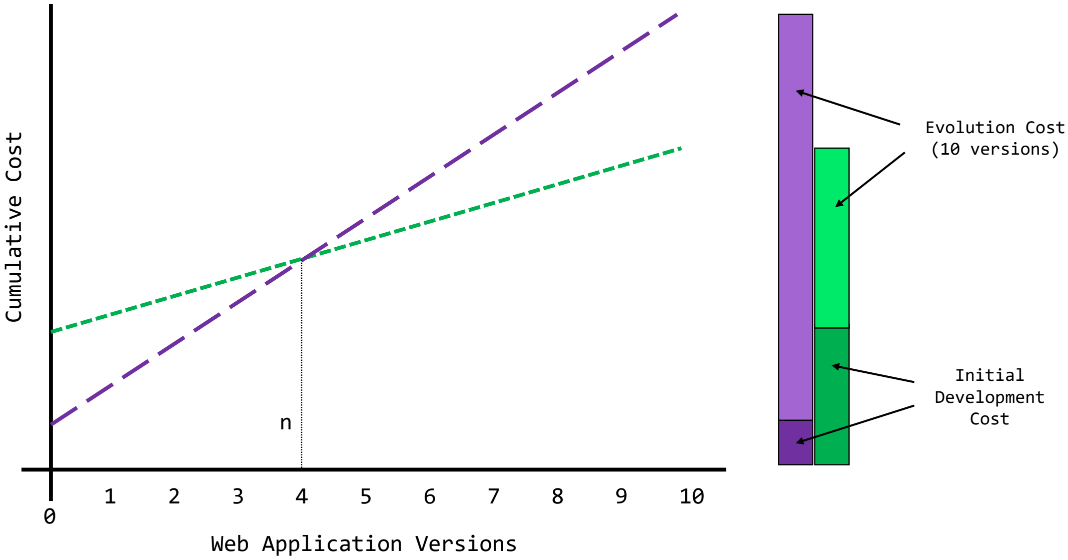


FIGURE 5 Example of Initial, Evolution, and Cumulative costs when comparing two web testing approaches

More in detail, under the simplifying hypothesis useful to make the calculation feasible 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 versions), the equation above can be solved as follows: $\frac{N_0 - C_0}{C - N}$. Thus, after n versions, the cumulative effort of the initial development and evolution of NLT test scripts is lower than the one of CRT test scripts. In the example reported in Figure 5 this corresponds to $n = 4$.

In general, the approach that has the lower evolution cost per versions will be the more convenient in the long term regardless of the initial cost. It is interesting to note that negative values of n can be obtained: this means, for example in case of NLT vs CRT, that the cumulative cost of the NLT is always lower than the one of CRT and happens whether also the initial development effort of NLT is lower than that of CRT (see an example in Figure 6 where $n = -1.25$). In the rare case the two straight lines are parallel a value for n cannot be computed. With the same formula we can estimate the value of n for NLT vs PT and CRT vs PT. By computing n, we provide the practitioners of an estimate about when the cumulative investment (considering both development and evolution effort) due to the adoption of a given testing approach could become favorable with respect compared testing approaches.

3.6 | Procedure

In our empirical evaluation we compared three different approaches, i.e., programmable web testing (PT), capture&replay web testing (CRT), and NLP-based web testing (NLT) while executing two different testing tasks: (i) test suite development and (ii) test suite evolution. Thus, two subsequent application versions (i.e., the experimental objects of the study) have been considered for each of the nine web applications included in the study (e.g., $V_1^{ExpressCart}$ and $V_2^{ExpressCart}$).

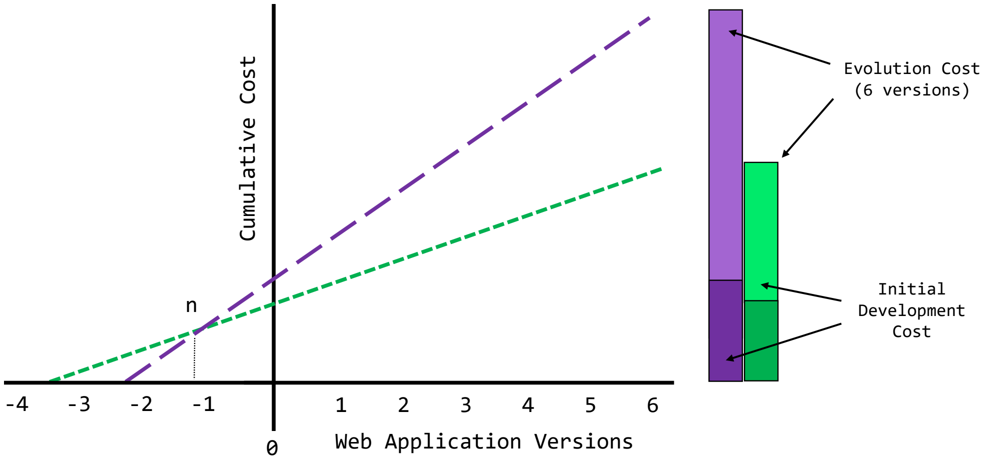


FIGURE 6 Example of Initial, Evolution, and Cumulative costs when comparing two web testing approaches. Case in which the estimated *n* assumes a negative value.

Initially, a preliminary training phase has been carried out to ensure a good and uniform level of familiarity with all three tools (each implementing one of the considered approaches). In particular, we first provided the Software Testers, that already have a good knowledge of Selenium IDE and WebDriver, with some guides, tutorials on the tool implementing NLT and asked them to develop some test suites to better understand its functioning. Then, as a final training task, we asked them to create a test suite for the PetClinic[w] application with all the three different testing approaches, by starting from the Gherkin test specifications we provided. In this way, a uniform level of knowledge of the approaches, and their corresponding frameworks/tools, is ensured among the three Software Testers.

In detail, the following procedure has been executed to complete each case study (i.e., for each web application considered) by one of the three Software Testers (i.e., the one in charge of carrying out development and evolution tasks for that specific app) and one of the authors. Note that each Software Tester worked on three different web applications applying on each of them all the three different testing approaches in turn.

1. *Web application selection*: the Tester under the supervision of one of authors selected an open-source web application following the criteria explained in Section 3.3.

2. *Web application analysis*: the application has been analyzed by both the Software Tester and one of authors of this paper to obtain knowledge about it. In particular, they analyzed the functionalities provided, the documentation, and the technology used to implement the application.

3. *Test cases definition*: a test suite specification has been defined for the web application at hand by one of the authors. This required to precisely describe a set of end-to-end functional test cases. The main functionalities provided by the first version of the application V_1^{app} (and still available in the second version V_2^{app}) have been covered at least once. To maintain reasonable the effort required to complete the development and evolution tasks of the test suites we mainly considered the normal case behaviors (e.g., a successful login scenario) and only a few incorrect or unexpected corner case behaviors (e.g., only incorrect password, but not all the possible cases of login test scenarios with incorrect/incomplete credentials). This has been done also to reduce (as much as possible) redundancy among test scripts. The Gherkin language has been employed to specify the test cases.

4. *Test scripts development*: for the initial version V_1^{app}, the Software Tester implemented the test scripts using PT, CRT and NLT, following the previously created test cases specifications. In this way, each Tester developed three executable test suites for testing V_1^{app} by using the three different tools considered in this case study. The three test suites developed for each application are completely equivalent from the functional point of view. Indeed, they test exactly the same functionalities in the same way since they have been developed precisely following the defined Gherkin test specifications. To balance as

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

much as possible the learning effects in the experiment, the order of test suites development has been alternated. Thus, each Tester developed the test suites for the three web applications assigned to him in the following orders: [PT, CRT, NLT], [CRT, NLT, PT], and [NLT, PT, CRT]. Moreover, the development of the test suites for the three applications has been interleaved in order to reduce learning about the same app: this allowed to separate the development of two test suites for the same app by at least 15 days. This means that basically the Tester developed the test suites for the three applications in the following order [PT *app*1, CRT *app*2, NLT *app*3, CRT *app*1, NLT *app*2, PT *app*3, NLT *app*1, PT *app*2, CRT *app*3].

Recorded data: for each test script, the Tester noted down the start and stop times (hh:mm:ss) of the development task.

5. *Test script evolution:* the three executable test suites built at the previous point (i.e., TS_1^{app}) have been executed, by the Tester, on V_2^{app}. This allowed to identify the failing test scripts, i.e., those test scripts that, due to application changes between the first and the second application version, report a failure or an error. The reasons behind the detected failures can be due to either structural or logical changes implemented in the second version V_2^{app} with respect to the previous version of the same application. Thus, the Tester repaired the failed test scripts so that the full test suites produced (TS_2^{app}) can be executed without problems also in the second version (V_2^{app}) of the applications under test. Also in this case, to balance as much as possible the learning effects in the experiment, the order of test suite evolution has been alternated. Thus, each Software Tester evolved the three web applications in the following orders: [PT, CRT, NLT], [CRT, NLT, PT], and [NLT, PT, CRT]. Moreover, the evolution of the test suites for the three applications has been interleaved (as described above for the development) in order to reduce learning about the same app: this allowed to separate the evolution of two test suites for the same app by at least 15 days.

   Recorded data: for each test script, the Tester noted down the start and stop times (hh:mm:ss) of the evolution task (in case a test script runs without problems the evolution time is zero). We thus have also the number of failing test scripts (i.e., the ones with evolution time > 0).

From the data recorded during the execution of the procedure we computed the total development and evolution effort for each test suite. Moreover, to analyze the statistical significance of the differences between couples of treatments (e.g., test scripts development times for PT vs. NLT approaches), we used both (a) the Kruskal–Wallis test [22] (considering all three approaches) and (b) the Wilcoxon paired test [23] with Holm correction [24] to compare the effects when considering couples of approaches (PT vs. NLT and CRT vs. NLT) while counteracting the problem of multiple comparisons. In all the performed statistical tests, we decided, as it is customary, to accept a probability of 5% of committing Type-I-error (α) [20]. Finally, we used the Vargha and Delaney's A test [25] to measure the strength of the relationship between couples of treatments (e.g., PT vs. NLT and CRT vs. NLT), where a statistically significant (Wilcoxon-based) value was observed. The Vargha and Delaney's A [25] value ranges between 0 and 1 and it can be interpreted according to four categories indicating the effect of the observed statistical relationship: negligible (N), small (S), medium (M), and large (L). These categories are useful for quantifying the observed effect size.

4 | RESULTS OF THE STUDY

In this section, we first report the quantitative results (Section 4.1) to answer the four research questions of our study, then we discuss the results (Section 4.2) also considering contexts different w.r.t. our setting (such as, e.g., large industrial test suites). Finally, we discuss the threats to validity (Section 4.3) of our empirical study.

4.1 | Quantitative Results

4.1.1 | RQ1: Developing Time

Table 2 reports general information about the developed test suites in terms of number and characteristics (e.g., lines of code) of test scripts developed. To compare the CRT and PT code, we exported the native CRT Selenese code (column "Sel lines") — the language used by Selenium IDE — in Java using the export feature provided by Selenium IDE.

In five test suites out of nine, as expected, the PT Java test code (excluded the Page Objects - POs) is shorter than the CRT one (column Java LOCs): since many technical details are moved in the Page Object methods, test scripts become simpler and shorter. However, for four test suites out of nine, the LOCs number for PT test scripts (Page Objects excluded, column test LOCs) exceeds the LOCs number for the CRT code exported in Java (column Java LOCs). This may seem counter-intuitive, but

Application	#Test scripts	Code						
		PT				CRT		NLT
		test LOCs	PO LOCs	Total LOCs	#PO	Sel lines	Java LOCs	Lines
expressCart	40	842	932	1774	18	635	934	361
Shopizer	28	506	483	989	7	273	417	150
OIM	32	462	1065	1527	18	552	765	351
PrestaShop	21	951	977	1928	21	343	596	224
Kanboard	20	735	724	1459	21	260	372	176
Bludit	21	684	683	1367	21	271	445	173
Joomla!	14	264	599	863	21	369	677	138
MantisBT	14	247	595	842	19	231	356	130
SMF	14	293	697	990	30	213	337	110

TABLE 2 Test suites code details

it can be explained by the structure of test suites. For three involved applications (PrestaShop, Kanboard, Bludit) the test suite has been organized with one class per test script (this was a choice of the Tester that developed them). In Java this creates a great abundance of LOCs, since all class declarations and, most importantly, all imports, must be repeated for each test script. The other application instead (Shopizer) makes extensive use of thread sleeps in the PT test suite (placed between invocations of Page Object methods). Thread sleeps are instructions that pause the test script execution for a given amount of time, and in E2E web testing they are widely used to wait for the page to be loaded to provide execution stability. Thread sleeps in Java are particularly verbose from a LOCs perspective since they must be enclosed in a try-catch block: in this way, a single thread sleep takes usually 5-6 lines of code. The Shopizer test suite contains 56 thread sleeps, and so the LOCs difference between PT and Java export of CRT can be easily explained.

Moreover, it is interesting to note that the number of NLT test script lines is always lower 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., two lines are required) while NLT provides a higher level command set (e.g., write “John” in the “name” field: i.e., only one line).

Table 3 reports the total test suite and average test script development effort (expressed in minutes) and the statistical difference observed (if any) between the distributions of the test development effort. In particular, we report: a) the result of the Kruskal–Wallis test for the three treatments, b) the results of the Wilcoxon paired test (with Holm correction) to compare PT and CRT against NLT and, c) the Vargha and Delaney’s A effect size, when a statistically-relevant relationship is observed. The last two columns of the Table report the effort ratio measured between PT and CRT against 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)			Kruskal Wallis	Wilcoxon paired p-value		Effect size VD.A		Ratio	
	PT	CRT	NLT	PT	CRT	NLT	p-value	PT-NLT	CRT-NLT	PT/ NLT	CRT/ NLT	PT/ NLT	CRT/ NLT
expressCart	316	46	157	9.6	1.4	4.7	**<0.01**	**<0.01**	**<0.01**	L	L	2.02	0.29
Shopizer	225	48	75	8.0	1.7	2.7	**<0.01**	**<0.01**	**<0.01**	L	L	2.98	0.63
OIM	310	86	93	10.0	2.8	3.0	**<0.01**	**<0.01**	0.07	L	-	3.33	0.93
PrestaShop	305	47	45	14.5	2.2	2.1	**<0.01**	**<0.01**	0.60	L	-	6.82	1.05
Kanboard	335	46	53	16.7	2.3	2.6	**<0.01**	**<0.01**	0.31	L	-	6.28	0.86
Bludit	315	42	61	15.7	2.0	3.0	**<0.01**	**<0.01**	**0.01**	L	M	5.16	0.68
Joomla!	256	45	99	18.2	2.3	7.0	**<0.01**	**<0.01**	**0.02**	L	L	2.58	0.33
MantisBT	263	47	102	18.7	3.3	7.3	**<0.01**	**0.04**	0.05	L	-	2.57	0.46
SMF	199	48	103	14.2	3.4	7.3	**<0.01**	**0.02**	0.09	L	-	1.92	0.46

TABLE 3 Test suite development time (minutes)

The Table shows that the development effort for PT is always higher than for NLT (p-value < 0.01 and large effect size – L), while there is also a trend, statistically relevant (with medium – M – to large – L – effect) for four out of nine 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 3), indeed PT required more effort than NLT in all the applications (PT/NLT ratio value is always higher than 1) and CRT required less effort than NLT in eight applications out of nine (CRT/NLT ratio is lower than 1 for eight out of nine cases). The result shows unequivocally that PT requires more development time than NLT, since the former requires to write the testing code (e.g., in Java) using the Selenium WebDriver API and the latter requires only to describe the test scenarios with a simpler step-by-step natural language description (e.g., derived from the Gherkin descriptions and enriched with several details to allow the tool to interact with the web page elements). At the same time, the results of our case study show also that CRT allows to produce test scripts 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 locators strings (e.g., "Cart" for the command click "Cart") to precisely locate the web page elements to interact with (with the CRT approach the locators are automatically generated during the recording phase). Conversely, the CRT approach is simpler as it is sufficient to replicate the various Gherkin steps directly on the application. By analyzing the developed NLT test scripts, we noticed that the Software Testers tried to describe the test actions/steps by using a simple natural language thus avoiding complex linguistic constructs; this has been done to simplify the task and to avoid NLP understanding problems with the NLT tool.

Figure 7 shows the box plots of Development time (in blue) partitioned by treatment for the test suites associated with the nine considered web applications: each box plot represents the distribution of the time required to develop the various test scripts for a specific testing approach. From the Figure, it is evident what has already been observed from the previous statistical analyses, i.e., the higher development time required by the PT approach w.r.t. the other approaches; on the contrary, the development times required by CRT and NLT are comparable even if in general CRT requires slightly less time.

RQ1. Summarizing, with respect to the research question RQ1, we can observe that: (i) PT requires the largest initial development effort; and (ii) there is a trend for which NLT requires more development effort compared to that required for CRT.

4.1.2 | RQ2: Reuse

Table 4 reports some information about the number of fixed/repaired test scripts, i.e., those test scripts developed for testing the application version V_1^{app} and that failed in exercising the application version V_2^{app}, thus requiring some effort to be fixed. In particular, Table 4 reports, for each testing approach: the number of fixed test scripts (the three columns “Fixed”), the result of the Kruskal–Wallis test on the three approaches, then the statistical difference (if any) between PT and CRT with NLT distributions, computed by using the Wilcoxon paired test with the Holm correction and finally, the Vargha and Delaney’s A effect size, when a statistically-relevant relationship is observed.

	PT	**CRT**	**NLT**	**Kruskal**	**Wilcoxon paired**		**Effect size VD.A**	
Application	**# Test**	**#Test**	**#Test**	**Wallis**	**p-value**		**PT/**	**CRT/**
	Fixed	**Fixed**	**Fixed**	**p-value**	**PT-NLT**	**CRT-NLT**	**NLT**	**NLT**
expressCart	19	23	19	0.34	0.33	1.00	-	-
Shopizer	17	16	17	0.95	1.00	1.00	-	-
OIM	26	28	15	**<0.01**	**0.01**	**0.03**	M	M
PrestaShop	21	17	20	0.06	1.00	0.23	-	-
Kanboard	14	20	20	**<0.01**	**0.02**	-	S	-
Bludit	16	16	15	0.90	0.76	0.76	-	-
Joomla!	12	13	9	0.14	0.23	0.71	-	-
MantisBT	11	14	6	**<0.01**	**0.04**	**<0.01**	M	L
SMF	13	14	11	0.15	0.34	0.15	-	-
total	149	161	132	-	-	-	-	-
average	16.5	17.8	14.6	-	-	-	-	-

TABLE 4 Test suites evolution: changes

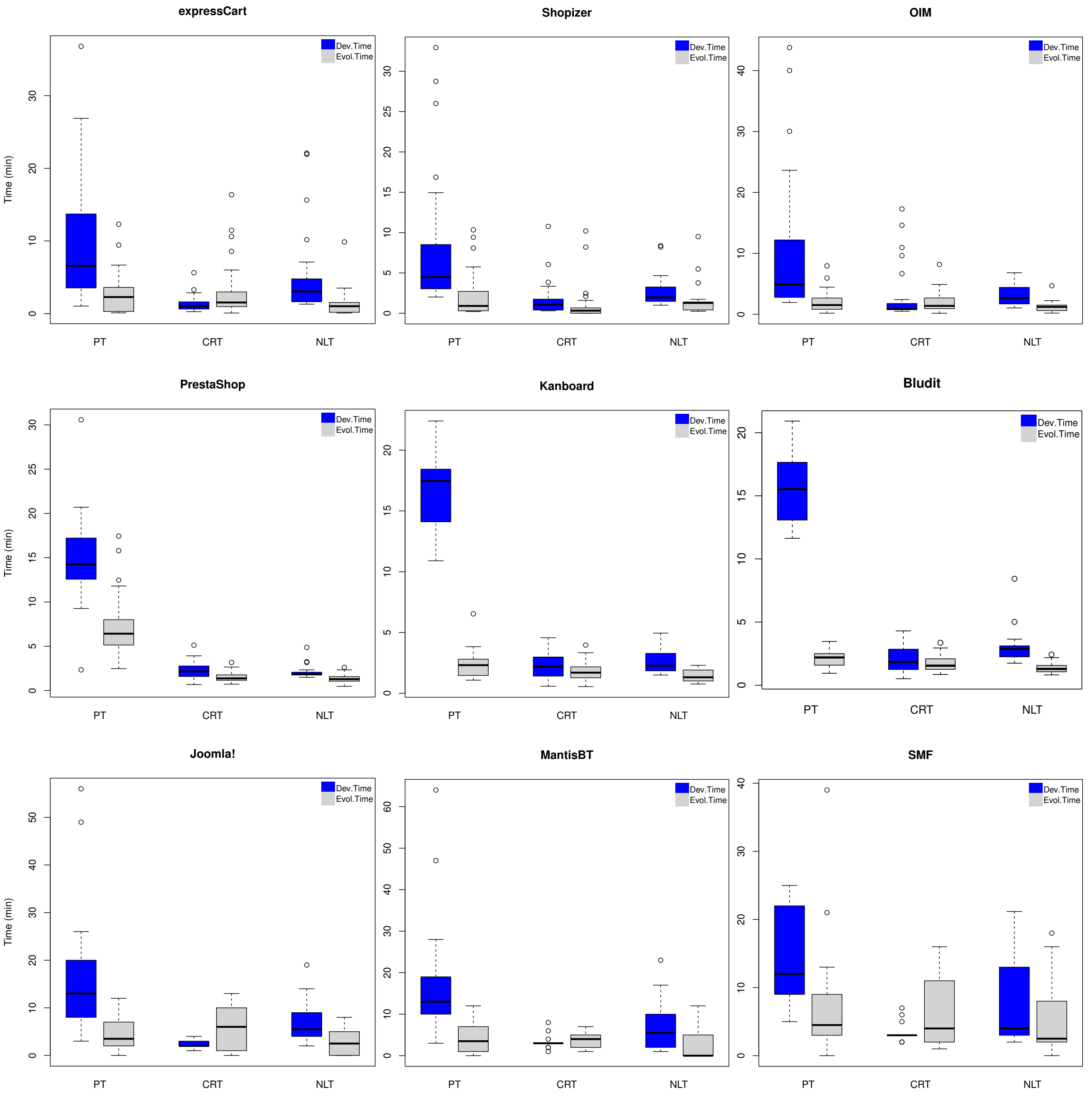


FIGURE 7 Boxplots of Development (blue) and Evolution (grey) time for the test suites associated with the eight considered web applications: distributions for PT, CRT, and NLT approaches are reported.

About the fixed test scripts, we mainly observe trends that are not statistically relevant in most of the cases, apart for OIM, MantisBT, and, partially, Kanboard. While for OIM and MantisBT, we observe that tests to be repaired differ significantly between PT/CRT and NLT (with medium – M – to large – L – effect), for Kanboard the observed difference exists only between PT and NLT (with small – S – effect). Finally, for the other applications no relevant difference has been observed. In general, we can observe that a large amount of test scripts needs to be fixed (in the range between 42.8% and 100%). CRT has the largest number of test scripts to be fixed, on average 86%, with respect to PT and NLT (respectively, on average 78.8% and 69.7%). While, in terms of variability for application, NLT has the largest variability, on average 20.5%, with respect to CRT and PT (respectively, on average 14.6% and 14%).

As we have already said, the changes between the two selected versions of the web applications V_1^{app} and V_2^{app} considered in the experimentation were of two types: structural and logical. Table 5 summarizes the type of changes for each application. It is possible to note that the number of changes is well distributed both between applications (except MantisBT) and types.

Application	Logical changes			Structural changes		
	PT	CRT	NLT	PT	CRT	NLT
expressCart	8	7	3	11	17	16
Shopizer	7	12	5	9	5	12
OIM	17	19	6	11	7	9
PrestaShop	6	14	7	17	7	10
Kanboard	8	6	4	9	14	7
Bludit	6	10	8	16	11	13
Joomla!	6	5	5	7	7	5
MantisBT	0	0	0	13	14	6
SMF	6	9	9	7	5	3

TABLE 5 Type of changes per application

As expected, PT and CRT show overall a similar levels of reusability (see Table 4) since they are based on the same DOM-based interaction paradigm [15]. 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. Probably, this is due to the fact that the NLT steps 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), suffering more from changes to the DOM.

RQ2. Summarizing, with respect to the research question RQ2, we can observe that: (i) CRT shows the lowest reusability, while (ii) NLT shows the highest test script reusability. This is probably due to the fact that NLT steps are more abstract with respect to the other approaches and not DOM based, and thus less affected by DOM structural changes.

4.1.3 | RQ3: Evolution Time

Table 6 reports: (a) general information about the evolution effort of the test suites in terms of time (expressed in minutes) required to fix the failed test scripts, (b) the result of the Kruskal–Wallis test on the three approaches, (c) the results of the Wilcoxon paired test with the Holm correction between PT-NLT and CRT-NLT and, (d) the Vargha and Delaney’s A effect size, when a statistically-relevant relationship is observed. The Table also reports the evolution effort ratio measured between PT and CRT with NLT. A value higher than 1 in the ratio between X and Y means that the X test suite required more evolution time than the corresponding Y test suite.

From Table 6 it is apparent that: PT and CRT required a higher evolution effort than NLT in almost all applications, even if the difference is statically relevant for respectively five out of nine applications for PT (with a medium/large – M/L – effect size for four application, while for Shopizer the effect is negligible – N) and one out of nine applications for CRT (with a small – S – effect size). Indeed, the penultimate column of Table 6 shows that PT has a ratio greater than 1 with respect to NLT for all the applications. While CRT shows, with respect to NLT (last column), a ratio greater than 1 in eight out of nine applications (the only exception is Shopizer).

Figure 7 shows the box plots of evolution time (in gray) partitioned by treatment for the test suites associated with the nine considered web applications: each box plot represents the distribution of the time required to evolve the test suite for a specific testing approach in order to make it working on V_2^{app}. From the Figure, it is evident the higher evolution time required by the PT approach; on the contrary, the evolution time required by CRT and NLT are comparable even if in general NLT requires slightly less time.

The fact that NLT takes less time to evolve failed test scripts than PT is reasonable since no programming effort is required in that case; to complete the evolution task with an NLT approach it is sufficient to modify the descriptive text of the test. On the other hand, NLT is also faster than CRT for eight applications out of nine (with only an exception, Shopizer): also in this case

Application	Total Time (min)			Average Time (min)			Kruskal Wallis	Wilcoxon paired p-value		Effect size VD.A		Ratio	
	PT	**CRT**	**NLT**	**PT**	**CRT**	**NLT**	**p-value**	**PT-NLT**	**CRT-NLT**	**PT/ NLT**	**CRT/ NLT**	**PT/ NLT**	**CRT/ NLT**
expressCart	88	96	45	2.7	2.9	1.3	**0.03**	**0.01**	**0.04**	M	S	1.97	2.14
Shopizer	62	30	42	2.2	1.1	1.5	**<0.01**	**0.03**	0.66	N	-	1.46	0.71
OIM	63	61	38	2.1	2.0	1.2	0.09	0.08	0.06	-	-	1.63	1.58
PrestaShop	156	32	30	7.4	1.5	1.4	**<0.01**	**<0.01**	0.50	L	-	5.28	1.09
Kanboard	48	37	29	2.4	1.8	1.4	**<0.01**	**<0.01**	0.26	L	-	1.63	1.24
Bludit	43	35	27	2.1	1.7	1.3	**<0.01**	**<0.01**	0.05	L	-	1.57	1.28
Joomla!	62	85	40	4.4	6.0	2.8	0.09	0.39	0.07	-	-	1.50	2.12
MantisBT	62	54	39	4.4	3.8	2.7	0.10	0.46	0.95	-	-	1.58	1.38
SMF	120	83	71	8.5	5.9	5.0	0.40	0.54	0.69	-	-	1.69	1.16

TABLE 6 Test suite evolution time (expressed in minutes)

editing the test description text seems to be simpler than directly editing the Selenese code, or re-recording the entire scenario. In Shopizer the trend is different, probably due to a banner for the user-management of the cookies, added in $V_2^{Shopizer}$, difficult to manage with the NLT tool. Indeed, in NLT, the banner required a few attempts to find 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. In our analysis, this motivates the different evolution time observed.

RQ3. Summarizing, concerning the research question RQ3, we can observe that: (i) PT requires a higher evolution effort compared to NLT; and (ii) the evolution effort required by CRT shows a high variability (but in eight cases out of nine is higher than the one required for NLT).

4.1.4 | RQ4: Cumulative Effort

Table 7 reports the estimated application version *n* in which we foresee a change of the cumulative testing effort trend. Concerning the adoption of NLT, Table 7 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 nine 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 seven 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 versions the cumulative costs of NLT are lower since it requires lower evolution costs. An exception is PrestaShop that exhibits a negative value in the column CRT-NLT, meaning that in this case, also the initial development effort of NLT is lower combined with a lower evolution effort of the NLT than CRT. Another exception is 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 difference w.r.t. the other applications could be attributable to the introduction of the banner (see the answer to RQ3).

Application	Application versions: 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
PrestaShop	NLT costs less for *n* > –2.0	NLT costs less for *n* > –0.9
Kanboard	NLT costs less for *n* > –15.0	NLT costs less for *n* > +0.9
Bludit	NLT costs less for *n* > –16.1	NLT costs less for *n* > +2.5
Joomla!	NLT costs less for *n* > –7.1	NLT costs less for *n* > +1.4
MantisBT	NLT costs less for *n* > –7.0	NLT costs less for *n* > +3.6
SMF	NLT costs less for *n* > –1.9	NLT costs less for *n* > +4.5

TABLE 7 Evolution cost: an approach that costs less starting from a version *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 NLT requires 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.

4.2 | Discussion

In this section, we describe the strengths and weaknesses of the three testing approaches, relying on the experience gained during the execution of the empirical study. Thus, we try to identify the best-use scenario for each approach also considering different application scenarios (such as, e.g., large industrial test suites).

4.2.1 | Programmable Web Testing (PT): Pros & Cons

The PT approach, in our opinion, is the most flexible and powerful one among the traditional E2E web testing approaches (i.e., vs. CRT). The developer has complete control over almost every aspect of the test script. Troubleshooting the problems is often easier than in other approaches, and there are plenty of frameworks for PT in different languages. But this comes with a cost: the development time of test suites using the PT approach is almost always higher than the ones required by the other approaches (this is also apparent in our series of case studies). In fact, manually writing the code for test suites is a time-consuming task, even for experienced web Testers. This is particularly true when the PO pattern is employed. As described in Section 2.2, the PO pattern prescribes to define a class for each web page of the application under test, and to encapsulate all page accesses inside of such class. This provides better readability and maintainability of the test suite's code, but it requires more LOCs to implement and thus more effort to adopt it. The advantages of the PT approach become evident during the evolution of the E2E test suites: in fact, having the test suite code organized in POs allows a quicker evolution process since the developer can quickly identify and fix the code that must be changed. This is particularly true as the number of test scripts increases to fully cover many test scenarios of the tested functionalities. In fact, as the reuse of PO methods between E2E test scripts increases, the advantage of being able to carry out the evolution task in a centralized way increases, i.e. in the methods of the POs rather than in the body of the test scripts. Moreover, adopting the PT approach allows the developer to choose the resilient locators [26, 27, 28] used to locate web elements and this can help to write more robust test suites (although this can be done, to some extent, also in the CRT approach). Finally, the PT approach allows to adopt more powerful assertions than the other two E2E approaches: PT testing frameworks offer plenty of different assertion kinds, and they can even be extended using frameworks dedicated to this aspect (for example, Hamcrest and AssertJ in the Java language [29]). Advanced assertions can make test scripts more efficient, with a lower rate of false negative (i.e., test executions that pass when they should fail). Finally, another advantage of the PT approach is the existence of free and open source tools, e.g., the Selenium ecosystem. These kind of tools can be executed also locally, so there is no need to pay fees/licenses for their usage or having vendor lock-in issues.

A downside of the PT approach is that, to be effective, it requires experienced developers that extensively know: the programming language used for developing the E2E test suite, the employed testing frameworks, and the application under test. An example of situation experimented in our series of case studies is that in the PT approach is harder to manage (w.r.t. CRT and NLT) the presence of 'iframes' in the web pages. The testing framework we employed for the PT approach, Selenium WebDriver, can handle iframes, but it must be done manually: when the test script has to interact with an element contained inside an iframe, the developer has to: (1) switch the WebDriver context to the desired iframe, (2) interact with the web element, and (3) restore the content to the main page. This is not a difficult task, but the problem is that the presence of iframes may not always be evident, at first sight, looking at the page layout. This may require, in advance, an inspection of the HTML code of the page (and so a considerable amount of time) to detect the presence of iframes. In our evaluation, even if the web Testers that developed the test suites had experience with handling iframes with Selenium WebDriver, a test script for the Joomla! web application required a considerable amount of time to be developed (56 minutes, out of 4h and 16 minutes of the total development time for the test suite) because the page contained multiple dynamically loaded iframes. Instead, for the other two approaches this problem did not occur because these testing frameworks are able to manage the iframes automatically, without the need for human intervention.

Given these characteristics, in our opinion and experience [30], the PT approach is an interesting solution for large-sized, industrial E2E test suites, composed by many complete and complex test scripts sharing many portions of the test sequences, even if it requires not trivial development skills. The adoption of the PO pattern and robust "ad-hoc" locators provide, in this context, a major advantage in terms of test suite evolution. Clearly, given that the experimental setting of our case studies is different, novel and specific empirical studies are needed with large industrial test suites to understand which approach is the best choice.

4.2.2 | Capture&Replay Web Testing (CRT): Pros & Cons

The main strength of the CRT approach is that it allows recording E2E test scripts very easily and quickly: no knowledge about coding is required. To create (or record) a test script, it is sufficient to perform the actions that an end-user would do while using the web application functionalities. The results of our evaluation reflect this aspect since in 8 applications out of 9 the CRT approach requires the lowest initial test suite development time. Another advantage of the CRT approach, shared with the PT one, is that free and open source tools implementing it are available on the Web (e.g., Selenium IDE and Katalon recorder). Also in this case, these tools can be executed locally, so there is no need to pay fees/licenses for their usage.

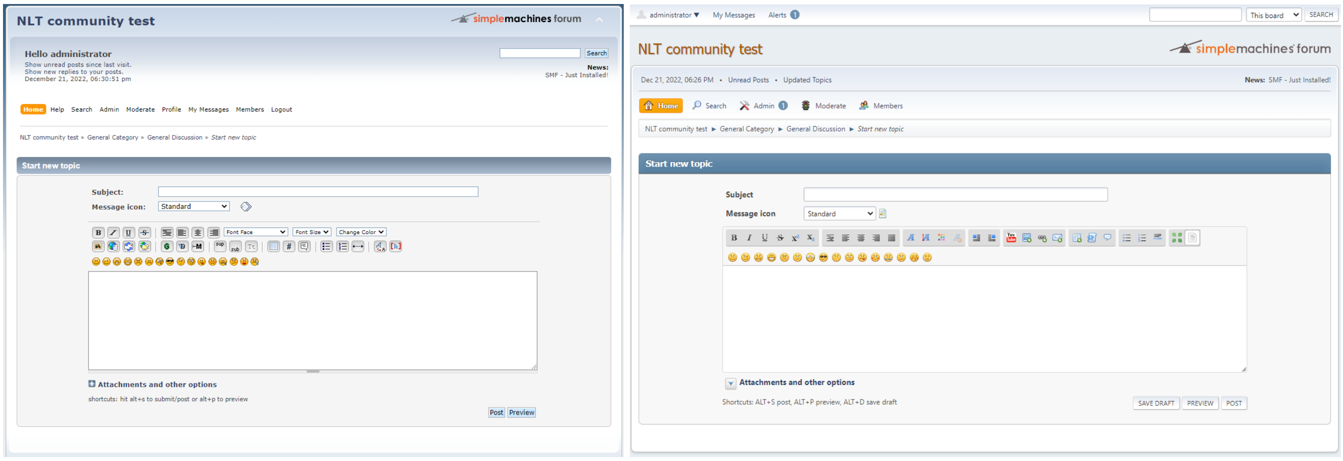
The main weakness of the CRT approach is that CRT test suites are hard to maintain, mainly because of the absence (or limitation) of features which promote code reuse: in fact, in E2E web testing, code reuse is fundamental to provide maintainability. E2E web test scripts usually share common parts that must be executed in many test scripts. A typical example is the set of actions required to login in a web application. Without the ability to factor and recall this group of actions, they must be manually repeated in each test script (another approach is to create a login-specific test script and then call it from all test scripts that need it, but this approach is rarely used by Software Testers). This is not a big problem when a small or medium-sized test suite is considered: since in the CRT approach test scripts are recorded quickly, if the common actions are not too long, recording them for each test does not have a significant impact on the initial development time, and allows the CRT approach to still maintain an advantage against PT and NLT. However, this could become a problem for large test suites made up of complex test scripts having many repeated actions in common. The evolution phase, instead, is the most affected by the absence of code reuse, even in small-medium test suites (as the ones considered in our study). In fact, if in the new version of the application under test there are changes affecting the common repeated parts, these parts have to be re-recorded for each test script that contains them, thus severely increasing the evolution time of the test suite. Even relying on copy and paste, although it is widely recognized as a bad practice in coding, it is not a quick task since the CRT tool we employed (Selenium IDE) does not allow to select a group of commands to be copied and pasted, but it must be done one command at a time. In 4 cases out of 9, evolving a CRT test suite required more time than developing it from scratch: in these cases, it would have been more convenient to re-record the test suites from scratch rather than evolving the already existing ones. The only upside of the CRT approach from an evolution perspective is that Selenium IDE provides an auto-correct functionality for locators: if an interaction with a web element fails (because for example it is not present in the page or because the considered locator points to a non-existent web element), the test execution does not fail immediately: indeed the tool tries to access the web element using another previously computed locator (in practice, each web element is associated with many locators and Selenium IDE maintains a list of them). In our experiment, this functionality provided a good success rate. However, this feature is also a downside from a test suite execution time perspective, since searching the web element using alternative locators can require several tenths of seconds. Ideally, to improve the performance, the developer should replace the original failing locator with the one suggested by the auto-correct functionality. Finally, another issue with Selenium IDE is that there have been problems with JavaScript alerts: even if they are supported by Selenium IDE, often the Software Tester must rearrange the recorded instructions related to a JavaScript alert acceptance in order to correctly replay the E2E test scripts.

To conclude, we think the CRT approach is a good choice for developing small test suites that don’t need major evolution activities, but should be avoided on large and complex test suites for heavily evolving web applications. CRT can be considered a good solution in case it is necessary to test quickly a small web application locally, employing a free tool (without fees) and with Software Testers having limited programming skills.

4.2.3 | NLP-based Web Testing (NLT): Pros & Cons

Finally, the NLT approach, in our opinion, represents a good compromise between the other two E2E approaches. It provides better maintainability than the CRT approach, and better development times than the PT approach: writing E2E test scripts with a tool supporting the NLT approach is easier than in the PT approach, but not as fast and easy as in the CRT approach. The better maintainability w.r.t. the CRT approach is mainly provided by the possibility of reusing code, which is almost missing in practice in the CRT approach. The NLT tool we employed in the empirical work provides code reuse by supporting the definition of functions (i.e. blocks of natural language instructions) that can be called inside of test scripts, similarly to functions of programming languages. Moreover, NLT test scripts are often more resilient to evolution since steps/actions are specified in a more abstract way than CRT and PT test scripts.

The main weaknesses of the NLT approach are the reduced expressiveness of assertions and the complex troubleshooting required when the most natural way to express a command does not work. For what concerns the assertions, in the selected NLT tool they are performed with the command `assert that`, that requires to specify the condition that must be checked. For example the assertion `assert that page contains "Welcome, user!" above "Home"` checks that in the web page at hand the text "Welcome, user!" must be present above the text "Home". The problem is that it may not always be easy to describe an element's position relative to other elements using natural language, especially when the other elements are images, which cannot be easily described in the tool. In these cases, the developer may be tempted to write weaker assertions, that check only the presence of an element but not its position on the page. However, these assertions should be avoided since they can lead to false negatives, where the test execution succeeds when it should have failed. Otherwise, the developer may spend more time to try to define a stronger assertion in natural language, but this can reduce one of the main advantages of the NLT approach, which is the speed in easily producing E2E test scripts. The difficult troubleshooting instead is due to the fact that depending on the structure of the DOM of the page and the position of the elements, the most natural way to express a command may not work. In these cases, the documentation does not provide great support, since only standard cases are described, and the developer is left alone trying to debug the test script until it works: in this sense, the possibility of using full XPaths (as web element locators) in an NLT test provides an excellent solution. A practical example of complex troubleshooting happened to one of the Testers during the evolution of the SMF test suite. Specifically, a test script in the SMF test suite required a specific text value to be entered into a field to post a message to a forum. The page containing the text field can be seen in Figure 8 for both the versions considered of the web application: it is evident that there are no major visual differences in this page between the two versions of the application.


NLT community test

Hello administrator
Show unread posts since last visit.
Show new replies to your posts.
December 21, 2022, 06:30:51 pm

News:
SMF - Just Installed!

NLT community test » General Category » General Discussion » *Start new topic*

Start new topic

Subject:
Message icon: Standard

Attachments and other options

shortcuts: hit alt+s to submit/post or alt+p to preview

administrator ▾ My Messages Alerts 1

NLT community test

Dec 21, 2022, 06:26 PM • Unread Posts • Updated Topics

News: SMF - Just Installed!

NLT community test ► General Category ► General Discussion ► *Start new topic*

Start new topic

Subject
Message icon Standard

Attachments and other options

Shortcuts: ALT+S post, ALT+P preview, ALT+D save draft

FIGURE 8 Screenshots from the Simple Machine Forums application, version 2.0.1 and 2.1.2

However, in the first version of the application, the text box at the center of the page had HTML name "message", so it is possible to insert text on it using the command `enter "text" into "message"`. But in the second version of the application, the HTML name was missing, so the Tester started trying some intuitive commands, like `enter "text" in the text area at the center of the page`, and `enter "text" in the text area above "Attachments and other options"` but they all failed, although according to the documentation they were syntactically correct. Even using the full XPath, in the form `enter "text" into xpath "/html/body/div[1]/textarea"`, did not succeed. In the end, the Tester found a working solution by clicking the element using the full XPath, then using the command `type "text"` to enter the text in it, but coming to this solution required the Tester 16 minutes, a considerable amount of time w.r.t. the total evolution time for this test suite (1 hour and 11 minutes). Finally, like in the CRT approach, also in the case of NLT we encountered issues with JavaScript alerts.

Keeping these considerations in mind and looking at the results of the study, the NLT approach appears to be a good choice for small to medium-sized test suites (as the ones considered in our empirical study), when minimizing the overall costs (development and evolution) is a priority even with Software Testers without programming skills. As aforementioned, additional evaluations are required with complex industrial E2E test suites to assess the overall costs in particular in comparison with the PT approach. Finally, it must be considered that the only existing NLT solutions are commercial and subject to a fee with the problem of vendor lock-in.

4.3 | Threats to Validity

Internal validity threats concern factors that may affect the dependent variables and that are not considered in the study. The most relevant threats to the internal validity of this study concern the subjectivity and variability of the test scripts implementation task: this includes the selection of the web applications to test, the choice of their functionalities to be tested, the definition of the test steps, and the input data to use. We tried to limit these threats by involving different persons, one for the definition of the test specifications and three junior Testers for the test script development/evolution, and by applying well-known testing criteria together with a precisely defined experimental procedure. E2E testing normally takes place after functional and system testing and its starting point is the application user's perspective. Hence, as expected for this type of testing, for each application under test, one of the authors tried to identify the test scenarios that could simulate the typical operations performed by the application's users (mimic the typical user's gestures). This, together with the choice of using Gherkin, allowed us to separate the definition of the test scenarios, focused on the application's domain and functionalities, with the test implementation done by the three junior Testers. Gherkin has been used by one of the authors to specify, and document in a quite structured way, the set of test scenarios to be implemented, by the three junior Testers using the different testing tools (i.e., PT, CRT, NLT). By adopting a domain-specific language such as Gherkin, we aim to limit the subjectivity and variability that typically affect the adoption of informal and/or free-text based test cases. Another (possible) impacting threat is related to the learning effect during test scripts development and evolution tasks. As explained, we tried to consider this threat in the experiment design by altering the order of test suite development and evolution. Since our study requires evaluating three different approaches, we asked to each Tester to implement (and then evolve) three test suites in different orders: this allowed us to consider all the possible implementation orders.

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 reasonable proxy for estimating the testing effort since it is a widely adopted practice in empirical software engineering research. Another threat concerns the fact that Gherkin has been employed to specify the test scripts and that such specifications can be considered quite similar to the one used for NLT (thus providing an advantage for what concerns the initial development effort to this approach). On the one side, however, Gherkin test cases are abstract (they must be understandable to a human being), while NLT test scripts are more concrete. Indeed, NLT test scripts are characterized by steps that must be executed, so they must contain precise values to (1) localize the web elements to interact with (including positional properties in the web page), (2) insert values into forms' field, and (3) evaluate assertions. Furthermore, we can consider that often in industry E2E test cases are specified in natural language and textual documentation, thus the adoption of Gherkin test cases allows us to mimics, in our experiment, what happens typically in the industry.

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., Kruskal-Wallis and 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. Since we had to compare three treatments, we first applied the Kruskal-Wallis statistical test [22]. This test is used for comparing two or more independent samples of equal or different sample sizes. It extends the Wilcoxon–Mann–Whitney test, which is used for comparing only two treatments. Finally, we adopted the Vargha and Delaney's A test [25] to measure the strength of the relationship between couples of treatments (e.g., PT vs. NLT and CRT vs. NLT), when a statistically significant effect was observed.

External validity threats are related to the generalization of the results. One of the most relevant threats to external validity concerns the involvement of only three junior Testers. Concerning this point, the involved Testers have industrial experience in the web testing domain, particularly employing Selenium WebDriver. Thus, they are good representatives of junior web

Testers in general. Moreover, it is essential to underline that the case study is challenging and time-consuming to complete; therefore, finding subjects available to participate in it is not simple and takes work. Another critical threat could be related to the complexity of the tasks performed during the experiment in relation to the participants experience. We observed that having 2/3 years of experience (as our Junior Testers) is by far sufficient to complete them. As a result we believe having test engineers with more years of experience may not significantly change the results. On the contrary, having Testers with very little (or even no) experience could give some further advantage to NLT (and partially also to CRT), given that to develop test suites with the PT approach it is necessary to have some programming skills and therefore a non-negligible experience can be a plus. Another critical threat could be related to the web applications adopted in the study. The chosen applications are medium-sized, realistic, representative of their domain, and based on modern technologies and languages. The fact that, for maintaining the case study feasible, we did not consider the development of test suites for large industrial applications requires care regarding the generalization of the results to this class of applications (as we mentioned multiple times while reporting and discussing the results); but also offers an important starting point for future works. Other potentially impacting threats are related to the developed test suites and the chosen tools representative of the three approaches. Test suites have been designed as much as possible by following a systematic approach aiming at mimic the typical user's actions with respect to the considered applications (as well as needed for E2E testing) and defining at least one test script for each significant application's functionality. In terms of chosen tools, we used third-party frameworks/tools, well-known and available on the Internet, thus avoiding any authors' bias. Moreover, the frameworks/tools selected as representative of the PT and CRT approaches (respectively Selenium WebDriver and Selenium IDE) are among the most used in E2E web testing [10]. Concerning the NLT tools there are no published statistics on their usage, but we selected one that appears to be mature, complete, and mentioned in many online guides and tutorials: moreover, we compared its functionalities with the ones offered by the other NLT solutions and they appear to be equivalent.

5 | RELATED WORK

In the literature, we are witnessing a progressive interest in the adoption of NLP-based techniques to support software testing and test automation. However, to the best of our knowledge, there is a lack in the literature of objective and comparative evaluations (i.e., empirical studies) of the proposed NLP methods with respect to more traditional approaches (e.g., programming and capture&replay [6]) to develop, generate, represent and evolve test cases. Differently from most of the existing literature, in our work, we focus on an empirical evaluation in the Web context by considering different types of test case representations. We decided to structure the related works into three subsections considering three very important aspects connected with software testing and the use of natural language.

5.1 | Empirical Studies and Surveys on NLP in the Context of Software Testing

Perhaps the most representative work in this field is that Garousi et al. [31] which reviews the state of the art of NLP in the context of software testing by means of a survey in the form of a systematic literature mapping. Many of the papers contained in this survey present approaches and techniques to conduct and automate NLP-based analysis (i.e., morphologic, syntactic and semantic NLP approaches) for assisting software testing and generating: (1) test cases from natural language (NL) requirement specifications; (2) textual input values for test cases; and, (3) test oracles aiming at verifying exceptional software behaviors. Some approaches adopt an intermediate representation between the natural language specifications and the generated test cases or test artifacts, as for example behavioral models represented as state machines and activity diagrams. In this work, Garousi et al. emphasize the fact that the generation of executable test cases is a complex task when NLP techniques are employed. This applies regardless of the intermediate representation adopted.

Gupta et. al. [32] investigates recent research trends in the adoption of NLP in the context of software testing and highlights the following interesting issues: (i) the adopted requirement specifications are often constrained to a specific structure that limits their expressiveness; (ii) intermediate behavioral models inferred from NL requirements need to be precise and comprehensive enough, thus often leading to large and complex models difficult to use; (iii) manual rectification of models is often required when models are extracted from NL requirements; and (iv) test cases generated by adopting NLP techniques are often not executable and need an additional intervention, e.g., providing test input data.

Ricca et al. [12] provide the results of a survey of the grey literature concerning the use of artificial intelligence (AI) and NLP to improve test automation practices. The authors filtered 136 relevant documents from which a taxonomy of problems that AI aims to tackle is extracted, along with a taxonomy of AI-enabled solutions to such problems. The paper concludes by distilling the six most prevalent AI based testing tools on the market. Among these, there are TestSigma and Functionize, tools adopting natural language to express test cases for web applications.

5.2 | Test Case Generation

Carvalho et al. [33] present a NLP-based method for test case generation by starting from NL requirements. The proposed method converts the NL requirements in an intermediate and hidden formalism named Software Cost Reduction (SCR). The produced specifications are then used by a tool (NAT2TESTSCR) to generate test cases. Several industrial case studies have been conducted to evaluate the effectiveness of the proposed approach, in terms of performance in generating test cases and of their fault detection capability. The obtained results have been compared with randomly generated test cases.

Sarmiento et al. [34] present a NLP-based tool offered as web application and named C&L able to translate NL requirements into behavioral models expressed by means of activity diagrams. In a second time, test scenarios and elements are derived from the generated behavioral models and used to produce abstract test cases, then enriched with input values to make them executable.

Fischbach et al. [35] describe a NLP-based approach and a tool (Specmate) for deriving test cases from requirements expressed by means of user stories. The considered user stories follows the template established by Cohn [36], i.e., "As a <type of user>, I want <goal>, so that <some reason>". A Cause-Effect-Graph, generated by the dependency tree extracted from the user stories, is used as an intermediate model. By traversing the Cause-Effect-Graph, a set of test cases is finally defined. A case study of 961 user stories in cooperation with the industry has been conducted to evaluate the effectiveness of the proposed approach/tool.

Gröpler et al. [37] propose the use of NLP to extract sequence models and model specifications from functional requirements written in natural language. These models and specifications are then used in a toolchain for generating test cases. The toolchain for requirements-based model and test case generation is applied with success to an industrial use case from the e-mobility domain: a system for charging approval of an electric vehicle in interaction with a charging station.

Wang et al. [38] investigate the adoption of a NLP-based method for analyzing textual documents that aim at extracting information relevant for the construction of test cases and for the identification of test data. As result, the authors present UMTG, a tool-based approach able to support the generation of executable acceptance test cases from requirements specifications written in natural language, with the goal of reducing the manual effort required to produce test cases. UMTG has been validated in two industrial case studies with excellent results. It was able to generate test cases that exercise all the test scenarios manually implemented by experts, but also some other test scenarios not previously considered.

5.3 | Gherkin

As introduced in Section 2, Gherkin is a structured quasi-natural language that let Software Testers to specify test cases by using a constrained natural language structured around a set of pre-defined keywords. Gherkin is mainly used to define scenarios in Behavior-Driven Development (BDD) [39], an agile software development technique also referred to as 'Specification by Example' that drives developers to define executable and testable requirements.

Oruc et al. [40] apply the BDD approach for testing web services. The authors develop a tool which, starting from test cases expressed in Gherkin, automatically generates JMeter test scripts[x].

Colombo et al. [41] present an approach to convert Gherkin-based specifications into models that can subsequently be used to generate and execute Selenium WebDriver test scripts. The authors instantiated the proposed approach using QuickCheck (a lightweight tool for random testing of Haskell programs) [42] and shown its applicability via a case study on the national health portal for Maltese residents.

Finally, both the works presented by Marchetto et al. [43] and Longo et al. [44] evaluate the adoption of Fitnesse[y] and Fit tables [45] as tools/techniques to define acceptance tests in constrained natural languages. In particular, while Marchetto et al. [43] compares Fit tables for traditional systems and Web specific Fit tables in maintenance tasks, Longo et. al. [44] compares Fitnesse and Gherkin projects from the point of view of Test data uniformity. Uniform test data, as explained by the authors, are expressions that are common to various test documents.

[x] https://jmeter.apache.org/

[y] http://docs.fitnesse.org/FrontPage

6 | CONCLUSIONS

This paper reports the results of a series of case studies we conducted to compare NLP-based test automation (NLT) and two more traditional approaches, i.e., programmable web test automation (PT) and capture&replay web test automation (CRT). The comparison is based on four factors: (1) the effort required for developing test suites (computed in minutes); (2) the resilience to changes of the test suites; (3) the effort required to evolve the test suites (computed in minutes); and (4) the cumulative effort computed combining conveniently development and evolution effort.

Results show that the NLT approach appears to be the best option for small to medium sized test suites such as those considered in our empirical study. In fact, compared to competitors, the NLT approach minimizes the total cumulative cost (development and evolution) and does not require Testers with programming skills. NLT is also the best in terms of resiliency being more robust to the evolution of a web application. CRT is slightly faster during development but then loses when evolution is considered. In conclusion, these new NLP-based tools appear to be very promising and will probably be even more performing in the near future when the adopted NLP algorithms used to transform natural language based test cases into executable test cases improve further.

For the future, we plan to increase the generalizability of the results of this study. In particular, we would like to: (1) conduct a larger study by extending the set of the considered web applications and involving others developers, possibly professional developers; (2) experiment with different testing tools/frameworks, in particular considering other testing tools belonging to the NLP category; and (3) consider many corner cases of the tested web app functionalities to analyze the increment of the maintenance effort due to test suite redundancy. Such effort could be different among the approaches (PT, CRT, NLT). Indeed, specific testing design patterns such as the PO pattern could reduce the maintenance effort in such cases, thus the PT could have some advantages from an evolution cost perspective at least compared to the CRT approach. Finally, we would like to conduct another study to understand the actual potential of these new NLP-based tools in terms of language expressiveness. In fact, in some cases the Testers who conducted the case studies realized that the NLP algorithm was not able to interpret some test case steps even if they were described correctly. So it would be interesting to test this aspect empirically to understand and asses the limits of the NLP algorithms in the web context.

References

[1] Callaghan D, O'Sullivan C. Who should bear the cost of software bugs? *Computer Law & Security Report* 12 2005; **21**:56–60, .

[2] Marchetto A, Ricca F, Tonella P. An empirical validation of a web fault taxonomy and its usage for web testing. *J. Web Eng.* 12 2009; **8**:316–345.

[3] Ocariza F, Bajaj K, Pattabiraman K, Mesbah A. A study of causes and consequences of client-side javascript bugs. *IEEE Transactions on Software Engineering* 01 2016; **43**:1–1, .

[4] Software Testing Help. What is software quality assurance (sqa): A guide for beginners apr 2023.

[5] Laporte C, April A. *Software Quality Assurance*. Wiley-IEEE, 2018.

[6] Leotta M, Clerissi D, Ricca F, Tonella P. Capture-Replay vs. Programmable Web Testing: An Empirical Assessment during Test Case Evolution. *Proceedings of 20th Working Conference on Reverse Engineering (WCRE 2013)*, IEEE, 2013; 272–281, . URL https://doi.org/10.1109/WCRE.2013.6671302.

[7] Berner S, Weber R, Keller RK. Observations and lessons learned from automated testing. *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, Association for Computing Machinery: New York, NY, USA, 2005; 571–579, . URL https://doi.org/10.1145/1062455.1062556.

[8] Ebert C, Gallardo G, Hernantes J, Serrano N. DevOps. *IEEE Software* May 2016; **33**(3):94–100, .

[9] García B, Gallego M, Gortázar F, Organero M. A survey of the selenium ecosystem. *Electronics* 06 2020; **9**:1067, .

[10] Cerioli M, Leotta M, Ricca F. What 5 million job advertisements tell us about testing: a preliminary empirical investigation. *Proceedings of 35th ACM/SIGAPP Symposium on Applied Computing (SAC 2020)*, ACM, 2020; 1586–1594, . URL https://doi.org/10.1145/3341105.3373961.

[11] Leotta M, García B, Ricca F, Whitehead J. Challenges of End-to-End Testing with Selenium WebDriver and How to Face Them: A Survey. *Proceedings of 16th IEEE International Conference on Software Testing, Verification and Validation (ICST 2023)*, IEEE, 2023; (in press).

[12] Ricca F, Marchetto A, Stocco A. Ai-based test automation: A grey literature analysis. *2021 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2021; 263–270, .

[13] Trudova A, Dolezel M, Buchalcevova A. Artificial intelligence in software test automation: A systematic literature review. 2020; 181–192, .

[14] Leotta M, Ricca F, Stoppa S, Marchetto A. Is nlp-based test automation cheaper than programmable and capture & replay? *Proceedings of 15th International Conference on the Quality of Information and Communications Technology (QUATIC 2022)*, *CCIS*, vol. 1621, Vallecillo A, Visser J, Pérez-Castillo R (eds.). Springer, 2022; 77–92, . URL https://doi.org/10.1007/978-3-031-14179-9_6.

[15] Leotta M, Clerissi D, Ricca F, Tonella P. Approaches and Tools for Automated End-to-End Web Testing. *Advances in Computers* 2016; **101**:193–237, . URL https://doi.org/10.1016/bs.adcom.2015.11.007.

[16] Leotta M, Biagiola M, Ricca F, Ceccato M, Tonella P. A family of experiments to assess the impact of page object pattern in web test suite development. *Proceedings of 13th IEEE International Conference on Software Testing, Verification and Validation (ICST 2020)*, IEEE, 2020; 263–273, . URL https://doi.org/10.1109/ICST46399.2020.00035.

[17] Leotta M, García B, Ricca F. An empirical study to quantify the setup and maintenance benefits of adopting WebDriverManager. *Proceedings of 15th International Conference on the Quality of Information and Communications Technology (QUATIC 2022)*, *CCIS*, vol. 1621, Vallecillo A, Visser J, Pérez-Castillo R (eds.). Springer, 2022; 31–45, . URL https://doi.org/10.1007/978-3-031-14179-9_3.

[18] García B, Ricca F, del Alamo JM, Leotta M. Enhancing web applications observability through instrumented automated browsers. *Journal of Systems and Software* 2023; **203**, . URL https://doi.org/10.1016/j.jss.2023.111723.

[19] Moro A, Raganato A, Navigli R. Entity linking meets word sense disambiguation: a unified approach. *Transactions of the Association for Computational Linguistics* 2014; **2**:231–244, . URL https://aclanthology.org/Q14-1019.

[20] Wohlin C, Runeson P, Hőst M, Ohlsson MC, Regnell B, Wesslén A. *Experimentation in Software Engineering*. Springer, Berlin, Heidelberg, 2012, .

[21] Raemaekers S, van Deursen A, Visser J. Semantic versioning versus breaking changes: A study of the maven repository. *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, 2014; 215–224, .

[22] McKight PE, Najab J. *Kruskal-Wallis Test*. John Wiley & Sons, Ltd, 2010; 1–1, .

[23] Wilcoxon F. Individual comparisons by ranking methods. *Breakthroughs in statistics*. Springer, 1992; 196–202.

[24] Holm S. A simple sequentially rejective multiple test procedure. *Scandinavian journal of statistics* 1979; :65–70.

[25] Vargha A, , Delaney H. A critique and improvement of the cl common language effect size statistics of mcgraw and wong. *Journal of Educational and Behavioral Statistics* 02 2000; **25**:101–132, .

[26] Nass M, Alégroth E, Feldt R, Leotta M, Ricca F. Similarity-based web element localization for robust test automation. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 2023; **32**(3), . URL https://doi.org/10.1145/3571855.

[27] Leotta M, Ricca F, Tonella P. SIDEREAL: Statistical adaptive generation of robust locators for Web testing. *Journal of Software: Testing, Verification and Reliability (STVR)* 2021; **31**, . URL https://doi.org/10.1002/stvr.1767.

[28] Leotta M, Stocco A, Ricca F, Tonella P. ROBULA+: An algorithm for generating robust XPath locators for web testing. *Journal of Software: Evolution and Process (JSEP)* 2016; **28**(3):177–204, . URL https://doi.org/10.1002/smr.1771.

[29] Leotta M, Cerioli M, Olianas D, Ricca F. Two experiments for evaluating the impact of Hamcrest and AssertJ on assertion development. *Software Quality Journal (SQJ)* 2020; **28**:1113–1145, . URL https://doi.org/10.1007/s11219-020-09507-0.

[30] Olianas D, Leotta M, Ricca F. SleepReplacer: A novel tool-based approach for replacing thread sleeps in selenium webdriver test code. *Software Quality Journal (SQJ)* 2022; **30**:1089–1121, . URL https://doi.org/10.1007/s11219-022-09596-z.

[31] Garousi V, Bauer S, Felderer M. NLP-assisted software testing: A systematic mapping of the literature. *Information and Software Technology* oct 2020; **126**:106 321, .

[32] Gupta A, Mahapatra RP. A circumstantial methodological analysis of recent studies on NLP-driven test automation approaches. *Intelligent Systems*. Springer Singapore, 2021; 155–167, .

[33] 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* 2014; **95**:275–297, .

[34] Sarmiento E, do Prado Leite JCS, Almentero E. C&l: Generating model based test cases from natural language requirements descriptions. *1st International Workshop on Requirements Engineering and Testing (RET)*, IEEE, 2014, .

[35] Fischbach J, Vogelsang A, Spies D, Wehrle A, Junker M, Freudenstein D. SPECMATE: Automated creation of test cases from acceptance criteria. *13th International Conference on Software Testing, Validation and Verification (ICST)*, IEEE, 2020, .

[36] Cohn M. *User Stories Applied: For Agile Software Development*. Addison Wesley Longman Publishing Co., Inc.: USA, 2004.

[37] Gröpler R, Sudhi V, García EJC, Bergmann A. Nlp-based requirements formalization forautomatic test case generation. *29th International Workshop on Concurrency, Specification and Programming (CSP)*, CEUR (ed.), 2021.

[38] Wang C, Pastore F, Goknil A, Briand LC. Automatic generation of acceptance test cases from use case specifications: An NLP-based approach. *IEEE Transactions on Software Engineering* feb 2022; **48**(2):585–616, .

[39] Patkar N, Chis A, Stulova N, Nierstrasz O. Interactive behavior-driven development: a low-code perspective. *2021 International Conference on Model Driven Engineering Languages and Systems Companion (MODELS-C)*, IEEE, 2021, .

[40] Oruç AF, Ovatman T. Testing of web services using behavior-driven development. *6th International Conference on Cloud Computing and Services Science*, SCITEPRESS, 2016, .

[41] Colombo C, Micallef M, Scerri M. Verifying web applications: From business level specifications to automated model-based testing. *Electronic Proceedings in Theoretical Computer Science* mar 2014; **141**:14–28, .

[42] Claessen K, Hughes J. Quickcheck: A lightweight tool for random testing of haskell programs. *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, ICFP '00, Association for Computing Machinery: New York, NY, USA, 2000; 268–279, . URL https://doi.org/10.1145/351240.351266.

[43] Marchetto A, Ricca F, Torchiano M. Comparing "traditional" and web specific fit tables in maintenance tasks: A preliminary empirical study. *12th European Conference on Software Maintenance and Reengineering*, IEEE, 2008, .

[44] Longo DH, Vilain P, da Silva LP. Measuring test data uniformity in acceptance tests for the FitNesse and gherkin notations. *Journal of Computer Science* feb 2021; **17**(2):135–155, .

[45] Melnik G, Read K, Maurer F. Suitability of fit user acceptance tests for specifying functional requirements: Developer perspective. *XP/Agile Universe 2004: Extreme Programming and Agile Methods - XP/Agile Universe 2004*, 2004; 60–72, .

□