Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Computer Science

Identification of Potentially Reusable Subroutines in Suboptimally Structured Automated Test Scripts

by

MARTIN FILIPSKY

A dissertation thesis submitted to
the Faculty of Electrical Engineering, Czech Technical University in Prague,
in partial fulfilment of the requirements for the degree of Doctor.

Ph.D. programme: Electrical Engineering and Information Technology
Specialization: Information Science and Computer Engineering

Prague, August 2018

Supervisor:
doc. Ing. IVAN JELÍNEK, CSc.
Department of Computer Science
Faculty of Electrical Engineering
Czech Technical University in Prague
Karlovo nám. 13
121 35 Prague 2
Czech Republic

Co-Supervisor:
Ing. MIROSLAV BUREŠ, Ph.D.
Computer Science and Engineering Department
Faculty of Electrical Engineering
Czech Technical University in Prague
Karlovo nám. 13
121 35 Prague 2
Czech Republic

Abstract and contributions

With the growing complexity of current information systems, the testing and quality assurance of these systems also grow in importance. Manual testing proves to be inefficient in a number of cases, and test automation is employed as a logical alternative. Automated tests can be very efficient because of their very low execution costs compared to manual testing. However, automated tests, especially those that focus on the user interface of the System Under Test (SUT), also have their issues that need to be solved.

In these tests, one of the main issues is the mutually exclusive combination of the fast development of automated tests and their subsequent cost–saving maintenance. Test code maintenance is necessitated by changes made to the SUT, which also implies subsequent changes in the automated test scripts to keep them up–to–date with the SUT. If the automated tests are not up–to–date, they can fail in the proper detection of defects, lead to false defect reports or simply terminate in an error state and not provide any information about the actual state of the SUT. The developers of automated tests have to ask a question before the actual start of the project: *is our goal to develop tests in an economical way or to maintain them economically in the later phases of the project or operation?* The prospective test maintenance overhead (which is frequently reported as the main drawback of the user interface–based test automation) can be reduced by the appropriate structuring of test scripts and by using reusable code parts such as objects and components. The identification of these reusable parts can be a tricky task for test automation developers. Hence, the overall economics of the project and the comfort of the test script developers can be improved by a semi–automated process for the identification of these reusable objects.

In this thesis, we present an approach for the automated identification of reusable components in a user interface–based test automation code. The approach mainly focuses on the identification of the repetitive parts of test scripts and is most efficient in the case of plainly structured linear automated test scripts, which can be a result of a record and replay approach or naive test automation programming style (which is still, in the current

industry praxis, a frequently repeated anti–pattern, thus leading to significant inefficiency in terms of the prospective maintenance of the created test scripts).

Such an improvement of the conventional test automation approaches like the aforementioned record and replay allows the enhancement of the overall efficiency of both the creation of automated tests and their subsequent maintenance.

In the proposed approach, we automatically provide suggestions for potential reusable components for test automation developers to give them continuous guidance to refactor the created tests. On this point, the proposed approach differs from the standard static analysis tools, which analyze the source code solely based on the repetitive code fragments.

In a test automation code, a particular sequence of the same actions exercised in the SUT user interface can be expressed by different notations. In a number of test automation Application Programming Interfaces (APIs), this case is common. The standard static analysis of code with the aim of searching for common code elements usually reaches a limit at this point. Hence, in our approach, we analyze the test step semantics by a proposed mechanism of abstract signatures, which gives our approach the capability to detect more relevant potential reusable objects.

The entire proposed process consists of several phases.

1. The source code of the automated tests is converted into a model of the test script, which abstracts particular implementation details and captures the actual actions that the tests perform in the SUT.

2. Then, we analyze these abstract signatures using a specially designed solver for evolutionary computations to find the potential common subroutines in the tests.

3. The found subroutines are then postprocessed and filtered, and their relevance is assessed using a set of proposed metrics.

During the project, we implemented two functional prototypes (an initial and the final version) of the method, which we verified in a set of experiments. In these experiments, we exercised the initial and final versions of the prototype against automated test scripts for web and mobile applications. The obtained results are promising. We achieve a better efficiency of identification of potential reusable objects than a manual optimization of automated tests. Moreover, compared to conventional approaches that are solely based on the direct analysis of structured data searching for duplicate script fragments, our proposed system localizes more relevant potentially reusable script components.

The main contributions of the thesis can be summarized as follows:

1. A novel approach that can improve the overall efficiency of test automation projects. The approach can resolve the traditional dilemma between (1) recording or quickly

creating the automated tests and pay the hidden costs later in terms of the high maintenance of these tests or (2) investing in suitable test automation architecture and the identification of the reusable objects, which can return subsequent lower maintenance costs.

2. An innovative model of automated tests that allows one to abstract the tests and make them independent of a particular language and/or framework. This abstraction allows one to conduct a more efficient search for truly relevant potential reusable objects.

3. A fully parameterizable set of algorithms for the identification of potentially reusable objects in a set of user interface–based automated tests. The parametrization of the algorithms increases the flexibility of the method and gives its users an opportunity to configure the proposed method to various test automation projects, used languages and test automation APIs.

4. The entire proposed method is implemented as a practically applicable framework supporting re–factoring process of the automated tests.

5. Experimental data show the results of the framework application for automated tests in the web–based and mobile domains. These data also provide insight into the particular parametrization of the proposed algorithms and suggest a suitable parametrization for further applications in industry projects.

Regarding the state of the art, the area of the optimization of recorded test scripts is relatively unexplored. Individual works such as the BlackHorse project by Carino et al.[1] can be found, but they do not aim for the exact same goal as ours. The closest area is a search for common subsequences of source code, which is a part of the static analysis discipline. However, this thesis focuses on the specific problem of test automation, which makes this contribution quite original in the research literature and also in the variety of current industrial test automation frameworks.

Keywords:

Test Automation, Test Scripts, Test Refactoring, Test Recording, Descriptive Programming, Longest Common Subsequence Problem, Reusable Objects.

Abstrakt a přínosy práce

S rostoucí složitostí současných informačních systémů roste význam testování a procesu zajištění kvality těchto systémů. Manuální testování softwaru se v mnoha případech ukazuje jako neefektivní, což vede k logickému využití automatizace testů. Automatické testy mohou být velmi efektivní z důvodu velmi nízké ceny vykonávání testů. Nicméně technologie automatizovaných testů, a to zejména testů zaměřených na uživatelské rozhraní testovaného systému, má i své problémy, které je zapotřebí systematicky řešit.

Jedním z hlavních problémů v těchto testech je vzájemně se vylučující se kombinace rychlého vývoje testů a jejich následných nízkých nákladů na údržbu. Údržba kódu testů je vynucena změnami v testovaném softwarovém systému. Jestliže automatické testy nejsou aktualizované, mohou selhávat v řádné detekci defektů, což vede k falešným chybovým hlášením, nebo jednoduše skončí v chybovém stavu a nemohou tak poskytnou žádnou informaci o stavu testovaného systému. Vývojáři automatických testů si před skutečným počátkem prací na projektu musí položit otázku: *Je naším cílem vyvinout testy s nízkými náklady, nebo je s nízkými náklady později spravovat a udržovat?* Potenciální náklady s údržbou (které jsou často označovány jako hlavní nevýhoda automatizace testů pracujících s uživatelským rozhraním testovaného systému) mohou být sníženy vhodným strukturování testovacích skriptů a využíváním znovupoužitelných částí kódů jako jsou objekty a komponenty. Identifikace těchto znovupoužitelných částí může pro vývojáře automatických testů být obtížným úkolem. Z tohoto důvodu mohou být celková ekonomika projektu automatizace testů a komfort vývojářů zlepšeny poloautomatickým procesem identifikace těchto znovupoužitelných objektů v kódu testů.

V této disertační práci představujeme metodu pro automatickou identifikaci opětovně použitelných komponent v automatizovaných testech založených na interakci s uživatelským rozhraním testovaného systému. Metoda se zaměřuje především na jednoduše strukturované lineární automatizované testovací skripty, které mohou být výsledkem použití metody Record a Replay (nahrání a přehrání testů) nebo naivního stylu strukturování automatizovaných testů. Tento neoptimální styl je stále v současné průmyslové praxi často opakovaný

antivzor, vedoucí k významné neefektivitě z pohledu budoucí údržby vytvořených testovacích skriptů.

Automatizovaná identifikace opětovně použitelných komponent v automatizovaných testech umožní vylepšení celkové efektivity jak tvorby automatických testů, tak jejich následné údržby.

Principem navrhované metody je automatizovaný analýza kódu testů, která poskytuje návrhy potenciálních znovupoužitelných komponent vývojářům automatických testů. Tyto návrhy jsou vývojáři využity při refaktoringu vytvořených testů. V tomto bodě se navržený postup liší od standardních nástrojů statické analýzy, které analyzují zdrojový kód výhradně na bázi opakujících se fragmentů kódu.

V kódu automatických testů mohou totiž být konkrétní posloupnosti stejných akcí v uživatelském rozhraní testovaného systému vyjádřeny různými notacemi. Tato situace je v současné průmyslové praxi zcela běžná. Standardní statická analýza kódu s důrazem na vyhledávání společných elementů je z tohoto pohledu obvykle výrazně neefektivní. Na rozdíl od běžných postupů statické analýzy, v navržené metodě analyzujeme sémantiku jednotlivých kroků testů pomocí navrženého mechanizmu abstraktních signatur. Díky tomuto přístupu má navržená metoda schopnost detekovat více relevantních potenciálně opětovně použitelných objektů.

Celý proces se skládá z několika fází:

1. Zdrojový kód automatických testů je konvertován do modelu automatizovaných testovacích skriptů, který odstiňuje konkrétní implementační detaily a zachycuje skutečné akce, které test vykonává v testovaném systému.

2. Poté analyzujeme tyto abstraktní signatury s pomocí speciálně navrženého modulu pro evoluční výpočty tak, abychom nalezli potenciálně znovupoužitelné sekvence v jednotlivých testech.

3. Vyhledané sekvence jsou poté následně zpracovány a filtrovány a jejich relevantnost ohodnocena pomocí sady navržených metrik.

Během experimentálního ověřování navržené metody jsme implementovali dva funkční prototypy řešení (průběžný a výsledný). V experimentech jsme otestovali jak průběžnou verzi prototypu, tak i jeho výslednou verzi s několika sadami automatizovaných testovacích skriptů pro webové a mobilní aplikace. Získané výsledky jsou slibné. Dosáhli jsme lepší efektivity v identifikaci potenciálně opětovně použitelných objektů než při ruční optimalizaci automatických testů. Navíc v porovnání s konvenčními postupy, které jsou výhradně založeny na přímé analýze strukturovaných dat vyhledávající duplicitní fragmenty kódu, naše řešení lokalizuje více relevantní potenciálně znovupoužitelné komponenty v kódu automatizovaných testů.

Hlavní přínosy této práce jsou především:

1. Inovativní řešení, které může vylepšit celkovou efektivitu projektů automatizace testů. Metoda řeší tradiční dilema mezi (1) nahráváním automatizovaných testů pomocí specializovaného nástroje nebo jejich rychlou tvorbou a pozdějšími vysokými skrytými náklady ve smyslu údržby těchto testů nebo (2) investicí do vhodné architektury automatizace testů a nalezení opětovně použitelných objektů, které mohou vrátit investici v podobě nižších nákladů na údržbu.

2. Inovativní model automatických testů umožňující abstrahovat akce vykonávané těmito testy a zajistit jejich nezávislost na konkrétním programovacím jazyku anebo frameworku. Tato abstrakce umožňuje provést efektivní vyhledávání skutečně relevantních potenciálně opětovně použitelných objektů.

3. Algoritmy pro identifikaci potenciálně opětovně použitelných objektů v kódu automatizovaných testů založených na uživatelských rozhraních. Parametrizace algoritmů zvyšuje flexibilitu metody a dává uživateli příležitost konfigurovat navržené řešení pro různé projekty automatizace testů, použité programovací jazyky a API nástrojů pro automatizaci testů.

4. Celé navržené řešení je implementováno jako prakticky aplikovatelný framework, který podporuje proces refaktoringu automatických testů.

5. Experimentální data ukazují výsledky aplikace frameworku pro automatické testy v doménách webových a mobilních aplikací. Tato data rovněž poskytují detailnější informace o vhodné parametrizaci navržených algoritmů a navrhují vhodné nastavení parametrů pro další budoucí aplikace řešení v softwarových projektech.

Oblast optimalizace nahraných testovacích skriptů je nedostatečně pokryta stávajícím výzkumem. Existuji jednotlivé práce jako projekt BlackHorse od autorů Carino a kol. [1], ale nezaměřují se přesně na stejný cíl jako je cíl této disertační práce. Nejbližší oblastí je vyhledávání shodných posloupností ve zdrojovém kódu, které je součástí disciplíny statické analýzy kódu. Nicméně tato teze se zaměřuje na specifický problém optimalizace kódu v doméně automatizace testování, což je jak ve výzkumné literatuře, tak rovněž v současném spektru používaných frameworků pro automatizaci testů poměrně originální počin.

Klíčová slova:
automatizace testování, testovací skripty, refaktoring testů, nahrávání testů, deskriptivní
programování, problém nejdelší společné posloupnosti, znovupoužitelné objekty

Acknowledgements

First of all, I would like to express my gratitude to my supervisor for the dissertation thesis, Doc. Ing. Ivan Jelínek, CSc. He has been a wonderful supporter of my research activities and gave me a lot of inspiration during my studies as well as for my professional development.

I would like to thank my co–supervisor Ing. Miroslav Bureš, Ph.D. who was always willing to help me with the topic of the dissertation thesis and provided me with dozens of consultations, and gave me intensive support during writing the thesis.

Special thanks go to the staff of the Department of Computer Science, who maintained a pleasant and flexible environment for my research. I would like to express special thanks to the department and faculty management for providing most of the funding for my research.

My research has also been partially supported by the Grant Agency of the Czech Technical University in Prague, grant No. SGS12/192/OHK3/3T/13, SGS15/084/OHK3/1T/13, and SGS16/090/OHK3/1T/13.

Finally, my greatest thanks go to my family members, my parents, my wife, and children, for their infinite patience and care while I was working on this thesis.

Dedication

To my Family

Contents

List of Figures

List of Tables

List of Algorithms

Used Symbols

List of Used Symbols

p	Potentially reusable subroutine found in analyzed automated tests
P	Set of potentially reusable subroutines in analyzed automated tests
s	(Abstract) signature of step of automated test
S	Set of all signatures
t_c	Chromosome used in the Longest Common Subroutine (LCS) search
t_a	Abstracted automated test script
T_A	Set of abstracted test scripts
T_P	Set of prospective abstracted automated test scripts for further LCS search
T_F	Set of analyzed abstracted test scripts, in which a p is present
SQ	Subroutine Quality
AVQ	Analysis Variant Quality
ASQ	Averaged Value of Sequence Quality

Acronyms

List of Acronyms

AJAX	Asynchronous JavaScript And XML
AVQ	Analysis Variant Quality
ASQ	Averaged Value of Sequence Quality
API	Application Programming Interface
CORBA	Common Object Request Broker Architecture
CPD	Copy-Paste-Detector
DOM	Document Object Model
CI	Continuous Integration
DSL	Domain Specific Language
EJB	Enterprise Java Beans
HTML	Hypertext Markup Language
IDE	Integrated Development Environment
IFML	Interaction Flow Modeling Language
LCS	Longest Common Subsequences
OCL	Object Constraint Language
OCR	Optical Character Recognition
ORM	Object-Role Modeling Language
REST	Representational State Transfer
RMI	Remote Method Invocation
SAT	Satisfiability Problem
SUT	System Under Test
SMT	Satisfiability Modulo Theories
SQ	Subroutine Quality

UAT	User Acceptance Testing
UI	User Interface
UML	Unified Modeling Language
VBScript	Visual Basic Script Edition
QA	Quality Assurance
XML	Extensible Markup Language
XSD	XML Schema Definition

Chapter 1

Introduction

In this chapter, we introduce the area of test automation, which is the subject of this thesis. We describe the current problems, which represent the motivation for our research. Then, based on this motivation, we summarize the goals of this thesis, including the theoretical and practical aspects of the problem. Here, we also introduce the achievements of the thesis (thesis highlights). Finally, we introduce the structure of the thesis.

The increasing complexity of contemporary software systems, including new architectures and technologies, brings new challenges for software developers. Cloud computing [2], Virtualization [3], and Software as a service [4] replace old systems and concepts of how software is developed and sold. However, in parallel with that, software project budgets and resources for new tools are usually limited. The demand of customers for new software features and added value from every investment is constantly present. Additionally, with the growing dependency of users on the implemented systems, better performance and scalability, reliability and high on–line availability are often required. Facing this competitive software market pressure, the market leaders compete to provide cutting edge technologies and use efficient approaches to be able to deliver software projects successfully.

In this process, the usage of open source–based tools and frameworks, instead of buying traditional commercial software tools, grows in importance.

The costs of software testing usually represent an important part of the total costs of software development. For instance, study [5] estimates these costs to be even between 40 % and 80 % of the total costs of software development.

In such a situation, the automation of the software tests represents a prospective approach to achieve a better efficiency for the software testing. However, automated testing requires considerably significant investments for test development and the subsequent maintenance. This initial investment can be returned by a very economical repetitive execution of the automated tests. If the right test automation approach is chosen, the benefits of test automation may outweigh the investment, while if an improper approach is used, the investment might not be returned.

Every team planning a test automation project faces some of the following issues and challenges. These include not having enough knowledge and expertise to develop the automated tests, technological issues, required test automation tools that might not be available, the testability of the System Under Test (SUT) by automated tests, a managerial requirement for the fast and cheap development and execution of automated tests, the challenging maintenance of developed tests, and the requirement to test the SUT on multiple platforms or configurations.

In this thesis, **we focus on the automated tests which are interacting with the SUT by invoking actions and events in its user interface**, as well as retrieving information from this user interface. Further on, when using the term "automated tests", we mean this type of tests, unless stated otherwise explicitly.

A very frequent issue in automated testing is the maintenance of the developed tests [6, 7, 8]. Test scripts become inaccurate and obsolete as the SUT changes. As a result, the teams responsible for testing reduce their usage of automated testing to only regression tests, smoke tests and performance and load tests, and even in these cases, maintenance problems can cause the test automation project to fail. The usage of automated testing to replace conventional manual testing is quite rare [7, 8].

Current research has dealt with those issues and presents solutions for some of them, such as automated test development and maintenance [9, 10]) that may increase the efficiency of the test automation. A number of techniques, approaches and tools for the acceleration of the test automation process and the increase of its efficiency have already been adopted by many quality assurance teams such as mockup techniques [11] and test case generation from application models [12]. Mockups of the system under test (SUT) are a necessity in short Agile software development style sprints, in which testing teams do not have a working prototype of the SUT available (or they do not have enough time to deliver automated tests with the SUT features actually being developed). Agile software development methodologies [13] increase the need for the development of automated tests.

This software development style justifies the Record and Replay test automation approach (examples of particular solutions include the Selenium IDE[1] and the Robotium Recorder[2]). The recording of automated tests in the SUT front–end is very popular and fits the need for fast test development. Time–consuming preparation is not needed in those cases because no sophisticated test automation framework or advanced test code architecture is developed.

Practically, in the Record and Replay style, testers can start to develop their automated tests immediately. However, this approach usually suffers from significantly high maintenance costs [14, 15]. The created test recordings are not automatically well–organized into a reasonable architecture (reusable objects, flexible building blocks and test suites), and

[1] https://www.seleniumhq.org/projects/ide/

[2] https://plugins.jetbrains.com/plugin/7513-robotium-recorder

tests usually contain a number of duplications, making the update of the tests to the actual state of the SUT costly and prone to developers' mistakes.

To improve the quality and maintainability of the automated test code, these duplications shall be refactored into reusable functions or objects. In the case of the Record and Replay approach, an effort to refactor the duplications might be significant, and so an automated method aiding in this refactoring would decrease the overall test automation costs significantly. However, this refactoring step is rarely performed in industry projects, and this missing optimization is responsible for increased costs later on when the automated tests have to be updated and maintained.

Development teams follow various test management approaches and test automation approaches. Companies that need to sell their products in markets and to compete on price pursue an indicator of the returns on investments in which every decision that is taken has to be considered and evaluated from a number of viewpoints. Typically, is it worth making an investment in the development of automated tests, or is it sufficient to perform only manual tests? Can we achieve returns on the investments? Ultimately, how much does it cost us to run automated tests? Apart from the financial limitations, projects are also limited by agreed–upon deadlines, dependencies on third parties and various technical limits including the testability of the SUT by automated tests.

1.1 Motivation

In the software development process, the typical responsibility of the developers is to propose an architecture for the system and develop it, while testers develop tests and conduct them. On this point, test automation might increase the productivity for both the testing team and the developers. They can run the automated tests to verify that they did not introduce any regression defects during the feature development or removal of reported defects. The testing team can concurrently focus on the development of new tests to increase the test coverage.

On the other hand, automated tests can also require more time for the test script preparation and their maintenance, and so the testing team could have less time for the actual testing. If the balance is improper, the situation could lead to increased costs of testing, and the return on investment can be put at risk. Better and more innovative test automation approaches might help achieve this balance and improve the overall efficiency of the test automation project.

In the current test automation praxis, various approaches allow one to achieve different goals but, unfortunately, not all of them concurrently. From the test script preparation viewpoint, the most economical option is to employ a plain test script **Record and Replay** approach unless there is a requirement for the automation of a complex test case testing across multiple environments with verifications of complex data. In this approach,

a standard test case based on a simple use case can be created in a reasonable time that is comparable to the time required for the manual test preparation. However, the maintenance costs of the automated test scripts created by such an approach are typically the highest of all of the options because the recording creates unstructured linear test scripts with a number of potential code duplications.

In contrast, test automation approaches based on **Descriptive Programming** (one of typical examples is WebDriverIO[3] for Node.js[4]) require significantly more resources for preautomation activities (based on the definition of the test automation architecture, agreement on the coding style rules and the structuring of the tests) and for subsequent test development, particularly compared with the recording approach. Descriptive Programming is the usual approach taken by experienced test automation staff who organize their test scripts into maintainable building parts and usually perform continuous refactoring of the created test automation code. As a result, the maintenance costs of the code are significantly lower. This style also enables more flexible and economical updates of the tests to make them fit with the SUT.

Logically, **hybrid approaches** can also be used to achieve the advantages of the Record and Replay and Descriptive Programming while minimizing their disadvantages.

These approaches, which are usually implemented by various test automation frameworks (for instance Mocha[5] or the already mentioned Robot Framework), may help decrease the test preparation and maintenance overhead, but they still struggle to achieve a low script preparation time and low maintenance time simultaneously.

In our work, we focused on the minimization of the mentioned weaknesses of the Record and Replay and Descriptive Programming approaches. Based on our research, development and industrial experience, we understand that more efficient code refactoring support is an important factor in minimizing these drawbacks.

The automated support of the refactoring of the code to create reasonable reusable objects can significantly decrease the maintenance of the source code in both major test automation approaches, which can improve the overall efficiency of the process.

Therefore, our main goal was to make the refactoring of the created code easier and more accessible to smaller teams and more junior test automation developers, and we achieved this by an automated method to search for potential reusable objects. In this thesis, we proposed a method that is more efficient than the conventional analysis of source code, as it reflects the specific context of the test automation and the fact that particular actions performed by the test can be expressed by various notations in the source code and test automation Application Programming Interface (API).

[3] http://webdriver.io/
[4] https://nodejs.org/
[5] https://mochajs.org/

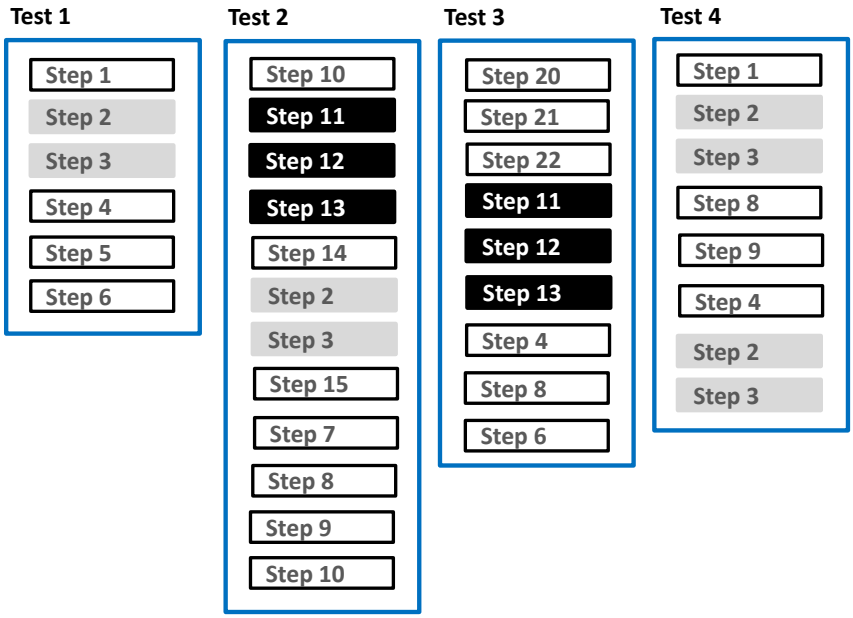

Figure 1.1: Example of duplicate code fragments causing subsequent test maintenance difficult [A.1].

1.2 Practical Effect of the Proposed Method

How does the proposed method improve the overall efficiency of the development of user interface–based automated tests?

In the case of the **Record and Replay** approach, the method aids in identifying reusable objects in an automated and efficient way. Through this, it allows the refactoring of the test automation code, which would otherwise be rarely performed in the industrial praxis.

Let us recall the technical background. The Record and Replay approach allows users to record the actions performed in the application under the test front–end and replay them later using a test automation tool. The resultant code of the automated tests typically contains many duplicate code fragments, which are the source of increased maintenance costs and thus an inefficiency of the overall test automation approach. An example is depicted in Figure 1.1. In four test routines, two code fragments that are performing the same action in the SUT front–end user interface are repeated. They are depicted by the black (Steps 11–13) and gray (Steps 2–3) colors.

In front–end–based testing, the duplication may occur in several places of the automated tests. Usually, we can categorize them into three groups: (i) a common functionality such as the set–up or tear–down procedure of the test, the clean–up procedure after the test is

performed, or the seeding of the testing data ; (ii) the localization of the elements of the front–end user interface and the actions performed with these elements such as clicking on the links or buttons, inputting data into forms or reading values from the screen; and (iii) the sequences of steps representing particular steps of a business case procedure (for instance, a login procedure). Reducing these duplications and replacing them with reusable objects provides a significant opportunity to decrease the subsequent code maintenance costs when the SUT changes. The decreased maintenance enables one to extend the test coverage achieved by the automated tests, enables a better return on investment, and prolongs the effective lifetime of the created tests. All these factors contribute to the overall efficiency of the test automation process.

Nevertheless, not only Record and Replay can benefit from the proposed method. In the case of the **Descriptive Programming** approach, which is used to develop automated tests using a conventional way similar to the development of any software product, an automated refactoring method also finds its application.

In case of the existing sophisticated structuring of the test automation code (the defined test automation architecture, the defined coding rules, and the identification of reusable objects and other rules), an automated search for potential reusable objects can reduce the remaining number of code duplications and make the development team more confident since the created test automation code is closer to the optimum from the subsequent maintenance viewpoint. However, the advantage of the proposed method is more significant when used by a team of inexperienced test automation developers who had taken the Descriptive Programming approach. Because of their lack of experience with test automation, they create naively structured, linear automated tests with insufficient levels of coding rules and the absence or only the partial employment of reusable objects. In extreme cases (which are not rare in the industry praxis), such a code can resemble the output from a Record and Replay test automation tool with many code duplications, which makes the subsequent maintenance difficult.

1.3 Goals of the Dissertation Thesis

To achieve the overall aim described in the previous sections, the goals of this Dissertation Thesis are as follows.

1. To define a model of an abstraction layer for the optimization of unstructured test automation scripts that are either recorded or coded in a suboptimal way.

2. To define the data structures that are necessary for the representation of the information that is needed for processing, such as the internal representation of single test steps, the structures for test suites, suggestions for the proposed reusable code fragments and the control parameters.

3. To propose new algorithms based on the defined formal model for finding potential common reusable objects in the analyzed code of automated tests. The proposed algorithms will be more efficient than the current approaches based on text search algorithms or algorithms for the code refactoring since they will respect (1) the specific context of the automated tests and (2) the fact that the actions performed by the test can be expressed by a variety of code notations, which makes the task more challenging.

4. To design a method for the suggestion of test script refactoring opportunities to the test developers that finally lead to the easier and more economical maintenance of the created tests.

5. To implement the proposed method using a framework prototype that aids in the refactoring of the test automation code. The framework shall be applicable to a variety of test automation projects and, besides its applicability in the classical waterfall development model, it shall also be applicable in the Agile development model.

6. To use the implemented framework prototype to conduct experiments to verify and demonstrate the correctness of the proposed approach and assess its efficiency in front–end–based test automation for web–based and mobile applications.

Apart from that, two minor goals are closely related to the main goals.

1. To define an architecture of the framework that allows for flexible extensions of the method and makes it platform–, tool– and language–independent (in the sense that various test automation languages and APIs can be supported by the framework in the future).

2. To design open APIs for the flexible and transparent user control of the proposed method.

1.4 Thesis Highlights

In this Dissertation Thesis, we present the following.

1. A novel method that automatically aids in the code refactoring of automated tests to reduce the potential duplication in their code and the test code maintenance costs caused by these duplications. The method is applicable to two major test automation styles, Record and Replay and Descriptive Programming. The main effect of the method is to improve the recorded or suboptimally structured test scripts to and increase the overall efficiency of the test automation process.

2. An innovative model of automated test scripts that allows for the following.
    a) The abstraction of the actual actions performed by the tests interacting with the elements of the user interface of the SUT. This model enables a more efficient identification of the potential common parts of the tests that might be achieved by searching for the repetitive parts of the code or the currently established code static analysis methods.
    b) The verification of the applicability of the approach for various implementation languages and test automation APIs, which also includes various types of SUTs such as web–based applications, mobile applications and thick–client type applications for various operating systems.
3. A set of parameterizable algorithms for the identification of potentially reusable fragments of the test scripts that respects the specific context of the test automation. The parametrization increases the flexibility of the method and gives its users the opportunity to adjust the refactoring support to the conditions of the particular test automation language, APIs, used frameworks or other project specifics.
4. The created method is implemented as a practically applicable TestOptimizer framework, which aids in the test automation code's refactoring by searching for potential common subroutines.
    a) The framework is designed and implemented as a platform–independent open method that is flexible to configuration and further extension due to its modular architecture and the defined set of open APIs that control the framework.
    b) The framework is applicable to various development styles spanning from the waterfall model to Agile development styles. It can also be flexibly extended and configured to support various test automation languages and APIs. In the presented prototype, the Java, Selenium and Appium languages and APIs are directly supported.
5. A set of experimental data from the pilot applications of the developed framework for automated tests for web–based and mobile SUTs, their discussion and their evaluation. These experimental data also provide feedback for the parametrization of the proposed method to allow for its future practical applications.

The proposed method implemented in the TestOptimizer framework prototype can be flexibly used in test automation projects as follows:

- for the optimization of the recorded automated tests, which presents a significant opportunity to decrease the potential maintenance of the recorded tests and thus increases the applicability of this approach in the industry praxis;

- for the optimization of naively or suboptimally structured automated tests created by descriptive programming, which, by decreasing the potential maintenance overhead, reduces the risks of the test automation project and enables a better return on investment of the test automation activity; and

- as an auditing mechanism for current test scripts to assess duplications in the tests and the level of employment of reusable objects in the created test automation code.

From the test automation project viewpoint and the viewpoint of the practical applicability of the proposed approach, the main contributions of the proposed method and the implemented framework can be summarized as follows.

1. The method decreases the time that is required for the refactoring and optimization of the automated tests to make them more robust and stable in terms of their prospective maintenance. The methods aids in the refactoring of the tests so that they are less affected by changes in the front–end user interface (UI) of the SUT, which contributes to the easier and more economical maintenance of the test code.

2. As a consequence, the applicability of the automated tests might increase in a number of projects. The tests can be used in project scopes where it was not possible previously because of negative return on investment predictions.

3. The lower maintenance and increased stability of the created tests also allow for the better coverage of the SUT by automated tests. With this higher test coverage, the created automated tests exercise the front–end of the SUT with all its underlying functionality more intensely, and the probability of finding defects in the SUT increases.

4. The previous effects decrease the production risks caused by the potential SUT's improper functionality, as the more efficient test automation method provides the opportunity to test the SUT intensively in a given amount of time.

1.5 Structure of the Dissertation Thesis

The Dissertation Thesis is organized into nine chapters as follows.

1. *Introduction*: This chapter introduces the problem domain and discusses the main challenges that form the motivation for this doctoral project. It briefly summarizes the proposed approach from a conceptual view and outlines its applicability. Then, it defines the goals of the thesis. Finally, it summarizes the contributions of this dissertation thesis.

2. *Overview of Industrial Practice*: To obtain the context of the problem domain, this chapter provides the necessary theoretical background and discusses common industrial practices in the area of automated testing. This chapter discusses the problem from an industrial viewpoint.

3. *State of the Art*: This chapter summarizes the state of the art in research. As several areas overlap in the scope of this thesis, all of them are discussed in this chapter, namely current approaches to test automation, a generation of test cases and test data used as a test oracle, and code refactoring and employment of reusable objects in the test automation code.

4. *Introduction to the Proposed Approach*: This chapter provides an overall picture of the proposed method and framework from a conceptual viewpoint. Its goal is to give the reader a complete picture of the method and its functionality before discussing the particular details of the model, algorithms and technical issues of the implemented framework.

5. *Abstract Model of Analyzed Test Code*: This chapter explains the concept of signatures that is the core of the model of the automated test code. The model abstracts the actual actions of the test scripts from the particular notation in the test automation language or APIs. The defined model serves as a basis for the formulation of the algorithms and the definition of the data structures of the implemented framework, which are described in the following chapters.

6. *Algorithms to Solve the Problem*: This chapter describes the algorithms that are employed in the search for potential common parts in a set of automated test scripts. The algorithms are presented in a pseudo–code notation, and their parametrization is explained.

7. *Framework Prototype*: This chapter describes the architecture of the framework that implements the proposed method, presents its implementation and other technical details and gives an overview of the APIs through which the framework is controlled.

8. *Experiments*: This chapter describes the experimental verification of the proposed approach, starting with a description of the experiment design. Then, it defines the metrics that are used to evaluate the results, presents the experimental data and finally discusses the results and the threats to its validity. In this chapter, the results of experiments on two framework prototypes are presented, and a comparison of them is made.

9. *Conclusions*: The final chapter of this thesis summarizes the results of our research, suggests possible topics for further research, and concludes the thesis.

Chapter 2

Overview of Industrial Practice

The goal of this chapter is to introduce the context of this thesis from an industrial praxis viewpoint. The chapter introduces the necessary theoretical background and discusses common industrial practices in automated testing. We discuss different testing approaches, the structure of the software testing process and the principal approaches to test automation. Then, we introduce different techniques for test automation and various types of test automation frameworks and tools. Finally, the chapter concludes with a discussion of the test automation praxis in different software development methodologies.

2.1 Quality Assurance

The development of any software product (of nontrivial functionality and size) is a complex process that involves a number of activities. Having a satisfied customer at delivery requires a development team with considerable experience, good management and excellent communication skills.

At the beginning of the project, software company representatives interview the customer to acquire the basic requirements concerning the software product. In larger software teams, business analysts work with the customer or customer representatives to define requirements for the software product. They usually prepare a solution proposal in cooperation with software architects. The architects are then responsible for the technical side of the proposal. They should ensure that the solution is feasible, define the technology and tools to be used during development and propose a production environment. Business analysts with project managers break the requirements down into atomic pieces (e.g., a user story in Agile Scrum) that can be processed unambiguously by production teams. The management chain (project management, development lead, and quality assurance lead) prepares a development plan based on the analysis and available resources. Depending on the chosen process model, e.g., Agile or Waterfall, the project requirements and line management drive the entire process.

Software testing and Quality Assurance (QA) are essential to the success of a software project; they evaluate the quality of the product being developed and identify risks in implemented functionality as well as in nonfunctional requirements. The sooner defects can be found, the less costly it is to detect them, as was observed by Barry W. Boehm more than 30 years ago [16, 17]. It is economically more logical to correct SUT defects revealed during early development than to fix defects found by the customer during user acceptance testing (UAT) or during the production run of the SUT. Generally, earlier defect detection helps to complete the project on time, within budget, and at a defined quality level. When the product is released, maintenance engineers become responsible for customer support and for resolving defects in the delivered product.

Quality Assurance involves undertake systematic activities such as systematic measurements and comparisons against standards defined in a quality system to ensure that the product or service fulfills the requirements. In contrast, software testing is a subset of quality assurance. Software testing represents an investigation that provides product owners with information about the quality of the product under test.

2.1.1 Testing Approaches

In software testing, development teams may follow many approaches for assessing if the software meets certain criteria, required needs and required level of quality and reliability. Available resources (for example, size of the available testing team), project budget or a type of the SUT play important roles when the team decides what approach to take. Test automation is also one of these options. For some types of tests, as usability tests or accessibility tests, manual testing is a logical choice, as these types of tests can be hardly automated. On the other hand, load, performance and stress tests are hardly feasible to be performed manually without any automation. Unlike usability and accessibility tests where human users are subject of observations when they work with software, or they try to asses a level of accessibility for users with disabilities, load and performance tests can be hardly executed with 10,000 manual concurrent users for instance. There is a high need to run automated scripts in such scenarios. Otherwise, it is almost impossible to manually measure, e.g., a response of the system under test or an average transaction processing time. Human testers cannot reach such level of testing in comparison to specialized tools.

However, the most frequent type of tests which are being employed in the software projects is functional tests aiming at detection of defects in the SUT functionality. In this type of tests, both manual and automated tests can be and are employed in the current industry praxis. Manual functional testing may win over automated testing in certain cases. For instance, if a senior tester conducts an experience–based testing and searches for new defects or anomalies (e.g. to avoid a pesticide paradox [18]), test automation can hardly replace such an activity efficiently.

Testers use different testing techniques based on their knowledge of a SUT (in this

point, let's not substitute the knowledge of SUT for the knowledge of the business domain, in which the processes supported by the SUT are performed). If the SUT implementation (e.g., a complete system architecture, technology or a source code) is known to the testers, this situation is referred to as the white box testing style. If testers do not know the implementation details, black box testing is established as a term describing such style. A situation when testers have limited knowledge about implementation details so they can design more efficient tests based on them is called grey box testing. White box testing is typical to the software developers and design of unit tests. QA and testing teams usually do not design tests at this level of detail. User Acceptance Testing can be seen as an ideal representative of black box tests in contrast to white box tests.

2.1.2 Common Structuring of the Testing Process

Business requirements play an important role in the software development process (see the top left part of Figure 2.1). Requirements define the functionality of the SUT as well as the user interface desired by the investor of the software project. Test cases exercise features and the expected behavior of the SUT. For this purpose, testers use the user interface to control the SUT and verify that behavior of the SUT corresponds to the defined requirements.

Let us imagine the following example. A customer wants to track all shipments and wishes to see top sold items from the stock. Now let's discuss a difference how software development team (for this example consider it as a team of software developers only) and software testers process the business requirements and which consequences it has for the subsequent software development process and duplications in the created software test cases. Software developers design and implement a solution based on the requirements. Furthermore, they select an implementation technology fulfilling their and customer's requirements. The software testers and quality engineers verify that implemented SUT features work as expected and validate that the product does what is supposed to do. Most often, they compare the actual functionality of the SUT to the requirements, which are processed in the form of test cases.

In our example with the shipment overview required by the customer, the feature is going to be implemented as a Model–View–Controller–based application. A model is stored in a relational database, a web user interface represents the view, and a controller handles the user interaction and calls saved SQL procedures to get required data from the database. The testing team should focus on several tasks when testing particular features of the SUT. First of all the testing team should verify whether components of the SUT work according to the SUT specification.

For example, the components under test are exercised if a connection to the database works properly in all required cases, if web pages of SUT user interface present correct data from the SUT database or if the application controller reacts on user actions in different web

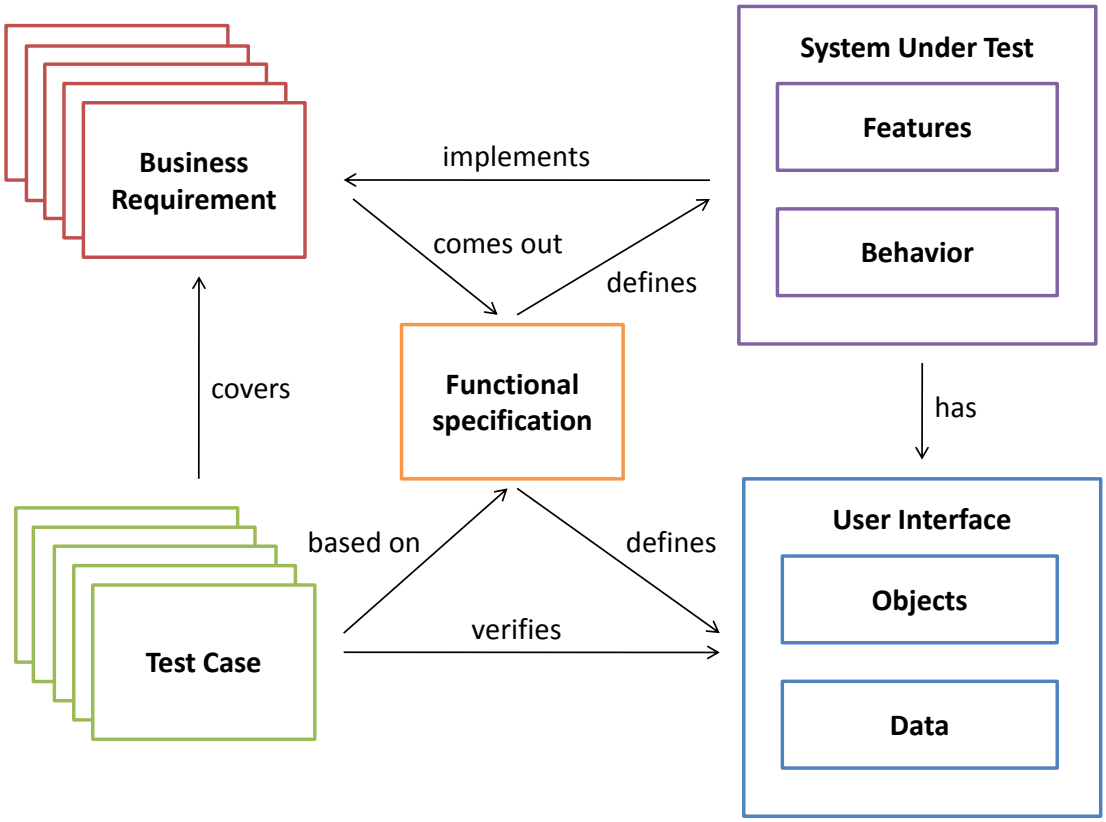

Figure 2.1: A relationship among business requirements, test cases and the system under test.

browsers. Moreover, testers should validate whether the application provides an output to the customer what he/she wanted. In our example, are displayed top sold items from the stock in the way the customer wanted? During the validation, testers do not focus only on the functionality but also on methods how the functionality is achieved, and if it is in accordance with the customer's expectations.

Further, in the software development process, a functional specification defines required features, mockups or wireframes of the user interface, and requested behavior of the application. Well–written tests should be based on business requirements. Otherwise, the team may test something else that was not previously required by the investor, because all parties may introduce their different point of view of the problem and different interpretations of the particular requirements may occur. The more vaguely defined the software requirements are, the higher this risk of misinterpretation is and, subsequently, the probability of software defects caused by these misinterpretations is increased.

As a standard software testing practice, the test cases are based on the functional

specification and on the business requirements to verify that the SUT complies with the business requirements. During the analytical phase of the software project, sets of business requirements are usually grouped into one SUT feature (an example of the situation is depicted in Figure 2.2).

In such a situation, if the testing team focuses only on covering the whole SUT features by tests and do not reflect on also covering the business requirements, some of these requirements might be forgotten. Therefore, covering business requirements by tests might seem as more suitable approach compared to covering the SUT features by the tests. On the other hand, covering business requirements by tests might represent a source of potential duplications in created test cases – the tests may overlap, and a particular business requirement might be covered by them several times.

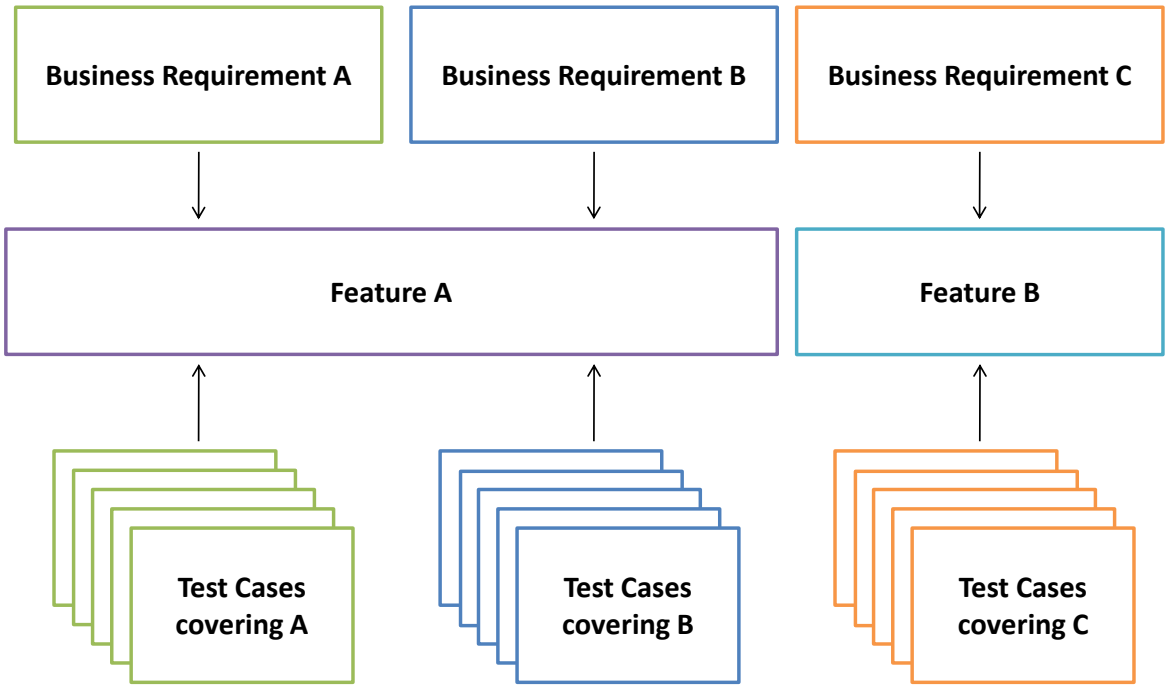

Figure 2.2: The feature A contains the requirements A, and B. Currently, the test cases A, and B cover the business requirements A, and B. The Feature B implements the requirement C, and it is covered by one test case C.

The Figure 2.2 illustrates an example of the described situation. Feature A represents business requirements A and B. The feature B includes the requirement C. In our model

example, a testing team prepared three sets of test cases covering all the requirements.

For instance, the first requirement A represents an option to print a text file in a Windows format, the requirement B asks to print a text file in a Unix format. To do that, the user has to login to a user interface. The feature A represents the following workflow: the user opens a wizard in the SUT, logs in to the system, and displays the file in the required format. This workflow is the same for the requirements A and B, only the format of the file differs. However, both requirements shall be tested by different sets of test cases – test cases A and test cases B in the Figure 2.2.

In such a situation, it is more than likely that the created test cases would contain similar steps. A problem may arise when the team is asked to update those tests because of changes in the SUT caused by a change in the workflow specification. Such updates might take the significant amount of resources of the testing team, which is missing in the actual testing effort. This problem increases, when the created tests are automated, as we explain later in this thesis.

In automated testing, the best practice is to extract common test steps into a separate reusable procedure (Figure 2.3), and employ this procedure in relevant test cases. Unfortunately, in many cases, testing teams do not design and create tests according to the best practices – usually due to time constraints and lack of experience. This fact has a fundamental impact on the complexity of subsequent test maintenance and arises especially in the case of automated tests.

To better understand the issues related to the maintenance of software tests, let's analyze the IEEE 829 Test Case Specifications. The IEEE standard defines a test case as a set of (i) the Test Case Specification Identifier, (ii) Test Items, (iii) Input and Output Specifications, (iv) Special Procedural Requirements, and (v) Intercase Dependencies. Furthermore, it can include a Test Procedure Specification. The Test Case Specification Identifier serves as a unique identifier of the test. Test Items define a subject of deliverable and test. Input Specifications specify user inputs or files, and Output Specifications define expected results, including screens, files, and timing. Special Procedural Requirements define possible operator interventions and required permissions.

The special procedural requirements can influence testability of the SUT by automated tests. Let us give the following example from the payment application domain. The user is asked to use a credit card when paying an invoice. Conducting a manual test is simple in this case because all actions are carried out manually, but automated tests are almost impossible to carry out without any human intervention. The operator has to swipe the credit card in the middle of the transaction to finish the transaction.

Testing of complex software systems requires to define Intercase Dependencies between individual test cases, which are in other methodologies also referred as preconditions. Let us explain a precondition by an example from the payment application. If we are expected to execute a test for a credit card payment, we need to have a test infrastructure up and running, e.g., a server processing the payment transactions and a terminal for the credit

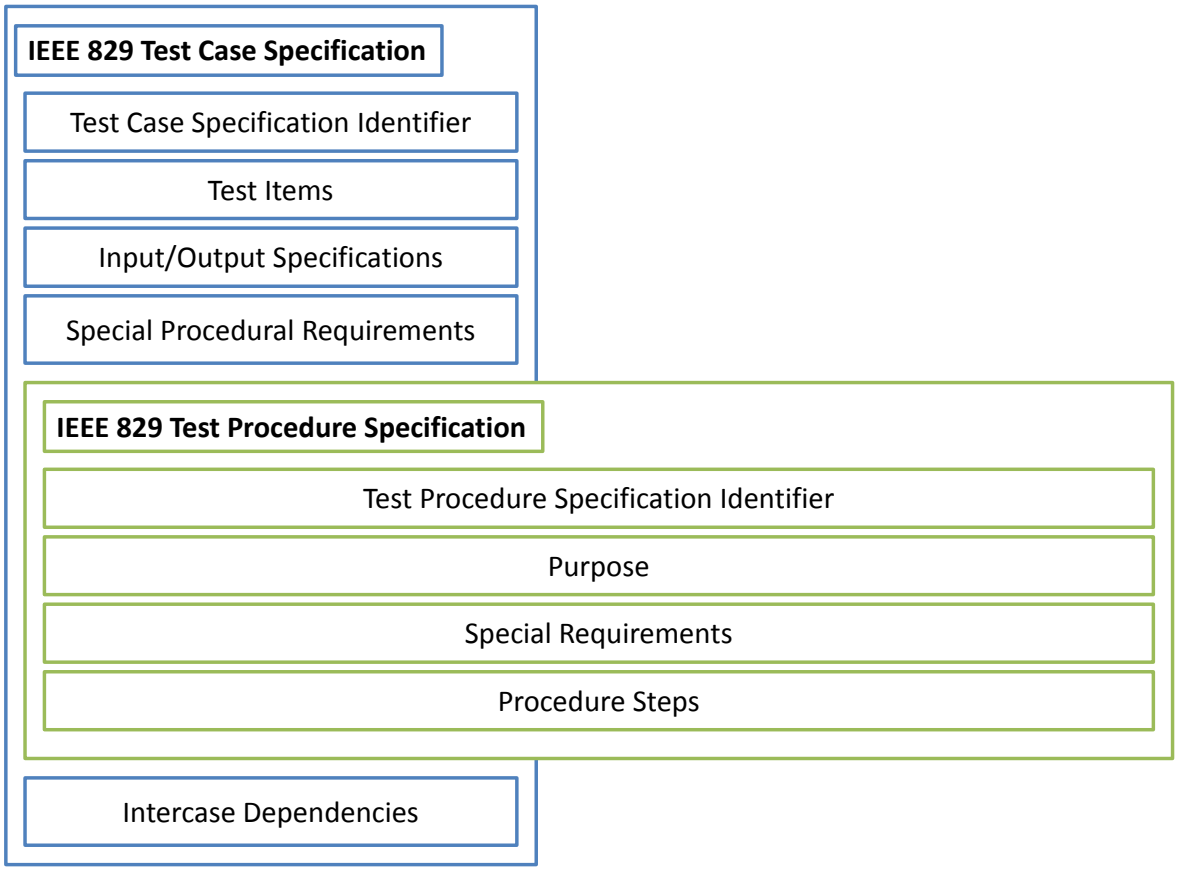

Figure 2.3: IEEE Test case and procedure specification.

card.

Preconditions help testers to set expectations in which state the SUT prior to the execution of particular test case should be. As we already mentioned above, the recognized best–practice in the domain is to create test cases as independent units, so in order of their execution does not influence the ability to execute them or their result. Otherwise, a test execution may fail if one or more tests in the cascade fail. Tests (and especially automated ones) need to have the system in a known and defined state in accordance with preconditions. If this promise is not fulfilled, tests may report the defects that are not relevant, or they may even damage the testing data in the SUT.

Finally, a test procedure specification describes running of test cases. It includes the following sections:

1. *Test Procedure Specification Identifier*,

2. *Purpose*,

3. *Special Requirements*,

4. *Procedure Steps*.

The *Purpose* states what tests shall be run. The *Special Requirements* section describes requirements on testers, what permissions are necessary, and how environment shall be configured (for example, data must be seeded in the system under test before testing takes place). *Procedure Steps* list typical activities within the pre–testing such as login or set–up (for instance, it corresponds to `@Before` annotation, i.e., `setUp()` methods in JUnit), and within the post–testing such as a measurement of results, clean up, shutdown or restart if needed (again imagine a JUnit example – this part corresponds to `@After` annotations and `tearDown()` methods). The overall situation is outlined in Figure 2.4. The Test Step Definition provides an exact description of an object under test, i.e., object locators or unique properties like id or name in the SUT (in the Figure 2.4 depicted as *Test Object Definitions*). Then it specifies an *action* being carried out either by an action in the SUT (for example, open contacts in an e–mail client) or by the tester and/or automation tool – like verify the data in the SUT mapped to actual results and compare them with *expected results* of the test. *Parameters* are used to distinguish between cases having identical test objects and actions but covering different semantics, e.g., a test of a login procedure involving different usernames and so roles in the SUT.

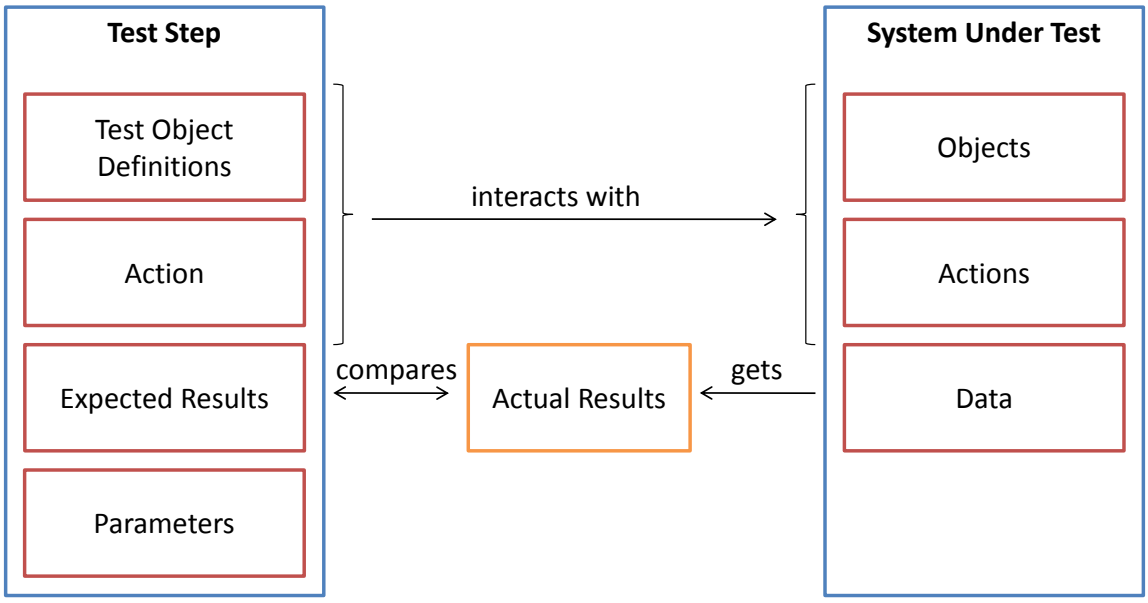

Figure 2.4: An interaction of the tester with the application represented by the test step.

2.1.3 Manual Testing

As the software project budgets are usually limited, a number of these projects require to find a good balance of resources spent on the quality assurance process and getting an appropriate resultant quality of the software product. Costs on software testing are usually minimized minimum unless the SUT is not mission critical. For these purposes, manual testing with ad–hoc approaches [19, 20] have their justification. Manual testing is beneficial if tests are executed only once without repetitions, or tests are only run on an irregular basis and quite rarely. If standard functional tests are repeated regularly, it is usually more effective to consider employing automated tests to cut down costs on multiple test runs.

Tasks that require intensive testing including repetitive test runs may appear as ideal candidates for test automation. However, some of them such as accessibility testing should be excluded from automation efforts. In the accessibility testing, it is crucial to evaluate and asses the system under tests from a human viewpoint, which is very difficult to record in automated tests. Furthermore, each person is an individual, and what is working for one human user it is not working for someone else.

Software methodologies like **TMap NEXT**[1] allow to introduce the structured testing process that increases a systematic nature of testing and thus its efficiency. However, despite this methodological effort, main issues of manual testing remains. Manual test runs are not consistent over their different runs, which is typically caused by random errors during the test execution or a different interpretation of test steps by different human testers. This issue is more relevant for tests with a low level of details and/or for testers on a contractor basis that change from test run to test run. Furthermore, some test steps can be omitted during the test execution if testers do not pay proper attention to testing. All those issues speak against manual testing for regression testing and tests with a significant number of repetitions.

2.1.4 Test Automation

The efficiency of a testing process can be improved in several ways such as a better test analysis, employing test data optimization methods like Constrained Interaction Testing [21], or automation of the test execution. If a testing team adopts well this technique well, automated tests offer a number of advantages in comparison to manual testing. First of all, automated tests can be run repeatedly on multiple platforms and with different configurations. As a result, test execution costs are low. Secondly, they provide more accurate results than manual tests. Last but not least, automated tests can be executed in non–productive time slots during a software development cycle — they perfectly fit into a Continuous Integration (CI) concept.

[1] http://tmap.net/tmap-next/

Table 2.1: Typical script preparation and maintenance times of principal test automation approaches.

Type	**Script Preparation Time**	**Script Maintenance Time**
Plain recording of test cases	low	high
Test scripting using a programming language	high	medium
Test automation frameworks (Data–driven frameworks, Keyword–driven frameworks and Model–based frameworks)	medium	medium
Hybrid approaches	May decrease the overhead	May decrease the overhead
Requirements of software industry	medium	low

However, the test automation has also its disadvantages. An effort to prepare the automated tests is usually higher than an investment into the preparation of manual tests cases. Automated tests always run according to a predefined scenario and do not offer a space for a creative human improvisation. That means if the SUT changes (and a change does not have to be drastic like a complete change of a use case) automated tests usually fail. In contrast, human testers use in those cases their intelligence, so they are able to overcome small inconsistencies in the definition of the test case and to continue in tests. Consider an example of a small change in the name of a button from Cancel to Discard, when a name of the button is used to identify the button. Human tester very likely continues with the test, differently to an automated test, which might fail, because it has not found the Cancel button.

Moreover, if automated tests are developed in a sub–optimal manner or if changes in the SUT are frequent, maintenance costs of the automated tests might grow rapidly.

From our observations [6] and other publications [7, 14, 8, 22], relatively high maintenance of automated tests is considered as one of the biggest challenges in the domain of automated testing, see the comparisons in Table 2.1. Moreover, the issue occurs regardless of a test organization model being employed and influences the test automation process [23]. Typically, high costs of maintenance are closely related a simulation of user actions in the front–end of the system under test, which is required for functional user interface tests.

2.2 Techniques in Automated Testing

The following subsections give an overview of the main techniques used for automated functional tests. We start with a problem of generating data used as a test oracle[2], then we introduce the most common approaches in software test automation, and we finish with a list of currently available test automation tools and frameworks.

2.2.1 Initial Overview

Automated testing represents a complex domain of methods, techniques and approaches for validation and verification of software. In the broader context, this discipline starts with an automated preparation of testing data, generation of tests from SUT models or specifications of the SUT, continues with an automated test execution and it finally ends with an automated verification of the test results. **Test automation** is usually understood as a subset of automated testing. It is focused only on automating of test execution, with the aim to replace human testers with specialized software or physical machines. Currently, a number of test automation methods is available. These methods are starting from naive approaches such as a simple test recording that basically captures a user activity when the user interacts with the system under tests. Senior test automation experts usually prefer descriptive programming, which is similar to developing scripts or employs various frameworks based on data–driven, keyword–driven or hybrid approaches. Those frameworks may be developed proprietary or developers can adopt open source frameworks like a Robot Framework[3] for instance. Their main contribution is to speed up test development, simplify test maintenance and to enable beginners to develop automated tests.

A recent growing trend in test automation is using a visual recognition of objects. This recognition is based on the Optical Character Recognition (OCR) approach, which helps to recognize user controls in a way like humans do. Furthermore, this technology introduces a platform–agnostic approach so test developers do not need to deal with, for example, a Document Object Model (DOM) of a web page or mobile application, but they work immediately with the visual object such as buttons, sliders, etc. An ability to run the same test on different platforms (web, mobile) and in various front–ends (iOS, Android) but with the same look and feel, is impressive. However, this approach has also its drawbacks – when the user interface of the SUT changes, the new recording of the respective screen part has to be done, which could imply considerable maintenance overhead, compared to advanced, well–structured descriptive programming approach.

[2]In software testing, we understand this term as a mechanism that allows deciding if a particular test passed or failed.

[3]http://robotframework.org/

2.2.2 Test Data Generation

Test data generation plays an essential role during the software verification process. The test data determine the initial state of the SUT before the actual test, and in this sense, they are essential to ensure this state. Moreover, various test data enables to exercise the test cases in a number of variants and possible combinations, increasing the probability to detect important defects in the SUT.

Apart from manual methods of test data preparation, test oracles are usually used to generate the test data for more complex cases. A test oracle can be created by several methods. The simplest method can be applied in cases in which a legacy system with an already proved functionality, same or similar to the SUT is available. Then we may use the data from the legacy system as a test oracle when an upgrade of the system to the newer version or a new platform is tested.

When a completely new SUT is developed, practically usable test oracle might not be available, and if the SUT is not based on computations (for example, a system performing computations according to math equations), the creation of a test oracle may be a challenging task.

However, the test oracles are one of the favourite subjects of test automation. Particular state of he art in this area is further discussed in Section 3.1.2.

2.2.3 Test Automation Approaches and Frameworks

When a testing team decides what strategy to automate functional user interface tests to chose, it has two principal options. The tests can be created quickly, and subsequent test maintenance is going to be complicated and demanding, or the test can be created slower with higher initial costs, but the test maintenance is not going to be demanding like in the first case. Finding a right trade–off between development speed and maintenance could be a challenging task. The principal approaches to the user interface based test automation, as already mentioned in the introduction to this Thesis, are:

(i) **Record and Replay**, which is based on capturing all steps what the test engineers conduct in the application front–end by a test automation tool, and

(ii) **Descriptive Programming**, which is similar to coding any program and is based on the writing of an automated test script.

Both methods can be combined to achieve the better balance between the test script preparation speed and subsequent maintenance costs. In the beginning, we may want to develop first drafts of tests quickly, so we record user walkthroughs in the SUT user interface. In the second phase, we need to stabilize tests and to add needed verifications;

eventually may want to add additional logic to drive the test execution and data flow. Then we script tests using descriptive programming. The test recording approach is generally considered as a fast method of development of tests but very inefficient from a test maintenance point of view [24, 25, 26].

In the effort to asses the maintenance viewpoint, usually, we cannot easily determine what an optimal level of the record and replay and the descriptive programming approaches is since many conditions and factors come into the place.

Firstly, how complex the SUT is, i.e., how many screens are available, what typical use cases are, or what data are required and processed by the system (consider an example of a calculator vs an integrated development environment). Secondly, a required structure of automated tests and a level of knowledge of the development team. Thirdly, we have to consider a technological side of the system under test as well as what test automation tools are available, and last but not least, what is a purpose of tests.

In general, a conclusion is that the Record and Replay approach creates not enough robust tests, but costs on development are relatively low. The Descriptive Programming approach helps to create tests robust enough using more advanced techniques like manually choosing best descriptive properties for locators of user interface elements or using escape strategies for scenarios when some objects are not found or when errors occur. Descriptive programming demands, on the other hand, a higher initial investment but a resultant test script quality may be significantly higher. Test maintenance costs are then lower [24, 27]. We can confirm these observations, as we witnessed this effect in a number test automation projects in our industry praxis.

From a technical viewpoint, Record and Replay, as well as descriptive programming conducted in a naive and unstructured mode, usually results in test scripts having a potentially high ratio of repeating fragments (an example has been depicted already in Fig. 1). These code–fragment duplications may cause issue and may lead to an increased need for maintenance. Moreover, they cause tests are not robust to small issues during the test execution. Suppressing duplications in the code of automated test scripts has been considered as an approach that can have a significant impact on the economics of testing [28].

Lanubile et al. [29] presented a study in agile software development environments focused on software inspections. They studied quality attributes of automated tests with the main focus on unit tests and proved that several approaches such as a software inspection could significantly improve the quality of resultant automated tests. Advanced test automation approaches for a semi–automation are based on models being used for a test case generation, for example [30, 31, 32, 33, 34]. If the system under test is changed, the model or the specification is updated, and new test cases can be re–generated. When a detailed model of the system under test does not exist, this fact makes use of the model–based methods difficult and a total efficiency of those methods disadvantageous. Another option for harnessing a test automation is using of partial models that can represent only one fea-

ture or story and so they can be quickly developed. It simplifies the preprocessing phase and makes the whole process of test generation more flexible in comparison to complex models of the whole system. For rapidly changing systems not available models in case of limited resources or team skills, testers directly script tests or record them. These test development approaches, however, require a manual work and overall costs on automated tests then significantly rises with a number of changes in the system under test. Manual development of test scripts can be accelerated by automation frameworks. Garcia defined a testing framework as *"a set of abstract concepts, processes, procedures and the environment in which automated tests will be designed, created and implemented"* [35].

The Automated Testing Institute[4] recognizes three generations of frameworks:

- **1st generation frameworks** represent a linear development of automated test scripts by record and replay approaches,
- **2nd generation frameworks** introduce a concept of re–usability and
- **3rd generation frameworks** involve advanced approaches for driving the execution by keyword and allow to generate automated tests based on SUT models.

First generation frameworks represents a linear approach for the development of test scripts. Tests are usually developed using the Record and Replay approach and are understood as an extension of their manual opposites. They are not structured and do not handle duplication in the test source code. Furthermore, they do not provide any added value in the sense of a quality of the product. On the other hand, they are easy to use and usually work fine in small–sized projects. Return on investment is achieved faster (is the SUT does not change) but maintenance costs are typically very high. The **Second generation frameworks** are typically improved in two aspects. They also employ linear scripts, but data are stored in a database (Data–driven frameworks) that allows reusing one test with multiple parameters, i.e., it introduces test parameterization. Secondly, these frameworks introduce a functional decomposition (reusable code), so test creators can build tests on existing components. Finally, the **Third generation frameworks** are divided into two main groups:

1. Keyword–driven frameworks and
2. Model–based frameworks.

Keyword–driven frameworks process automated tests being represented as tables with a set of keywords, i.e. available actions of the framework associated with application–specific or framework–specific functions and scripts. A framework runner interprets actions with

[4]http://www.automatedtestinginstitute.com/

data parameters, and the interpreted actions are executed by a test automation tool to control interactive user controls in the user interface of the SUT.

Model–based frameworks are provided with additional information about the SUT, i.e., a specification or a model, which is used to generate test cases automatically and execute them in a semi–intelligent manner. With model–based frameworks, automating tests actually means to develop a model of the system under tests and not to develop tests themselves. Initial costs of development of the automated tests are better for the first two framework generations, however, the third generation is more suitable if a higher demand on test maintenance is expected.

In the following paragraphs, we give an overview of currently available and the most known and applied test automation frameworks.

Microfocus Application Lifecycle Management (formerly HP)[5] is a suite of software quality tools for test management, requirements, and the whole application lifecycle. The current suite is built on top of products from the Mercury Interactive Corporation, which was acquired by HP, and the newly created HPE software division from HP was sold to the Microfocus corporation later on. It comprises a Microfocus Application Lifecycle Management web–based tool for the test management, Microfocus Unified Functional Testing integrating graphical user interface functional tests and API service tests in one tool (formerly HP QuickTest Professional).

IBM Rational Quality Manager[6] is a suite of products helping to drive quality of the system under tests through its application lifecycle. Similarly as the Microfocus ALM, it contains tools for functional testing (Rational Functional Tester), test management (Rational Quality Manager), performance testing (Rational Application Performance Analyzer) or security testing (IBM Security AppScan).

Selenium[7] is an open–source framework for web applications in the browser. Therefore, it can be used not just for the creation and execution of automated tests but also for automation of different tasks in web browsers. It is being developed under the Apache 2.0 license, and everyone can contribute. Selenium consists of several projects. Apart from the Selenium IDE, which is an add–in for a Firefox browser that allows to record and replay actions taken in the system front–end, it includes other projects:

- Selenium WebDriver – original Java-based testing system.

[5] https://software.microfocus.com/en-us/solutions/software-development-lifecycle
[6] https://www.ibm.com/us-en/marketplace/quality-management
[7] https://www.seleniumhq.org/

- Selenium Remote Control – a client/server system to control web browsers locally or on remote machines,
- Selenium Grid – system for parallel test execution.

ThoughtWorks that originally developed Selenium, also developed a framework **Twist**[8] for the creation of tests, their maintenance, and finally for the test execution using any Java or Groovy–based test driver. The Twist framework supports out–of–the–box many popular open source automated software–testing drivers.

Robotium[9] is a test framework, which allows users to write robust automated black–box tests for Android applications. With the support of Robotium, testers can write functional test scenarios, spanning multiple Android activities. It offers better robustness of test cases due to the run–time binding to GUI components.

AutoIt[10] is a freeware scripting language (similar to the BASIC programming language) designed for automating a Windows graphical user interface. It was initially designed for automation and configuration of thousands of PCs. Since it supports complex expressions, user functions, or loops, it can be also used for the test automation of functional tests.

Parasoft **SOAtest**[11] provides a comprehensive solution for a cloud, SOA, and API testing. It is designed for automation of complex scenarios across a web user interface, service layer, ESBs, databases, and mainframes. The SOAtest is shipped with a Parasoft Load Test. Both the tools can be integrated with Parasoft Language products such JTest.

SilkTest[12] was a tool formerly develop by a Borland corporation. Currently, the project is maintained by the Microfocus corporation. It is a tool for automated functional and regression testing of enterprise applications. The whole solution ensures that the application being developed meets business needs and increases an test efficiency.

The **Software Testing Automation Framework** (STAF[13]) is an open source, multi–platform, and multi–language framework offering reusable components for test creation. The idea of the framework is to remove the tedium of building an automation infrastructure and enable the user to focus on building the automated tests.

[8] https://www.thoughtworks.com/products/twist-agile-testing/
[9] http://www.robotium.org/
[10] https://www.autoitscript.com/site/autoit/
[11] https://www.parasoft.com/products/soatest/
[12] https://www.microfocus.com/products/silk-portfolio/silk-test/
[13] http://staf.sourceforge.net/

TestComplete[14] is an automated testing tool for creating, managing and running tests for desktop, web or rich client applications. Tests can be recorded, manually scripted or created in a script–free mode and for the playback. **WATIR**[15] (Web Application Testing in Ruby) is an automated testing tool using the Ruby language to control web browsers. The tests are scripted in Ruby and executed in the web browser.

A **Robot**[16] framework is a generic test automation framework mainly for acceptance tests and acceptance test–driven development. It utilizes a keyword–driven test approach and its capabilities can be improved by custom Java or Python libraries.

Cucumber[17] in comparison to the Robot framework merges a design specification and test documentation into one single source of SUT specification. This framework principally supports the behavior–driven development style.

Perfecto Mobile[18] provides a cloud–based platform for testing on mobile devices and provide integration with all major test frameworks.

Espresso[19] is a framework for Android UI tests. Requirements on cross–platform testing, unfortunately, exclude this solution from the radar of many test automation teams.

Soasta[20] focuses on load and performance testing, optimization and measuring with a capability to simulate different network conditions with test clients from all over the world.

Mocha[21] is a popular JavaScript framework running on Node.js and in the browser. It allows asynchronous testing.

Appium[22] is an open source test automation framework aimed at mobile web applications and native mobile applications. Appium supports multi–platforms and drives iOS, Android, and Windows applications using the WebDriver protocol.

[14] https://smartbear.com/product/testcomplete/overview/
[15] http://watir.com/
[16] http://robotframework.org/
[17] https://cucumber.io/
[18] https://www.perfecto.io/platform/test-automation/
[19] https://developer.android.com/training/testing/espresso/
[20] https://www.soasta.com/load-testing/
[21] https://mochajs.org/
[22] http://appium.io/

2.3 Specifics of Test Automation in Different Software Development Methodologies

In this section, we analyze and discuss specifics of the test process and its consequences for test automation process in waterfall–type and Agile software development methodologies. Understanding this context would help to get a better overall picture of the current test automation issues in the industry praxis.

2.3.1 Test Automation in Waterfall Methodologies

Waterfall software development methodology became a standard in the software in the past decades as it allows software development teams to drive development in an organized and scheduled manner. All activities are broken down into phases and a next activity does not start until the previous is finished, hence it is named waterfall. Testing phase follows the end of the development phase – in consequence the testing team is usually not involved in test–as–you–go activities before a handover of the SUT to testers. Later versions of the waterfall methodologies encourage the testers to start creating test scenarios during the development phase of the project, when design phase is finished. However, still, this later involvement of the testers to the process may be understood as a potential limitation of the method in comparison to agile approaches. On the other hand, this characteristic simplifies a test plan creation because all features to be implemented are actually done before testing starts.

From the test automation viewpoint, Waterfall methods (either a standard version or a modified Waterfall with iterations) can remove potential obstacles that developers of automated tests may experience. Such obstacles typically are: (1) a definition of the user interface is not ready yet, (2) not all functions of the front–end are ready, or, for instance (3) alternative measures must be taken in order to proceed with test activities according to the use case, as the SUT is not completely finished in the time of development of the automated tests.

In Waterfall development style, project actions run sequentially and the number of potential dependencies to other team members or features is reduced to a minimum. The testing scope is then more clear which results in simpler test automation that is not interrupted by not an unavailable functionality or unstable environment. However, this does not apply in the situations, when the SUT is already developed, but the presence of defects in it is so high, that this state practically postpones the start of the test automation activity.

2.3.2 Test Automation in Agile Environments

Unlike the Waterfall software development style, Agile methods (e.g. Scrum or Kanban) introduce new challenges for the testing teams since project activities run less sequentially and more in parallel in shorter development cycles. A benefit of those dynamic approaches is less costly fixing because when the defect is introduced in early development phases, it can be fixed sooner than at the end of the product cycle. This is an essential property of Agile methods because testers may test earlier than in the Waterfall planned test phase. This advantage is also one of the reasons, why many teams decide to employ Agile methodologies in the current software development praxis.

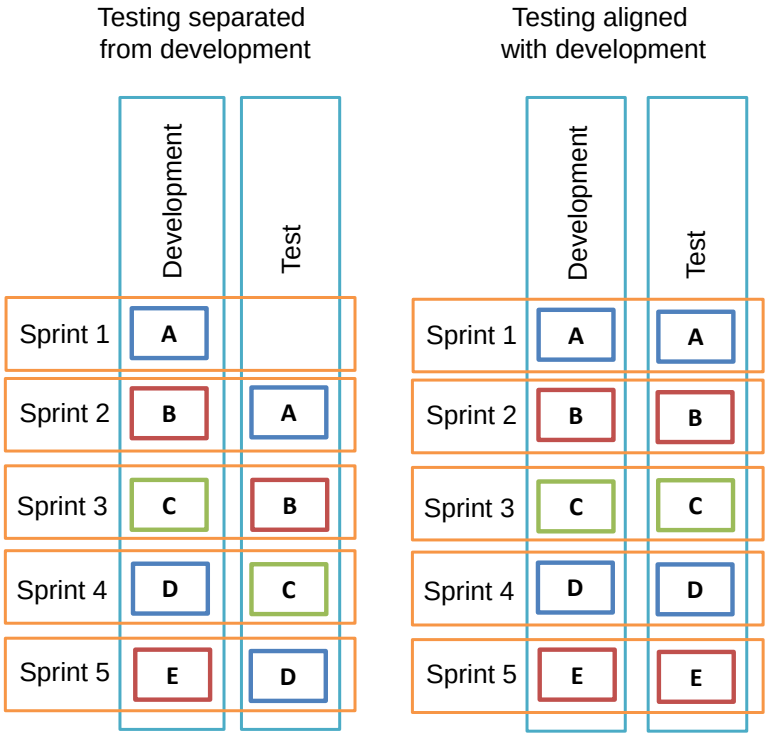

Figure 2.5: A comparison of a testing separated from SUT feature development and testing aligned with SUT feature development (features are depicted by letters) in Agile environments.

In Agile environments, testing can be driven in several possible modes. A set of test automation practices for Agile development environments were presented by Collins [36]. In this overview, we discuss two typical options for organization of software testing in such environment [37]. In the first method (left side of Figure 2.5), testing is separated from development. While the first feature A is under development in the first sprint, testers wait for the feature to be completely developed, and they start to test it the next sprint. In this model, testing is always at least one sprint behind.

Such an approach is similar to a Waterfall approach at a low scale of single sprints. Developers and testers are not aligned and do not work on the same set of features in one sprint. Developers elaborate and implement the user stories in the first sprint. While the testers conduct preparation activities for the upcoming tests, i.e., they install a test infrastructure, analyze requirements or define acceptance tests. Alternatively, testers can participate in a feature design phase and review of functional requirements as well as the technical design. After the first sprint, the development team starts to work on the user stories of the second sprint and testers start to test features developed in the first sprint.

In this style, developers have enough time to fix found defects (they can use the whole sprint to fix them), which, in the ideal case, should result in a situation where the backlog of issues to be fixed does not grow during the project. A complete sprint available for testers gives them an opportunity to develop automated tests. Mockups techniques are usually not necessary in this style, as the SUT functions to test are available.

The second option of testing in Agile software development methodologies is a parallel alignment of testing and development activities. This approach represents an extreme challenge for the testing team if they want to automate tests for stories being developed in the actual sprint. It requires a very good collaboration among all core team members and spreads traditional well–bounded roles between developers and testers at the same time. Aligning testing with feature development brings new issues for testers. At the beginning of the sprint, testers have to wait until a partial development is done and they can start to propose tests based on the implemented features. It is obvious if a feature being implemented is not in a runnable state, automated tests cannot be developed. In such cases, test engineers may use SUT mockups to prepare tests based on them and to speed up test development during the final phase at the end of the sprint when the feature is delivered. This style results in a very limited time that test engineers have to design, create and debug automated tests, and to test the user stories.

A figure 2.6 shows a proposal for a collaboration of the core team members. The development team is assigned with a user story. Then developers start to work on this user story – they design and discusses an initial proposal with the tests during the first week of the sprint. Based on that, testers prepare requirements on the test cases according to test recommendations from developers. At the same time, development uses the test requirements as a checklist of what is going to be validated by the testers. Before testers start to create the test cases, they should have a UI wireframe sketch of the user interface or proper mockups. Otherwise, they will not be able to define locators of the SUT user interface elements to be handled by the automated tests, and basic behavior of the tests. Every delay then pushes the testing team in a situation when they will not be able to verify the developed feature by automated tests. Not later than in the third week, developers should have the first prototype ready. This is the key moment of the sprint because the testers can finish the creation of tests based on the real SUT prototype, and start to

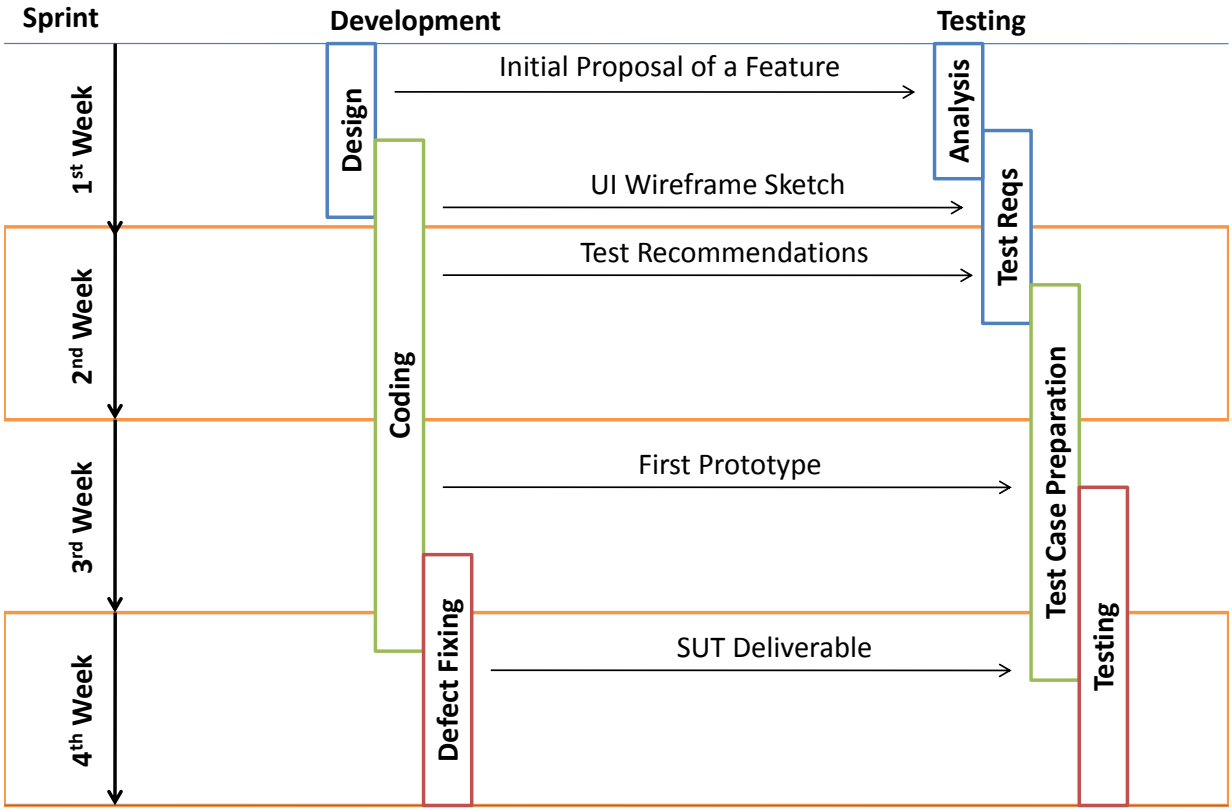

Figure 2.6: A sample model of a collaboration of testing aligned with feature development.

conduct initial tests of the user story. The sooner is the SUT prototype available for test development, the better quality and efficiency of automated can be achieved, and it prepares more space for an actual verification of the SUT feature. In the last week of the sprint, the final SUT deliverable should be ready, and development should also freeze the SUT code with the exception of defect fixes. At this moment, testers run tests over and over again to verify the user story using the defined acceptance criteria and developers focus on fixing of found defects as much as possible. However, in this model, test automation may be difficult or even impossible because testers may face an incomplete user interface design or unstable application.

A typical four–week sprint (Figure 2.6) gives the testing team approximately only two weeks for test automation. However, test scripts based on application prototypes delivered during this period are usually not stable and have to be updated as the new builds are available. As the particular feature is implemented a list of implemented sub–features grows and so it may change a resultant user interface as well a behavior of the SUT. In this volatile environment, it is quite difficult to develop, stabilize and deliver automated

tests exercising the developed SUT features. Testing aligned with feature development is, on the other hand, comfortable for product owners because they get new features with every end of the sprint. Furthermore, resultant deliverables are verified and signed off by the quality assurance team in the same sprint.

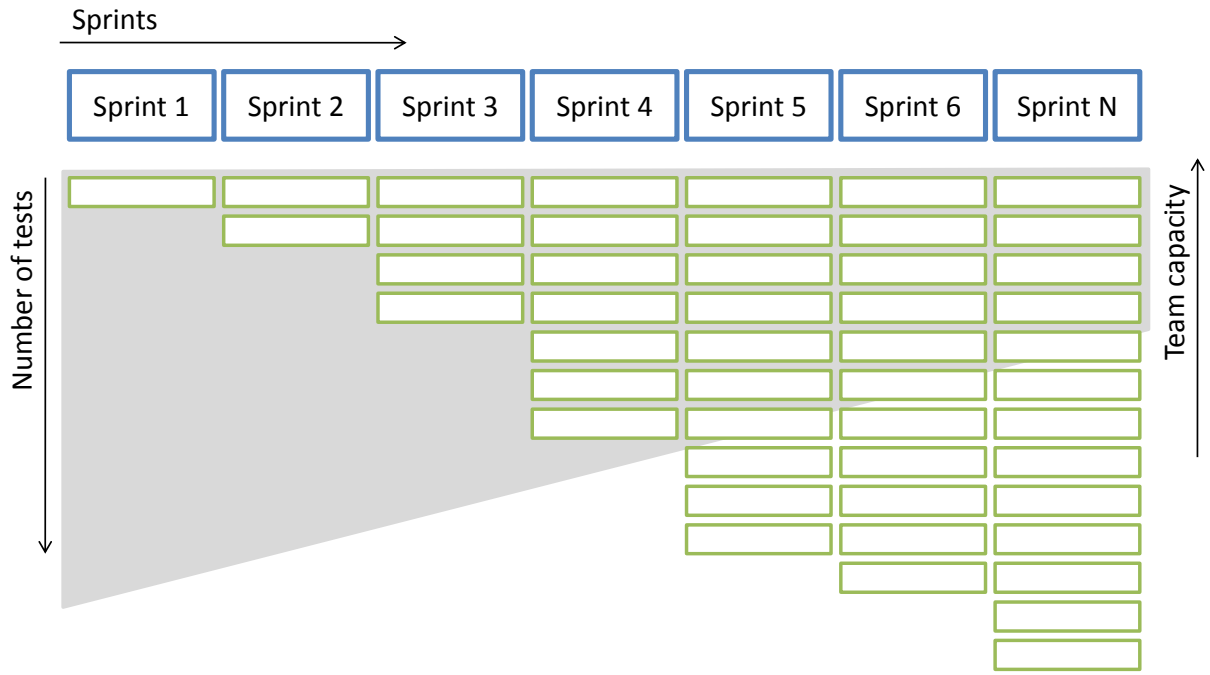

Figure 2.7: An impact of test maintenance on a team capability of testing team to develop new tests in Agile environments.

As development of the product continues and the core team finishes more and more SUT features (based on the user stories) within sprints, testers then have to not only develop new automated tests for features being developed but they have also to start to maintain the current automated tests because the tests may get obsolete due to changes in the SUT. The problem is depicted in Figure 2.7. Green rectangles represent the count of tests that are maintained in the given sprint, and the grey polygon depicts a capability of testing team to develop new tests.

In an initial sprint, the team capability to develop new tests is not impacted by a need to maintain tests since they are up to date and their number is not significant. However, with the growing number of developed automated tests, the team capacity decreases and the need to maintain tests becomes more influent. Hence, the team cannot develop so many new automated tests as it could do in the beginning. An ability to maintain tests

easily is then crucial for successful test automation on projects, regardless if the software development style is Waterfall or Agile based.

2.4 Summary

In this chapter, we introduced a necessary theory and background related to the problem of test automation and testing of software products in different methodologies. We also analyzed issues that arise during a user interface automation in Agile environments.

We also summarized current available frameworks and tools for automation of functional tests. Apart from that, we introduced different generations of test frameworks and explained the differences among these categories.

Finally, we analyzed differences of test automation process in the Waterfall based and Agile based development styles. We arrived at the conclusion, that maintenance of the created automated tests is critical to the success of test automation activity.

This chapter has given the overview of the area from an industrial praxis viewpoint. In the next chapter, we are going to summarize and analyze state of the art from the research and development viewpoint.

Chapter 3

State of the Art

This chapter summarizes and discusses related works in areas relevant to the scope of this doctoral project. Here, we summarize the methods and approaches for test automation, test data generation, automated test case generation, common substring search algorithms on both unstructured and structured data and automated test refactoring.

3.1 Current Approaches to Test Automation

We start the state of the art survey by first summarizing and then providing a deeper analysis of the current mainstream approaches employed in user interface (UI)–based test automation. In addition to conventional approaches to test automation (Record and Replay and Descriptive Programming), other alternatives have been published and employed in individual industry projects—mainly model–based and specification–based approaches.

3.1.1 Conventional Approaches

The current widely used test automation approaches in the software industry can be divided into two principal groups:

1. Record and Replay approaches that record user actions and transform them to a set of UI–based tests using a special tool or framework

2. Descriptive Programming approaches use standard software development techniques to create automated tests.

While test recording is usually suitable for less experienced teams, for ad hoc automation of functional tests in the user interface, or in cases when programming is not economically feasible, descriptive programming is a better fit for more advanced test automation

developers or when the SUT is changing rapidly and requires mature methods to reduce the brittleness of test scripts (e.g., the UI changes with every system release).

Because of the relatively fast speed of automated test creation, the Record and Replay approach can be considered suitable for the Agile development style. Automated test creation was investigated by Meszaros [38], who focused on the experiences of a development team using this method during both test automation and test executions, and concluded that Record and Replay approach is beneficial for automation of regression test suites of legacy systems without any automated tests. He also suggests to develop a support for test recording directly in the SUT if no other tools are available.

A study in the domain of developing robust and helpful automated tests for dynamic web applications using the TestComplete tool [1] was conducted by Al-Zain et al. [39]. The authors introduced a robust solution test script that does not depend on the TestComplete recorder tool and addresses some issues of the dynamic web, i.e., changes in a hierarchy of objects under test and in their attributes.

Another study focused on comparing popular Record and Replay tools and was conducted on both a custom application and on three popular industrial applications downloaded from the Google Play store [40]. They used three popular industrial applications according to downloads from the Google Play store to evaluate those tools. They concluded that none of tools used in the study was convenient for developers to use on real test automation projects.

As an alternative, study by Hammoudi et al. analyzed methods based on test recording [41] and addressed a stability of recorded tests, taxonomized the causes of defects in the subsequent test execution and developed automated techniques to repair them. Moreover, this method presented techniques to prevent test failures from occurring during test runs and developed frameworks for root cause analysis.

Unlike the test recording and playback approaches, the descriptive programming approach, when implemented properly, can reduce test maintenance overhead. One contribution in this area was a method to avoid the overhead of maintaining a TestObject map file with the script [42], allowing test scripts to run independently.

The classical approaches mentioned in this section can be aided by several accelerators or concepts supporting the test automation process. In the case of web applications, crawlers can be used to analyze the SUT front–end user interface structure [43, 44, 45]; this approach can also be used to assess automated testability of the SUT [46] by calculation of indicative metrics [47] as presence of element IDs, attributes, difficult front–end elements and others.

Also, attempts to define a system for assessing the level of structuring of created automated tests in the economic context of the test automation project has been done [48]. To better synchronize the changes in SUT with the automated tests, the system to aggregate

[1] https://smartbear.com/product/testcomplete/overview/

the information about possible change and propagate it to the test maintenance process has also been proposed [49].

3.1.2 Generation of Test Cases and Test Data Used as Test Oracle

In software testing, an ability to determine if a test passed or failed is essential, and proper data for exercising the SUT or expected test results is a key property of a good test. A verification of large and complex systems may benefit from a generation of test data that would be very difficult to prepare manually. In the next paragraphs, we summarize the current state of the art in this area.

Feather [50] discusses a process for an automated generation of test oracles. Authors explain potential issues that may appear when test data are automatically generated and they present it on examples of case studies up to their implementations. They discuss required properties of test oracles if one wants to efficiently verify the system under test. Smaragdakis et al. [51] shows issues of test data generation and lists semantic constrains what test data must fulfill: (i) records in a given table must contain a unique selected values (the cell is a key), (ii) a part of the table can be a subset of another table, and (iii) values in any cell of a record shall be within a defined range.

Data modeling languages such as the Unified Modeling Language (UML) [52]) or Object–Role Modeling Language (ORM), which is a conceptual modeling language focused on database applications, can be used to define presented constraints on test data. Models of SUT are suitable for a generation of large test data volumes. However, this approach has one drawback preventing it from using the method in large scales. Test data generation is a complex problem resulting in a complexity of NP–hard problems (a satisfaction constraint problem), which also applies for significant simplifications of the generation of test data. Furthermore, a complexity of validations of real constraints is typically more difficult and results in more than exponential computational complexity [51]. Sabharwal [31] presented an approach to solving the problem of generated data using a genetic algorithm. [30] employed modified SAT solvers; [51] reduced a complexity of the problem by reducing a subset of data modeling language to solve it in a reasonable time.

A problem of the generation of suitable test data for web applications was discussed by Sabharwal et al. [31]. Their approach is based on UML state diagrams and a genetic algorithm with a variable length of a chromosome that encodes states and transitions. They do not generate standard test data, i.e., inputs, from database tables, but sequences of triggers for the UML state diagram. They consider sequences as test cases.

Smaragdakis et al. [51] based a solution for test data generation on a data modeling language ORM. They concluded the satisfaction constraint problem of an ORM diagram as an NP–hard problem. Moreover, the problem is undecidable for certain formulations of the ORM language. They observed that simple computations based on a brute–force or by using SAT solvers may not bring reasonable results because current SAT solvers cannot

find data that would satisfy defined constraints and a condition of uniqueness in a real time. Moreover, it applies also to small instances of the problem. The presented approach uses a constrained subset of the ORM language; they can decide in a polynomial time whether given constraints are consistent or rather if constraints can be used to generate test data.

Fujiwara et al. [30] utilized a UML class diagram supported by an OCL language (Object Constraint Language). They used the language to specify a behavior of the system under test and data constraints. Authors made an observation of an ability to describe key behaviors of web applications as well as data constraints (constraints on foreign keys) using constraints on a table size. A principle of the method is to transform an OCL specification to an equivalent constraint specification using an expression for the table size. Test data are later on generated from those transformed data by a solver Satisfiability Modulo Theories (SMT). However, the SMT solver does not accept the OCL language; the authors had to propose an iterative process to simplify defined constraints. The proposed data method works with test data that correspond to a state before, and have to satisfy all conditions before and data constraints. Apart from generating test data, they can also generate a set of test cases covering all different behavior of the system under test. Authors carried out experiments and compared time demands on a manual preparation of test data with time spent when their approach is used. Their approach may generally speed up test data preparation and shorten the time needed to up to one–third of a time for conventional approaches, but it requires modeling the system under test in both the UML and the OCL. From reasons, the approach does not have to be suitable for agile software projects, where a model of the system is usually missing and/or not appropriate for the test data generation.

3.1.3 Model–based Approaches

Model–based approaches play a significant role among descriptive programming approaches as it allows to generate tests directly from models of the SUT without a need to actually code the test scripts (or they might reduce the amount of work of the programmers by generation initial parts of the automated tests). Model–based approaches generally bring a higher level of agility and help to test applications in a more scalable and manageable way. This approach has also its drawbacks, which is a high initial investment to set–up the whole process and necessity to update the model to be actual with the SUT.

Peleska et al. introduced an automated model–based test case and data generation that is based on constraint types for real–time systems [53]. In their approach, an expected behavior of the system is represented by a model and generated symbolic test cases represent logical constraints for calculations over the model. During the test case generation process, they exclude invalid test cases to accelerate searching for a solution. However, the presented approaches are limited to real–time systems. Xu presented an Integration and

System Test Automation tool for automated test generation and execution from models [12]. They employ high–level Petri nets as finite state test models. The specification of the system under tests contains the Petri net as well as mapping functions from the Petri net elements to implementation constructs. The resultant source code can be generated in various scripting languages. Koopman et al. describe a model–based testing system for on–the–fly testing of thin–client web applications that are defined by Extended State Machines [54]. The approach is based on a simplification of web applications into a plain HTML, i.e., they do no support flex parts, or Java–applets or AJAX. The system under tests is modeled by states and transition functions specifying reachable states over outputs. Plasmeijer and Achten replaced a plain text, i.e., HTML, by an internal data representation named iData [55, 56], but in this approach, some limitations remain like an impossibility to execute operations for data comparison within web pages or data comparisons with third–party applications.

Beek presented an approach for conformance testing of web applications [57, 58]. The main focus is on black–box testing of a behavior of the system under tests however without a need to conduct a data validation. Besson et al. introduced an interesting approach for automation of testing within Acceptance Test–Driven Development [59], i.e., test case modeling and their execution. They employ two tools: (i) first enables customers to write acceptance tests and unifies test requirements, and (ii) the second tool is a framework for automated testing powered by Selenium. Tests are internally represented as an acyclic graph which they call Test Tree. Every path in the tree represents another test case.

Finite state machines are quite popular for modeling systems under test in test automation. States of the system are represented by states, i.e., nodes, and actions to be taken in the front–end are represented by transitions, i.e., edges. Andrews et al. presented a testing at a system level that can be used for a hierarchical modeling of the system under test [60]. They generate subsequences of states of the finite state machine as test requirements so they natively reduce a state space to states that are reachable by performed actions in the front–end, for example, if the user clicks a button and a dialog appears. Nevertheless, test requirements cannot define requirements on data validation such as correct values, data comparisons or data flow in the systems. Resultants tests can be created by combining subsequences into oriented paths in the graph.

3.1.4 Specification–based Approaches

A natural language is comfortable to humans for expressing and describing objects and it is widely used to define test requirements in test automation as well as to specify the SUT. However, to achieve more exactness of the specification, which is crucial for a smooth run of the project, specific, formally defined and machine–processable languages shall be used. Stepien et al. introduced a web application testing [61] with a TTCN–3 language [62]. A test specification language can be used for a specification of tests with a different level of

abstraction. Using an abstractions takes some advantages: (i) test case development does not need to wait for final deliverables of the system, (ii) tests are robust to changes in the front–end because the abstraction allows overcoming such issues.

However, such an effort may be counterproductive on small projects or for teams who did not a certain level of maturity in test automation as there is a strong need of right skills and expertise to create automated tests from formal specifications. Moreover, specification–based approach adopters need to solve issues with technological stack and overcome inconsistency of heterogeneous environments with AJAX, Adobe Flash or JavaScript. Jia and Liu presented an Extensible Markup Language (XML) specification–based approach of testing of web applications [63]. They specify test cases in XML documents that hide actual implementation details and introduce an abstraction of tests. Those XML files are used as inputs for their testing tool. Object locators are defined by XPath[2] expressions and they use regular expressions for data validation. The testing tool parses XML input files and based on them, it generates test scripts for a JUnit framework.

3.1.5 Other approaches

Niese et al. presented a system for a graphical representation of tests and a complex business logic exercised by these tests [64]. The application behavior is depicted graphically and third–party testing tools are supported by a CORBA/RMI–based communication layer but they an identification of identical objects remains unsolved. Duplications are handled manually by users, the system does not offer any support for them. Benedikt et al. developed a VeriWeb tool for a systematical exploring of paths in web applications (a web crawler) [65]. In contrary to conventional crawlers which are able to explore only static links, the proposed VeriWeb tool can explore user objects in web applications. This approach, unfortunately, reaches its limit, because many frameworks for user interfaces define their own object and a simple navigation based on HTML tags may be impossible.

García presented a complex approach for functional test automation of web applications [35]. The method is based on a validation of the SUT with a browser that carries out transitions from a state to state according to a pre–built navigation model. Each application state represents one step in the model. Three possible inputs are possible: (i) UML models, (ii) XML files, and (iii) recorded scripts. The UML language is in the present time understood as a de–facto standard for modeling of software applications, which leas García to focus on this notation. To automatically process the navigation model, three UML diagrams are necessary: a use case diagram, an activity diagram, and a presentation diagram. Since a presentation diagram is not usually available on commercial software projects (in contrast to using case and activity diagrams) authors decided to use Navigation Development Techniques [66] to model web applications with UML. The second supported format is an XML–based file defining a model of the navigation according to an XSD

[2] https://www.w3schools.com/xml/xpath_intro.asp

schema in the system under test. The website meta–model that supports an event–driven nature of web applications defines then for a given web a complete navigation structure as a collection of states and transitions in the XSD schema. The last option for the input is to use recorded scripts. The authors conclude that the Record and Play approach is more comprehensive in comparison to the models created in UML or XML because the recording is carried out with a real application while other approaches only model a behavior of the system under test. They introduce an enhancement of adding expected results, i.e. a test oracle, in the navigation model so they can use recordings as a data source for a generation of the application model based on the meta–model.

However, a full verification of the SUT behavior, as well as data presented on screens according to the user activity when the user interacts with the SUT controls, requires the testing approach to be more comprehensive than the presented one. Consider an example of a data grid that is presented on a web page or in a mobile application. The user can change an order of columns and also can filter data presented in all columns. He/she wants to verify a sum of values from different columns and rows. Moreover, the sum has to match with a pre–defined value; for example, a sum of three fields should equal to one hundred percent. In our opinion, a generation of scripts for automated tests that cover the required functionality from the recordings may result in NP–hard problems because information on relationships between controls and data may be completely missing. The tester actually checks the table without any interaction as he/she uses only a visual check. García tried to cover the whole process and to automatically build the navigation model [67, 68] of the SUT from the inputs, and to use this navigation model to automatically generate test cases that cover all navigation paths in the model as well as to automatically generate the test data.

A simple solution of the problem of covering edges of the navigation model (based on a graph theory, which is combined with an option to manually provide an additional input and output) may not be sufficient for a test case that requires to validate data in one application with data in another application. Such a case would need to develop mapping functions and additional test logic that simply cannot be captured during the test recording. Moreover, the presented approach does not represent test cases in a higher level of abstraction so the solution is not applicable to different scripting languages for other tools like VBScript or Python. Furthermore, the approach is based on computations performed with application states and transitions between them, objects of the user interface captured by the test recording are not involved in the verification. In consequence, test developers cannot design their tests in an object–oriented way. For example, the test developer needs to verify on–the–fly whether an error warning related to an input box shows an error while the user types an invalid input. Testing tools record only the testers' activity in the SUT front–end user interface but they do not capture responses of the SUT.

Besides using the UML specifications to generate the automated test from the model, usage of the Interaction Flow Modeling Language (IFML) has been also previously explored

[69, 70]. In the context of this thesis, this type of modeling is closer to the user interface based test automation, as the IFML[3] models user interaction and control behavior of the front–end of software applications.

3.2 Code Refactoring and Employment of Reusable Objects

Optimizations of automated test scripts in terms of removing potential duplications and replacing them with reusable objects or reusable functions is a subject of an intensive research. A research focused on a structural and an architectural level brought a number of approaches and frameworks for resolving reusability issues [28, 71, 72, 73]. The latest trend aims at an Object Character Recognition (OCR) approach which seen as an alternative approach [26] that may reduce maintenance costs of the automated tests. Generally spoken, an automation framework or architecture may be usually inspired by such approaches and it means an initial investment to define a proper structure that would support reusable objects or blocks of code. The approach proposed in this thesis brings a new perspective into this problem as it automatically analyzes test recordings and/or test scripts developed by descriptive programming approaches in a naive or suboptimal style.

The problem of reusability has been continuously solved and partially resolved in the top used test automation platforms such as former HP QuickTest Pro (now called Micro Focus Unified Functional Testing) or Test Complete and Selenium WebDriver [74, 75]. Although these platforms address the reusability issue and provide some functionality that optimizes the test recordings it is only focused on atomic reusable objects, i.e., to avoid duplications in terms of object locators, but not on reusable blocks of code, i.e., reusable functions. A direct mechanism for reducing of potential duplications in recorded tests is not available.

A need to reduce potential duplications rises a question of estimating a reusability of test components in automated testing. Kaner proposed a method based on a model of return on investment [22]. They can then estimate a minimum reusability of an automated test component. This work inspired Kan to focus on this problem in more depth to propose a method for estimating the potential reusability of test automation components [76]. Those results inspired us to propose a model of test automation architecture that enables to estimate potential code reuse in a design phase [6]. Moreover, it allows the proposed TestOptimizer to be used for estimates of the potential reusability ratio in the analyzed automated test scripts, however, we did not intend this as a primary use case of the method.

At the unit test refactoring level, an identification of repetitive code is covered much more better [77, 78, 79, 80] than at the level of functional tests for a system under test front–end [81]. A domain of recorded functional tests is covered significantly less than

[3] http://www.ifml.org/

refactoring of the unit tests. In recorded tests, the BlackHorse project [1] is close to our intentions including its goal: to record tests and then optimize them to make them more robust, and it is definitively relevant to our proposal. However, we identified major differences between the BlackHorse project and our approach in TestOptimizer that are in both implementations as well as the general intended use case. Carino et al. developed and employ a proprietary recording tool to save test traces that are later on converted into a Java code in order to minimize potential consequences of changes in the application front–end. Moreover, it serves for a better test robustness that prevents tests from being out–of–date. In contrast, TestOptimizer analyzes already created test scripts, which can be recorded by any tool that can produce scripts in various supported languages, and it searches for potential common subroutines that are offered to test creators as suggestions for test refactoring. In contrast to BlackHorse framework, the TestOptimizer is then naturally platform–agnostic.

In the BlackHorse project, the concept of test traces is conceptually similar to our concept of signatures but goals and implementations of these two concepts are different. The BlackHorse approach is based on using the test traces from test script recordings as an input to create the final test scripts in Java. In our approach, we use the signatures to describe steps of tests and to identify potential common test steps from a business logic point of view. Moreover, we abstract the tests from particular source code notation. These significant use case differences make very difficult to directly compare TestOptimizer with BlackHorse and as a result, we decided not to compare these two frameworks in the experimental verification of the TestOptimizer.

Fang and Lam introduced a method for the identification of test refactoring candidates based on assertion fingerprints [82] encoding a control flow. Their method is relevant to our research intentions as they aim to detect duplications in the test scripts caused by copy and paste, which result in maintenance overhead. They use tailored static analysis techniques to find similar test cases. To identify similar test cases, they determine an ordered set of assertion calls in test methods. However, we identified several points that limit the usage of their approach in comparison to the TestOptimizer framework. The proposed method supports only Java sources and JUnit tests. Furthermore, the authors rely on direct detection of assertions. However, many functional tests are based on some design patterns (for example, a page object pattern), and assertions may be encapsulated. Moreover, their approach cannot detect tests that are similar but do not contain any assertions (for instance, load tests).

Vahabzadeh et al. presented an automated method for eliminating fine–grained redundancies [83] in automated tests. They enable automated test reorganization within test cases at the test step level, and they preserve the test coverage and assertions of the test set. In the TestOptimizer framework, we focus more on the localization of reusable subroutines in tests with variable semantics than on automated refactoring of the test set.

Another method of increasing a reliability of automated tests is introduced by Kumar.

In this alternative method, authors determine if a result of an automated test shall be inspected manually by a human tester based on an analysis of the system front–end [84]. They compare baselines of the SUT front–end and if a difference between baselines is found, they use it for the subsequent analysis.

An approach aiming at increasing quality of the automated test is presented by Chen and Wang [85]. They analyze automated test cases to detect code smells of automated tests and propose refactoring methods that can be applied to remove them from automated tests.

When we refactor automated tests, we need to regression test in order to prevent automated tests from failing in detecting defects in SUTs. Bladel and Demeyer introduced a tool [86] that can help test automation engineers to verify whether a refactored test set preserves its behavior before and after refactoring.

Apart from the test refactoring, we need to pay an intention to a source code refactoring, which is also conceptually similar to the TestOptimizer use case. In this area, there is a number of previous solutions, for instance, [87, 88, 89]. Tools for the source code refactoring are proposed to analyze a general source code, and they do not reflect the specifics of automated test scripts. In the domain of automated tests, two different fragments of the test script source code can carry out practically the same action in the front–end user interface of the SUT. However, in this context, the functionality of tools supporting code refactoring can be compared to the functionality of the proposed TestOptimizer framework. In the experimental evaluation in this thesis, We used the copy–paste–detector (CPD) [90] from the commonly available solutions for a comparison with TestOptimizer. Results of these experiments are in presented in Chapter 8. CPD is part of the PMD Eclipse plug–in project [4] and employs the Karp-Rabin algorithm for string matching.

For searching for common subsequences in two strings, relevant initial algorithms were published already some 40 years ago [91]. An algorithm that solves the problem in quadratic time and linear space was presented by Baker [92]. Pessoa et al. [93] presented an improved technique for a detection of code duplications in a Java–based source code using a dynamic statistical process calibrated by expert's knowledge.

In the problem solved in this thesis, we search for common subroutines in larger sets of strings and that is principally an NP–complete problem [94] in which evolutionary computations techniques such as genetic algorithms gain satisfactory results in a reasonable time [95, 96]. Inspired by these results, we used the genetic algorithm for the longest common subsequences (LCS) problem as the basis of the TestOptimizer Solver component.

3.2.1 Analysis of Textual Information and Text Search Algorithms

Automated tests can be represented by many textual notations, spanning from standard programming languages like Java or Python to other types of descriptive languages, which

[4] PMD plug–in for Eclipse project, http://pmd.sourceforge.net/

are actually proprietary dialects of particular test automation frameworks. In contrast to understanding automated tests as sole scripts written in conventional programming languages, some approaches suppose to work only with simple commands written in plain English (as a successful example, we can give a behavior–driven testing concept implemented in the Cucumber framework[5]). Those formats can be considered as a set of text strings and their purpose is to help product managers or customers to define automated tests without a need to actually know how to programme. Therefore, automated tests can be processed in the same way as text documents and the problem of searching potentially reusable code can be transformed to the problem of a finding of the longest common subsequences of test steps.

In automated testing as well as in manual testing, test cases represent structured data that define what is tested (object under test), how to conduct the test (test steps), and what is expected (expected results). Test automation introduces a higher level of the complexity of the problem. Tests can be represented by various programming languages with a different grammar and even within the same test suite. Locators of the SUT interface elements controlled by the tests as well as user expected results are usually parametrized, and tests are usually chained into cascades. Apart from that, test scripts contain a complementary code, which is not relevant for actual actions performed by the test in the SUT, but it is necessary to run the automated tests. As an example, we can give various try–catch blocks, error handling or test tracing.

In our approach, we consider the automated tests to be analyzed as plain unstructured text. Hence, in this survey, we should not forget to analyze string search algorithms for the plain text. They are subject of research for decades and with some modifications, they can be tailored to be applicable to the longest common subsequences problem, which is understood as a special case of string search algorithm. The Boyer–Moore algorithm [97] is an efficient string search algorithm that is generally considered as a standard benchmark for other string search algorithms. This algorithm employs preprocessed patterns, i.e, strings being searched for, to accelerate the searching. However, the input text itself is not preprocessed. When compared to brute–force searches of all occurrences, the Boyer–Moore algorithm takes an advantage of information gathered during the preprocessing step to skip sections of the text as many as possible. The result is a lower constant factor than many other string search algorithms have. Moreover, the algorithm gives better results compared to other algorithms as the pattern length grows.

The Rabin–Karp algorithm [98] is an example of another string searching algorithm that is based on hashing to find some occurrences from a set of string patterns in the input text. The algorithm gives a try to accelerate tests of pattern equalities in the input text by using a hash function instead of using a sophisticated skip like the Boyer–Moore algorithm. The hash function converts every string into a numeric value and the Rabin–

[5]https://cucumber.io/

Karp algorithm works on a premise that if two strings are equal, their hash values are also equal. Good results of the algorithm rely on the choice of the hash function. Hash values should be tiny otherwise memory demands would be potentially enormous for various texts. It means that some different strings have identical hashes, which is, of course, a violation of the assumption that two strings are equal if their hash values are equal. To preserve the solution consistent, additional tests are necessary to verify that the strings are equal. This can take a long time for long substrings however with a good hash function on most reasonable inputs, this issue does not occurs too often.

The Knuth–Morris–Pratt algorithm [99] improves skip search algorithms. It uses buckets of positions for each character of the alphabet and consists of two phases: the phase is dedicated to preprocess a shift table that is utilized later on in the second searching phase. The algorithm searches for occurrences of a string in the input text. When a mismatch occurs (i.e., when single characters of both strings being processed do not match), the algorithm utilizes observations that the string itself contains enough information to find where the next match could begin with the help of the shift table. A better efficiency is achieved by bypassing re–examination of previously matched characters.

3.2.2 Advanced Search Approaches

In contrast to plain text, structured data contain also elements being used to organize data. These elements or tags do not convey any information but they define a structure of the text document like in, for instance, XML. Usually, it does not make a sense to use the tags without an actual content either in a form of tag attributes or tag values. A document that contains hybrid data is a text supplemented by tags, i.e., it contains both the structured and unstructured data. Although tags may appear irrelevant for an analysis, they are necessary for an advanced text search on hybrid data.

Zhu et al. observed that a text search on hybrid data may result in bad ranks of search results [100]. They demonstrated and proofed on an example with structured data why the text search fails or gives improper results when structured data are not taken into account. They defined cosine as a similarity function and showed that some valid results are discarded because their similarity score for the query text is low. Zhu et al. see a solution to this problem in finding of a method for measuring of a relevance between records and the query text. They remark that taking structured data into account would definitely improve the text search so they re–defined the relevance function and defined additional rules for the structured data being used to filter duplicate results. They conducted experiments and verified that their newly defined relevance function is adequate, and the proposed approach is effective.

The XML format is suitable for a representation of test cases because it allows representing descriptions of user activities in the system under test front–end in a structured way. Apart from standard text search approaches for searching of the longest common

sequences of test steps, there is another option to conduct semantic searches in XML documents. Search engines over these XML documents can be divided into two main categories: (i) information retrieval and (ii) database–oriented. The database–oriented approach [101] is based on a decomposition of XML documents which are stored then in relational databases. A query processing may become very expensive for this method due to an excessive number of joins that are required to recover all the information from fragmented data. In contrast to the database–oriented approach, information retrieval approaches employ other computational techniques like genetic algorithms [102] to overcome the complexity of the problem.

Srinivasa et al. introduced an advanced approach for a mechanism of information retrieval from XML documents [103]. They explored how to retrieve and rank XML fragments based on keyword queries. The authors employ genetic algorithms to learn information about XML tags, i.e., to classify the XML tags as frequently or occasionally used, and this information is used together with a proposed distance metric between keywords among the XML documents to retrieve semantically interconnected fragments of a document. In their viewpoint, XML tags represent data semantics and they can contribute to improving an accuracy of the keyword search. The chromosome keeps two types of information: (i) a set of tag weights, and (ii) a total number of distinct tags appearing in the document. The genetic algorithm uses two indices; the first stores the information for frequently used tags and the second keeps occasionally used tags. Authors also discuss a generalization of the problem and they suggest to build create several indices that are prioritized according to the frequency of their usage.

3.2.3 Evolutionary Computational Algorithms

Evolutionary computational algorithms may when they are appropriately configured, achieve interesting results in a reasonable time even for problems with a non–linear operational time complexity. A research focused in this area on the longest common subsequence problem was conducted by Hinkemeyer et al. They employed a modified genetic algorithm to search for a solution; candidate sequences are encoded as binary strings as long as the shortest of given string [96]. A proposed fitness function penalizes sequences not found in all the strings and the population is initialized conventionally with random genotypes. They conducted experiments with several problem instances and demonstrated that their method always found an optimum solution. Moreover, they showed that it runs in a reasonable time even on large instances. Apart from the experiments with different problem instances, they compared the proposed method with a dynamic programming algorithm, which runs faster on small instances, but they proofed that for larger instances, the overall time required to find an optimum solution by the genetic algorithm is lower the for dynamic programming.

Julstrom and Hinkemeyer observed that evolutionary algorithms may find solutions

good enough quicker when a problem is one of constrained optimization and the initial population is empty [95], i.e., genotypes of an initial population are represented by empty solutions. This premise allows the algorithm to construct valid solutions as much as search for them. Experiments showed that results for the approach based on the initial population with encoded empty sequences are better than results of the population with encoded random sequences. The initial population with encoded empty sequences shortened according to the experiments the time required to identify an optimum subsequence by one third up to four–fifths.

We were inspired by this report in a definition of the proposed method and construction of the TestOptimizer Solver module, but in our concept, we completely changed the representation of the genotype and introduced the concept of signatures as we explain later in Chapter 5.

3.3 Summary

In this chapter, we summarized current related work that is relevant to the scope of our project. This survey covers fields of test automation, generation of test data, automated generation of test cases as well as common substring search algorithms on unstructured as well as structured data and refactoring of automated tests.

We discussed the advantages and potential issues of analyzed approaches and methods. We also discussed the reasons that led us to propose the new approach solving a problem of optimization of automated test scripts.

Chapter 4

Introduction to the Proposed Approach

In this chapter, we introduce the proposed method from a conceptual viewpoint and provide a broad overview of the TestOptimizer framework and its functionality. The goal is to give the reader an overall grasp of the entire concept before discussing particular automated test models, algorithms to search for potential common subroutines and other technical details of the framework.

4.1 Principle of the Proposed Method

To solve the problem of reducing potentially duplicate fragments in a set of automated test scripts, we were inspired by concepts in test automation that are generally understood as best practices in the field. Generally, these use an abstraction layer to reduce test brittleness and adopt an automation framework concept to hide the internal complexity of the method from end users. Moreover, our goal was to create an open, scalable, platform–agnostic and tool–agnostic design as well as to choose approaches that would not require developing a new recording tool or cause vendor or specific–solution lock–in issues. Hence, we propose the following concept.

We first explain the basic principles of the TestOptimizer framework and briefly introduce the novel parts of our approach. The proposed method provides a computer–aided optimization of recorded and/or suboptimally structured automated tests to the testers or test automation developers. The framework automatically analyzes a set of test scripts. Then, based on this analysis, it suggests the potentially reusable subroutines that can be used in a subsequent refactoring process. At this point, we leave the final decision concerning whether a particular identified subroutine is reusable up to the developer because the desired optimization level depends on many factors, such as how the test suite should be structured or user preferences. For example, users may prefer only longer reusable subroutines for a common functionality to a large number of shorter subroutines. After the automated analysis, we create a list of available subroutines and specify their positions in

the test source code. Based on this list, the user can determine the reusable objects or refactor the test script.

During the identification of repetitive fragments, we consider fragments of the test scripts that have the same semantics regarding the actions performed in the SUT user interface. In contrast, if we were to consider only physically identical source code fragments, our approach would identify only trivial cases of source code redundancy. To better grasp this issue, consider an example: one method that takes one parameter. This method occurs twice in the code but both times with a different value. Using only a common substring search, these methods would not be considered identical. Although this example is trivial, many more complex cases of this test redundancy exist; these are discussed in more detail in Section 5.1.

Given this approach, we can objectively assert that our semantic analysis still yields only approximate results: it cannot detect all possible refactoring opportunities as human users can. However, using this approach, we can achieve higher efficiency than is possible through simple source code–fragment comparisons. More detail is presented in Chapter 5.

In the proposed approach, we do not differentiate between complete reusable routines and partially reusable routines. This flexibility is maintained intentionally. Based on the suggestions of the potentially reusable parts of the analyzed tests, test developers can decide what structure of the subroutine fit their needs. They can manually exclude certain steps from the proposed subroutine or add new steps if suitable, for example, add assertions to verify data or the application state.

The proposed TestOptimizer framework can be practically applied with any testing tool or a test automation language because it only analyzes a resultant source code from automated tests. No code dependencies are imposed by using this framework. If a particular testing tool or test management tool does not allow to integrate directly with the TestOptimizer by available API, we can still integrate at the level of source code via version control systems. Test scripts can be written in any supported programming languages such as Java for Selenium WebDriver [1] or in a domain specific language for a particular tool, for instance, Selenese [2]. The current version of the TestOptimizer supports Java, Selenium WebDriver for automation of tests for web–based SUTs and Appium for automation of tests for mobile applications. Support for other languages or test automation APIs can be easily created by adding a new **Converter** component to the framework modular structure.

Figure 4.1 depicts main principles of this method. In the beginning, test scripts are submitted to the TestOptimizer server (step 1). In step 2, the source code of the automated tests is converted into an abstract layer which captures the semantics of the individual steps of the converted tests. Then the analysis of repetitive parts is carried out over the abstraction of the automated tests (steps 3 and 4). In the last step (5), a set of identified

[1] http://www.seleniumhq.org/projects/webdriver/

[2] http://www.seleniumhq.org/docs/02_selenium_ide.jsp

potential reusable subroutines is prepared and offered to the user. The user can utilize provided information in the refactoring of the automated test scripts. The parameters for the analysis rules are saved in a configuration file.

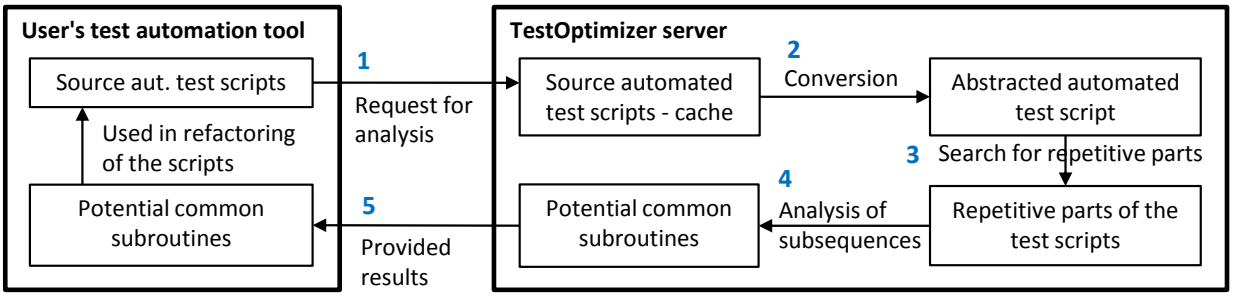

Figure 4.1: The conceptual schema of the proposed method [A.1].

4.2 Summary

The principle of the proposed method is to analyze input automated test scripts and to transform them into abstracted, particular language–independent shape. Then we process these abstracted scripts in several iterations to find potentially reusable components. We use newly proposed algorithms to filter raw data and to exclude inaccurate subroutines from a final set of test script refactoring suggestions offered to the user.

In the proposed approach, we do not directly conduct automatic refactoring of the automated test scripts. This responsibility is left on test developers because a decision on the optimal reusable part of the code cannot be completely left to an automated algorithm. Optimal structure and size of test suites and reusable objects may vary from project to project and many factors might play role in such a decision.

In the described features, the proposed method differs from previous work done in this area [1], as discussed in the Chapter 3.

In the following chapters, we describe the entire process in more detail. Following the process steps outlined in Fig 4.1, Chapter 5 describes step 2, Chapter 6 describe steps 3 and 4, respectively, and Chapter 7 describes the TestOptimizer server interface, corresponding to steps 1 and 5.

Chapter 5

Abstract Model of Analyzed Test Code

In the following sections, we introduce the concept of test step signatures which are used in an abstract model of analyzed automated test scripts. The abstract model enables searching for relevant common parts of the tests rather than searching for code subsequences based solely on string similarity. Through this abstraction, the method is independent of specific implementation details in diverse test scripts that use various coding styles, test automation languages or particular test automation APIs.

In the further text of this thesis, we refer to this ***abstract model*** *as to the "****set of abstracted test scripts****". The set of abstracted test scripts is formally defined in Section 6.2 using the concepts defined and explained in this Section.*

5.1 Concept of Abstract Signatures

The advantages of introducing abstraction into the automated test script analysis process and converting the analyzed scripts into abstracted data before conducting further analyses are as follows:

- the abstraction ensures the method's platform–independence;
- during automated test analysis, this approach reflects the true code semantics; hence, the analysis is not limited to the source code similarity level;
- the abstraction hides differences caused by different notations and coding styles in the test scripts; therefore, it enables analysis of automated test scripts created in different programming styles or in situations when parts of the scripts are coded in a domain–specific language of an automation framework.

The proposed approach offers all of the above advantages. However, it also preserves the ability to reuse existing algorithms when searching for longest common subsequences, which is a text–based operation.

Analysis of the code of automated tests performed by the TestOptimizer framework is based on a modified genetic algorithm that was previously explored for the LCS problem [95].

In case of automated tests, analysis of the real semantics of their steps is important to make a proper decision which parts of the code can be identified as a reusable fragment. Therefore, we introduce a concept of test step **signatures**. In this concept, each step of the analyzed test script is converted to a signature. These signatures hide specifics of programming languages, unifies their syntax a reduces differences in their grammars. As a result, the signatures enable to compare steps of test cases at the semantic level. Sequences of identical signatures promise potentially common subsequences of test scripts. We use this property to find repetitive parts in the analyzed test code with higher effectiveness and relevance than a pure comparison of the source code fragments might provide.

Let's illustrate the problem on an example before describing the problem formally. Automated tests consist of sequences of steps, typically expressed by commands of the programming language in which the test is created (or recorded). Those commands usually represent actions performed by the automated test in the user interface of the SUT, but not exclusively. Some commands may represent an assertion of the data displayed in the SUT interface, or other auxiliary commands, needed for the run of the automated test. The problem is, that syntax of high–level programming languages is richer in comparison to low–level languages, so one particular action in the user interface can be expressed by several different notations in the source code of the automated test. To be able to process such tests implemented in high–level languages, we need to make an abstraction of the actions actually performed by these tests, instead of parsing the source code. As an example, listing 5.1 presents two possible implementations of one identical test step in Java and Selenium WebDriver.

```
//case 1
driver.findElement(By.id("login")).click();

//case 2
Element e = driver.findElement(By.id("login"));
e.click();
```

Listing 5.1: An example of differences in the syntax of two different test steps that have the same semantics (API objects appear in bold, methods in italics, and user–defined parameters are underlined)

In general, the automated test code that performs particular actions in the SUT user interface is mainly composed of test steps that consist of:

- **test** automation API **objects**,
- **methods** of these objects, defined by these API, for instance, click on an element in the user interface, and
- **parameters** of these methods, such as identification of user interface elements that are handled by the script, or testing data to be entered to the SUT via the user interface by an automated test.

In practice, it is definitely a not rare situation that two test steps performing the identical action in the SUT user interface have different specific notations in the actual source code, even in the same programming language and in the same test automation API. We solve this issue by translation of particular test steps into their signatures. A **signature** s is defined as follows:

$$s =< o, X_o, a, X_a >$$

where o is an object, a is an action, X_o is a set of additional attribute parameters to identify object o in the SUT user interface and X_a is a set of parameters that specify testing data used by the action a.

Object $o =< c, n >$ represents an abstraction of a user interface element, e.g., an input box or a button. This abstraction does not depend on the physical implementation of this element in HTML or another markup language for a web–based user interface of the SUT.

An object o is composed of a pair: a **class**, c, and a **name**, n. The class defines the type of object (for example a radio button or input text box), while the name acts as a unique identifier for a given instance of the object and it also links the object to its physical representation in the SUT user interface.

The object o represents an atomic element of the SUT user interface that can be accessed by the automated test (technically, which is controlled by the test automation tool or API during the test). In the real applications, the object name n is not sufficient to identify the user interface element in some cases uniquely (for instance, several elements of the SUT user interface can have the same IDs, despite the fact, that this can be considered as a flaw in the user interface programming, considered as an anti–pattern making test automation more difficult). To overcome such a situation, other attributes from X_o can be used for its identification.

An **attribute parameter**, $x \in X_o$, is a pair of an attribute name and its value.

An **action**, a, is executed on an object o in the user interface of the SUT (as an example, we can give a navigation from one screen to another one by clicking a button). A is the list of all possible actions based on possible capabilities of a particular test automation API.

Further, S is a set of all signatures derived from the set of all analyzed automated test scripts. The signature has to be represented in a text form for a further processing since the search for potential common subsequences in the test scripts is based on text data. Hence, for further processing, we represent the signature as a text string, defined by the following regular expression:

```
obj:<class>{.<object_name>{+<attribute_parameter>=<value>}}&
act:<action>{+<parameter>:<value>}
```

The individual elements of text representation of the signature are explained in Table 5.1:

Table 5.1: Elements of a test step signature.

Element	**Description**
obj	Keyword
class	Class c of the object o
object_name	Name n of the object o
attribute_parameter	Name of attribute parameter x. The attribute parameters are used for a specification of a physical representation of an object o if the object_name is not sufficient to localize uniquely the element in the front–end.
act	Keyword
action	Action a identifier
parameter	It specifies what data are used when the action is performed
value	General parameter value type of String

Our proposal employs three types of signatures with different levels of detail that are listed in Table 5.2.

Table 5.2: Levels of test step signatures.

Signature Level	**Signature**
0	**obj:**<class>&**act:**<action>
1	**obj:**<class>.<object_name>{+<attribute_parameter>=<value>}&**act:**<action>
2	**obj:**<class>.<object_name>{+<attribute_parameter>=<value>}&**act:**<action>{+<parameter>:<value>}

Although the signatures seem to be similar to a real code, using signatures takes three significant benefits. Firstly, the signatures allow us to isolate the TestOptimizer core algorithms from the specifics and complexity of particular possible notations used in the source code of the automated test scripts – we are able to analyze different parts of the source code that actually means the same action performed by the automated test in the SUT user interface. Secondly, TestOptimizer can be applied to multiple languages like Python, Java or C# and different notations in the automated tests. The isolation creates conditions for a modular extension of the TestOptimizer framework without the need to make any changes to the core algorithms. Finally, defined signatures support a flexibility in a level of detail to configure which parts of the automated test scripts are relevant for the analysis, and which can be neglected during this process.

The signature levels defined in Table 5.1 have the following practical applicability in the search for potentially reusable parts in the code of automated tests:

- The **Level 0** signature describes only the objects of the SUT user interface and actions performed on these objects. If the Level 0 is used for an analysis, we search for groups of potentially similar actions performed on objects of the same classes (i.e., practically identical types of the objects). The analysis at this level does not depend on any particular elements of the SUT user interface and it is suitable for a high–level search for potential candidates for common reusable objects and procedures.

- The **Level 1** signature defines particular physical elements of SUT user interface and actions performed with these elements by an automated test. However, the parameter values of actions are not reflected by the analysis. At this level, we can identify repeating common subroutines that carry out a certain action on a particular element of the SUT user interface.

- The **Level 2** signature analysis is the most detailed. It takes the parameter values of performed actions into account. To achieve more accurate comparison when searching for repeated common subsequences, instead of the original parameters that have a variable length and may contain special characters, we use their hashed values that replace long data strings in the signatures. We convert only action parameters but not object locators.

Domain–specific languages of various test automation frameworks of different test automation APIs do not allow to represent single tests steps by signatures of all levels defined in Table 5.1. For instance, the original Micro Focus Unified Functional Testing framework is based on Visual Basic Script Edition (VBScript) language; not just its syntax but also its capabilities enable to use signatures for all three levels, i.e., 0–2. However, automated tests developed in Selenium WebDriver can be expressed only using the levels 1-–2. The limitation here is not the programming language, (mainly Java in case of Selenium

WebDriver), but capabilities of the Selenium WebDriver API, which does not support the object repository concept.

The principal difference between the signatures of type Level 1 and 2 is in the manner how the parameters of individual actions are handled. We considered an option to distinguish between identical steps of the automated tests that differ only by parameters of particular actions as very useful as they may represent the different semantics of an actual test step. There are use cases where developers want to preserve the observed differences in the automated tests, e.g., in role–based tests, such test fragments with identical steps are then complemented by verification steps after creating the tests to check a particular behavior of the SUT. Test automation developers can then use Level 2 signatures for those cases. When the action parameters are not important, Level 1 signatures can be used for the analysis instead.

Table 5.3 compares examples of these levels using sample commands of Micro Focus UFT and Selenium WebDriver. A sample command for Micro Focus UFT in VBScript is, for example:

```
WebField("username").Set "john"
```

This is a construction of UFT VBScript working the object repository that uses object location properties stored there. The value `username` servers as a unique identificator of the object and may not be related to any particular value in the DOM of the SUT. An equivalent command in Java for Selenium WebDriver is then:

```
driver.findElement(By.id("username")).set("john");
```

Data that we process and extract from the commands of the automated test scripts at a particular signature level are presented (as underlined bold text) in Table 5.3.

Now we explain the process of conversion of the test steps of the particular test automation language to a set of signatures. An example of the conversion is depicted in the Figure 5.1. We proposed a converter that uses a lexical and semantic analysis of the source code to convert the original tests into an abstract test $t = (s_1, s_2, \ldots, s_n)$. In the lexical analysis, we tokenize the input code and classify every token as one type from a group of numbers, keywords, special characters, etc. The processed tokens are subsequently used in the semantic analysis of the test code.

Since our intention was to have a full control over the whole process of conversion, we did not employ any existing parser for Java language and we implemented our semantic analyzer based on the reference grammar defined by Oracle [1]. The semantic analyzer parses the tokens from the lexical analysis and creates a syntax tree of the code being analyzed. The resultant syntax tree is used then to translate source code statements to defined test step signatures. In the pilot implementation of the TestOptimizer framework,

[1] https://docs.oracle.com/javase/specs/jls/se7/html/jls-2.html

Table 5.3: Examples of information extracted from automated test commands for signature levels 0, 1 and 2 in different frameworks.

Signature Level	**Information extracted from Micro Focus UFT command**	**Information extracted from Selenium WebDriver command**
0	**WebField**("username") .**Set** "john"	not relevant
1	**WebField**("**username**") .**Set** "john"	driver.**findElement**(By.id("**username**")) .**set**("john");
2	**WebField**("**username**") .**Set** "**john**"	driver.**findElement**(By.id("**username**")) .**set**("**john**");

we implemented and debugged the model parser in the method with Java for Selenium WebDriver and the JUnit[2]/TestNG[3] frameworks. In the final version of prototype we verified that Selenium WebDriver commands for Appium are also correctly parsed.

As not all steps are interesting and relevant for the potential optimization of automated tests we classified these steps as:

- **convertible** test steps, which represent test actions in the front–end user interface of the SUT; they are translated to test step signatures, and
- **non–convertible** code, which is not relevant from a test semantics point of view, hence we do not translate it to the signatures.

The convertible test steps represent actions performed in the SUT front–end user interface that change the state of the SUT and assertions of the expected results. In Table 5.4 we give some examples of convertible and non–convertible steps. Examples in the first two rows are considered as convertible steps because WebDriver navigates to a particular page of the SUT front–end user interface in the first row, and then in the second row, the user checks if a given element is present in the front–end under test. However, in the third row, the step causes WebDriver to shut down, which does not actually represent a change of the system under tests state. JUnit or TestNG test frameworks annotations other than `@Test` such as `@BeforeSuite`, `@AfterClass`, and `@BeforeMethod` typically do not involve operations in the SUT user interface as they serve as common methods for test setup, data preparation or clean up after the test execution. Therefore, they are not part of the analysis. Moreover, commands in such annotated methods were already identified by developers of the automated tests as a reusable component.

[2]https://junit.org/junit5/
[3]https://testng.org/doc/index.html

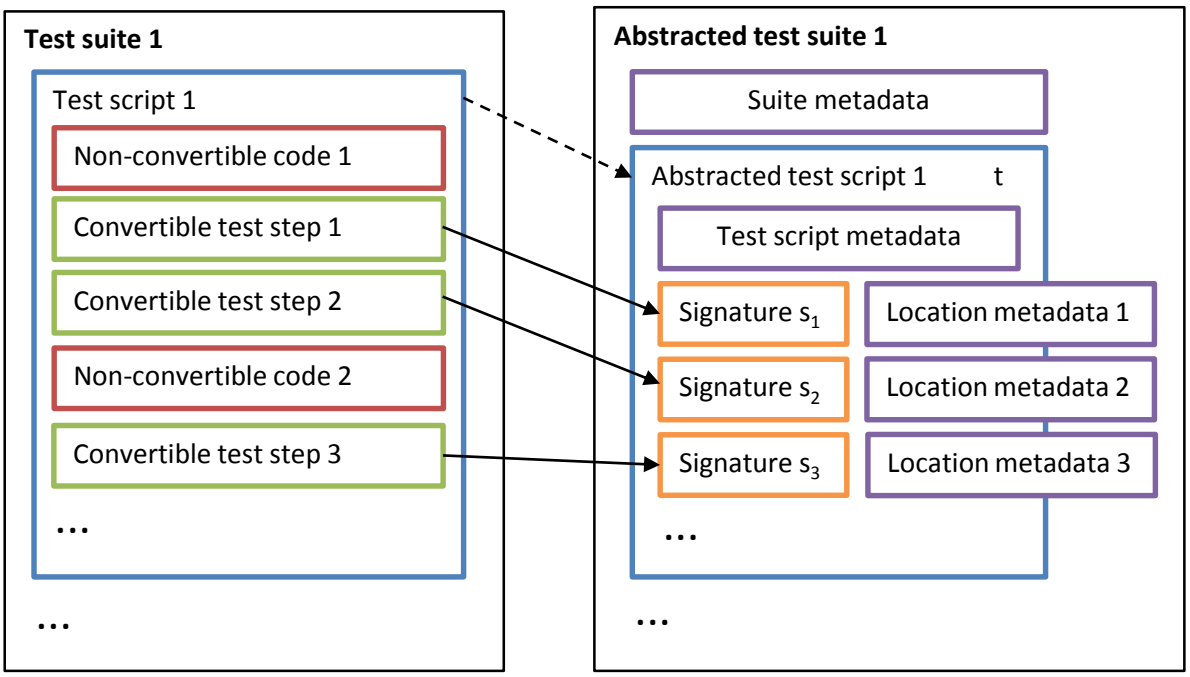

Figure 5.1: A conversion of source automated test scripts to their abstractions [A.1].

Table 5.4: Examples of convertible and non–convertible steps in an automated test scripts developed in Selenium WebDriver and in the JUnit format.

Type of Step	**Example of Source Code**
convertible	An action in SUT front–end `driver.get(baseUrl + "/ui/index.html");`
convertible	Assertion of results `assertTrue(isElementPresent(By.id("empTab")));`

not convertible	JUnit framework code except `@Test`
	`@After`
	`public void tearDown() throws Exception {`
	`  driver.quit();`
	`  ...`
	`}`

For the analysis, we defined and stored parsing rules in the converter configuration. The converter then makes decisions based on these rules whether the step is convertible or it is not. Since it is important to trace the transformation of original test script steps to the signatures, we use location metadata (see Figure 5.1) for this task. Location metadata describe the respective code line–number range of the analyzed test as well as the original code fragment. We do not process the metadata during the analysis, they are only used for debugging purposes. Apart from that, every abstracted test script contains a complementary set of metadata (refer to Figure 5.1) and that includes the source filename and the count of signatures. The analyzed test suite (refer to the suite metadata depicted in Figure 5.1) contains then both the source code and a count of the analyzed test scripts.

5.2 Summary

In this chapter, we introduced the concept of signatures used in the analysis of automated tests with the goal to identify potential repetitive parts in the code. The abstraction achieved by the test step signatures brings a universal approach that does not depend on any particular platform, and it helps to focus on the core of the problem. Besides that, it allows us to search for the potentially repetitive parts of the tests with higher relevance – by this approach, we are able to find the code fragment doing actually the same actions in the SUT user interface, despite the fact, that particular notations of these actions might differ in the source code.

The analysis can be performed at three different levels of details, allowing better flexibility of the analysis in the particular cases. We classify test steps as convertible or non–convertible if they are not relevant for the analysis.

In the next chapter, we present details of the algorithms that are used to identify potential common subroutines in the textual representation of the created test step signatures. The algorithms are used in the TestOptimizer framework, which is a pilot practical implementation of the method proposed in this thesis.

Chapter 6

Algorithms to Solve the Problem

The chapter describes the algorithms used in the proposed method. The chapter starts by defining the inputs and outputs of the main algorithm; then, it presents the proposed metrics used to evaluate the method quality, and finally, it defines all the algorithms that compose two main algorithms to solve the problem.

6.1 Algorithm Overview

The proposed method employs several algorithms that process abstracted automated test scripts. The whole process starts by selecting prospective tests to be analyzed (details are described in Section 6.3). In a set of these test scripts, search for the longest common subsequences is performed (its strategy is described in Section 6.4).

The next step is finding of common test steps in the analyzed abstracted test scripts (the Algorithm 6.1 described in Section 6.5).

When the individual common steps of the automated tests are identified, we analyze them using the Algorithm 6.2 for finding particular common subroutines described in Section 6.6. This part employs two auxiliary algorithms to find candidate subroutines – the Algorithm 6.3 and Algorithm 6.4 to filter the identified candidate subroutines.

The Section 6.7 of this chapter discusses the available modes in which the TestOptimizer framework can be operated. An evolutionary computational technique used in our approach allows the user to choose between a manual mode (Algorithm 6.5 described in Section 6.7.1), when the user is interested to analyze if a particular subroutine occurs in the rest of the test automation code, or an automatic mode (Algorithm 6.6 described in Section 6.7.2), when TestOptimizer analyzes general occurrences of potential common subroutines. In both cases, users set criteria that determine what they prefer, i.e., what the quality of any resulting reusable routines should be. For example, some users prefer less usage but longer routines, while other may prefer shorter routines with many occurrences.

6.2 Input and Output

The **input** of the analysis is a set of **abstracted test scripts**, $T_A = t_1, t_2, \ldots, t_n$. Each of tests is composed of an ordered sequence of signatures: $t_x = (s_1, s_2, \ldots, s_n), s_1, s_2, \ldots, s_n \in S, t_x \in T_A$.

We denote a **potential common subroutine** as $p = (s_1, s_2, \ldots, s_m), s_1, s_2, \ldots, s_m \in S$, where p must be present in two or more abstracted test scripts from T_A. Then, T_F is a set of analyzed abstracted test scripts in which p is present, $T_F \subseteq T_A$.

The **output** of the analysis is a set of potential common subroutines, P. In general, more potential common subroutines can exist in T_A.

6.3 Selection of Prospective Tests to Analyze

When the analyzed automated test scripts are converted to the textual representation of sets of tests signatures, the next step in the process is to identify the longest common subsequences in T_A. Because we are processing the text strings, we can reuse already defined algorithms for the problem of finding LCS, which generally represents an NP–complete problem. In the proposed method, we employed the genetic algorithm that was suggested for the LCS problem previously by [95, 96]. As we mentioned in the previous paragraph, metadata of an abstracted test script are not taken into account during the analysis based on the LCS search.

To optimize the input set T_A prior to run of the LCS search genetic algorithm, we reduce the T_A set by excluding test scripts that do not have at least one common signature with the other test scripts in T_A. This optimization contributes to a better performance of the genetic algorithm as it does not search reusable test fragments in disjunct tests. For this reduction, we implemented the function SELECT_PROSPECTIVE_TESTS, where T_P denotes a set of prospective test scripts to analyze, $T_P \subseteq T_A$, and $T_P :=$ SELECT_PROSPECTIVE_TESTS(T_A). The function is based on an adopted Karp–Rabin string matching algorithm [98]. The SELECT_PROSPECTIVE_TESTS function is designed to search for simple duplications in signatures among the set of analyzed abstracted test scripts, T_A. An abstracted test script, $t_n \in T_A$, is excluded from T_P when $\forall s_n \in t_n : \forall t_o \in T_A / \{t_n\} : s_n \notin t_o$.

6.4 LCS Search Algorithm

Once the input set T_A is reduced to T_P, we start the LCS search on T_P. The chromosome of the genetic algorithm is denoted as $t_c \in T_P$. The analysis runs in a pre–defined number of iterations. In every iteration, the solver evaluates potential variants of results using the proposed fitness function to select the best current population as the set of candidate sequences. Moreover, the solver applies genetic operation like mutations and cross–selection when it evolves the population. We experimented with several strategies trying to find an optimal configuration of the fitness function for the LCS of test steps. In this effort, we focused on a proposal of the fitness function that will prefer the longest possible sequences and the sequences that occur most often. The overall results of the analysis depend on a selection of the chromosome t_c (a major factor) and also on the selection of control parameters (a minor factor) for the LCS algorithm. These parameters are listed in more detail in Table 7.1.

In the further explanation of the algorithms let's assume given chromosome t_c. More details to the process by which the chromosome is determined are explained later in Section 6.7. The LCS search algorithm we use is a modified genetic algorithm based on [95, 96] with an empty initial population. Computations start from scratch without any random initialization and we utilize an elitism to preserve promising individuals. To evolve the population, we use genetic operators for a mutation and crossover as well as a selection of the best individuals to the elite population. We apply genetic operators based on a roulette wheel selection.

6.4.1 Metrics Used to Evaluate the Quality of the Found Method

The optimization of the automated tests focused on reusable components rises a question what is more interesting for test developers from a refactoring point of view. Is it a few long common subroutines or a large set of shorter common subroutines? Instead of making a conclusion which option to prefer we left those decisions on the end user, because the coding style of automated tests and their structure may depend on many factors as an intended maintenance or a strategy of test suite design.

The user may be then interested in suggestions at different levels, and that is why we introduced set of parameters that can be used to express the user's preferences and metrics to select the best result for the potential common subroutines in the set of analyzed scripts, i.e.,: Subroutine Quality (SQ), defined by Eq. (1), and Analysis Variant Quality (AVQ), defined by Eq. (2).

$$SQ(p, T_a) = |p| \cdot |T_p| \tag{1}$$

T_F is a set of analyzed abstracted test scripts in which p is present, and $T_F \subseteq T_A$.

$$AVQ(P, T_A) = \Sigma_{p \in P} SQ(p, T_A) \tag{2}$$

The user can control the analysis by adjusting $lengthWeight$ and $testCountWeight$ parameters, which are real number constants whose values are set before the analysis of the automated test set is initiated. The user can determine the best values for both parameters experimentally by running several iterations of the analysis with different values.

If the user increases the $lengthWeight$ parameter then longer potential common subroutines are preferred during the analysis but they tend to occur less often in analyzed automated test scripts. If the $testCountWeight$ parameter is increased then potentially shorter common subroutines are preferred during the identification of subroutines, and these cases occur more often in analyzed automated test scripts.

We defined that:

$lengthWeight + testCountWeight = 1$, $lengthWeight > 0$ and $testCountWeight > 0$.

6.4.2 Fitness Function Used in the LCS Algorithm

We use the $lengthWeight$ and $testCountWeight$ parameters in several parts of the analysis. First of all, we use these parameters in the proposal of the fitness function for the LCS search performed by the genetic algorithm. This configuration is presented in Listing 6.1.

```
fitness = |p| * lengthWeight + |T_F | * testCountWeight;
if (|p| == |t_c|) then {
    fitness = fitness * 10;
}
if (|T_F| == |T_P|) then {
    fitness = fitness * 5;
}
```

Listing 6.1: The definition of the proposed fitness function for Longest Common Subsequences (LCS) search

At the end of the analysis, if $P = \emptyset$, it may mean that:

1. either there are no common steps in the analyzed abstracted test scripts T_A,

2. or the user set inappropriate values for the configuration of one or more input parameters.

Consequently, the solver has a natural tendency to find a local optimum in a single test. If this case occurs we recommended the user to restart the analysis with a different configuration of the control parameters.

6.5 Search for Common Test Steps

The chromosome t_c is encoded as a binary string by the following method. If a given signature that represents a test step of chromosome t_c is present in at least in one abstracted test scripts $T_F, T_F \subseteq T_P, t_c \notin T_F$, it is encoded by 1 in the binary string. If the signature does not have any multiple occurrences in any of abstracted test scripts T_F then the signature is encoded by 0. Other tests in the analyzed test set are represented by simple arrays that contain references to test step signatures. Since Julstrom [95] proved on an example of unstructured text that genetic algorithm achieves better results when it starts from scratch, i.e., the initial population is empty (encoded by zeros) and it is not initialized, we did not adopt an approach to set up the population at the beginning.

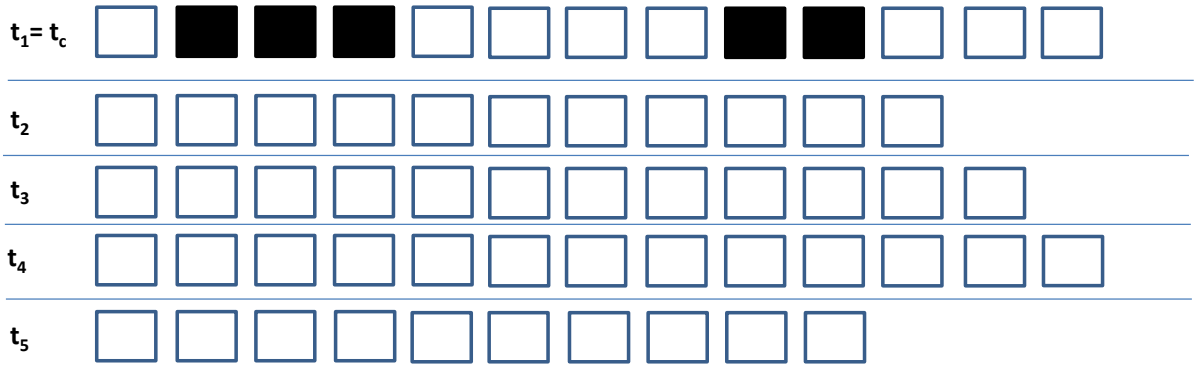

Figure 6.1: A sample output of the LCS algorithm: potential common steps for the chromosome t_c [A.1].

Figure 6.1 illustrates an example with $T_P = \{t_1, \ldots, t_5\}$, $t_1 = t_c$ is the chromosome and T_F has not yet been computed. Potential common steps from t_c will be analyzed with an objective to find potential common subroutines, and are depicted by black rectangles. Steps that are not part of the set of potential common steps are depicted by white rectangles. After identifying the potential common steps with the chromosome t_c, we identify these

steps in $T_P \setminus \{tc\}$ using the Algorithm 6.1. The asymptotic complexity of the algorithm is determined by the count of found signatures in the chromosome, the size of the analyzed set and by the number of signatures in a test from the analyzed set. It yields $O(|T_P||t_c||t_{max}|)$, where t_{max} is an abstracted test script with the highest number of signatures, $t_{max} \in T_P$.

Algorithm 6.1 FIND_COMMON_STEPS
Searching of the potential common steps in the abstracted test scripts based on the common sequence in the chromosome.

Require: $t_c \neq empty$ and $T_P \neq empty$
1: **for all** s_c in t_c **do** ▷ for each signature from the chromosome
2: **for all** t in $(T_P \setminus \{t_c\})$ **do** ▷ for each abstracted test script in the analyzed set, excluding the chromosome
3: **for all** s in t **do** ▷ for each signature from the analyzed abstracted test script
4: **if** $sc = s$ **then**
5: s.common(sc)
6: **end if**
7: **end for**
8: **end for**
9: **end for**
10: **return** T_P

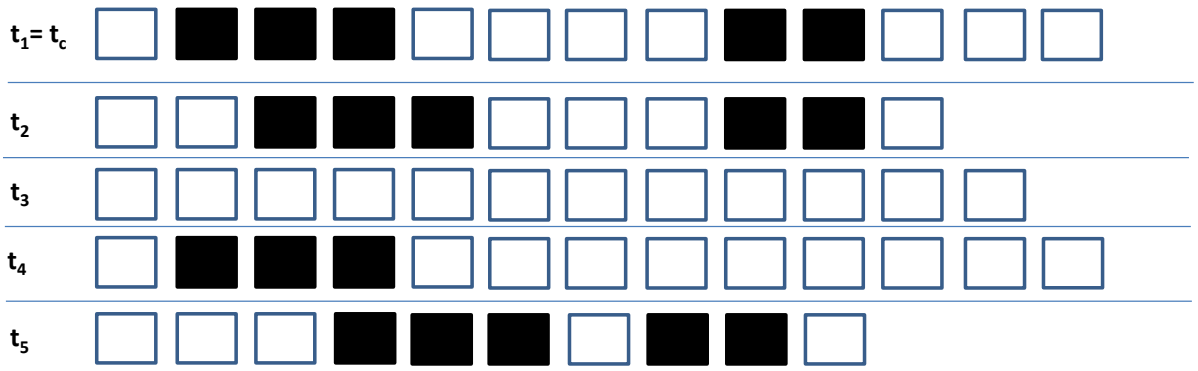

Figure 6.2: The potential common steps identified in the set of abstracted test scripts T_P [A.1].

A sample output of Algorithm 6.1 is presented in Figure 6.2: potential common steps are localized in the set of analyzed abstracted test scripts. The results promise that a set of abstracted test scripts $\{t_1, t_2, t_4, t_5\}$ may contain potential common subroutines as

abstracted test scripts t_1, t_2, t_4 and t_5 are expected to potentially contain two subroutines: one three–step subroutine and one two–step subroutine. The abstracted test script t_4 is only expected to contain a three–step subroutine. The next step is the identification of the common subroutines based on the localized potential common steps.

6.6 Search for Potential Common Subroutines

As the subsequent step, the Algorithm 6.2 finds a set of potential common subroutines P in the set of prospective tests T_P. This algorithm takes one parameter $tequiredMinLength$ on input that specifies a minimal length of potential common subroutines (minimal number of test step signatures) that will be identified. During the computation, we use an array *occurrence* to keep information about the number of abstracted test scripts from T_P in which a particular signature occurs, and a temporary *max_occurrence* variable that contains the largest number of abstracted test scripts in which some of the signatures occurred during the processing so far.

Asymptotic complexity of this algorithm is determined by the size of a set being analyzed and the number of signatures in the test being processed for the first phase. The second phase of computation is influenced by the number of detected potential subroutines. It yields $O(|T_p|(|t_{max}| + |t_{max}||R_{max}|))$, where t_{max} is the abstracted test script with the highest number of signatures, $t_{max}TP$, and R_{max} represents the largest of the sets of detected potential common subroutines created for each abstracted test script in the analyzed T_P.

The implementation of the algorithm is extended by an auxiliary data structure that saves a pointer to a physical location of the test step represented by a signature. This structure connects signatures with the original analyzed source code so any third party tool can integrate with TestOptimizer and process suggestions of reusable components.

The auxiliary data structure contains two principal records then for each of the signatures:

1. A location of signature s in the abstracted test script, $t \in T_P$ and
2. location of the signature s in the original analyzed test script, captured by the location metadata presented in Section 7.1 (refer to Figure 5.1).

In our example, the Figure 6.3 illustrates two potential common subroutines that are marked as Subroutine A and a Subroutine B. Steps being common for several tests are depicted by black rectangles while unique steps are depicted by white rectangles.

It holds that $\exists\ T_1 \subseteq T_P : |T_1| > 1, \forall\ t_x \in T_1 : A \subset t_x$ and $\exists\ T_2 \subseteq T_P : |T_2| > 1, \forall\ t_x \in T_2 : B \subset t_x$. As depicted in Figure 6.3, the potential common subroutine B follows the subroutine A in each of the abstracted test scripts where they are present. However,

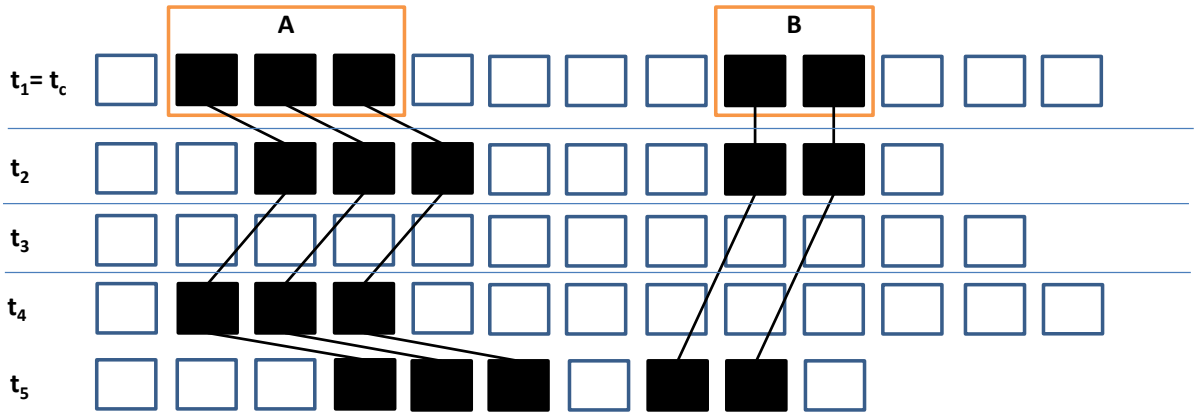

Figure 6.3: All the potential common subroutines identified in abstracted test scripts being analyzed [A.1].

this is only an example, and we cannot generally conclude that in all cases, the order of detected potential common subroutines found by TestOptimizer corresponds to the actual order in the analyzed automated test code. The order of subroutines can be the other way around, i.e., the subroutine B could occur before the subroutine A. However, for the final presentation of results to the user of the TestOptimizer, this issue is not relevant.

Algorithm 6.2 FIND_SUBROUTINES
Finding common subroutines in in the abstracted test scripts.

Require: $t_c \neq empty$ and $T_P \neq empty$ and $requiredMinLength \geq 1$
1: **set** *occurrence[]* to 0 ▷ set all elements of the array to 0
2: $max\_occurrence \leftarrow 0$ ▷ set max_occurrence to 0
3: $occurrence \leftarrow FIND\_CANDIDATES(t_c, T_P, requiredMinLength, occurrence[], max\_nce)$
▷ build potential candidate subroutines from common steps in tests
4: $C \leftarrow$ *empty set* ▷ reset the sequence of candidate steps
5: $P \leftarrow$ *empty set* ▷ reset the set of final detected potential common subroutines
6: $P \leftarrow$ *FILTER_CANDIDATES()* ▷ exclude low quality candidate subroutines
7: **return** P

The Algorithm 6.2 uses the Algorithms 6.3 and 6.4 as its subroutines. The Algorithm 6.3 iterates over the T_P and assembles a candidate subroutine from steps that occur in other tests of T_P. The algorithm adds the step into the candidate sequence and if the sequence is longer than the set threshold, it is saved in the set of candidate sequences R. During the analysis, we update the information about the count of step occurrences in the subroutines.

Algorithm 6.3 FIND_CANDIDATES
The algorithm for finding candidate sequences from the test steps of T_P.

Require: $t_c \neq empty$ and $T_P \neq empty$ and $requiredMinLength \geq 1$ and $nce[] \neq empty$ and $max\_occurrence \geq 1$

1: **for all** t in $(T_P \setminus \{t_c\})$ **do** ▷ for each abstracted test script in the analyzed set
2: $C \leftarrow$ *empty set* ▷ reset the potential sequence of common steps
3: $R \leftarrow$ *empty set* ▷ reset the set of detected potential common subroutines
4: **for all** s in t **do** ▷ for each signature from the analyzed abstracted test script
5: **if** s being marked as common with any other signatures **then**
6: $C \leftarrow (C,(s))$ ▷ append the potential common step sequence by signature s
7: **else**
8: **if** C.Length$\geq requiredMinLength$ **then**
9: $R \leftarrow R \cup \{C\}$ ▷ add the potential sequence of common steps C to the set of detected potential common subroutines
10: **end if**
11: $C \leftarrow$ *empty set* ▷ reset the potential sequence of common steps
12: **end if**
13: **if** C.Length$\geq requiredMinLength$ **then**
14: $R \leftarrow R \cup \{C\}$ ▷ add the potential sequence of common steps C to the set of detected potential common subroutines
15: $C \leftarrow$ *empty set* ▷ reset the potential sequence of common steps
16: **else**
17: $C \leftarrow$ *empty set* ▷ reset the potential sequence of common steps
18: **end if**
19: **end for**
20: **for all** r in R **do** ▷ for each of detected potential common subroutines
21: **for all** s in r **do** ▷ for each signature in a detected potential subroutine
22: $occurrence[s] \leftarrow occurrence[s] + 1$
23: **if** $max\_occurrence < occurrence[s]$ **then**
24: $max\_occurrence \leftarrow occurrence[s]$
25: **end if**
26: **end for**
27: **end for**
28: **end for**
29: **return** $occurrences[]$

The Algorithm 6.4 is a subroutine of Algorithm 6.2 and it filters candidate common subroutines found in the chromosome t_c.

In principle, the algorithm counts the occurrences of the abstract test steps in the candidate sequences. If an occurrence of the particular step equals *max_occurrence* we append the step into the current sequence C and if the candidate subroutine exceeds the given value of $requiredMinLength$ we save it as a resultant common subroutine. The algorithm returns the set of potential common subroutines P, which is the result of the analysis.

Algorithm 6.4 FILTER_CANDIDATES
Filters candidate subsequences, excludes not relevant subsequences and returns a set of potential common subroutines.

Require: $t_c \neq empty$ and $T_P \neq empty$ and $requiredMinLength \geq 1$ and $occurrence[] \neq empty$ and $max\_occurrence \geq 1$ and $C \neq empty$ and $P \neq empty$

1: **for all** s_c in t_c **do** ▷ for each signature from the chromosome
2: **if** $occurrence[s_c] = max\_occurrence$ **then**
3: $C \leftarrow (C, (s_c))$ ▷ append the potential common step sequence by signature s
4: **else**
5: **if** C.Length$\geq requiredMinLength$ **then**
6: $P \leftarrow P \cup \{C\}$ ▷ add the potential sequence of common steps C to the set of detected potential common subroutines
7: **end if**
8: $C \leftarrow$ *empty set* ▷ reset the potential sequence of common steps
9: **end if**
10: **end for**
11: **return** P

6.7 The Main Algorithms and Processing Modes Related to the Chromosome Selection

Apart from the selection of configuration parameters for the genetic algorithm for the LCS search, overall results significantly depend on the selection of chromosome t_c. The better the chromosome is the more valuable results are. Let us illustrate the problem in the following example. Let's have a set of abstracted test scripts A, B, C and D, where scripts A, B and C have common test steps while the script D not. In such a case, selection of the script D as the chromosome does not make any sense – no common test step can be found in the chromosome.

Moreover, the problem of the selection of a suitable abstracted test script as the chromosome has another dimension. Let's consider the lengths of the abstracted test scripts in

our example. For instance, the script A has 11 steps, the script B 58 steps, and the script C 120 steps (we do not consider the test D as it is not relevant to be a chromosome), it may not be worth to select the shortest test A as the chromosome. When selecting A, we may miss some potential subroutines between the test B and C. Therefore, we proposed two modes of operation for the TestOptimizer framework:

- The user wants to check if a particular subroutine (e.g., an already defined reusable subroutine) is present in a set of analyzed test scripts. In this case, we set a test that contains this subroutine as the chromosome t_c.
- The user wants to know whether there are some potential common subroutines in the test suite in general. In this case, the chromosome t_c is not known in advance.

This leads us to define two operation modes, CHROMOSOME_MANUAL and CHROMOSOME_AUTO. Table 6.1 presents the details of these modes.

Algorithm 6.5 CHROMOSOME_MANUAL_SEARCH
Processing in the CHROMOSOME_MANUAL mode.

Require: $T_A \neq empty$ and $requiredMinLength \geq 1$ and $0 \geq lengthWeight \leq 1$ and $0 \geq testCountWeight \leq 1$

1: $T_P \leftarrow SELECT\_PROSPECTIVE\_TESTS(T_A)$
2: $LCS(t_c, T_P, lengthWeight, testCountWeight)$
3: $FIND\_COMMON\_STEPS(t_c, T_P)$
4: $P \leftarrow FIND\_SUBROUTINES(t_c, T_P, requiredMinLength)$
5: **return** P

6.7.1 Chromosome Manual Mode

Algorithm 6.5 defines the computations in the **manual mode** (CHROMOSOME_MANUAL). The algorithm reduces T_A to T_P by calling the SELECT_PROSPECTIVE_TESTS (described in Section 6.3). In the next steps, we run algorithms LCS, FIND_COMMON_STEPS (Algorithm 6.1) and FIND_SUBROUTINES (Algorithm 6.2) for the given t_c.

6.7.2 Chromosome Semi–automated Mode

Algorithm 6.6 processes data in an **semi–automated mode** (CHROMOSOME_AUTO option). It starts with optimization of the input test set T_A to exclude disjunctive tests using the SELECT_PROSPECTIVE_TESTS (described in Section 6.3) and it yields T_P. Based on the T_P, we select prospective chromosomes T_C, $T_C \subseteq T_P$. For each chromosome

Table 6.1: Methods of a selection of the chromosome.

Method	**Selection of t_c**	**Description**
CHROMOSOME_MANUAL	by the user	**Principle**: The user has a full control over the process and she/he selects the chromosome and the set of test scripts to be analyzed. **Task**: To find out if a particular reusable routine occurs in some tests across the test set. **Algorithm**: CHROMOSOME_MANUAL_SEARCH, Algorithm 6.5 defined later in the text.
CHROMOSOME_AUTO	automated, for selected suitable $t_c \in T_P$	**Principle**: The user chooses a set of test scripts to be analyzed and the chromosome is selected by the solver automatically. **Task**: To find out if some common subroutines exist in the given set of scripts. **Algorithm**: CHROMOSOME_AUTO_SEARCH, Algorithm 6.6 is defined further on.

in T_C, the LCS search is performed, followed by FIND_COMMON_STEPS (Algorithm 6.1) and FIND_SUBROUTINES (Algorithm 6.2). At the end of the analysis, we collect the results and select the best results according to the AVQ value. The $lengthWeight$ and $testCountWeight$ user parameters are then very important as they influence a selection of the final results.

The SELECT_PROSPECTIVE_CHROMOSOMES function identifies tests that are suitable to be used as a chromosome. To do that, we utilize the Karp–Rabin [96] algorithm, which counts duplications in the abstracted test scripts from T_P. We use these results (the number of duplicates found) to select the abstracted test scripts whose steps are as different as possible. Such identified tests are then considered as the prospective chromosomes T_C.

The asymptotic complexity of Algorithm 6.6 is given by the asymptotic complexity of the LCS search algorithm, which is $O(n|Tp|)$, where n is the sum of $|t|$ for all $t \in TP$ [104].

The parameter *iterationsCount* drives the count of iterations for various chromosomes and it servers as a limit for performance reasons. The maximum value of *iterationsCount* is given by $|T_C|$.

Algorithm 6.6 CHROMOSOME_AUTO_SEARCH
Processing in the CHROMOSOME_AUTO mode.

Require: $T_A \neq empty$ and $requiredMinLength \geq 1$ and $0 \geq lengthWeight \leq 1$ and $0 \geq testCountWeight \leq 1$ and $iterationsCount \geq 1$

1: $X \leftarrow$ *empty set* ▷ reset the potential sequence of common steps
2: $T_P \leftarrow SELECT\_PROSPECTIVE\_TESTS(T_A)$ ▷ select prospective tests to analyze
3: $T_C \leftarrow SELECT\_PROSPECTIVE\_CHROMOSOMES(T_P)$ ▷ select prospective chromosomes
4: **if** $iterationsCount >\mid T_C \mid$ **then**
5: $iterationsCount \leftarrow\mid T_C \mid$ ▷ adjust the number of iterations
6: **end if**
7: **for all** t_c in T_c **do** ▷ for all prospective chromosomes
8: **if** $iterationsCount > 0$ **then** ▷ start a new iteration if not zero
9: $LCS(t_c, T_P, lengthWeight, testCountWeight)$ ▷ longest common subsequence search
10: $FIND\_COMMON\_STEPS(t_c, T_P)$ ▷ post–processing, see algorithm 1
11: $F \leftarrow FIND\_SUBROUTINES(t_c, T_P, requiredMinLength)$ ▷ post–processing, see algorithm 2
12: $X \leftarrow X \cup \{F\}$ ▷ save potential results
13: $iterationsCount \leftarrow iterationsCount - 1$ ▷ decrease iteration counter
14: **end if**
15: **end for**
16: $P \leftarrow x, x \in X, x$ is having the highest AVQ(x, T_P) ▷ select the best results according to user's preferences
17: **return** P

6.8 Summary

In this section, we presented the algorithms for the search for potentially reusable subroutines in a set of abstracted automated tests, including input and output of this processing and metrics that are used to evaluate the method quality.

In this chapter, we also discussed the problem of the selection of the chromosome that significantly influences the resultant quality and a usability of the results. TestOptimizer can operate in two modes The first offers the manual selection in which the user defines a subroutine which will be searched in other automated tests, while in the second mode,

TestOptimizer runs in the automatic mode and selects chromosomes automatically. In this mode, a general search for common parts of the tests is identified.

The user controls the whole processes with the help of defined parameters to achieve the best result of the analysis tailored to the particular project context.

Chapter 7

Framework Prototype

In this chapter, we present an API specification to control the initial prototype TestOptimizer framework. Then, we describe the implementation details of the final prototype TestOptimizer framework used for experimental verifications of the proposed method.

7.1 API of the TestOptimizer Framework

Proper parameter settings that achieve mutual balance are crucial in obtaining useful results from the proposed method during the analysis process. From our experiments, we concluded that an inappropriate combination of control parameters, even when executed against highly suitable data, may affect the analyzed results and cause the refactoring suggestions to not be optimal for further usage. Hence, to increase the applicability of the proposed method, the process is flexibly parameterized so that users can find the best configuration for applying the TestOptimizer framework to a particular test automation project. The parameters can be entered via the framework API.

In this section, we explain the parameters used in the TestOptimizer API. Table 7.1 summarizes all the parameters fundamental to test analysis and/or for controlling the entire process. We use these parameters to describe an output set of suggestions of potential common subroutines returned by the framework that serves as the basis for subsequent test script refactoring.

Table 7.2 defines the result of analysis, which consists of technical metadata and a set of resulting potential common subroutines indicating potentially reusable objects in the automated test scripts. For each of these subroutines, details about particular scripts and individual repetitive signatures are provided.

Information concerning the location of the potential common subroutines in the scripts includes records regarding a particular script name, the starting and ending offset of the repetitive subroutine within the script and a copy of a code fragment that is considered part

of this subroutine. The details about repetitive signatures include the signature name and pointers to relevant code fragments. Based on these results, the end user can manually refactor the test scripts. To simplify this process, we provide the user with subroutine quality metrics that can help in determining which identified potential subroutines to consider.

An example of result data is depicted in Figure 7.1. Technical metadata such as an ID of the request, an ID of the server cache record and a link to a log is not depicted in the figure.

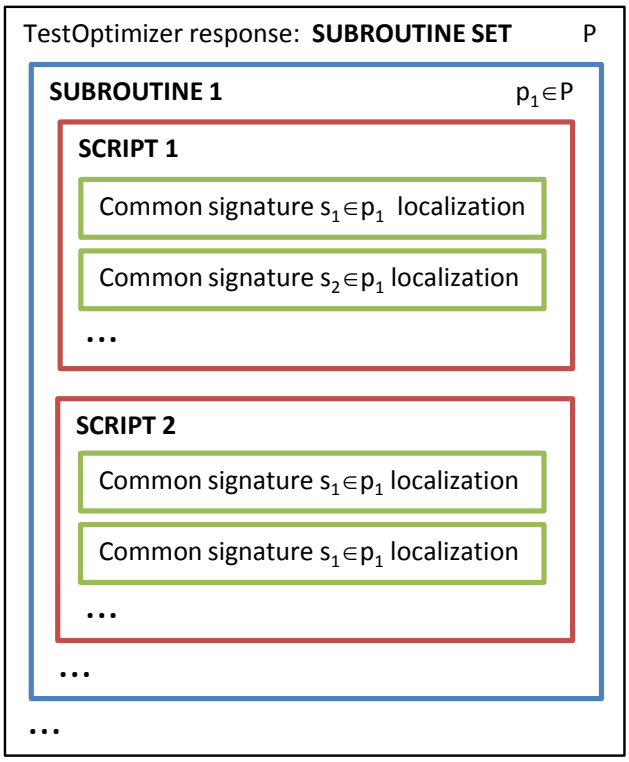

Figure 7.1: Structure of analysis results provided by the TestOptimizer server [A.1].

For an easier association of signatures with scripts, we pair the parameters as *scriptName*, *lineNumberFrom* and *lineNumberTo* (location metadata) with the signatures. The parameter *codeFragment* is determined from the source scripts stored in the cache. For this purpose, the location metadata saved in this cache are employed. At the script level, we assembly allocation data in blocks according to their order in which signatures of the potential common subroutine are present in the physical automated test script.

Table 7.1: Request parameters influencing the analysis.

Parameter	**Description**
requestID	ID of the request specified by the user. The ID allows tracking the submitted request during processing on the TestOptimizer server
converterType	The type of converter to use (which translates original test scripts to their abstractions). TestOptimizer provides different converters for different languages and test automation APIs.
startPoint	The user can select one of the following options: NEW_UPLOAD – Set of automated test scripts for analysis are uploaded to the server. The scripts are then converted to abstracted test scripts T_A and analyzed for potential common subroutines by specified parameters. SCRITPS_CACHED – Uploaded scripts are taken from cache and converted to abstracted test scripts T_A, then analyzed for potential common subroutines by specified parameters. ABSTRACTION_CACHED – Created T_A is analyzed again with different parameters.
scriptCacheID	In case of SCRITPS_CACHED: ID of set of already uploaded test scripts for analysis.
scripts	In case of NEW_UPLOAD: Stream of automated test scripts for analysis to upload.
signatureLevel	Level of detail, which is captured by signatures during the conversion to abstracted test script. For an overview of levels, refer to Table 5.2. Expected in case of NEW_UPLOAD and SCRITPS_CACHED options.
requiredMinLength	A minimal length of potential common subroutines, which will be identified (refer to $requiredMinLength$ parameter in Algorithm 6.2, 6.3, 6.4)
lengthWeight	A preference of longer potential subsequences in less test scripts by $lengthWeight$ parameter (refer to fitness function in LCS and 6.5)
testCountWeight	A preference of shorter potential subsequences in more test scripts by $testCountWeight$ parameter (refer to fitness function in LCS and Algorithm 6.5)
chromosomeMode	CHROMOSOME_MANUAL (user's preference to manually select a chromosome) or CHROMOSOME_AUTO mode, refer to Table 6.1.
chromosome	In case of CHROMOSOME_MANUAL mode: an explicit selection of chromosome $t_c \in T_A$
iterationsCount	In case of CHROMOSOME_AUTO mode: $iterationsCount$ parameter, specifying how many iterations for various chromosomes will be executed (refer to Algorithm 6.6)

Table 7.2: A structure of the response, including parameters that influenced the analysis.

Parameter	**Description**
Technical Metadata	
requestID	$requestID$ (copied from the request call)
scriptCacheID	ID of the server cache record in which the analyzed original test scripts are saved
Log	An access to the log, which lists details from the conversion and analysis process (a link to the log file stored on the server)
SUBROUTINE SET	For the set of potential common subroutines P
totalNumberOfScripts	A total number of analyzed test scripts, $\|T_A\|$
lengthWeight	$lengthWeight$ parameter specified in the request
testCountWeight	$testCountWeight$ parameter specified in the request
AVQ	AVQ(P, T_A, $lengthWeight$, $testCountWeight$) value to help the user to decide, if the analysis result is good enough for particular T_A, or if to try to analyze T_A again with different parameters.
Subroutine	For each potential common subroutine $p \in P$:
subroutineID	ID of a potential common subroutine p
SQ	SQ(p, T_A) value to help the user to decide, which of the potential subroutines to prefer in identification of reusable objects.
numberOfScripts	A number of test scripts, where the potential common subroutine p occurs
scriptNames	A list of names of test scripts, where the potential common subroutine p occurs
Script	For each of test scripts, where the potential common subroutine p occurs
scriptName	A name of test script, where potential common subroutine p occurs
lineNumberFrom	A number of line in the original test script code, where potential common subroutine p starts.
lineNumberTo	a number of line in the original test script code, where potential common subroutine p ends.
codeFragment	An aggregated fragment of analyzed original test script code.
Signature	For each occurrence of signature sp in an original test script
subroutineIDref	Reference to ID of potential common subroutine p
scriptName	Names of test script, where s occurs
lineNumberFrom	A number of line in the original test script code, where signature s starts.
lineNumberTo	A number of line in the original test script code, where signature s ends.
codeFragment	A fragment of analyzed original test script code. Can be different for individual signatures from p.

The reasons why we include the *codeFragment* parameter are the following:

1. Automated test scripts are abstracted in order to capture the semantics of their steps. The abstraction may hide potential differences of two identical signatures at the source code level.

2. The *codeFragment* field enables more easy integration with third–party tools.

3. A presence of this parameter simplifies testing of TestOptimizer framework because the returned *codeFragment* value can be quickly compared to the fragment of the original source code that is specified by the triple of parameters: *scriptName*, *lineNumberFrom* and *lineNumberTo*.

7.2 Conceptual Architecture of the TestOptimizer

The user can connect to the TestOptimizer server web console via the API and carry out main administrative tasks or she/he can use the command–line interface of the TestOptimizer server. Figure 7.2 presents the high–level architecture of the TestOptimizer system from a conceptual viewpoint. We illustrated storage components as gray rectangles whereas arrows indicate main data flow in the process. Storage components are implemented by a PostgreSQL relational database.

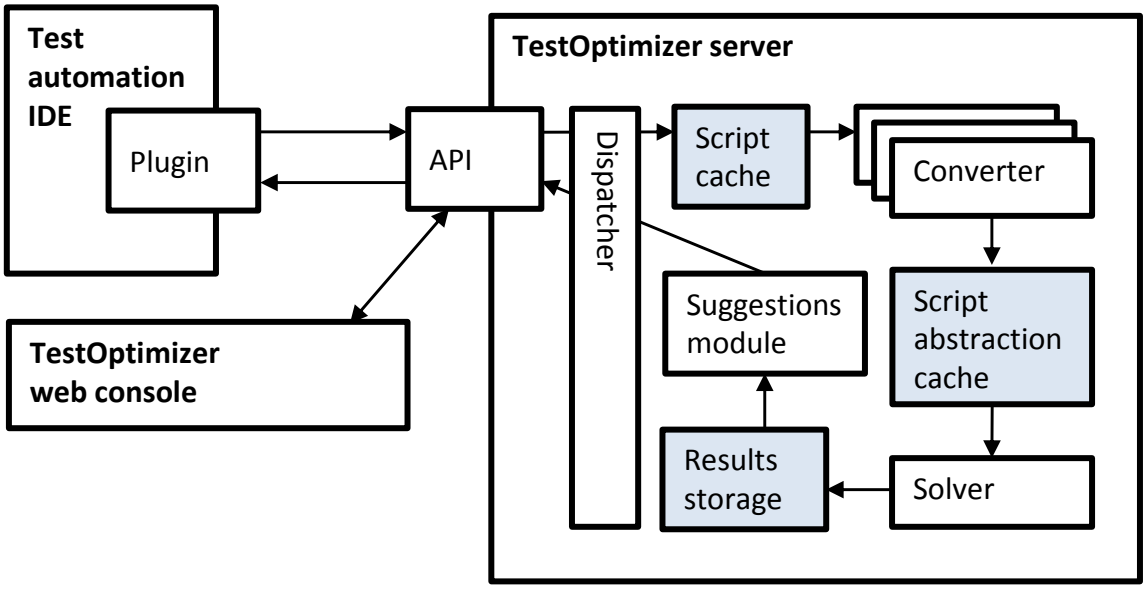

Figure 7.2: An architecture overview of the TestOptimizer prototype [A.1].

The whole process is implemented as a modular pipeline in which all the individual parts can be flexibly adopted according to the user needs. In the beginning, the user sends a request to optimize scripts, which is handled by the *runAnalysis* API function at the beginning of the pipeline. When the user uploads test data to the server via the REST API, TestOptimizer saves analyzed scripts in the **Script** cache to reduce an amount of the network traffic and to shorten the processing time. This feature is important if the user decides to re–run the analysis with a different set of parameters on the same set of scripts. We use the *startPoint* parameter in the *runAnalysis* request for this purpose.

The **Converter** module is responsible for converting original tests to the abstracted test scripts (refer to Chapter 5). The module is flexible to be adapted to a particular programming language and test automation API (for instance Java with the Selenium WebDriver API or WebdriverIO with Mocha JavaScript) but it requires to implement the particular extension of the converter for the given combination. Our implementation supports Java in its specification 7 and Selenium WebDriver API [1] including Appium. The selection of the particular converter that is used for the analysis is made by the converterType parameter of the *runAnalysis* request.

We keep the abstracted test scripts in the **Script Abstraction** cache and we link them with the source code of analyzed automated test scripts. When the user starts the optimization with the *startPoint* parameter defined as ABSTRACTION_CACHED, TestOptimizer checks whether the previously created abstractions for the specified *scriptCacheID* and *signatureLevel* are cached. If the records are loaded in the cache then the cache is used as the main data source. If relevant data are not found in the cache then the analysis process is executed again with the new configuration.

In the next step, the analysis continues in the **Solver** component. This module is responsible for finding the longest common subsequences in the chromosome using the core LCS algorithm. Apart from that, **Solver** identifies common steps in the analyzed abstracted automated tests (FIND_COMMON_STEPS specified in Algorithm 6.1) and assembles the potentially reusable subroutines from these common steps (FIND_SUBROUTINES specified in Algorithm 6.2). On demand, the **Solver** can analyze the input test suite using the CHROMOSOME_AUTO mode (CHROMOSOME_AUTO_SEARCH specified in Algorithm 6.6). After the computations, results are saved in the **Results** storage and further processed by the **Suggestion** module in order to prepare a response for the user.

Since the processing is time demanding, we implemented API calls as asynchronous. To start the analysis, the user sends a *runAnalysis* request. Once the *runAnalysis* is called, the system assigns to the job a *requestID* that the user uses to track the status of the job. The **Dispatcher** component controls the execution in the whole pipeline and manages the pool of possible concurrent requests to the TestOptimizer sever.

[1] https://seleniumhq.github.io/selenium/docs/api/java/

7.3 Physical Architecture of the TestOptimizer

To increase its practical applicability, our goal was to implement the TestOptimizer framework prototype in an open, scalable and high–performance architecture that would enable a parallelization of the test code analysis in the future. Therefore, we proposed a server–client architecture with a server being responsible for all the time–demanding and performance requiring computations, and a client being used as a user interface and a proxy for the server.

A server architectural layout uses a layered pattern for which we choose reference Oracle Enterprise Java Beans (EJB) as a core technology [2]. The implementation is written in Java Enterprise Edition 7. Java Beans are very suitable for developing enterprise applications as they allow to create distributed, transactional and scalable applications that can be deployed on several nodes grouped in clusters.

The Figure 7.3 presents the implementation layers of the TestOptimizer prototype. We use a PostgreSQL relational database (the database layer) to store abstracted test scripts and auxiliary data structures described in Chapter 5. For the persistence layer, we used a Hibernate framework for an object–relational mapping and we implemented entities being processed as data access objects. All the major computations during the analysis are carried out in the business layer that encapsulates its API using the presentation layer. This presentation layer publishes a REST API and servers as main endpoints for the client applications. It is implemented as a facade for the underlying APIs. The application is deployed on a WildFly[3] application server (formerly JBoss).

7.4 Final Version of Prototype

Based on the feedback from the first phase of the experiments, we adjusted the TestOptimizer in the several areas. In this thesis, we refer to the version of the prototype with these adjustments as to the final prototype.

Firstly, we reworked the LCS search part of the method. We changed genetic operators for a mutation and crossover, and also tailored the process of applying these operators on the population. In the initial prototype, we applied mutation and crossover based on a roulette wheel selection. In the final prototype, we changed the roulette to prefer particular operators depending on a phase of the computations. At early phases, when the population does not contain any useful longer subsequences, it is not worth to crossover the blank population but it is better to prefer mutation to crossover because there is a quite good chance that some test steps in mutated individuals or their parts become a part of the final solution. Secondly, we made changes to a representation of population and

[2] http://www.oracle.com/technetwork/java/index-jsp-140203.html
[3] http://www.wildfly.org/

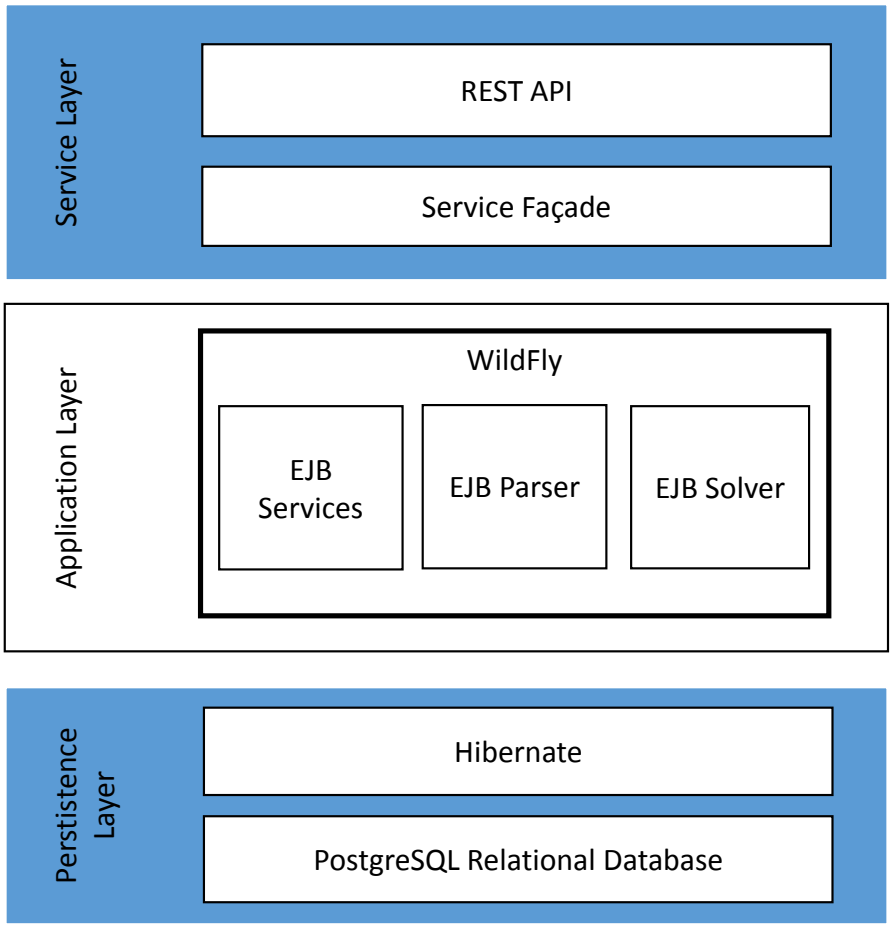

Figure 7.3: An overview of implementation layers of the method.

the selection process. We emphasized the concept of elitism but we focused on a higher diversity of the population meaning that we do not preserve all the candidate individuals with high rates but similar sequences. In contrast to that, we also support candidates with lower rates but with significantly different test steps.

7.5 Summary

In the proposal of the TestOptimizer architecture and its APIs, we laid an emphasis on several qualities that we understood as critical: user's experience, a capability to integrate TestOptimizer with third–party tools and a possibility to trace and evaluate the analysis. A rich REST–based API gives the user a good opportunity to tailor settings of the tool to

the best results of the analysis. Furthermore, it makes easier to integrate TestOptimizer to other development, test management tools or eventually source code management tools because the RESTful APIs are today considered as a standard for a communication between two systems.

The selected Java–based technology allows running the TestOptimizer framework on different platforms including Linux/Unix based systems or Windows environments. The modular structure of the framework enables adding a new functionality (for instance, a support for another test scripting language) without a need to change the application dramatically. Moreover, the architecture of the framework is prepared for a future parallelization of the analysis. In the present time, we use only one node to run TestOptimizer on the application server, but using EJBs, we can improve the performance out–of–the–box by deploying other instances on additional nodes and thus, to process larger sets of analyzed test scripts and/or process a higher count of user requests.

Chapter 8

Experiments

We start this section by definition of the research questions, which are answered by the experimental results. Then, we describe the method used in the experiments and provide their results, which helped us to adjust configuration details of the proposed method and verified its practical functionality. In the first part of this chapter, we present details of the experiments conducted with an initial prototype of the TestOptimizer applied on several sets of automated tests created for web–based applications. In the second part of this chapter, we describe the experiments conducted with the final prototype of the framework, which was applied on several sets of automated tests created for SUTs with the web–based user interface, as well as for mobile applications.

8.1 Research Questions

For the verification of the proposed concept, we proposed a set of experiments, in which several sets of automated tests developed within various real test automation projects were analyzed by the TestOptimizer. In the experiments, focused on the following principal issues:

1. **A functional aspect of the proposed method**: is the proposed approach suitable for different automated tests and able to process them, and does our approach achieve better results than conventional methods for an identification of repetitive fragments of the source code?

2. **An applicability aspect of the whole method on test automation projects**: we were interested in a fact whether test automation developers using the TestOptimizer save time during the refactoring of automated test scripts.

To evaluate the proposed method, we defined the following research questions, which are answered by the conducted experiments described further in this thesis:

RQ 1 Does the TestOptimizer framework identify potentially reusable subroutines, which are relevant for the optimization of the automated tests?

RQ 2 What is the performance of the proposed method in comparison to manual identification of the potentially reusable subroutines in terms of accuracy of the method (real relevance of the identified subroutines)?

RQ 3 What is the performance of the proposed method in comparison to manual identification of the potentially reusable subroutines in terms of time effectiveness of this process?

RQ 4 What is the performance of the proposed method in comparison to the selected currently established mainstream tool used for code refactoring in terms of accuracy of the method (real relevance of the identified subroutines)?

RQ 5 What are the usability of the proposed method and the TestOptimizer framework in comparison to the selected currently established mainstream tool used for code refactoring.

RQ 6 How flexible is an application of the proposed method to the automated tests created for another type of SUTs than systems with the web–based user interface.

The experiments performed with the initial prototype of the TestOptimizer framework answers research questions **RQ 1**, **RQ 2**, **RQ 3** and **RQ 4**. Additional set of experiments conducted with the final prototype of the framework provides answers to research questions **RQ 1**, **RQ 4**, **RQ 5** and **RQ 6**.

8.2 Experiments With Initial Prototype

We started verification of the proposed method and the pilot implementation of the TestOptimizer framework after finalization of its first prototype. In this Section, we present respective details.

8.2.1 Experiment Design

To answer **RQ 1**, **RQ 2**, **RQ 3** and **RQ 4**, in the experimental verification of the first TestOptimizer prototype, we used several sets of automated test scripts, which were processed by the proposed approach. Table 8.1 presents properties of these automated tests. The test scripts were taken from selected real software development projects since our main

Table 8.1: Test suite properties used for experiments with the initial prototype.

Script Set ID	**Language**	**Number of Scripts**	**Lines of Code**
1	Java, JUnit Selenium WebDriver	70	6728
2	Java, JUnit Selenium WebDriver	99	9345
3	Java, JUnit Selenium WebDriver	50	4507
4	Java, JUnit Selenium WebDriver	150	16347
5	Java, JUnit Selenium WebDriver	120	11704

objective was to conduct tests with real test data and not artificial tests. The test scripts were recorded with the Selenium IDE and exported to the Java and JUnit format with Selenium WebDriver API. We selected this scenario since we wanted to simulate the main use of TestOptimizer, which is the optimization of recorded automated tests.

Furthermore, to answer **RQ 4** we need to conduct a comparison of TestOptimizer framework with another common approach for code refactoring, being based on a direct analysis of repetitive fragments of the source code. From the group of tools offering the direct analysis, we selected a static source code analyzer PMD[1] for the Eclipse IDE employing the modified Rabin–Karp string search algorithm to search for code duplicates.

During the experiments aimed at answering **RQ 1**, **RQ 2** and **RQ 3**, we analyzed prepared sets of automated test scripts (Table 8.1) using the proposed TestOptimizer framework.

In addition, in the experiments aimed at answering **RQ 4**, we analyzed prepared sets of automated test scripts (Table 8.1) using both the PMD tool and the TestOptimizer.

Since our intention was to perform fine–grained comparison from various aspects, we introduced an Averaged Value of Sequence Quality (ASQ) metric in addition to the Analysis Variant Quality (AVQ) metric defined in Section 6.4.1

$$ASQ(p, T_A) = \Sigma_{p \in P} SQ(P, T_A)/|P| \tag{3}$$

[1]https://pmd.github.io/

As the next step of the experiment, we performed a manual analysis of the potentially reusable subroutines found by the TestOptimizer and PMD, as we had to decide which of the identified potentially reusable subsequence are really relevant for the test refactoring. Not all identified reusable subsequences are suitable to be a generally reusable function for automated tests, so a manual verification of the results had to be performed, despite the fact that this verification was considerably resource demanding during the experiments.

The manual analysis assessing the true relevance of the identified potential reusable subroutines was carried out by two subject matter experts in test automation. We asked them (i) to exclude all identified subroutines that were already included in any longer subroutine, (ii) to exclude all the subroutines in which object descriptors were marked as the action parameters, and (iii) to exclude all the subroutines in which instances of classes and static members are wrongly identified (for instance, a `driver.findElement` test step may be wrongly identified as assertion like `Assert.AssertThat` as both commands contain only an instance of a class and its method). To reduce human errors, we let each expert perform the manual analysis independently and then we cross–compared the results twice.

8.2.2 Experimental Results

Table 8.2 present the following information gathered to answer **RQ 1** and **RQ 4** during the experiments: (i) raw data acquired by the applied PMD and TestOptimizer, (ii) truly relevant subroutines filtered by manual analysis of the raw data, and (iii) their relative ratios. For the first part, we evaluated all the identified potentially reusable common subroutines with the AVQ metric. In the second part of Table 8.2, the manual analysis described in the previous subsection excluded irrelevant results and we evaluated only truly relevant candidates for reusable subroutines using the same metric. In the last section of Table 8.2, we calculated the relative ration of relevant candidate subroutines. We used the same organization (raw data, truly relevant subroutines and their relative ratio) also for Table 8.3, but with evaluated by the ASQ metric.

For experiments with all the test suites except the test suite 2, we set the **lengthWeight** parameter to 0.9 and **testCountWeight** parameter to 0.1. However we wanted to emphasize the length of candidate sequences in the test suite 2 so we set the **lengthWeight** parameter to 0.95 and **testCountWeight** parameter to 0.05. Apart from that, we set the **chromosomeMode** to CHROMOSOME_AUTO and **iterationsCount** to 5.

During the analysis, we considered only candidate sequences whose length was longer or equal 3 test steps, i.e., $p >= 3$. For the verification with the direct analysis of repetitive fragments using the PMD tool, we used the standard configuration from Eclipse (Kepler edition 2016). For PMD, we also set the minimal length of common code blocks to three.

Based on the findings presented in Tables 8.2 and 8.3, we can make several conclusions.

Table 8.2: A comparison of TestOptimizer results with PMD by AVQ metrics.

Script Set ID	**AVQ for PMD**	**AVQ for TestOptimizer, signatureLevel = 0**	**AVQ for TestOptimizer, signatureLevel = 1**	**AVQ for TestOptimizer, signatureLevel = 2**
Raw data				
1	320	363	176	98
2	502	1421	798	181
3	324	197	183	66
4	700	652	312	120
5	592	438	288	158
Truly relevant potential subroutines after manual analysis				
1	215	233	172	96
2	411	671	798	162
3	179	154	179	63
4	312	425	312	112
5	282	259	284	155
Relative ratio of relevant potential subroutines				
1	68.0 %	64.2 %	97.7 %	98.0 %
2	81.9 %	47.2 %	100.0 %	89.5 %
3	55.2 %	78.2 %	97.8 %	95.5 %
4	44.6 %	65.2 %	100.0 %	93.3 %
5	47.6 %	59.1 %	98.6 %	98.1 %

Considering the initial data without further assessment of the relevance of the found potential reusable subroutines, PMD achieves better results in average based on the AVQ metrics on the experimental sets of automated tests, TestOptimizer can compete for the PMD tool only with the lowest *signatureLevel*. However, PMD was worse in the case of the second script set. We analyzed a root cause of this effect. The reason was, that TestOptimizer selected an ideal test with many common test steps as the chromosome, which resulted in a high count of valid subroutines.

After the manual analysis, assessing the real relevance of the findings, the AVQ metrics show that TestOptimizer produces a comparable count of relevant potential subroutines (*signatureLevel* = 1 and 2) for an automated test script optimization than PMD but with a better relevance (**RQ 4**). When we compare relative ratios of relevant potential subroutines, PMD reaches an interval approximately between 45 and 85 %, whereas TestOptimizer reaches an interval 80 up to 100 % for *signatureLevel* equals to 1 and 2.

Similar effect can be observed for the ASQ results (Table 8.3, part for Raw data).

Table 8.3: A comparison of TestOptimizer results with PMD by ASQ metrics.

Script Set ID	**ASQ for PMD**	**ASQ for TestOptimizer, signatureLevel = 0**	**ASQ for TestOptimizer, signatureLevel = 1**	**ASQ for TestOptimizer, signatureLevel = 2**
Raw data				
1	40	60.5	35.3	18.9
2	167.3	355.3	199.5	90.5
3	36	49.3	43.5	24.8
4	140	163	122	60
5	74	109.5	48	39.5
Truly relevant potential subroutines after manual analysis				
1	26.9	38.8	34.7	18.4
2	137	167.8	199.5	81
3	19.8	38.5	42.8	23.3
4	104	106.3	122	56
5	47	64.8	47.3	38.8
Relative ratio of relevant potential subroutines				
1	67.2 %	64.2 %	98.1 %	97.4 %
2	81.9 %	47.2 %	100.0 %	89.5 %
3	54.9 %	78.2 %	98.5 %	93.9 %
4	74.3 %	65.2 %	100.0 %	93.3 %
5	63.5 %	59.1 %	98.6 %	98.1 %

Without subsequent manual verification the relevance of potential common subroutines for $signatureLevel = 0$, TestOptimizer achieved better results than PMD. For the analysis with a higher of detail at $signatureLevel = 1$ and 2, it might seem, that the PMD tool performed better.

However, when subject matter experts analyzed the results returned by the TestOptimizer and PMD and excluded all irrelevant potential common subroutines (presented in "Truly relevant potential subroutines after manual analysis" sections of Table 8.2 and Table 8.3), different conclusions can be made:

1. The finest level for signatures ($signatureLevel = 2$) is very specific, which prevents TestOptimizer from finding candidate subsequences that occur in more analyzed automated test scripts. TestOptimizer at this level also considers the specific testing data being used in actions. Such data do not repeat usually across the analyzed scripts, for example, consider a particular combination of a username and password. As a result, at this level, TestOptimizer locates a lower number of potential common

subroutines than PMD (**RQ 4**).

2. The analysis of data after excluding irrelevant potential common subroutines shows that the initial $signatureLevel = 0$ is too abstract and the TestOptimizer finds a high number of common subroutines that are not relevant from the test automation point of view, and thus it decreases the efficiency of the whole approach. Analyzing the test script source code at this level makes sense in cases when the user wants to check whether unknown test suites may contain some reusable routines to process it later on at a different level but it is not practical from the refactoring suggestion point of view.

3. Experiments yield $signatureLevel = 1$ as the best option when considering the level of abstraction to the relevance of the suggestions. The values of AVQ and ASQ metrics show that TestOptimizer is very efficient at this level and identified potentially reusable candidates are relevant. Moreover, the overall results of TestOptimizer are better than results returned by the PMD tool when compared at the middle level of abstraction (**RQ 4**).

4. When focused on a structure of automated test scripts and their specific context (see Table 8.2 and 8.3, the section "Relative ratio of relevant potential subroutines"), we can conclude that the TestOptimizer framework at $signatureLevel = 1$ and 2, identifies more relevant results than the general search for common code fragments by the PMD (**RQ 4**).

The results of experiments with the initial prototype demonstrated that TestOptimizer finds and localizes more relevant potential reusable subroutines, while the general PMD tool with a direct analysis simply reports on the number of sequences (**RQ 1**, **RQ 4**). Results provided by the PMD were less relevant from the automated test semantic viewpoint because it mismatches commands having a completely different semantic meaning for automated test steps and considers them as an identical command across the identified set of potentially reusable subroutines (**RQ 4**).

To answer **RQ 1** in more depth, we need to answer two sub–questions: (i) are the suggested reusable subroutines meaningful from an automated test script refactoring point of view, and (ii) is the original functionality of automated tests preserved? Regarding the first sub–question, we conducted a manual review, which task was to exclude invalid or irrelevant subroutines. Relative ratios of relevant potential subroutines show that TestOptimizer meets this requirement. For the problem of an intact functionality of automated tests, we proposed a verification of the method by a direct comparison of test runs with SUTs. We ran all tests with stable SUTs prior to the optimization, recorded test results and then, we ran all tests after the optimization made in accordance with TestOptimizer suggestions (after the manual analysis of relevance of the tests) again. Afterwards, we

Table 8.4: An efficiency of TestOptimizer in relation to manual processing.

Script Set ID	**ASQ for performed manual analysis**	**ASQ for TestOptimizer, signatureLevel = 1**	**ASQ for TestOptimizer, signatureLevel = 1, after manual revision**
1	38.4	35.3	34.7
2	218.1	199.5	199.5
3	30.1	43.5	42.8
Script Set ID	**Average time spent by developers performing the manual analysis [hours]**	**Time spent using Test Optimizer [hours]**	**Time saved by using Test Optimizer [percentage]**
1	3.2	0.9	71.88 %
2	4.1	1.5	63.41 %
3	1.7	0.6	64.71 %

compared test results from both test runs and double–checked that the test results are identical. We performed these tests for the sets 1–3. Functionality and results of the refactored automated tests has been maintained.

Regarding the **RQ 2** and **RQ 3**, we focused on testing of the practical applicability of the method. In this phase, we conducted a comparison of independent manual analyses of the potential common subroutines being carried out on three sets of the automated test scripts with the results obtained from the TestOptimizer framework. We evaluated:

1. The relevance of the potential common subroutines found (**RQ 2**), and,
2. The time savings when using the automated analysis with the TestOptimizer framework (**RQ 3**).

We asked 5 senior and 25 junior Java developers to carry out the independent review of the test scripts and to analyze and determine reusable code components. All reviewers were trained in Selenium WebDriver and test code refactoring principles for test automation. We did not lay any restrictions on a usage of any refactoring or code duplication search tools during the manual analysis except the TestOptimizer tool. The results of this experiment are concluded in Table 8.4.

The Time spent using TestOptimizer column does not include the server computation time. This experiment showed that:

1. The scores of relevance expressed by the ASQ metric of both the manual analysis and the TestOptimizer results are comparable for the automated test script sets used in these experiments (**RQ 2**),

2. Employing our approach significantly reduced the time required for processing linearly processed test scripts (by 65% on average) (**RQ 3**).

The measured times are in hours. However, the main advantage and contribution of TestOptimizer are the following: when the test suite refactoring tasks are conducted repeatedly, the time savings accumulate during test automation. As a result, such time savings are then more significant and justify employing the proposed method in software development projects (**RQ 3**).

8.3 Experiments With the Final Prototype

Based on the feedback from the first phase of the experiments, described in Section 8.2, we adjusted the prototype of the TestOptimizer framework (these adjustments are described in Section 7.4). With this final prototype and we conducted another set of experiments to provide more data to the research questions **RQ 1** and **RQ 4** (already addressed in the first phase of the experiments) as well as answer the research questions **RQ 5** and **RQ 6**.

8.3.1 Experiment Design

In the second phase of experiments with the final prototype, we analyzed automated tests created for SUTs with a web–based user interface as well as for mobile applications to test a practical application of TestOptimizer in different environments.

In addition to the automated tests created in Java, JUnit and SeleniumWebdriver API used in the first phase of the experiments (Table 8.1), in this phase, we used another set of automated tests for the mobile applications implemented in Java, JUnit and SeleniumWebdriver and Appium. Properties of these tests are summarized in Table 8.5. Also here, we put emphasis on employing automated test scripts from real software projects. However, recording of user actions in the front–ends of mobile applications is not so straightforward as in the case of web applications, due to the following possible technical limitations:

1. **Different operating systems such as Android and iOS** that require to use different test automation tools producing different test scripts for the same test case. Furthermore, both platforms may differ in a structure of the mobile SUT and IDs used.

2. **Native mobile applications** on for example an Android platform do not simply expose their structure and require to use special SDK tools[2] to scan and analyze them, while web–based mobile applications can be simply investigated using a Chrome[3] browser.

3. **Web–based contexts are not exposed** in hybrid applications on real hardware devices that combine a native with the web–based context within one mobile application. The web–based context of applications in a release mode is out a control of test automation tools.

Moreover, development of web applications is established for the last two decades, but development of mobile applications is incomparably younger discipline, which also influences the available test automation tool for the mobile platforms. As a result, test recording tools are generally less mature and capable than their web counterparts. For these reasons, we could not use just simply recorded tests we used for the previous experiments but we had to use automated scripts developed by test developers manually using the descriptive approach and to simulate naive, sub–optimal programming style in this process. However, this set–up is perfectly relevant for the experiment, as a number of current industry test automation projects are challenged by using of this naive, sub–optimal descriptive programming, having a significant impact on subsequent maintenance of the tests.

Hence, for the experiments, we selected sets of automated tests, in which their developers have not used advanced approaches as reusable components (objects, functions) or a page object design pattern to mimic the test recording approach as much as possible.

The automated test scripts were developed directly in Java, JUnit, and Selenium WebDriver with Appium. We selected as the test runner again the JUnit framework. Nevertheless, our approach supports TestNG as well and there is no significant difference from the analysis point of view between these two Java test frameworks.

In this phase of experiments, we followed the same method as described in Section 8.2 to evaluate the correctness of the results for given sets of automated test scripts and to investigate whether TestOptimizer results are of a higher quality than conventional analysis that does not reflect specifics of the analyzed problem. Also in this phase, we benchmarked the TestOptimizer results with the PMD code static analyzer.

The properties of sets of automated test scripts used in this phase of experiments are summarized in Table 8.5. After analysis by TestOptimizer and PMD, we processed the results in the same way we did it in the experiments with the initial prototype of the TestOptimizer. Captured records were manually analyzed by two test automation domain experts in two phases to exclude irrelevant refactoring suggestions. The manual review included a cross–check to minimize possible human mistakes.

[2]https://developer.android.com/training/testing/ui-automator
[3]https://www.google.com/chrome/

Table 8.5: Characteristics of mobile test suites used for the experiments with the final prototype.

Script Set ID	**Language**	**Number of Scripts**	**Lines of Code**
6	Java, JUnit Selenium WebDriver with Appium	54	9727
7	Java, JUnit Selenium WebDriver with Appium	89	10883
8	Java, JUnit Selenium WebDriver with Appium	195	33042
9	Java, JUnit Selenium WebDriver with Appium	263	31561
10	Java, JUnit Selenium WebDriver with Appium	424	62052

8.3.2 Experimental Results

In the second phase of experiments we aimed at answering the research questions **RQ 1**, **RQ 4**, **RQ 5** and **RQ 6**.

Table 8.6 presents the AVQ values for the result raw data as well as AVQ values for the results after manual analysis to identify truly relevant potential reusable subroutines. Then, the table presents relative ratios of raw candidate suggestions and truly relevant subroutine suggestions. In the Table 8.6, benchmark values for PMD tool are also presented.

Since just the AVQ metric does not express all the aspects of quality of common subroutines, we evaluated the results including the results from the benchmark using the ASQ metric. The Table 8.7 presents the results. Obtained results (Tables 8.6 and 8.7) give an answer to **RQ 1** and **RQ 4**: TestOptimizer identifies potentially reusable subroutines that were considered after the manual review as valid. Moreover, relative ratios of relevant potential subroutines compared to the PMD benchmark indicate that the real relevance of identified subroutines by TestOptimizer at the signature level equals 1 achieve more than 90 % while PMD only 20 – 70 % (**RQ 4**).

The second round of experiments showed that in larger test suites (more than 100 tests) with a highly redundant test steps, in terms of initially suggested potential subroutines without further analysis of their real relevance, TestOptimizer cannot compete with the used benchmark, and PMD achieves significantly better AVQ values (for script set ID 9 and 10 in Table 8.6) in comparison to all signature levels used by TestOptimizer. In larger

test suites, TestOptimizer cannot produce so many combinations of potentially reusable subroutines as PMD because the overall count of subroutines is determined by a length of its chromosome and aggregation of the partial results from different runs of the analysis. After the manual analysis, TestOptimizer achieved better results than PMD in the large test set 8. The AVQ value for $signatureLevel = 0$ reached 571 and for $signatureLevel = 1$ 293 but the PMD value was 260.

In contrast to the AVQ metric, the ASQ metric (see Table 8.7) does not indicate a higher potential of PMD to deliver more fine test script suggestions than TestOptimizer. After the manual analysis of real relevance of the findings, TestOptimizer shows a better capability to localize potentially reusable subroutines that are more relevant for the user (**RQ 4**). In Table 8.7 for script sets 9 and 10, the ASQ values after the manual analysis are higher by 25 % percent on average for the most relevant $signatureLevel = 1$ from the user point of view. However, we did not observe this characteristic for other smaller test script sets.

In these experiments, we observed that all potentially reusable routines identified by TestOptimizer were relevant for the large script sets 9 and 10. This fact is a consequence of relatively high probability to hit a right combination of test steps for a subsequence in large test sets and long tests, which is simpler than in short tests in small test sets.

For **RQ 1**, we carried out similar tests as we did in the experiments with the initial prototype. We manually analyzed the suggestions for reusable subroutines to check whether they are relevant for an optimization of automated tests. Regarding the preserved functionality of automated tests, we conducted two test runs (with the original analyzed automated tests and with these tests refactored by the suggestions provided by the TestOptimizer after manual analysis of their relevance) and compared their test results if they match. These tests were performed for the test sets 6–8. The automated tests kept their functionality and results.

The experiments showed that balancing **lengthWeight** and **testCountWeight** parameters play an important and essential role for the resultant quality of candidate subroutine suggestions. Since one test step usually occurs in many tests concurrently, for example a simple command `androidDriver.findElementById("btn1").click()`, the **testCountWeight** parameter when set to a significant level start to dominate and TestOptimizer has then a tendency to search for local optimums, i.e., to search typically only two or three step subroutines. Such subroutines have a huge spread across the analyzed test suite but on the other hand, those subroutines do not make any sense from test script refactoring point of view.

Based on the results of experiments with the initial prototype, we preferred the **lengthWeight** parameter and set it to 0.95 and suppressed the complementary **testCountWeight** parameter to 0.05. We tried also the **lengthWeight** level at 0.9 and we ran another round of analysis with the **lengthWeight** parameter set to 0.995. However, these

Table 8.6: Comparisons of TestOptimizer results with the PMD benchmark using the AVQ metric.

Script Set ID	**AVQ for PMD**	**AVQ for TestOptimizer, signatureLevel = 0**	**AVQ for TestOptimizer, signatureLevel = 1**	**AVQ for TestOptimizer, signatureLevel = 2**
Raw data				
6	1697	1463	468	86
7	1421	1183	308	106
8	998	1047	302	72
9	1763	944	924	95
10	3480	1912	1045	123
Truly relevant potential subroutines after manual analysis				
6	444	609	453	81
7	302	439	302	99
8	260	571	293	69
9	948	748	924	94
10	1117	928	1045	113
Relative ratio of relevant potential subroutines				
6	23.2 %	41.6 %	96.8 %	94.2 %
7	21.3 %	37.1 %	98.1 %	93.4 %
8	26.1 %	54.5 %	97.0 %	95.8 %
9	53.8 %	79.2 %	100 %	98.9 %
10	32.1 %	48.5 %	100 %	91.9 %

test runs did not prove that additional increase of the **lengthWeight** parameter yields a higher average quality of identified reusable subroutines because all possible major subroutines were localized within the suites under test.

We also did not prove our working hypothesis that TestOptimizer may find optimal or suboptimal solutions within fewer iterations if we prefer the $lengthWeight$ parameter because it will prefer longer subsequences sooner. Increasing the **lengthWeight** parameter above the experimentally determined level (0.05) was counter–productive as TestOptimizer preferred very long subsequences that occurred only in one test. From additional TestOptimizer settings, we set the **chromosomeMode** to CHROMOSOME_AUTO and kept the major **iterationsCount** to 5 but increased the count of internal minor iterations to 10 per one major iteration.

For the subsequent manual analysis, we considered again only candidate suggestion sequences longer than 2 test steps, i.e., $p >= 3$. For the PMD toll benchmark, we used its standard configuration as set in Eclipse IDE (Luna edition). Also in the PMD tool, we

Table 8.7: A comparison of TestOptimizer results with PMD by ASQ metrics.

Script Set ID	**ASQ for PMD**	**ASQ for TestOptimizer, signatureLevel = 0**	**ASQ for TestOptimizer, signatureLevel = 1**	**ASQ for TestOptimizer, signatureLevel = 2**
Raw data				
6	242.4	243.8	91.4	30.3
7	118.4	197.2	51.3	26.5
8	58.7	130.9	37.75	12.0
9	135.6	137.2	136.1	23.8
10	165.7	216.8	105.7	25.5
Truly relevant potential subroutines after manual analysis				
6	88.8	101.5	87.7	27.8
7	50.3	173.2	50.3	24.8
8	32.5	71.4	36.6	11.5
9	86.2	106.3	136.1	23.5
10	85.9	116.9	105.7	23.8
Relative ratio of relevant potential subroutines				
6	36.6 %	41.6 %	95.9 %	91.7 %
7	42.5 %	37.1 %	98.1 %	93.4 %
8	55.4 %	54.5 %	97.0 %	95.8 %
9	63.5 %	77.5 %	100 %	98.9 %
10	51.9 %	53.9 %	100 %	93.5 %

were interested in code fragments with the minimal length of common steps equal to three.

The experiments with the final prototype showed on the ASQ raw data results that TestOptimizer achieves better results at $signatureLevel = 0$, whereas the PMD tools used as the benchmark achieves better results at $signatureLevel = 1$ and 2. However, after the manual review of the results, when irrelevant subroutines are excluded from the result set, TestOptimizer localizes more relevant subroutines.

The experiments show, that TestOptimizer identifies more truly relevant subroutines than PMD. Let's analyze this finding in more depth.

We explain the whole problem on examples of two automated tests taken from a real test automation project. Listing 8.3.2 and 8.3.2 represent parts of tests that were identified by the PMD tool as redundant. The majority of these steps are similar and have similar semantics, but these steps actually test completely different features. Furthermore, they partially use disjunctive code to test features. If the users decide to replace both fragments by one routine based on the PMD suggestion, then the user introduces a defect into tests and one feature will not be covered after this optimization anymore.

Listing 8.1: First sample Selenium WebDriver test fragment in Java used to illustrate the issue of false identified common subsequence.

```
Logger.log();
new HomePage(driver).sync();
RDBClient client = new RDBClient();
Menu menu = new Menu(driver);
menu.toggle();
menu.browseApplications().sync();
wait(1000);
menu.selectApplication("Connections");
menu.selectDetails();
Links links = menu.selectLinks();
links.syncPopulated();
links.create().setName("Relational").create();
links.selectType("SQL Database");
links.selectObjectForRelational("stored_entity");
links.setTableRelational("SAMPLES");
links.save();
wait(1000);
client.createTable();
menu.toggle();
Executor executor = menu.selectExecutors();
```

Listing 8.2: Another test fragment in Selenium WebDriver in Java with similar semantics but a completely different context.

```
Logger.log();
new HomePage(driver).sync();
Menu menu = new Menu(driver);
menu.toggle();
menu.browseApplications().sync();
wait(1000);
menu.selectApplication("Connections");
menu.selectDetails();
Links links = menu.selectLinks();
links.syncPopulated();
links.create().setName("NoSQL").create();
links.selectType("NoSQL Database");
links.selectObjectForNoSQL("stored_entity");
links.setTableNoSQL("SAMPLES");
links.save();
```

```
wait(3000);
TestConection testConection = links.testConnection();
Assert.assertTrue(links.getMessage().
  contains("Table created"))
menu.toggle();
Executor executor = menu.selectExecutors();
```

Table 8.8 summarizes the conflicts that arise when the user makes a decision to use PMD suggestion. First issue causes a command `RDBClient client = new RDBClient()`, which is an extra command related to the second test. Next conflicts are caused by steps:

```
links.selectObjectForRelational("stored_entity")
```

and

```
links.setTableRelational("SAMPLES")
```

that do not have their counter–parts in the second test and cannot be merged. Last conflict is the command `client.createTable()` that is also not present in the second test and in contrast to that, test steps

```
TestConection testConection = links.testConnection()
```

and

```
Assert.assertTrue(links.getMessage().contains("Table created"))
```

from the second test are missing. TestOptimizer searches for potentially reusable routines that are an intersection of two or more tests but not a union like the PMD tool. This property is essential if the automated test scripts shall be refactored without a loss of test coverage.

After the analysis of experimental results obtained from application of TestOptimizer to automated tests created for web–based applications (summarized in Table 8.2 and 8.3) and for mobile applications (summarized in Table 8.6 and 8.7), we can make the following conclusions:

1. The first level of the signature corresponding to $signatureLevel = 0$ is excessively abstract and cannot be used for the suggestions for a test script refactoring. Experiments with larger test suites (> 100 tests) with highly redundant test steps showed that PMD finds significantly more duplicates, i.e., potentially reusable routines, than TestOptimizer. However, we investigated after the manual analysis that a usability of such duplicates in terms of reusable routines is low, and the user is overloaded by data to process and analyze manually, which significantly decreases the user comfort of the process.

Table 8.8: Conflicts in reusable routines based on the PMD tool suggestions.

Reusable Routine Created According to PMD Suggestions	**Status**
`Logger.log();`	Valid step
`new HomePage(driver).sync();`	Valid step
`RDBClient client = new RDBClient();`	Conflict
`Menu menu = new Menu(driver);`	Valid step
`menu.toggle();`	Valid step
`menu.browseApplications().sync();`	Valid step
`wait(1000);`	Valid step
`menu.selectApplication("Connections");`	Valid step
`menu.selectDetails();`	Valid step
`Links links = menu.selectLinks();`	Valid step
`links.syncPopulated();`	Valid step
`links.create().setName("Relational").create();`	Valid step
`links.selectType("SQL Database");`	Valid step
`links.selectObjectForRelational("stored_entity");`	Conflict
`links.setTableRelational("SAMPLES");`	Conflict
`links.save();`	Valid step
`wait(1000);`	Valid step
`client.createTable();`	Conflict
`menu.toggle();`	Valid step
`Executor executor = menu.selectExecutors();`	Valid step

2. Experiments with initial and final prototype showed on AVQ and ASQ values of the results that TestOptimizer achieves the best results when $signatureLevel = 1$. Potentially reusable candidate subroutines are relevant and do not contain false test steps in contrast to the PMD tool. The used benchmark could not compete with TestOptimizer in sense of solution quality and accuracy (**RQ 4**).

3. The most specific level of signatures ($signatureLevel = 2$) is suitable for cases when the user searches for particular combinations of parameters within a limited set of actions. For other cases, the middle level is more beneficial and/or if the user does not search for more specific cases, the general purpose PMD tool may be also helpful (**RQ 4**).

4. Changes in the TestOptimizer prototype led to partial improvements observable at all signature levels but we did not achieve a major enhancement. The current prototype implementation using the LCS with one chromosome does not allow a significant improvement as the source set of test steps that could be used in common reusable routines is limited. Furthermore, the results showed that both tools reached a limit in a localization of potentially reusable routines in the set of test suites for web–based SUTs, and further improvements can be potentially achieved in a speed of searching or an accuracy of the solution rather than in a count of routines or its lengths. Table 8.9 summarizes results of application of the both prototypes to sets of automated tests 1–5, where L represents the signature level (**RQ 4**, **RQ 5**).

The results of both the initial and final prototype proved that the approach proposed in this thesis achieves better results than a general purpose tool for code static analysis. Furthermore, it can provide better accuracy and reliability in the sense of usability of such results. With the proposed approach, users spend less time to review the relevance of the suggestions for potentially reusable subroutines, which might bring an additional time–saving. Moreover, this is a critical feature because a small harm in the refactored test scripts may cause a defect leakage in the application under test.

Based on our observations, we concluded that the user can achieve this level of refactoring support with the PMD tool significantly less efficiently (**RQ 5**). With PMD, there is much higher risk of a loss of functionality in optimized automated tests.

When we conducted experiments with larger test sets that also contained highly duplicate test steps, we noticed that a manual review and processing of suggested automated test script duplications may be unfeasible in a real industrial project because of a huge amount of data. To illustrate this issue, Table 8.11 summarizes the sizes of sets of analyzed data with duplicates returned by PMD for sets of automated tests 6–9. In these sets, the user was typically required to manually review hundreds of combinations of test script duplicates. For every combination, the user needs to determine whether a suggested duplicate test script fragment is valid and matches with other test script duplicates in the given

Table 8.9: AVQ values for the initial and final prototype applied at sets of automated tests 1–5.

Script Set ID	**AVQ** $L = 0$ **Initial**	**AVQ** $L = 0$ **Final**	**Diff.**	**AVQ** $L = 1$ **Initial**	**AVQ** $L = 1$ **Final**	**Diff.**	**AVQ** $L = 2$ **Initial**	**AVQ** $L = 2$ **Final**	**Diff.**
Raw data									
1	363	398	9.6 %	176	188	6.8 %	98	103	5.1 %
2	1421	1385	2.5 %	798	798	0.0 %	181	181	0.0 %
3	197	201	2.0 %	183	177	-3.3 %	66	61	-7.5 %
4	652	670	2.8 %	328	332	1.2 &	120	124	3.3 %
5	438	449	2.5 %	300	317	5.7 %	158	166	5.1 %
Truly relevant potential subroutines after manual analysis									
1	233	243	4.3 %	172	182	5.8 %	96	100	4.2 %
2	671	675	0.6 %	798	798	0.0 %	162	162	0.0 %
3	154	157	1.9 %	179	175	-2.2 %	63	58	-7.9 %
4	425	432	1.6 %	312	324	3.6 %	112	116	1.6 %
5	259	270	4.2 %	284	298	4.9 %	155	163	5.2 %

Table 8.10: ASQ values for the initial and final prototype applied at sets of automated tests 1–5.

IDs	**ASQ** $L = 0$ **Initial**	**ASQ** $L = 0$ **Final**	**Diff.**	**ASQ** $L = 1$ **Initial**	**ASQ** $L = 1$ **Final**	**Diff.**	**ASQ** $L = 2$ **Initial**	**ASQ** $L = 2$ **Final**	**Diff.**
Raw data									
1, 6	60.5	66.3	9.6 %	35.3	37.8	6.8 %	18.9	20.2	6.6 %
2, 7	355.3	346.3	-2.5 %	199.5	199.5	0 %	90.5	90.5	0 %
3, 8	49.3	50.3	2.0 %	43.5	40.5	-6.9 %	24.8	23.5	-5.0
4, 9	163	167.5	2.7 %	130	132	1.5 %	60	62	3.3 %
5, 10	109.5	112.5	2.5 %	50	52.8	5.7 %	39.5	41.5	5.1 %
Truly relevant potential subroutines after manual analysis									
1, 6	38.8	40.5	4.3 %	34.7	37.2	7.3 %	18.4	19.4	5.4 %
2, 7	167.8	168.8	0.6 %	199.5	199.5	0 %	81	81	0 %
3, 8	38.5	39.3	1.9 %	42.8	42.2	-1.6 %	23.3	22	-5.4 %
4, 9	106.3	108	1.6 %	122	128	4.9 %	56	58	3.6 %
5, 10	64.8	67.5	4.3 %	47.3	49.7	4.9 %	38.8	40.8	5.1 %

Table 8.11: PMD Analysis Results.

Script Set ID	**Size of Results Set Describing the Code Duplicates**
6	753
7	418
8	815
9	429
10	1265

combination. That means that the overall number of test script fragments to be reviewed simply exceeds thousands of records and makes the manual processing impossible.

In contrast to that, TestOptimizer cannot produce huge sets of combinations, which is given by its proposal that is based on the analysis of test steps in one chromosome and finding its mutual combinations, but at the experimentally determined *signatureLevel* 1 achieves very good accuracy. The user can then focus on the test script refactoring itself and can optimize the test suite gradually (**RQ 5**).

From the experimental results, we could conclude that the proposed method is not sensitive to the type environment being analyzed and does not depend on the SUT but it depends more on the semantics of the programming language used in automated scripts (**RQ 6**). We analyzed the results that we measured during the experiments on the mobile platform (Table 8.6 and 8.7) with the web platform (8.2 and 8.3), and we concluded that the characteristics of TestOptimizer on the mobile platform correspond to results measured for the web platform. It suggests that TestOptimizer might be even employed on a cross–platform analysis and optimization (**RQ 6**), for example, if one set of tests is implemented in Selenium WebDriver for a web application and another set of automated tests is implemented in Robotium for an Android mobile application and the UI of the both applications is principally similar to the extent it can be modeled by the same set of abstracted test scripts.

However, this cross–platform applicability is a nice–to–have feature of the proposed approach and has its obvious limits. Namely, if the test steps are not similar to each other across the inspected test set and the steps represent actions of different environments with diverse semantics, then TestOptimizer cannot identify potentially common reusable routines of an identical business process.

8.4 Threats to Validity

Several concerns can be raised regarding the validity of the experimental data provided in this thesis, which might affect the resultant applicability of the proposed method. In this

section, we discuss these possible issues and estimate their extent, or describe a mitigation action taken to minimize these issues.

- **Relevance of identified potentially reusable subroutines**: we let expert test-automation-code developers conduct the manual analysis of the true relevance of identified potential subroutines of code fragments. Our primary objective was to find the best configuration for TestOptimizer in order to conduct an analysis of data in the most objective way. However, we could not completely suppress subjective factors during the analysis. We estimate that the subjective factors caused that up to 10% of the potential subroutines were marked as non–relevant.
- **Particular test automation platforms used in the experimental data**: for the first set of experiments with the initial prototype, we used 5 sets of real project test automation code for web applications recorded in Selenium IDE. The tests were exported to Java for Selenium WebDriver in the JUnit format. For the second experiments with the final prototype, we used the same sets of automated test scripts, extended by 5 new sets of automated tests, created for applications on mobile platforms. In this case, we used simply structured tests using the descriptive programming (verifying another possible use case of the TestOptimizer framework). Hence the results are related to these technologies and it might be possible, that the selection of the platforms affects the overall performance of the proposed method. In other languages, recording styles or test development strategies are used then the resultant efficiency of the proposed concept may be different.
- **Size of the experimental data**: Another concern may be raised regarding the size of the sets of automated tests used in the analyses. However, the analysis performed on 10 sets of tests consisting of 1514 individual automated test scripts, having 195896 lines of code in total can be considered as extensive enough sample to draw conclusions on the effectiveness of the proposed method.
- **Diversity of the set of automation developers participating in the experiments**: In the experiments, we employed the group of 30 test automation developers for the evaluation of the method efficiency. This size of the group can be considered as sufficiently extensive. However, only 5 developers were at the senior level and the majority of developers were at the junior level. The task of identifying potentially reusable subroutines in the test scripts is also appropriate for the junior expertise level, but here we might expect that senior test script developers should be faster in the identification of reusable functions. When we matched the overall performance of the senior developers with the performance of junior developers in the group, more experienced senior developers produced more relevant potential subroutines in less time that was on average by 30 %. However, even with this issue reflected, the presented results are still valid and significant.

- **PMD tool used for comparison with TestOptimizer**: A concern can be raised about the selection of PMD as a benchmark for comparison of the effectiveness and applicability of the TestOptimizer for refactoring of the automated tests. Other static analysis tools may yield different results. We selected PMD as the benchmark, as it is established, well–known, quality method for code refactoring. Moreover, considering the fact, that another alternative static analysis tools do not employ the principle abstracted automated test scripts, we can expect the results will be similar in case of comparison with another code static analysis method.

The results with the initial prototype encouraged us to pursue the experiments and to improve the whole method including searching for the optimal balance of parameters. In the experiments with the final prototype, we selected another platform that we used for the first experiments. We exercised the method with web and mobile applications, and we also changed the manner how those tests were created and replaced it by a sub–optimal manual development. Nevertheless, we are aware of the possible issue, that to the experimental results might be relevant for used test automation platforms (Java, JUnit, Selenium WebDriver and Appium) and for other platforms, the overall performance may differ.

8.5 Discussion

The presented method and its implementation on the form of TestOptimizer framework are directly aimed to search for potential reusable parts in specific test automation code. Therefore, the achieved results during the search for potential common subroutines are better than universal tools based on conventional approaches for searching of code duplications. Moreover, application of TestOptimizer compared with a manual analysis of the potential reusable parts achieves significant time savings. Although there are still some possible limitations or features of the proposed method to be discussed.

The overall efficiency of the proposed method in detecting potential common subsequences may be different for various sets of automated tests. In our experiments, we used real–project automated test scripts on two different platforms to reduce the problem of non–objective results. Conducting the experiments with automated tests created for web–based and mobile applications helped us to verify the functionality of the framework and its applicability on real industry projects.

Based on the conducted experiments, we can conclude that TestOptimizer framework performs best for:

1. recorded automated tests or automated tests that are coded in a suboptimal and non–structured (sometimes also called naive) style;

2. larger test automation projects with test sets containing typically dozens of tests or more;

3. automated tests that are expected to be run repeatedly over long time spans and with a need of good internal structure because of an expected maintenance overhead; and

4. whenever there is a requirement on the repeated refactoring of the set of automated tests.

The configuration of the Converter module influences what code of automated tests can be processed and converted to signatures. This feature is one hand a limitation of TestOptimizer because it can only process programming languages and domain–specific languages of test automation frameworks, which are currently supported by the Converter but on the other hand, it allows TestOptimizer to be flexible and dynamically extend a set of supported languages and work on the different level of parameter details.

The complexity of the problem being solved and the processing time currently limits the framework to batch processing. The real–time processing on standard hardware configurations and with the prosed architecture is unfortunately not available in the current version of the method. The batch workflow starts with uploading automated tests to the TestOptimizer server. Then the analysis is carried out as a batch job, and the user can send a query to the TestOptimizer server whether the results are ready to be downloaded. Processing time for the analyzed test sets in our experiments took up to 30 minutes. This slow response is not suitable for real–time advice in integrated development environments but as we discussed with several test automation developers, it is not considered as a roadblock for adopting this method in the test automation process. So, real–time processing remains as one of the features for the future improvements of the proposed method (see Section 9.2).

In this place we would like to quote one test automation developer giving us the feedback on the method during the experiments: *"You say, that it is worthwhile waiting ten minutes to get the results instead of doing this job manually which can take two days? You are perfectly right. But only to the point that I would not do the search for refactoring opportunities for two days. I would just not to bother with it when my boss asks for the new tests every second hour."* This practitioner's answer nicely illustrates another benefit of the proposed method. The TestOptimizer framework can help the more junior test automation developers to perform the refactoring activities in the situation, when a refactoring would not be performed at all – leading to significant maintenance costs and possibly even a fail of the test automation project relatively soon after the initial creation of the automated tests.

Improvements in the final prototype of TestOptimizer led to the partial improvement of results (Table 8.9). Since the whole framework is an expert–based system that is sensitive

to the mutual balancing of parameters used during computations, it is difficult then to find a general setting that would fit for every possible input test scripts. If it is well–tuned for some sets of test scripts, it can give significantly worse results on other sets. A resultant quality of the solution depends on many factors like the size of the test script being analyzed, a length of tests, a structure of the test set, a redundancy of test steps and others. Therefore, the settings of TestOptimizer is strongly influenced by the quality of input training data (in this case, analyzed sets of automated tests). For these purposes, it would be beneficial to implement an ability for TestOptimizer to self–reconfigure its settings based on a selection of reusable subroutines by the user.

During the experiments with large test suites (typically more than one hundred tests in the test suite with a higher level of redundancy of test steps), we observed that the current prototype implementation with one chromosome is limited in the sense of a possibility to localize more combinations of test steps in the reusable subroutines. Subroutines consisting of test steps that are not present in the chromosome cannot be identified, and thus the resultant size of the set of potentially reusable subroutines is smaller than it potentially could be in comparison to PMD. We compensate this limitation by an aggregation of results from several runs with different tests selected as the chromosome. However, this approach is not efficient from an operational time viewpoint. A possible solution to this problem may be the usage of multiple chromosomes, so the set of source test steps that are used covers the majority of test steps in the test suite.

Experiments with large test sets containing many duplications raised an issue of an ability to perform a manual analysis and processing of automated test refactoring suggestions. On such problem instances, the PMD tool produces a huge number of potentially reusable subroutine duplications (Table 8.11), which can be very difficult for a manual analysis as well as their subsequent utilization for the test script refactoring. This issue raises additional questions: what is an optimal size of the set for test set optimization? Can it be determined by some characteristics of the test set being analyzed? From our observations during experiments, hundreds of combinations of test script duplicate from PMD is too much for manual processing.

Another aspect, playing an important role in the process of optimization of automated tests is a quality of reusable subroutines. We made a decision to leave the preference for subroutines types on the user. The user decides what is his/her preference: whether long subroutines with low occurrences or shorter subroutines with high occurrences or eventually their combinations. However, the quality of suggestions is not just expressed by quantitative factors like a length, or a number of occurrences but also by characteristics like is the suggested subroutines relevant from the test optimization point of view, or are two suggested subroutines valid from the test case viewpoint (are they identical)? We concluded that PMD failed in these criteria while TestOptimizer at the experimentally determined $signatureLevel = 1$ achieved very good accuracy and the majority of suggestions were marked as relevant. With the used PMD tool as a common standard for the static code

analysis and as our benchmark, the user has to resolve many conflicts during the merging of particular suggestions into one reusable routine.

Last but not least is a question of how TestOptimizer would deal with a full stack automation involving a test automation of one use case on several platforms with different scripting languages and automation tools. Such a scenario was not in the scope of this doctoral project but it is a valid case that would be worthy of further investigations. A typical case of this scenario is a payment order in an Internet and mobile banking, where the web part is implemented, for example, in Java with Selenium WebDriver and mobile parts in Python with WebDriver (Android) and C# with Ranorex[4] (iOS). A scan of those tests can then serve as an auditing mechanism and help to eliminate hidden structural issues in the test sets. Nevertheless, a direct use case of automated test script optimization in different scripting languages is not likely.

[4]https://www.ranorex.com/mobile-automation-testing/ios-iphone-ipad-testing-automation/

Chapter 9

Conclusions

In this thesis, we propose a method addressing a major issue in test automation based on actions in the user interface of the SUT: the maintenance costs of the created automated tests when the SUT changes in the subsequent phases of the project or product development. Generally, this maintenance can be decreased by suitable structuring of the test automation code and the employment of reusable objects at various levels to minimize the code's duplication.

The main goal of the proposed approach is to decrease the expensive and resource–demanding maintenance of recorded tests or naively structured automated test scripts. Currently, test automation frequently requires the rapid preparation of automated tests and their easy updating when the front–end UI of the SUT is changed. These two requirements are, unfortunately, in practical contradiction.

In a number of industrial projects, the Record and Replay approach and the rapid unstructured programming of automated tests by an inexperienced team satisfies the first requirement for the economical creation of automated tests.

The automated identification of refactoring opportunities that is proposed in this thesis can contribute to satisfying the second requirement for the more economical maintenance of the created test code. However, the proposed method can also serve experienced test automation teams to make the refactoring process easier and to audit their test code to automatically identify new refactoring opportunities.

In the thesis, we introduced a novel approach to source code analysis that is based on an abstraction of the actions executed by the analyzed tests in the user interface of the SUT. With this method, test engineers are able to easily identify truly relevant potential reusable subroutines in the code with a higher accuracy than when using common tools for code refactoring, such as PMD. Furthermore, the proposed approach cannot cause a vendor or proprietary solution lock–in problem because it does not create any dependency of the optimized automated scripts on external tools, and nor is there any library that has to be compiled with the product code.

The results of our experiments that were performed on two initial prototypes of the proposed TestOptimizer framework with automated tests for web–based and mobile applications are promising. The employment of the framework can reduce the effort needed for the optimization of large test suites that were recorded and that would need to be updated manually to make them more robust to changes in the SUT and to decrease the subsequent potential maintenance costs. An automated process of finding potential common reusable objects in the developed test scripts helps test engineers identify possible refactoring opportunities, which saves the time that would be needed for refactoring. Moreover, it makes the refactoring process easier for less skilled test automation developers.

The TestOptimizer system takes the specific context of front–end functional automated test scripts into account so that it can achieve more relevant results than PMD or other common universal tools for searching for common subroutines in the source code.

During experiments with large test sets having more than 100 tests that also contained highly redundant test steps, we observed that a manual review of suggestions of reusable test script subroutines provided by PMD becomes unfeasible because of high amounts of data to be processed and reviewed. Furthermore, we observed that PMD used as the benchmark identifies many subroutines that are irrelevant and/or contains invalid steps causing functional conflicts in the automated tests. The user has to determine for every subroutine whether it is valid and matches with other reusable subroutines in the given combination. That means that the overall number of subroutines to be reviewed simply exceeds thousands of records and makes the manual processing impossible. In contrast to PMD, TestOptimizer achieves better results of the relative ratio of relevant potential subroutines: PMD reaches values in the interval between 45 and 85 % on average, whereas TestOptimizer reaches an interval 80 up to 100 % on average (for $signatureLevel = 1$ and $signatureLevel = 2$).

Although developers have to implement the optimizations suggested by the TestOptimizer framework, the achieved time and resource savings are still relevant.

Employing our approach significantly reduced the time required for processing linearly processed test scripts (by 65% on average). If the test suite, which is a subject of optimization, consists of at least tens of tests with redundant test steps, TestOptimizer saves time also in a direct comparison to PMD since this tool requires a significant manual pre–processing before the user can actually start to optimize the test set.

As we explained in the thesis, the framework does not change the source code of the tests automatically; it only suggests potential common subroutines, which represents an opportunity for code refactoring.

This semi–automatism is intentional, as the developers maintain complete control over the source code. Additionally, the manual verification and assessment of the suggestions provided by the framework is needed (achieving 100% relevance in the suggestions is practically impossible when we consider the fact that we do not analyze the code solely from

the text similarity viewpoint, but instead we assess the same actions that the test does in the actual user interface of the SUT).

The proposed approach employs the *signatureLevel* concept in which we can specify the required details of the conducted search for potential common subroutines. During the experiments, we observed the *signatureLevel 1* that defines actions and elements to be the most efficient level of abstraction. At this level, we do not reflect any particular testing data.

We designed the TestOptimizer framework as an open and scalable method that processes the analyzed automated test scripts in a modular pipeline. It enables future extensions of the system and the tailoring of the individual units that are participating in the test code analysis. An integration of the method can be achieved with different IDEs, such as IntelliJ IDEA or Eclipse, Selenium WebDriver IDE and others, through plugins. If support for another language, such as Python or C++, is required, it can be achieved by adding a new parser module into the pipeline. Since the principles of the analysis do not change, the remaining modules can be used.

9.1 Contributions of this Dissertation Thesis

The contributions of the Dissertations Thesis can be summarized as follows. From the **research and development viewpoint**, we can present the following achievements.

1. The proposal of an innovative approach for the automated refactoring of code duplications in automated tests, which leads to a better structuring of these tests and lower subsequent maintenance costs. As a consequence, the proposed method enables the quick development of automated tests combined with the reasonable maintenance of the created tests from financial and time viewpoints. It also enables the optimization of the test automation code that is created in the descriptive programming style.

2. The proposal of the model of the automated test scripts allows for the abstraction of the test automation code. The model uses signatures that are suitable for automated processing and exhibits two major features:

    a) The model removes the dependencies of the proposed approach on a particular language of the test scripts, test automation framework or test automation API; and

    b) The abstract code model also gives the method better efficiency in its search for truly relevant potential subroutines, since the actual actions that are performed by the automated test are analyzed, which is in contrast to the traditional static analysis approaches that detect potential code duplications.

3. A set of parameterizable algorithms that search for potential common subroutines in analyzed test code, abstracted by the defined model. In this section, the genetic algorithm [95] is adopted to solve the problem.

4. Two functional prototypes (initial and final prototype) of the proposed TestOptimizer framework that implement the proposed method and are designed to be highly applicable for industry projects due to their flexible integration with test–code development IDEs and their modular structure, which allows support for other test automation languages or APIs to be added.

5. Data and practical lessons learned from a set of experiments verifying the applicability of the created framework and from assessing its benefits and possible limits. These conclusions provide better insight into the framework's strengths, including algorithmic configuration, and contribute to its better future applicability.

6. This doctoral project focused on areas that are insufficiently covered by previous research but that are highly relevant to industrial praxis. While individual prior attempts to optimize suboptimally structured test automation code can be found, for instance BlackHorse project [1], the goals and conceptual details of BlackHorse differ, as explained in the Related Work in Section 3.2. TestOptimizer's openness, test automation platform independence, and the independence of the optimized test automation code supported by the proposed framework represent an original method in this area.

From an **industrial applicability viewpoint**, we present the following highlights and features of the proposed method and the TestOptimizer framework:

1. From a practical viewpoint, the main contribution of the proposed method and developed framework is that it reduces the refactoring time required by automated tests to identify and remove code redundancies. Such refactoring is widely recognized as a best–practice method to improve the robustness of automated tests against changes in the user interface of the SUT–refactoring contributes to simpler and less resource–intensive test maintenance.

2. Due to the maintenance effort reduction required to maintain the synchronization of automated test with the current state of the SUT, the TestOptimizer framework extends the use of automated tests beyond regression tests with a high number of test runs; they can be applied to projects with a low number of repetitions to benefit from automation on a broader basis. Because our approach reduces the time required for test script maintenance and updates, tests can be updated more quickly following changes in the SUT. Hence, the tests may find regression defects in the SUT faster. Consequently, this approach introduces an opportunity to improve the

overall efficiency of the testing process and reduce the production risks arising from SUT operation because the SUT can be tested more extensively in less time.

3. The method and framework can be used in common projects in the software industry without any special requirements on a technical equipment nor skill set of test automation engineers. The proposed method allows even software developers without any experience with test automation to create tests which are optimized by the suggestions given by the TestOptimizer framework.

4. In the TestOptimizer framework, the user has full control over the refactoring suggestion process and can tailor it according to the needs of the particular test automation project.

5. The TestOptimizer framework can be used to refactor and subsequently optimize recorded automated tests or suboptimally structured tests created by descriptive programming methods. In addition, it can be used as a support tool to conduct audits of automated test scripts created by more sophisticated methods.

6. The TestOptimizer framework is platform–agnostic and can be used to refactor user–interface–based automated tests created for web applications as well as for mobile or thick–client applications. The current final prototype of the framework provides support for Java, Selenium WebDriver for web applications and Appium for mobile applications. Due to the modular framework structure the, adding support for another test automation language or API simply involves creating another Converter module.

9.2 Future Work

Currently, we are working on improving several areas of the proposed method and framework and on adding future features. First, are developing a future extension (TBD) to the Converter module that would be capable of parsing the latest features in Java 8 and 9 (for example, lambda functions and factory methods) and creating abstractions based on the code written to those Java specifications. Furthermore, our goal is to continuously improve the selection of a set of prospective tests that may help to achieve better results.

Moreover, we plan to enhance the abstraction layer to work with recent tests written in asynchronous JavaScript. Although JavaScript syntax is similar to Java syntax, the TestOptimizer framework cannot process the test scripts because it does not support the features for synchronous test execution, such as callbacks, promises or arrow functions. Last but not least, there is a need to optimize computations related to the analysis of the automated test scripts to obtain relevant results at a speed closer to real time. Such an

improvement would result in increased user comfort and further improve the applicability of the method.

A potentially very promising feature is fully automated optimization that requires no human intervention. This feature would allow the system to scan the recorded or generally available tests and propose (i) an optimal test structure and (ii) optimal system test coverage when attached to a model of the SUT. Achieving fully automated optimization requires solving a number of issues, such as having a testing context available (i.e., a knowledge of test requirements and their relations to tests that influenced the initial test proposal and the structure of test suites). However, due to the complexity of the problem, fully automated test optimization is outside the scope of this doctoral project. We plan to perform future research and development in this area.

Another field of interest would be a TestOptimizer extension to integrate it with mainstream Integrated Development Environments (IDEs) and version control systems. This feature would simplify working with the tool and allow it to become an essential part of the test development process. Tight integration with IDEs is another possibility for future improvements, as is adding support for other test execution engines such as Mocha and automation frameworks such as the Robot framework.

Last but not least, the usability of the method rises significantly when it can produce results in real time. The current implementation of the framework does not support parallel processing; however, adding support for parallelism is a source of major potential future improvements. Current technologies such as Apache Hadoop allow the processing of large amounts of unstructured and distributed data; thus, finding a mechanism to segment the analyzed automated test sets into parts that can be processed in parallel or finding ways to run the analysis in parallel on multiple nodes would improve the TestOptimizer response time.

Bibliography

[1] Carino, S.; Andrews, J. H.; Goulding, S.; et al. BlackHorse: Creating smart test cases from brittle recorded tests. In *2012 7th International Workshop on Automation of Software Test (AST)*, June 2012, pp. 89–95, doi:10.1109/IWAST.2012.6228996.

[2] Furht, B.; (Eds.), A. E. *Handbook of Cloud Computing*. Germany: Springer Berlin Heidelberg, first edition, 2010, ISBN 978-1-4419-6524-0.

[3] M., C.; (Eds.), A. G. *Grids, Clouds and Virtualization*. Germany: Springer Berlin Heidelberg, first edition, 2009, ISBN 978-0-85729-049-6.

[4] Blokdijk, G. *SaaS 100 Success Secrets - How Companies Successfully Buy, Manage, Host and Deliver Software As a Service (SaaS)*. USA: Emereo Pty Ltd, 2008, ISBN 90-386-0564-1.

[5] Eldh, S.; Hansson, H.; Punnekkat, S.; et al. A Framework for Comparing Efficiency, Effectiveness and Applicability of Software Testing Techniques. In *Testing: Academic Industrial Conference - Practice And Research Techniques (TAIC PART'06)*, Aug 2006, pp. 159–170, doi:10.1109/TAIC-PART.2006.1.

[6] Bures, M. Automated Testing in the Czech Republic: The Current Situation and Issues. In *Proceedings of the 15th International Conference on Computer Systems and Technologies*, CompSysTech '14, New York, NY, USA: ACM, 2014, ISBN 978-1-4503-2753-4, pp. 294–301, doi:10.1145/2659532.2659605. Available from: `http://doi.acm.org/10.1145/2659532.2659605`

[7] Rafi, D. M.; Moses, K. R. K.; Petersen, K.; et al. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In *2012 7th International Workshop on Automation of Software Test (AST)*, June 2012, pp. 36–42, doi:10.1109/IWAST.2012.6228988.

[8] Kasurinen, J.; Taipale, O.; Smolander, K. Software Test Automation in Practice: Empirical Observations. *Adv. Soft. Eng.*, volume 2010, Jan. 2010: pp. 4:1–4:13, ISSN 1687-8655, doi:10.1155/2010/620836. Available from: http://dx.doi.org/10.1155/2010/620836

[9] Pettichord, B. Seven Steps to Test Automation Success. 1999, sTAR West.

[10] Hoffman, D. Cost Benefits Analysis of Test Automation. 2009, sTAR West.

[11] Rivero, J. M.; Rossi, G.; Grigera, J.; et al. From mockups to user interface models: an extensible model driven approach. In *Proceedings of the 10th international conference on Current trends in web engineering*, ICWE'10, Berlin, Heidelberg: Springer-Verlag, 2010, ISBN 3-642-16984-8, 978-3-642-16984-7, pp. 13–24.

[12] Xu, D. A tool for automated test code generation from high-level petri nets. In *Proceedings of the 32nd international conference on Applications and theory of Petri Nets*, PETRI NETS'11, Berlin, Heidelberg: Springer-Verlag, 2011, ISBN 978-3-642-21833-0, pp. 308–317.

[13] Martin, R. C. *Agile Software Development, Principles, Patterns, and Practices*. USA: Prentice Hall, first edition, 2002, ISBN 978-0135974445.

[14] Berner, S.; Weber, R.; Keller, R. K. Observations and Lessons Learned from Automated Testing. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, New York, NY, USA: ACM, 2005, ISBN 1-58113-963-2, pp. 571–579, doi:10.1145/1062455.1062556. Available from: http://doi.acm.org/10.1145/1062455.1062556

[15] Kresse, A.; Kruse, P. M. Development and Maintenance Efforts Testing Graphical User Interfaces: A Comparison. In *Proceedings of the 7th International Workshop on Automating Test Case Design, Selection, and Evaluation*, A-TEST 2016, New York, NY, USA: ACM, 2016, ISBN 978-1-4503-4401-2, pp. 52–58, doi:10.1145/2994291.2994299. Available from: http://doi.acm.org/10.1145/2994291.2994299

[16] Boehm, B. W. Software engineering economics. *IEEE transactions on Software Engineering*, , no. 1, 1984: pp. 4–21.

[17] Boehm, B. W. Software engineering economics. In *Software pioneers*, Springer, 2002, pp. 641–686.

[18] Beizer, B. *Software Testing Techniques (2Nd Ed.)*. New York, NY, USA: Van Nostrand Reinhold Co., 1990, ISBN 0-442-20672-0.

[19] Ramler, R.; Wolfmaier, K. Economic Perspectives in Test Automation: Balancing Automated and Manual Testing with Opportunity Cost. In *Proceedings of the 2006 International Workshop on Automation of Software Test*, AST '06, New York, NY, USA: ACM, 2006, ISBN 1-59593-408-1, pp. 85–91, doi:10.1145/1138929.1138946. Available from: http://doi.acm.org/10.1145/1138929.1138946

[20] Kumar, D.; Mishra, K. The Impacts of Test Automation on Software's Cost, Quality and Time to Market. *Procedia Computer Science*, volume 79, 2016: pp. 8 – 15, ISSN 1877-0509, doi:https://doi.org/10.1016/j.procs.2016.03.003, proceedings of International Conference on Communication, Computing and Virtualization (ICCCV) 2016. Available from: http://www.sciencedirect.com/science/article/pii/S1877050916001277

[21] Ahmed, B. S.; Zamli, K. Z.; Afzal, W.; et al. Constrained interaction testing: A systematic literature study. *IEEE Access*, volume 5, 2017: pp. 25706–25730.

[22] Kaner, C.; Bach, J. B.; Pettichord, B. *Lessons Learned in Software Testing: A Context-Driven Approach*. USA: Wiley, 2002, ISBN 978-0-471-08112-8.

[23] Doležel, M. Images of Enterprise Test Organizations: Factory, Center of Excellence, or Community? In *Software Quality. Complexity and Challenges of Software Engineering in Emerging Technologies*, edited by D. Winkler; S. Biffl; J. Bergsmann, Cham: Springer International Publishing, 2017, ISBN 978-3-319-49421-0, pp. 105–116.

[24] Fewster, M.; Graham, D. *Software Test Automation: Effective Use of Test Execution Tools*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1999, ISBN 0-201-33140-3.

[25] Persson, C.; Yilmazturk, N. Establishment of Automated Regression Testing at ABB: Industrial Experience Report on 'Avoiding the Pitfalls'. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering*, ASE '04, Washington, DC, USA: IEEE Computer Society, 2004, ISBN 0-7695-2131-2, pp. 112–121, doi:10.1109/ASE.2004.33. Available from: http://dx.doi.org/10.1109/ASE.2004.33

[26] Alégroth, E.; Nass, M.; Olsson, H. H. JAutomate: A Tool for System- and Acceptance-test Automation. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, March 2013, ISSN 2159-4848, pp. 439–446, doi:10.1109/ICST.2013.61.

[27] Fewster, M. Common Mistakes in Test Automation. Technical report, Groove consultants, 01 2001. Available from: http://www.agileconnection.com/sites/default/files/article/file/2012/XDD2901filelistfilename1_0.pdf

[28] Lonngren, D. D. Reducing the cost of test through reuse. In *1998 IEEE AUTOTESTCON Proceedings. IEEE Systems Readiness Technology Conference. Test Technology for the 21st Century (Cat. No.98CH36179)*, Aug 1998, ISSN 1088-7725, pp. 48–53, doi:10.1109/AUTEST.1998.713419.

[29] Lanubile, F.; Mallardo, T. Inspecting automated test code: a preliminary study. In *Proceedings of the 8th international conference on Agile processes in software engineering and extreme programming*, XP'07, Berlin, Heidelberg: Springer-Verlag, 2007, ISBN 978-3-540-73100-9, pp. 115–122.

[30] Fujiwara, S.; Munakata, K.; Maeda, Y.; et al. Test data generation for web application using a UML class diagram with OCL constraints. *Innovations in Systems and Software Engineering*, volume 7, 2011: pp. 275–282, ISSN 1614-5046.

[31] Sabharwal, S.; Sibal, R.; Sharma, C. A genetic algorithm based approach for prioritization of test case scenarios in static testing. In *Computer and Communication Technology (ICCCT), 2011 2nd International Conference on*, sept. 2011, pp. 304 –309.

[32] Palomba, F.; Panichella, A.; Zaidman, A.; et al. Automatic Test Case Generation: What if Test Code Quality Matters? In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, New York, NY, USA: ACM, 2016, ISBN 978-1-4503-4390-9, pp. 130–141, doi:10.1145/2931037.2931057. Available from: `http://doi.acm.org/10.1145/2931037.2931057`

[33] Fellner, A.; Krenn, W.; Schlick, R.; et al. Model-based, Mutation-driven Test Case Generation via Heuristic-guided Branching Search. In *Proceedings of the 15th ACM-IEEE International Conference on Formal Methods and Models for System Design*, MEMOCODE '17, New York, NY, USA: ACM, 2017, ISBN 978-1-4503-5093-8, pp. 56–66, doi:10.1145/3127041.3127049. Available from: `http://doi.acm.org/10.1145/3127041.3127049`

[34] Ramler, R.; Klammer, C.; Buchgeher, G. Applying Automated Test Case Generation in Industry: A Retrospective. In *2018 IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, April 2018, pp. 364–369, doi:10.1109/ICSTW.2018.00074.

[35] García, B. *Contribution to the Automation of Software Quality Control of Web Applications*. Dissertation thesis, Universidad Politécnica de Madrid, Spain, 2011.

[36] Collins, E. F.; de Lucena, V. F., Jr. Software Test Automation Practices in Agile Development Environment: An Industry Experience Report. In *Proceedings of the 7th International Workshop on Automation of Software Test*, AST '12, Piscataway,

NJ, USA: IEEE Press, 2012, ISBN 978-1-4673-1822-8, pp. 57–63. Available from: http://dl.acm.org/citation.cfm?id=2663608.2663620

[37] Crispin, L.; Gregory, J. *Agile Testing: A Practical Guide for Testers and Agile Teams*. USA: Addison-Wesley Professional, first edition, 2008, ISBN 978-0321534460.

[38] Meszaros, G. Agile Regression Testing Using Record & Playback. In *Companion of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '03, New York, NY, USA: ACM, 2003, ISBN 1-58113-751-6, pp. 353–360, doi:10.1145/949344.949442. Available from: http://doi.acm.org/10.1145/949344.949442

[39] Al-Zain, S.; Eleyan, D.; Garfield, J. Automated User Interface Testing for Web Applications and TestComplete. In *Proceedings of the CUBE International Information Technology Conference*, CUBE '12, New York, NY, USA: ACM, 2012, ISBN 978-1-4503-1185-4, pp. 350–354, doi:10.1145/2381716.2381782. Available from: http://doi.acm.org/10.1145/2381716.2381782

[40] Lam, W.; Wu, Z.; Li, D.; et al. Record and Replay for Android: Are We There Yet in Industrial Cases? In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, New York, NY, USA: ACM, 2017, ISBN 978-1-4503-5105-8, pp. 854–859, doi:10.1145/3106237.3117769. Available from: http://doi.acm.org/10.1145/3106237.3117769

[41] Hammoudi, M. Regression Testing of Web Applications Using Record/Replay Tools. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, New York, NY, USA: ACM, 2016, ISBN 978-1-4503-4218-6, pp. 1079–1081, doi:10.1145/2950290.2983942. Available from: http://doi.acm.org/10.1145/2950290.2983942

[42] Agarwal, T. A Descriptive Programming Based Approach for Test Automation. In *2008 International Conference on Computational Intelligence for Modelling Control Automation*, Dec 2008, pp. 152–156, doi:10.1109/CIMCA.2008.132.

[43] Tanida, H.; Prasad, M. R.; Rajan, S. P.; et al. *Automated System Testing of Dynamic Web Applications*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, ISBN 978-3-642-36177-7, pp. 181–196, doi:10.1007/978-3-642-36177-7_12. Available from: http://dx.doi.org/10.1007/978-3-642-36177-7_12

[44] Dallmeier, V.; Burger, M.; Orth, T.; et al. *WebMate: Generating Test Cases for Web 2.0*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, ISBN 978-3-642-35702-2, pp. 55–69, doi:10.1007/978-3-642-35702-2_5. Available from: http://dx.doi.org/10.1007/978-3-642-35702-2_5

[45] Memon, A. M. Automatically Repairing Event Sequence-based GUI Test Suites for Regression Testing. *ACM Trans. Softw. Eng. Methodol.*, volume 18, no. 2, Nov. 2008: pp. 4:1–4:36, ISSN 1049-331X, doi:10.1145/1416563.1416564. Available from: `http://doi.acm.org/10.1145/1416563.1416564`

[46] Bures, M. Framework for assessment of web application automated testability. In *Proceedings of the 2015 Conference on research in adaptive and convergent systems*, ACM, 2015, pp. 512–514.

[47] Bures, M. Metrics for automated testability of web applications. In *ACM International Conference Proceeding Series*, volume 1008, 2015, pp. 83–89, doi:10.1145/2812428.2812458.

[48] Bures, M. Model for evaluation and cost estimations of the automated testing architecture. In *New Contributions in Information Systems and Technologies*, Springer, 2015, pp. 781–787.

[49] Bures, M. Change Detection System for the Maintenance of Automated Testing. In *IFIP International Conference on Testing Software and Systems*, Springer, 2014, pp. 192–197.

[50] Feather, M. S.; Smith, B. Automatic Generation of Test Oracles – From Pilot Studies to Application. *Automated Software Engg.*, volume 8, no. 1, Jan. 2001: pp. 31–61, ISSN 0928-8910.

[51] Smaragdakis, Y.; Csallner, C.; Subramanian, R. Scalable automatic test data generation from modeling diagrams. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, New York, NY, USA: ACM, 2007, ISBN 978-1-59593-882-4, pp. 4–13.

[52] Fowler, M. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. USA: Addison-Wesley Professional, third edition, 2003, ISBN 978-0321193681.

[53] Peleska, J.; Vorobev, E.; Lapschies, F. Automated test case generation with SMT-solving and abstract interpretation. In *Proceedings of the Third international conference on NASA Formal methods*, NFM'11, Berlin, Heidelberg: Springer-Verlag, 2011, ISBN 978-3-642-20397-8, pp. 298–312.

[54] Koopman, P.; Achten, P.; Plasmeijer, R. Model-Based Testing of Thin-Client Web Applications and Navigation Input. In *Practical Aspects of Declarative Languages*, *Lecture Notes in Computer Science*, volume 4902, edited by P. Hudak; D. Warren, Springer Berlin Heidelberg, 2008, ISBN 978-3-540-77441-9, pp. 299–315.

[55] Plasmeijer, R.; Achten, P. The Implementation of iData - A Case Study in Generic Programming. Technical report, Proceedings Implementation and Application of Functional Languages, 17th International Workshop, IFL05, 2005.

[56] Plasmeijer, R.; Achten, P. iData for the world wide web – programming interconnected web forms. In *Proceedings of the 8th international conference on Functional and Logic Programming*, FLOPS'06, Berlin, Heidelberg: Springer-Verlag, 2006, ISBN 3-540-33438-6, 978-3-540-33438-5, pp. 242–258.

[57] van Beek, H. M. A.; Mauw, S. Automatic Conformance Testing of Internet Applications. In *Formal Approaches to Software Testing*, *Lecture Notes in Computer Science*, volume 2931, edited by A. Petrenko; A. Ulrich, Springer Berlin Heidelberg, 2004, ISBN 978-3-540-20894-5, pp. 205–222.

[58] van Beek, H. M. A. *Specification and Analysis of Internet Applications*. Dissertation thesis, Technical University Eindhoven, The Netherlands, 2005.

[59] Besson, F.; Beder, D.; Chaim, M. An Automated Approach for Acceptance Web Test Case Modeling and Executing. In *Agile Processes in Software Engineering and Extreme Programming*, *Lecture Notes in Business Information Processing*, volume 48, edited by A. Sillitti; A. Martin; X. Wang; E. Whitworth, Springer Berlin Heidelberg, 2010, ISBN 978-3-642-13053-3, pp. 160–165.

[60] Andrews, A. A.; Offutt, J.; Alexander, R. T. Testing Web applications by modeling with FSMs. *Software & Systems Modeling*, volume 4, 2005: pp. 326–345, ISSN 1619-1366.

[61] Stepien, B.; Peyton, L.; Xiong, P. Framework testing of web applications using TTCN-3. *International Journal on Software Tools for Technology Transfer (STTT)*, volume 10, no. 4, July 2008: pp. 371–381, ISSN 1433-2779.

[62] ETSI ES 201 873-1. The Testing and Test ControlNotation version 3. 2007, part1: TTCN-3 Core notation, V3.2.1.

[63] Jia, X.; Liu, H. Rigorous and Automatic Testing of Web Application. In *Proceedings of the 6th IASTED International Conference on Software Engineering and Applications (SEA 2002)*, Cambridge, MA, USA, 2002, pp. 280–285.

[64] Niese, O.; Margaria, T.; Steffen, B. Automated Functional Testing of Web-based Applications. In *Proceedings 5th Int. Conference On Software and Internet Quality Week Europe (QWE 2002)*, 2002, pp. 157–166, belgium.

[65] Benedikt, M.; Freire, J.; Godefroid, P. VeriWeb: Automatically Testing Dynamic Web Sites. In *Proceedings of the 11th International World Wide Web conference (WWW2002)*, 2002, honolulu, USA.

[66] Escalona, M. J.; Aragón, G. NDT. A Model-Driven Approach for Web Requirements. *IEEE Trans. Softw. Eng.*, volume 34, no. 3, May 2008: pp. 377–390, ISSN 0098-5589.

[67] García, B.; Duenas, J. C. Automated Functional Testing based on the Navigation of Web Applications. *WWV 2011*, 2011, reykjavik, Iceland.

[68] García, B.; Duenas, J. C.; Parada, H. A. Functional Testing based on Web Navigation with Contracts. In *Proceedings IADIS International Conference (WWW/INTERNET09)*, Rome, Italy, 2009.

[69] Frajták, K.; Bures, M.; Jelínek, I. Using the Interaction Flow Modelling Language for Generation of Automated Front-End Tests. In *FedCSIS Position Papers*, 2015, pp. 117–122.

[70] Frajták, K.; Bureš, M.; Jelínek, I. Transformation of IFML schemas to automated tests. In *Proceedings of the 2015 Conference on research in adaptive and convergent systems*, ACM, 2015, pp. 509–511.

[71] Kawakami, L.; Knabben, A.; Rechia, D.; et al. An Object-oriented Framework for Improving Software Reuse on Automated Testing of Mobile Phones. In *Proceedings of the 19th IFIP TC6/WG6.1 International Conference, and 7th International Conference on Testing of Software and Communicating Systems*, TestCom'07/FATES'07, Berlin, Heidelberg: Springer-Verlag, 2007, ISBN 3-540-73065-6, 978-3-540-73065-1, pp. 199–211. Available from: `http://dl.acm.org/citation.cfm?id=2391293.2391307`

[72] Nguyen, D. H.; Strooper, P.; Süß, J. G. Automated Functionality Testing Through GUIs. In *Proceedings of the Thirty-Third Australasian Conferenc on Computer Science - Volume 102*, ACSC '10, Darlinghurst, Australia, Australia: Australian Computer Society, Inc., 2010, ISBN 978-1-920682-83-5, pp. 153–162. Available from: `http://dl.acm.org/citation.cfm?id=1862199.1862216`

[73] Mu, B.; Zhan, M.; Hu, L. Design and Implementation of GUI Automated Testing Framework Based on XML. In *2009 WRI World Congress on Software Engineering*, volume 4, May 2009, pp. 194–199, doi:10.1109/WCSE.2009.91.

[74] Kaur, M.; Kumari, R. Comparative Study of Automated Testing Tools: TestComplete and QuickTest Pro. volume 24, 06 2011.

[75] Kaur, H.; Gupta, G. Comparative Study of Automated Testing Tools: Selenium, Quick Test Professional and Testcomplete. volume 3, 09 2013: pp. 1739–1743.

[76] Kan, H.-x.; Wang, G.-q.; Wang, Z.-d.; et al. A method of minimum reusability estimation for automated software testing. *Journal of Shanghai Jiaotong University*

(Science), volume 18, no. 3, Jun 2013: pp. 360–365, ISSN 1995-8188, doi:10.1007/s12204-013-1406-1. Available from: https://doi.org/10.1007/s12204-013-1406-1

[77] Meszaros, G. *XUnit Test Patterns: Refactoring Test Code*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2006, ISBN 0131495054.

[78] Deursen, A.; Moonen, L. M.; Bergh, A.; et al. Refactoring Test Code. Technical report, Amsterdam, The Netherlands, The Netherlands, 2001.

[79] Guerra, E. M.; Fernandes, C. T. Refactoring Test Code Safely. In *International Conference on Software Engineering Advances (ICSEA 2007)*, Aug 2007, pp. 44–44, doi:10.1109/ICSEA.2007.57.

[80] Janzen, D.; Saiedian, H. Test-driven development concepts, taxonomy, and future direction. *Computer*, volume 38, no. 9, Sept 2005: pp. 43–50, ISSN 0018-9162, doi:10.1109/MC.2005.314.

[81] McMahon, C. History of a Large Test Automation Project Using Selenium. In *2009 Agile Conference*, Aug 2009, pp. 363–368, doi:10.1109/AGILE.2009.9.

[82] Fang, Z. F.; Lam, P. Identifying Test Refactoring Candidates with Assertion Fingerprints. In *Proceedings of the Principles and Practices of Programming on The Java Platform*, PPPJ '15, New York, NY, USA: ACM, 2015, ISBN 978-1-4503-3712-0, pp. 125–137, doi:10.1145/2807426.2807437. Available from: http://doi.acm.org/10.1145/2807426.2807437

[83] Vahabzadeh, A.; Stocco, A.; Mesbah, A. Fine-grained Test Minimization. In *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, New York, NY, USA: ACM, 2018, ISBN 978-1-4503-5638-1, pp. 210–221, doi:10.1145/3180155.3180203. Available from: http://doi.acm.org/10.1145/3180155.3180203

[84] Kumar, B.; Singh, K. Testing UML Designs using Class, Sequence and Activity Diagrams. In *International Journal for Innovative Research in Science & Technology*, volume 2.3, 2015, pp. 71–81.

[85] Chen, W.; Wang, J. Bad Smells and Refactoring Methods for GUI Test Scripts. In *2012 13th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, Aug 2012, pp. 289–294, doi:10.1109/SNPD.2012.10.

[86] Bladel, B. v.; Demeyer, S. Test Behaviour Detection As a Test Refactoring Safety. In *Proceedings of the 2Nd International Workshop on Refactoring*, IWoR

2018, New York, NY, USA: ACM, 2018, ISBN 978-1-4503-5974-0, pp. 22–25, doi:10.1145/3242163.3242168. Available from: http://doi.acm.org/10.1145/3242163.3242168

[87] Yue, T.; Ali, S.; Briand, L. Automated Transition from Use Cases to UML State Machines to Support State-Based Testing. In *Modelling Foundations and Applications*, edited by R. B. France; J. M. Kuester; B. Bordbar; R. F. Paige, Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, ISBN 978-3-642-21470-7, pp. 115–131.

[88] Lipka, R.; Potuzak, T.; Brada, P.; et al. A Method for Semi-automated Generation of Test Scenarios Based on Use Cases. In *2015 41st Euromicro Conference on Software Engineering and Advanced Applications*, Aug 2015, ISSN 1089-6503, pp. 241–244, doi:10.1109/SEAA.2015.32.

[89] Potuzak, T.; Lipka, R.; Brada, P. Interface-based semi-automated testing of software components. *2017 Federated Conference on Computer Science and Information Systems (FedCSIS)*, 2017: pp. 1335–1344.

[90] Dobolyi, K.; Soechting, E.; Weimer, W. Automating regression testing using web-based application similarities. *International Journal on Software Tools for Technology Transfer*, volume 13, no. 2, Apr 2011: pp. 111–129, ISSN 1433-2787, doi:10.1007/s10009-010-0170-x. Available from: https://doi.org/10.1007/s10009-010-0170-x

[91] Hirschberg, D. S. Algorithms for the Longest Common Subsequence Problem. *J. ACM*, volume 24, no. 4, Oct. 1977: pp. 664–675, ISSN 0004-5411, doi:10.1145/322033.322044. Available from: http://doi.acm.org/10.1145/322033.322044

[92] S. Baker, B. A Program for Identifying Duplicated Code. volume 24, 07 1992.

[93] Pessoa, T.; Brito e Abreu, F.; Monteiro, M.; et al. An Eclipse Plugin to Support Code Smells Detection. 04 2012.

[94] Hamid, A.; Ilyas, M.; Hummayun, M.; et al. A Comparative Study on Code Smell Detection Tools. volume 60, 11 2013: pp. 25–32.

[95] Julstrom, B. A.; Hinkemeyer, B. Starting from Scratch: Growing Longest Common Subsequences with Evolution. In *Parallel Problem Solving from Nature - PPSN IX*, edited by T. P. Runarsson; H.-G. Beyer; E. Burke; J. J. Merelo-Guervós; L. D. Whitley; X. Yao, Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, ISBN 978-3-540-38991-0, pp. 930–938.

[96] Hinkemeyer, B.; Julstrom, B. A. A genetic algorithm for the longest common subsequence problem. In *Proceedings of the 8th annual conference on Genetic and evolutionary computation*, GECCO '06, New York, NY, USA: ACM, 2006, ISBN 1-59593-186-4, pp. 609–610.

[97] Boyer, R. S.; Moore, J. S. A fast string searching algorithm. *Commun. ACM*, volume 20, no. 10, Oct. 1977: pp. 762–772, ISSN 0001-0782.

[98] Karp, R. M.; Rabin, M. O. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, volume 31, no. 2, Mar. 1987: pp. 249–260, ISSN 0018-8646.

[99] Knuth, D. E.; Morris, J. J. H.; Pratt, V. R. Fast Pattern Matching in Strings. *SIAM Journal on Computing*, volume 6, no. 2, 1977: pp. 323–350.

[100] Zhu, H.; Yang, X.; Wang, B.; et al. Improving Text Search on Hybrid Data. In *Web-Age Information Management*, *Lecture Notes in Computer Science*, volume 7419, edited by Z. Bao; Y. Gao; Y. Gu; L. Guo; Y. Li; J. Lu; Z. Ren; C. Wang; X. Zhang, Springer Berlin Heidelberg, 2012, ISBN 978-3-642-33049-0, pp. 192–203.

[101] Luk, R. W.; Leong, H. V.; Dillon, T. S.; et al. A survey in indexing and searching XML documents. *J. Am. Soc. Inf. Sci. Technol.*, volume 53, no. 6, May 2002: pp. 415–437, ISSN 1532-2882.

[102] Yang, J.; Korfhage, R. R. Effects of Query Term Weights Modification in Annual Document Retrieval: A Study Based on a Genetic Algorithm. In *Proceedings of the Second Sym Symposium on Document Analysis and Information Retrieval*, 1993, pp. 271–285.

[103] Srinivasa, K. G.; Sharath, S.; Venugopal, K. R.; et al. GaXsearch: An XML Information Retrieval Mechanism Using Genetic Algorithms. In *Australian Conference on Artificial Intelligence*, 2005, pp. 435–444.

[104] Wagner, R. A.; Fischer, M. J. The String-to-String Correction Problem. *J. ACM*, volume 21, no. 1, Jan. 1974: pp. 168–173, ISSN 0004-5411, doi:10.1145/321796.321811. Available from: `http://doi.acm.org/10.1145/321796.321811`

Publications of the Author Directly Related to This Thesis

Article in Impacted Journal

[A.1] Bureš, M; Filipský, M.; Jelínek, I. Identification of Potential Reusable Subroutines in Recorded Automated Test Scripts. In: *International Journal of Software Engineering and Knowledge Engineering*. Vol. 28, No. 01. pp. 3-36. ISSN 0218-1940. 2018. (Q4, IF 0.4) [33%]

Article in Peer-reviewed Journal Indexed in WoS

[A.2] Filipský, M.; Bureš, M; Jelínek, I. Automated Optimization of Functional Recorded Tests. In: *International Journal on Information Technologies and Security*. Vol. 7, No. 01, pp. 3-14. ISSN 1313-8251. 2015. [33%]

Conference Proceedings Paper Indexed in WoS

[A.3] Filipský, M.; Bureš, M; Jelínek, I. Creating Smart Tests from Recorded Automated Test Cases. In: *ROCHA, A., et al., eds. New Contributions in Information Systems and Technologies*. 3rd World Conference on Information Systems and Technologies. Ponta Delgada, Portugal. Advances in Intelligent Systems and Computing, Heidelberg: Springer. pp. 773-780. ISSN 2194-5357. ISBN 978-3-319-16485-4. 2015. [33%]

Other Conference Proceedings Papers

[A.4] Filipský, M.; Bureš, M; Jelínek, I. Finding Common Subsequences in Recorded Test Cases. In: *Proceedings of ICSEA 2013: The Eighth International Conference on*

Software Engineering Advances. The Eighth International Conference on Software Engineering Advances. Venice, Italy. Advances in Intelligent Systems and Computing, Wilmington: IARIA. pp. 51-54. ISSN 2308-4235. ISBN 978-1-61208-304-9. 2013. [33%]

[A.5] Filipský, M.; Bureš, M; Jelínek, I. Framework for Better Efficiency of Automated Testing. In: *Proceedings of The Seventh International Conference on Software Engineering Advances*. The Seventh International Conference on Software. Lisbon, Portugal. Silicon Valley: International Academy, Research and Industry Association (IARIA), pp. 615-618. ISBN 978-1-61208-025-3. 2012. [33%]

[A.6] Filipský, M.; Bureš, M; Jelínek, I. Building Test Suites From Test Recordings of Web Applications In: *Proceedings of IADIS International Conference WWW/INTERNET 2012*. pp. 507-510. International Conference WWW/INTERNET 2012. Madrid, Spain. ISBN 978-989-8533-09-8. 2012. [33%]

Other Publications of the Author in the Area of Test Automation

Article to be Submitted to Impacted Journal

[B.1] Bureš, M; Ahmed, B.; Frajták, K., Filipský, M.; Jaroš, M., Bellekens, X. Flexibility between Physical and Simulated Smart Street Testbed: a Sample Use Case in the PatrIoT, an Open IoT Test Automation Framework. Being finished and submitted to the *Journal of Systems and Software*, Elsevier. (Q1, IF 2.3) [10%]

Conference Proceedings Paper Indexed in WoS

[B.2] Bureš, M; Filipský, M. SmartDriver: Extension of Selenium WebDriver to Create More Efficient Automated Tests. In: *Proceedings of the 6th International Conference on IT Convergence and Security (ICITCS 2016).* 6th International Conference on IT Convergence and Security, Prague. pp. 319-322. ISSN 2473-0122. ISBN 978-1-5090-3765-0. Red Hook: Curran Associates, Inc., 2016. [10%]