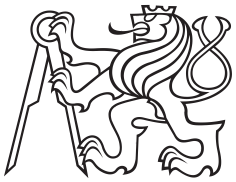

Czech
Technical
University
in Prague

Department of Computer Science

# Environment for evaluation of automated software tests effectiveness

Bc. Petr Syrovátka

Supervisor: Ing. Feras Abdul Hadi Mustafa Daoud
May 2024

# ZADÁNÍ DIPLOMOVÉ PRÁCE

## I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Syrovátka** Jméno: **Petr** Osobní číslo: **492281**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Specializace: **Softwarové inženýrství**

## II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

**Prostředí pro vyhodnocování efektivity automatizovaných testů pro software**

Název diplomové práce anglicky:

**Environment for evaluation of automated software tests effectiveness**

Pokyny pro vypracování:

Navrhněte a implementujte prostředí pro vyhodnocování efektivity automatizovaných testů pro software. Prostředí se bude skládat z testovaného open source systému (nebo systémů), sady automatizovaných testů, sady aktivních, historických nebo umělých chyb v testovaném softwaru a podpory pro vyhodnocení efektivity testů. Prostředí ověřte třemi experimenty vyhodnocujícími efektivitu automatizovaných testů oproti manuálnímu postupu.

Seznam doporučené literatury:

Kuhn, D. R., Kacker, R. N., & Lei, Y. (2013). Introduction to combinatorial testing. CRC press.
Ammann, P., & Offutt, J. (2016). Introduction to software testing. Cambridge University Press.
Nass, M., Alégroth, E., & Feldt, R. (2021). Why many challenges with GUI test automation (will) remain. Information and Software Technology, 138, 106625.

Jméno a pracoviště vedoucí(ho) diplomové práce:

**Ing. Feras Abdul Hadi Mustafa Daoud katedra počítačů FEL**

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **01.02.2024** Termín odevzdání diplomové práce: **24.05.2024**

Platnost zadání diplomové práce: **21.09.2025**

| Ing. Feras Abdul Hadi Mustafa Daoud<br>podpis vedoucí(ho) práce | podpis vedoucí(ho) ústavu/katedry | prof. Mgr. Petr Páta, Ph.D.<br>podpis děkana(ky) |
|---|---|---|

## III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

. Datum převzetí zadání

Podpis studenta

# Acknowledgements

I would like to thank my supervisor Ing. Feras Abdul Hadi Mustafa Daoud and doc. Ing. Miroslav Bureš, Ph.D. for all the time and help they provided me while writing my thesis.


# Declaration

I declare that this work is all my own work and I have cited all sources I have used in the bibliography.

Prague, May 24, 2024

Prohlašuji, že jsem předloženou práci vypracoval samostatně, a že jsem uvedl veškerou použitou literaturu.

V Praze, 24. května 2024

# Abstract

This thesis aims to assess the effectiveness of automated software testing, particularly with the help of Combinatorial Interaction Testing (CIT), in comparison with manual testing. The study measures defect detection rates through experiments that are performed on different open-source software applications, the efficiency of automated testing, and the practical implementation challenges of CIT. The study of the results demonstrates that the use of automated testing with the help of CIT improves the effectiveness of defect detection and increases the efficiency of testing but also reveals some difficulties in integration. The results offer useful recommendations for combining CIT with existing software development processes to enhance overall test efficiency.



**Keywords:** Software testing, Automated software testing, Combinatorial Interaction Testing, Test effectiveness, Defect detection



**Supervisor:** Ing. Feras Abdul Hadi Mustafa Daoud



# Abstrakt

Cílem této práce je posoudit efektivitu automatizovaného testování softwaru, zejména pomocí metody Combinatorial Interaction Testing (CIT), ve srovnání s manuálním testováním. Studie měří míru detekce chyb prostřednictvím experimentů, které jsou prováděny na různých open-source softwarových aplikacích, efektivitu automatizovaného testování a praktické problémy při implementaci CIT. Studie výsledků ukazuje, že použití automatizovaného testování pomocí CIT zlepšuje účinnost odhalování vad a zvyšuje efektivitu testování, ale také odhaluje některé obtíže při integraci. Výsledky nabízejí užitečná doporučení pro kombinaci CIT se stávajícími procesy vývoje software pro zvýšení efektivity testování.



**Klíčová slova:** Testovaní software, Automatizované testování software, kombinatorické testování interakcí, detekce defektů, efektivita testování

**Překlad názvu:** Prostředí pro vyhodnocování efektivity automatizovaných testů pro software

# Contents


**1 Introduction** **1**
1.1 Definition of Automated Software Testing . . . . . 1
1.2 Evolution and Importance of Automated Testing in Software Development . . . . . 1
1.3 Current Trends and Practices in Automated Testing . . . . . 2
1.3.1 Shift-left testing . . . . . 2
1.3.2 Test-driven development . . . . . 3
1.3.3 Integration with CI/CD Tools 3
1.4 Challenges in Assessing Test Effectiveness . . . . . 4
1.5 Objectives and Research Questions 4

**2 Theoretical Framework** **5**
2.1 Theoretical Models of Software Quality and Testing . . . . . 5
2.1.1 ISO/IEC 25010 . . . . . 5
2.1.2 Waterfall Model . . . . . 6
2.1.3 V-Model . . . . . 7
2.1.4 W-Model . . . . . 7
2.2 Concepts of Testability and Test-Driven Development (TDD) . . 8
2.2.1 Test Driven Development . . . . 9
2.3 Test Coverage Criteria and Adequacy Models . . . . . 10
2.3.1 Test Coverage Criteria . . . . . 10
2.3.2 Adequacy Models . . . . . 10
2.4 Frameworks for Evaluating Test Automation Return on Investment 11

**3 Literature review** **13**
3.1 Definition and Types of Automated Software Testing . . . . . 13
3.2 Types and Techniques of Automated Software Testing . . . . . 13
3.2.1 Unit Testing . . . . . 13
3.2.2 Integration Testing . . . . . 14
3.2.3 Database Testing . . . . . 14
3.2.4 System testing . . . . . 14
3.2.5 End to End testing . . . . . 15
3.2.6 Regression Testing . . . . . 15
3.3 Automated testing methods . . . . 15
3.3.1 Combinatorial testing . . . . . 15
3.3.2 Path based testing . . . . . 16
3.4 Benefits and Limitations of Automated Testing . . . . . 18
3.4.1 Benefits . . . . . 18
3.4.2 Limitations . . . . . 18
3.5 Factors Affecting Test Effectiveness . . . . . 20
3.6 Metrics for Evaluating Test Effectiveness . . . . . 21
3.7 Validation and Reliability Measures . . . . . 23
3.7.1 False Positive and False Negative Rates . . . . . 23
3.7.2 Test Suite Stability and Consistency . . . . . 23
3.7.3 Reproducibility and Replicability . . . . . 24
3.7.4 Comparative Analysis with Manual Testing . . . . . 24
3.8 Previous Studies on Automated Test Effectiveness . . . . . 25
3.8.1 Comparing the effort and effectiveness of automated and manual tests [7] . . . . . 25
3.8.2 Increasing the Effectiveness of Automated Testing [23] . . . . . 25
3.8.3 On the Effectiveness of Unit Test Automation at Microsoft [26] 26

**4 Methodology** **27**
4.1 Research Design and Approach . 27
4.1.1 Combinatorial test cases . . . . 27
4.1.2 Comparison of manual and combinatorial generated test cases 28
4.2 Selection of Test Subjects and Case Studies . . . . . 28
4.2.1 Redmine . . . . . 29
4.2.2 Trac . . . . . 30
4.2.3 Tracks . . . . . 31
4.3 Criteria for Selecting Automated Testing Tools and Frameworks . . . . 32
4.4 Data Collection Methods . . . . . 32
4.4.1 Data collection . . . . . 32
4.4.2 Redmine . . . . . 33
4.4.3 Trac . . . . . 34
4.4.4 Tracks . . . . . 36

**5 Results** **39**
5.1 Experiment 1 - Redmine . . . . . 39
5.2 Experiment 2 - Trac . . . . . 40
5.3 Experiment 3 - Tacks . . . . . 41

**6 Results Analysis** **43**
6.1 Effectiveness of Automated Testing with CIT . . . . . 43
6.2 Efficiency and Resource Utilization . . . . . 43
6.3 Limitations and Challenges . . . . 44
6.4 Practical Implications and Recommendations . . . . . 45
**7 Conclusion** **47**
**A Bibliography** **49**
**B Used Software** **53**
**C Code Examples** **55**

# Figures

1.1 V-model . . . . . 3

2.1 ISO 25010 11[1] . . . . . 5
2.2 Waterfall Model [19] . . . . . 7
2.3 W-Model . . . . . 7

3.1 Pairwise testing combinations [14] 16
3.2 Higher degree interactions [14] . . 17
3.3 Sample Pseudo Code with its C-program, CFG and Labeled CFG [16] . . . . . 17
3.4 Questions related to benefits of AST[22] . . . . . 19
3.5 Questions related to limitations of AST[22] . . . . . 20

# Tables

4.1 Parameters in New Issue Form in Redmine . . . . . 29
4.2 Parameters in Time Tracking Form in Redmine . . . . . 30
4.3 Parameters in New Issue Form in Trac . . . . . 30
4.4 Parameters in New Action form in Tracks . . . . . 31
4.5 Parameters in Configuration of Tracks . . . . . 31
4.6 Manual test cases for New Issue in Redmine . . . . . 33
4.7 Manual test cases for Time Logging in Redmine . . . . . 34
4.8 Manual test cases for New Ticket in Trac . . . . . 35
4.9 Manual test cases for New Action in Tracks . . . . . 36
4.10 Manual test cases for Configuration of Tracks . . . . . 37

5.1 Study 1: Summary of time requirement and test case count of experiments on Redmine . . . . . 40
5.2 Study 1: Comparison of defect detection rate and defect detection effectiveness of experiments on Redmine . . . . . 40
5.3 Study 2: Comparison of defect detection rate and defect detection effectiveness of experiments on Trac 41
5.4 Study 3: Comparison of defect detection rate and defect detection effectiveness of experiments on Tracks . . . . . 42

# Chapter 1

# Introduction

## 1.1 Definition of Automated Software Testing

Software testing is one of the most crucial phases in the software development cycle. The IEEE Standard Glossary of Software Engineering Terminology defines testing as *"The process of operating a system or component under specified conditions, observing or recording the results, and making an evaluation of some aspect of the system or component"* [1]. Restated, software testing is used to ensure that the behavior of software is as expected and meets specified requirements, and the primary goal is to identify defects, errors, or bugs in the software and to ensure its quality and reliability.

One of the more common approaches to testing is Automated testing. In automated testing, predefined tests are executed by testing tools or scripts on the system under test (SUT). Unlike manual testing, where testers must manually assess the software's quality, automated testing relies on automation tools to execute the test more efficiently and repeatedly [9]. This claim will be one of the main topics of this thesis.

## 1.2 Evolution and Importance of Automated Testing in Software Development

The evolution of automated testing in software development is highly influential and reflects the ever-changing nature of the industry. Automated testing has made remarkable progress and has evolved from basic scripts to advanced frameworks and tools. Advances in testing methodologies have been motivated by the need for more efficient and reliable approaches to testing, given the increasing complexity of software systems and the rapid pace of development.

In the beginning, automated testing emerged as a solution to the limitations of manual testing. With the increasing complexity and size of software systems, the time and effort required for manual testing became unsustainable. Automated testing solved this problem by offering faster and more reliable test execution, repeatability, and the ability to cover a much wider range of test scenarios. The move from manual to automated methodologies was

a major turning point that allowed development teams to decouple testing processes and improve the overall quality and reliability of the software.

In today's software development landscape, automated testing plays a crucial role, particularly in agile and DevOps environments. The demands of rapid iterations and continuous integration require testing processes and methods that are swift, effective, and reliable. Automated testing meets these demands by providing a framework for efficient and repeatable testing, ensuring that software changes can be tested comprehensively, repeatedly, and swiftly during the development process.

Beyond speed and efficiency, the importance of automated testing lies mostly in its contribution to early defect discovery and overall software stability. By automating repetitive and labor-intensive testing tasks, developers can focus on higher-level problem-solving while automated tests consistently verify the integrity of the software. This accuracy and reliability ultimately translate into creating robust and stable software.

In summary, the evolution and importance of automated testing in software development cannot be underestimated. It has become an irreplaceable component of modern development practices. As technology continues to advance, the role of automated testing is likely to expand further, shaping the future of software development by providing efficient, reliable, and scalable testing solutions.

## 1.3 Current Trends and Practices in Automated Testing

Software testing is a constantly evolving field, with new approaches being invented every so often. In this section, approaches with a rise in popularity, as well as some widely used approaches that are frequently used today, will be described.

### 1.3.1 Shift-left testing

The later a defect is discovered in the development cycle, the more challenging and expensive it becomes to address. This is why most testing methodologies are moving away from the waterfall scheme, where the testing is almost at the end of the development process. Shift-left testing is an approach modifying the V-model 1.1, in which each phase of the development process has a corresponding testing phase, and the emphasis is moved on the unit and integration testing [11]. The V-model itself will be explored more thoroughly in section 2.1.3.

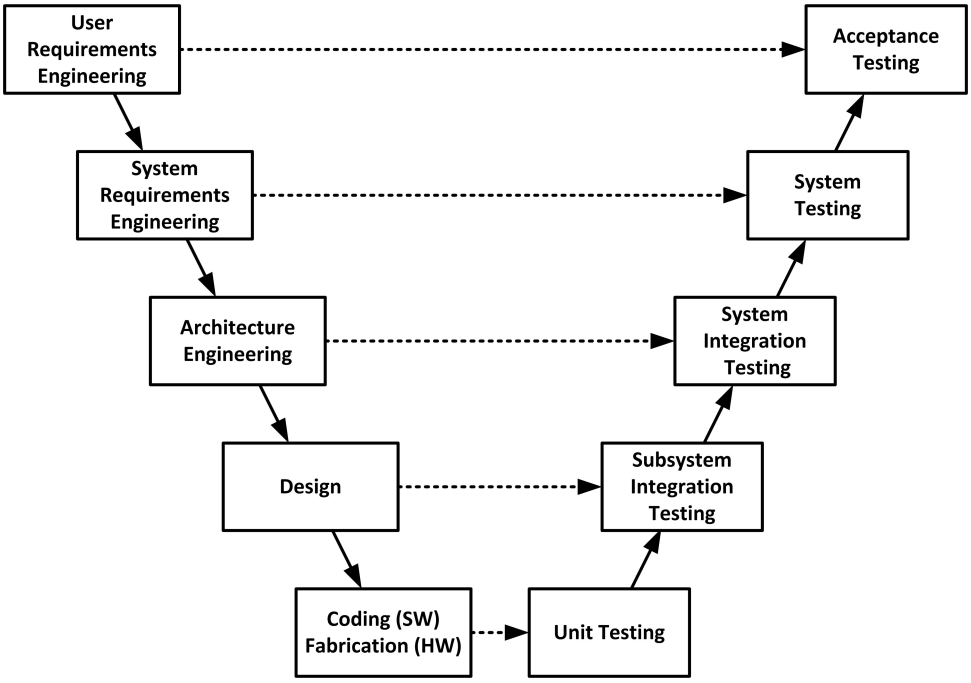


**Figure 1.1:** V-model

### 1.3.2 Test-driven development

Test-driven development is a software development process in which the functional requirements are converted into test cases before the software is developed. This approach ensures the quality and functionality of the software during its development, which is a significant advantage over the approach where tests are created and executed after. This concept will be further explored in section 2.2

### 1.3.3 Integration with CI/CD Tools

In software development, CI/CD is a practice that combines continuous integration and continuous deployment. With this approach, developers use tool-chains that automate building and deploying code stored in shared codebases. This streamlines the development process and enables the use of additional tools [10]. Integrating testing tools with this approach enables a constant quality evaluation with each addition to the software without the need for programmer interaction. In addition, these tools can execute tests that cover a wider range of systems in connection, which would be time-consuming to setup and execute manually.

## 1.4 Challenges in Assessing Test Effectiveness

Assessing the effectiveness of software tests poses many challenges, as the dynamic nature of software development continually introduces new approaches. One key challenge lies in defining comprehensive criteria for evaluating test outcomes, given the diverse range of software applications and their unique requirements. Additionally, the constantly evolving nature of technology demands continuous adaptation of testing methodologies, making it difficult to establish a standardized assessment framework. System complexity further complicates identifying root causes for defects, making it challenging to isolate and evaluate issues efficiently. Furthermore, the subjectivity in measuring testing efforts' success, such as user satisfaction and stakeholder expectations, adds another layer of complexity. Navigating these challenges requires a holistic approach, integrating both technical and non-technical perspectives to ensure a robust evaluation of test effectiveness in the dynamic landscape of software development.

## 1.5 Objectives and Research Questions

The primary objective of this thesis is to evaluate the effectiveness and efficiency of automated testing using Combinatorial Interaction Testing (CIT) in comparison to manual testing. This objective is crucial for understanding the potential of CIT-based automated testing to improve software quality assurance processes. Specifically, the thesis aims to:

- *Assess Defect Detection Rates:* Evaluate the capability of CIT-based automated testing to detect defects in software applications and compare it with traditional manual testing methods.
- *Measure Testing Efficiency:* Analyze the time and resources required for automated testing using CIT versus manual testing, focusing on the overall efficiency gains.
- *Identify Practical Challenges and Limitations:* Examine the practical challenges encountered when implementing CIT-based automated testing and identify any limitations that may affect its adoption and effectiveness.
- *Provide Recommendations:* Offer practical recommendations for integrating CIT-based automated testing into existing software development workflows, highlighting the balance between automated and manual testing.

# Chapter 2

# Theoretical Framework

## 2.1 Theoretical Models of Software Quality and Testing

### 2.1.1 ISO/IEC 25010

The quality model is the cornerstone of a product quality evaluation system. It determines which quality characteristics will be taken into account when evaluating the properties of a software product [1].

ISO/IEC 25010 is part of the ISO/IEC 25000 series, which encompasses a set of standards related to software quality. Specifically, ISO/IEC 25010 focuses on the quality model, which defines nine quality characteristics that are relevant to software products and systems shown in figure 2.1.

It's widely used by organizations, software developers, and quality assurance professionals to assess and improve the quality of software products and systems. It serves as a valuable reference for establishing quality requirements, conducting evaluations, and making informed decisions about software development.

| SOFTWARE PRODUCT QUALITY | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| FUNCTIONAL SUITABILITY | PERFORMANCE EFFICIENCY | COMPATIBILITY | INTERACTION CAPABILITY | RELIABILITY | SECURITY | MAINTAINABILITY | FLEXIBILITY | SAFETY |
| FUNCTIONAL COMPLETENESS<br>FUNCTIONAL CORRECTNESS<br>FUNCTIONAL APPROPRIATENESS<br>iso25000.com | TIME BEHAVIOUR<br>RESOURCE UTILIZATION<br>CAPACITY | CO-EXISTENCE<br>INTEROPERABILITY | APPROPRIATENESS RECOGNIZABILITY<br>LEARNABILITY<br>OPERABILITY<br>USER ERROR PROTECTION<br>USER ENGAGEMENT<br>INCLUSIVITY<br>USER ASSISTANCE<br>SELF-DESCRIPTIVENESS | FAULTLESSNESS<br>AVAILABILITY<br>FAULT TOLERANCE<br>RECOVERABILITY | CONFIDENTIALITY<br>INTEGRITY<br>NON-REPUDIATION<br>ACCOUNTABILITY<br>AUTHENTICITY<br>RESISTANCE | MODULARITY<br>REUSABILITY<br>ANALYSABILITY<br>MODIFIABILITY<br>TESTABILITY | ADAPTABILITY<br>SCALABILITY<br>INSTALLABILITY<br>REPLACEABILITY | OPERATIONAL CONSTRAINT<br>RISK IDENTIFICATION<br>FAIL SAFE<br>HAZARD WARNING<br>SAFE INTEGRATION |

**Figure 2.1:** ISO 25010 [1]

[1] https://iso25000.com/index.php/en/iso-25000-standards/iso-25010

### 2.1.2 Waterfall Model

The waterfall model was first introduced by Walter Royce in his 1970 article. Even though there is a wide agreement of problems connected to its use, the model is still a widely used way of working in software development companies [20]. One of the most recognized problems is the late detection of defects in the development process, which causes fixing them to be costly. This is one of the prevalent advantages of early testing, which has been mentioned several times in the previous chapter.

The waterfall model used for directing software development runs through phases of requirements engineering, design & implementation, testing, release, and maintenance, as seen in the figure 2.2, and between all phases, the software current documents have to pass a quality check. This approach is referred to as a stage-gate model. The following list describes each of the stages of the model [19]:

- *Requirements Engineering:* During this phase, customer needs are identified and documented at a high level of abstraction. These requirements are refined for use in the design and implementation phases, and they're stored in a requirements repository. At quality gates, stakeholders' agreement on requirements, alignment with business strategy, and understanding of all requirements are verified.
- *Design & Implementation:* The architecture of the system is designed and documented, followed by the development. Basic unit testing is conducted before passing the code to the testing phase. Quality gate checks include architecture evaluation, adherence to requirements, and adherence to planned timelines, effort, and scope.
- *Testing:* System integration is tested for quality and functionality, with performance measures collected for decision-making. Testing involves various hardware and software configurations to cater to different customer setups. Checklist reviews assess system verification, deviations from quality gate decisions, handover plans, and meeting customer requirements.
- *Release:* The product is prepared for shipment, with the finalization of release documentation and build instructions. Customization options may be enabled through build instructions. Quality gate checks ensure customer requirements are met and accepted by the customer and that quality requirements are delivered on time and fulfilled.
- *Maintenance:* After release, ongoing maintenance involves addressing customer-reported issues and delivering updates for system faults.

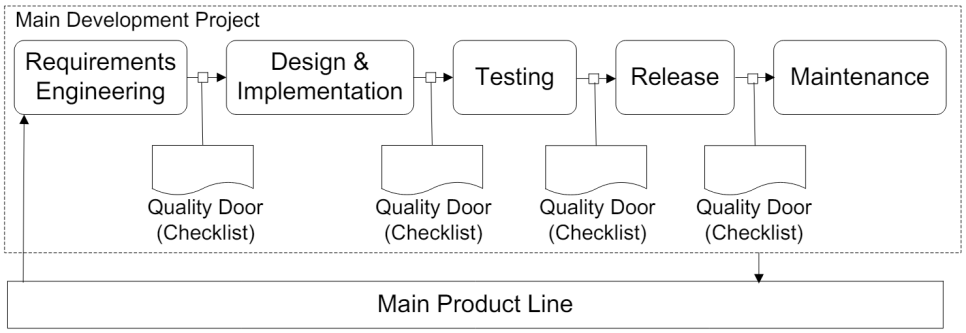


**Figure 2.2:** Waterfall Model [19]

### 2.1.3 V-Model

The V-Model, also known as the Validation or Verification Model, is a software development model that emphasizes the relation between each phase of the development cycle and its corresponding testing phase. It deploys a well-structured method in which each phase can be implemented using the detailed documentation created by the previous phase. The model is named "V" because of its visual representation, which resembles the letter V, as displayed in figure 1.1. Its shape comes from the fact that after all the development stages, which are very similar to the Waterfall Model, each stage has its corresponding stage, and the most recent stages are the ones that are first tested. This model is probably the most used model for software development and testing management [17]. Even though the V-Model emphasizes testing than the Waterfall Model, late detected defects may still occur, which is the reason why more suitable models were introduced.

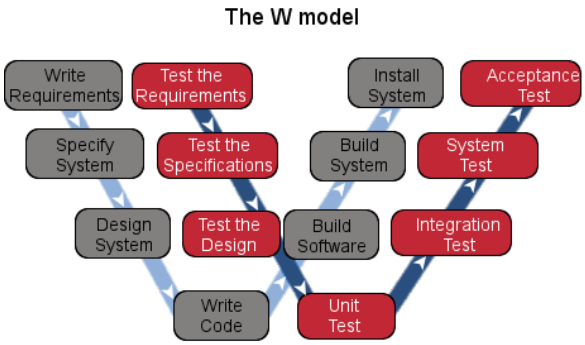


**Figure 2.3:** W-Model[2]

### 2.1.4 W-Model

The W-Model, which was introduced by Paul Herzlich in 1993, attempts to address and tackle the shortcomings of the V-Model. IT further emphasizes

[2]https://www.professionalqa.com/w-model

the testing activities in software development. It is called the "W-Model" because it visually represents the shape of a 'W', suggesting additional testing activities before and after the traditional V-Model phases. The figure 2.3 displays each stage and their corresponding test stages. In this approach, each development phase is immediately followed by its test phase so that if any defects are detected, they are easy and inexpensive to fix [24].

## 2.2 Concepts of Testability and Test-Driven Development (TDD)

Software testability is an indefinite metric that describes how well the software(software system, modules, design document) supports testing in the given context. If the testability is high, discovering defects in the system by means of testing is easier.

The quality of the product remains the main focus of software development, both in practice and literature. Sticking to quality concepts in development enhances the efficacy and quality of the resulting software. Given the competitive market and short release cycles, software organizations are discovering effective ways to monitor and instill ongoing quality. Testability, which reflects the ease of testing a software product, impacts testing cost and effort. As [25] suggests, the research consensus indicates that improving the testability of software has a direct, positive impact on overall quality but that challenges in defining then, measuring, and assessing software testability remain.

Defining and measuring testability has been a significant challenge, leading to extensive research efforts aimed at establishing its overall characteristics and associated measures. Various standards and individual studies have defined testability in diverse ways, reflecting different purposes and perspectives. Researchers such as Bache and Mullerburg (1990) [3] approached testability from perspectives such as test coverage, effort, time, and resources. These different views have led to the identification of multiple influential factors in software testability, setting the stage for further exploration in subsequent research.

Software testability is influenced by various factors, including the extent of required validation, the testing process, the tools used, and the representation of requirements. Identifying all potential factors and understanding their impact across different testing contexts is challenging due to their varying foundations. Effectively managing software testability requires modeling, measuring, analyzing, and interpreting its factors, similar to other quality attributes. This requires the use of software measurement and metrics to quantify the characteristics of interest in testing.

Software measurement plays a crucial role in software engineering, aiding project managers and engineers in planning and modeling, monitoring progress, and evaluating software performance. Software metrics, which quantify software characteristics, are fundamental to assessing software quality throughout its lifecycle. These metrics are often categorized into product,

process, and project metrics, with quality measurements commonly associated with product and process metrics. Product metrics, further classified into static and dynamic metrics, focus on different aspects of a system. Static metrics measure attributes without executing the program, reflecting the internal quality of the software, while dynamic metrics capture behavior during program execution. Static metrics, obtained through static analysis techniques, include measures like Lines of Code (LOC) and Cyclomatic Complexity (CC), providing coverage completeness but facing limitations in detecting dynamic dependencies. Dynamic metrics, collected during runtime, offer direct insight into system behavior but pose challenges in data collection and computation. While dynamic metrics have received less attention compared to static metrics, they offer valuable information for assessing software quality [25].

### 2.2.1 Test Driven Development

As mentioned in section 1.3.2, Test Driven Development is one of the testing methodologies, that is on the rise. To reiterate, TDD is a software development style where tests are written before the part of the software, the test should be evaluated on the basis of the functional expectations, and then the code is updated to a state that all of the tests pass.

An example of how the tests in TDD are structured is to take a typical method that takes one input parameter, returns an output value, and throws an exception in case of a wrong parameter. For such a method, unit tests would typically cover (1) a valid input parameter resulting in an expected output and (2) an invalid input triggering the appropriate exception, often including boundary values. It's common for the unit test method to be longer than the method under test. While unit tests may not cover every possible scenario, they generally focus on expected behaviors with valid inputs and a few exceptional cases. Additionally, extra unit test methods may be included for each method when its behavior relies on the object being in a specific state. In such instances, the unit test initiates with the correct method calls to ensure the object is in the intended state. After designing the tests, the actual method is programmed until it passes all of the previously mentioned tests.

Test-driven development offers several benefits, including reducing the gap between decision-making and feedback by employing a granular test-then-code cycle, which results in continuous feedback to developers. It facilitates early identification of defects, enabling easier and cheaper removal. TDD encourages the creation of automatically testable code, leading to benefits such as producing reliable systems, enhancing the quality of testing efforts, reducing testing time, and minimizing scheduling constraints. Moreover, TDD test cases establish a comprehensive regression test suite, enabling the detection of new changes that may break existing functionality and facilitating seamless integration of new features into the code base through continuous automated testing [18].

## 2.3 Test Coverage Criteria and Adequacy Models

Test coverage criteria and adequacy models are fundamental concepts in software testing. These metrics define the extent to which code is exercised by test cases, ensuring comprehensive testing. Test coverage criteria focus on specific code elements like statements, branches, and paths. Adequacy models provide guidelines for determining when testing is sufficient to ensure software quality. These models and criteria play a crucial role in software testing by providing guidelines for creating effective test sets, measuring test quality, and ensuring comprehensive coverage of the codebase [28].

### 2.3.1 Test Coverage Criteria

Test coverage criteria are metrics used to measure the extent to which the source code of a program has been exercised by a set of test cases. These criteria define specific aspects of the code that need to be covered during testing. Common types of test coverage criteria include:

- Statement Coverage: Ensures that each statement in the code is executed at least once during testing.
- Branch Coverage: Requires that all possible branches in the code are taken during testing.
- Path Coverage: Ensures that all possible paths through the code are executed.
- Function Coverage: Focuses on testing individual functions or methods within the code.
- Decision Coverage: Requires that all possible decision outcomes in the code are exercised.

### 2.3.2 Adequacy Models

Adequacy models, also known as test data adequacy criteria, define rules or guidelines for determining when testing is sufficient to ensure software quality. These models help assess the effectiveness of test sets and determine if additional testing is needed. Adequacy models can be used as stopping rules to decide when testing can be concluded or as measurements to quantify the degree of adequacy achieved by a test set. Examples of adequacy models include [28]:

- Statement Adequacy: Ensures that at least one test case covers each statement in the code.
- Branch Adequacy: Requires that all branches in the code are exercised by the test cases.

- Mutation Adequacy: Involves creating mutant versions of the code and checking if the test cases can detect these mutations.
- Data Flow Adequacy: Focuses on testing how data flows through the program to uncover potential issues.

## 2.4 Frameworks for Evaluating Test Automation Return on Investment

Return on Investment (ROI) is a financial metric used to evaluate the profitability or efficiency of an investment. It is calculated by dividing the net profit or benefit gained from the investment by the initial cost of the investment and then expressing the result as a percentage or ratio.

In the context of software development and testing, ROI can be used to assess the cost-effectiveness of implementing automated testing frameworks compared to manual testing. By analyzing the return on investment, organizations can make informed decisions about where to allocate resources and investments to maximize benefits and minimize costs [8].

The most used frameworks for evaluating ROI are: [3]

- *Basic ROI calculation:* $\frac{benefits-costs}{costs} * 100\%$

  The basic ROI calculation is calculated as benefits minus costs divided by costs, and multiplied by 100 to make it a percent metric. This metric is very simple, but it can very precisely determine if the investment in test automation is beneficial.

- *Cost-Benefit Analysis:* $TotalBenefits - TotalCosts$

  Cost-benefit analysis (CBA) compares the costs associated with implementing test automation, such as tool acquisition, training, and maintenance, to the benefits gained from test automation. It quantifies the financial impact of test automation by subtracting the total costs from the total benefits to determine the net benefit.

- *Payback Period:*

  $\frac{InitialInvestment}{AnnualSavingsOrBenefits}$

  The payback period measures the time it takes for the benefits of test automation to balance out the initial investment. It helps assess the speed at which the investment in test automation can be recovered. A shorter payback period indicates a faster return on investment.

[3] https://medium.com/slalom-build/what-is-the-roi-of-my-test-automation-10ae7bf0d9ed

# Chapter 3

# Literature review

## 3.1 Definition and Types of Automated Software Testing

In software testing, test automation refers to employing distinct software tools to manage the execution of tests and compare observed outcomes against expected results, separate from the software under examination [12]. This approach streamlines the testing process by reducing manual intervention, enhancing efficiency, and enabling repetitive testing tasks to be performed automatically and quickly. By automating testing procedures, software teams can achieve quicker feedback cycles, improve test coverage, and ensure the reliability and quality of their software products.

## 3.2 Types and Techniques of Automated Software Testing

There are many types and techniques of software testing that can be defined, each with its unique purpose. In this section, we will define many well-used techniques to understand their importance and purpose better.

### 3.2.1 Unit Testing

Nowadays, almost every programming language has its own library or framework for unit testing. One of the most commonly known is JUnit for Java, or NUnit for .NET languages, from which the most commonly used is C#. Unit testing has become one of the pillars of software testing, often even mandated by development processes [6]. The practice focuses on the smallest testable code units, such as functions or classes, and tests them in isolation. Using this approach, we can ensure that the foundation of the software is as reliable as possible and can be used to build on. Its aim is to be repeatable and fast, such that the tests can be quickly executed with every addition to the code and can provide the first check of flawlessness.

### 3.2.2 Integration Testing

Integration testing is the second part of the standard testing stack aside from Unit testing. This testing methodology is applied at the module level rather than the statement and function level, as in unit testing. It emphasizes the interaction of the modules, interfaces, and other connected systems, such as databases [15], so that the systems' flawlessness and communication are ensured. According to [4], almost 40% of software errors can be traced to module interaction defects discovered during integration. This shows the importance of integration testing since unit tests won't discover this type of defectas, and they can also be challenging to trace when more complex and broader scope testing methodologies are used.

### 3.2.3 Database Testing

Databases are one of the core parts of the majority of modern software. They serve as a central repository for software applications to store, manage, and retrieve data efficiently. By providing a structured data storage and retrieval framework, databases enable software to access, manipulate, and process information effectively. Even though databases are usually part of the software that is tested during Integration testing, with the complexity of the stored data, databases are often tested separately. Since Databases are tested individually, this type of testing is usually considered unit testing. When testing, there are usually 4 data integrity types that need to be verified[1]:

- Entity integrity: testing if the individual data components (tables, columns) have their own unique identifying primary key
- Referential integrity: ensuring that the relation of data between two or more tables is correct
- Domain integrity: guaranteeing uniform data types and formats for every column or row to ensure compatibility during data transfer to other systems, each with its own interpretation rules, preventing conflicts
- User-defined integrity: ensuring that user-defined rules, which serve additional needs, are valid

Some modern databases come with the support for database testing, such as Oracle SQL Developer[2], other frameworks for database testing are DbUnit or SQLUnit[1].

### 3.2.4 System testing

The IEEE Standard Glossary of Software Engineering Terminology defines system testing as *"Testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements"* [1]. While

[1] https://katalon.com/resources-center/blog/database-testing
[2] https://docs.oracle.com/cd/E15846_01/doc.21/e15222/unit_testing.htm#RPTUG45000

usually almost at the end of the testing stack, system testing is as important as other testing types. System testing verifies integration between different components, identifies defects, and documents test plans and results for stakeholders' reference. Ultimately, it plays a pivotal role in ensuring the software's quality, reliability, and functionality before end-user deployment.

### 3.2.5 End to End testing

End-to-end (E2E) testing is a software testing approach aimed at evaluating the seamless flow of functional and data processes throughout an application, encompassing multiple interconnected subsystems. Often, these subsystems are developed independently using diverse technologies by distinct teams or organizations. End-to-end testing ensures the holistic functionality of the application by scrutinizing its entirety, integrating all constituent components from inception to completion [3].

### 3.2.6 Regression Testing

In software testing, regression refers to the unintentional introduction of defects caused by changes in the codebase that appear on previously flawlessly working software [2]. This phenomenon may appear with the addition of new features, as well as library upgrades and even fixing other defects. Regression testing is a term used for repeated execution of existing tests to ensure that previously developed and tested software still performs as expected, even after an addition to the code. In the present day, this is often carried out by tools integrated with CI/CD pipelines.

## 3.3 Automated testing methods

### 3.3.1 Combinatorial testing

Combinatorial testing can be used to detect hard-to-find software defects more precisely than manual test case selection methods. Its strength lies in the exploration of each n-way interaction between each of the possible values of each parameter [13], and it can also greatly reduce the number of value combinations tested.

#### Pairwise testing

Suppose we want to test a software application with three parameters, each with two possible values. If we wanted to test each combination, we would have $2 * 2 * 2 = 8$ combinations. With pairwise (2-way) testing, we can test its behavior with each possible pair of values with only 4 tests. An example

[3] https://microsoft.github.io/code-with-engineering-playbook/automated-testing/e2e-testing/

| Test case | OS | CPU | Protocol |
|---|---|---|---|
| 1 | Windows | Intel | IPv4 |
| 2 | Windows | AMD | IPv6 |
| 3 | Linux | Intel | IPv6 |
| 4 | Linux | AMD | IPv4 |

**Figure 3.1:** Pairwise testing combinations [14]

can be seen in figure 3.1, where each pair of three two-value parameters is present in at least one test case.

Even though pairwise testing does not exhaust all possible combinations, it is useful for checking for simple yet potentially defective interactions with fewer tests. Imagine we have a large automation system with 20 controls, each with 10 possible settings. This would mean that we would have to test $20^{10}$ combinations, which would be impossible to test. With pairwise testing, we would test every possible pair of the settings with only a very surprising 180 test cases [14].

#### Higher-Degree Interactions

One of the most important thoughts that is discussed in [14] is *"Other empirical investigations have concluded that from 50 to 97 percent of software faults could be identified by pairwise combinatorial testing. However, what about the remaining faults? How many failures could be triggered only by an unusual interaction involving more than two parameters?"*. When investigating higher n-way interactions, progressively fewer faults were discovered. In figure 3.2, we can see that roughly 40 percent of defects in Web server application were caused by a single value, another 30 percent were caused by two value interactions, and almost 90 percent were triggered by three or less parameters.

Although pairwise combinatorial testing, the most fundamental form of this technique, is widely acknowledged and adopted by software testing practitioners, higher degree n-way testing is on the rise, but its utilization in the industry remains inconsistent. Nevertheless, the investment in additional training necessary for implementation proves to be highly beneficial.

### 3.3.2 Path based testing

Path based testing, or basis path testing, is a white-box testing technique that systematically tests the execution paths within a software application. It involves identifying and testing different paths or sequences of statements, branches, and loops in the source code to ensure comprehensive coverage [21]. Path-based testing is particularly useful for uncovering errors or defects related to control flow, such as logic errors, missing or incorrect conditions, and unreachable code. However, achieving complete path coverage can be challenging, especially in complex software with a large number of possible

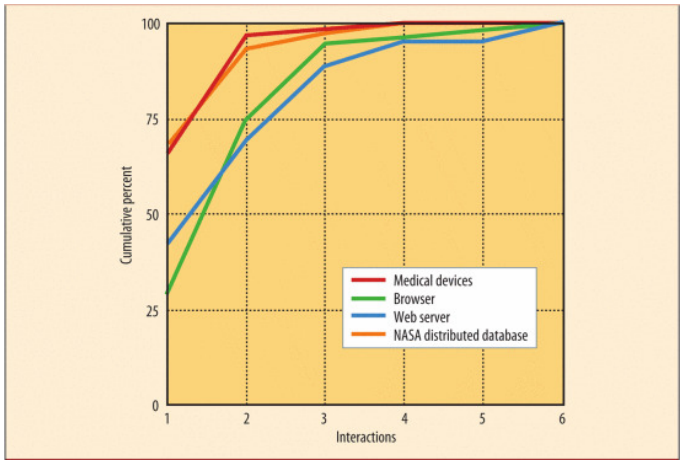


**Figure 3.2:** Higher degree interactions [14]

paths. Therefore, path-based testing is often used in combination with other testing techniques to achieve comprehensive coverage and ensure software quality. Path testing begins with the creation of a program flow graph, which serves as a simplified representation of all potential pathways within the program. This graph comprises nodes to denote decision points and edges to illustrate the flow of control. By replacing program control statements with equivalent diagrams, such as in the absence of goto statements, constructing the flow graph becomes straightforward. Each branch within conditional statements like if-then-else or case statements is delineated as an individual path, while loops are represented by arrows looping back to the condition node[4]. An example of creating a CFG for a pseudo-code or a code written in C can be seen in figure 3.3.

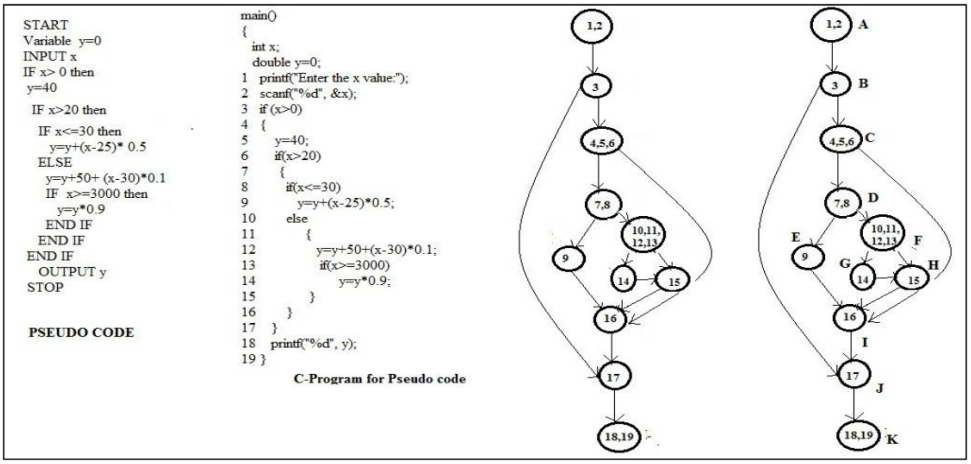


**Figure 3.3:** Sample Pseudo Code with its C-program, CFG and Labeled CFG [16]

[4]https://ifs.host.cs.st-andrews.ac.uk/Books/SE9/Web/Testing/PathTest.html

## 3.4 Benefits and Limitations of Automated Testing

When discussing the usage of automated testing, we have to consider its positives and negatives in relation to our situation to consider if automated testing is the correct solution for our application. In this section, we will discuss the benefits and limitations of automated testing further to explore its place in the category of software testing. As [22], a literature review of 25 publications, suggests, benefits often originate from stronger sources of evidence (experiments and case studies), while limitations often originate from experience reports.

### 3.4.1 Benefits

- *Improved product quality:* Enhances product quality by minimizing the number of defects in the software.
- *Test coverage:* Achieves extensive code coverage (e.g., statement, branch, path) through automation.
- *Reduced testing time:* Decreases the time needed for testing, allowing more tests to be executed within the same period.
- *Reliability:* Ensures more consistent test results, eliminating variations caused by different manual testing methods, though it cannot leverage the tester's expertise.
- *Increase in confidence:* Boosts developers' confidence in the system's quality.
- *Reusability of tests:* Enables frequent reuse of tests designed with maintenance in mind, providing benefits beyond a single execution.
- *Less human effort:* Reduces the need for human intervention, freeing up resources for other tasks, particularly those focused on defect prevention.
- *Reduction in cost:* Saves costs through a high degree of automation.
- *Increased fault detection:* Enhances the ability to identify a significant number of defects due to its high fault detection capability.

### 3.4.2 Limitations

- *Automation can not replace manual testing:* Some testing tasks, especially those requiring extensive domain knowledge, cannot be easily automated.
- *Failure to achieve expected goals:* Organizations may be attracted to the quick execution of tests but fail to realize long-term or substantial benefits.

- *Difficulty in maintenance of test automation:* Maintaining automated tests becomes challenging as technology changes and software evolves.
- *Process of test automation needs time to mature:* Developing the necessary infrastructure and tests for automation takes time, so the benefits of automation require patience.
- *False expectations:* Organizations often have unrealistic expectations about AST, aiming to cut costs by engaging in unproductive testing activities.
- *Inappropriate test automation strategy:* Deciding on the correct strategy for which test levels to automate and for what purposes is difficult, leading to suboptimal strategies that fail to maximize AST benefits.
- *Lack of skilled people:* Effective test automation requires a variety of skills, including knowledge of test tools, software development, and domain-specific insights, which may be lacking.

This publication identified an overview of benefits and limitations in a wide range of related literature and conducted a survey with 115 respondents. Respondents were primarily composed of quality assurance analysts and programmers with other roles, such as system architects and designers, as well as project managers. Their experiences ranged from less than five years to more than fifteen years. The benefits and limitations were transformed into statements, with which respondents could rate them in the range of one to five, with five representing total agreement.

When answering questions related to benefits displayed in figure 3.4, it is visible that the benefits of automated software testing that were found in the literature are strongly supported by the respondents, with at least half of them agreeing or fully agreeing to 8 out of 9 statements. Limitations in figure 3.5, which were presented in a similar manner as the benefits, show that the limitations are also strongly supported by practitioners [22].

| Rank (4+5) | Questions related to benefits | Answers on Scale[1] | | | | | Median |
|---|---|---|---|---|---|---|---|
| | | 5 | 4 | 3 | 2 | 1 | |
| B.R1 | B6: High reusability of tests makes automated testing productive | 46 | 53 | 13 | 3 | 0 | 4 |
| | | 40% | 46% | 11% | 3% | 0% | |
| B.R2 | B6/B3: AST enables the repeatability of tests, which gives the possibility to do more tests in less time | 38 | 59 | 10 | 6 | 2 | 4 |
| | | 33% | 51% | 8% | 6% | 2% | |
| B.R3 | B1/B2: AST improves product quality by better test coverage. | 28 | 59 | 20 | 7 | 1 | 4 |
| | | 24% | 51% | 18% | 6% | 1% | |
| B.R4 | B8/B3/B6: AST saves time and cost as it can be re-run again and again and hence is much quicker than manual testing | 29 | 54 | 15 | 15 | 2 | 4 |
| | | 25% | 47% | 13% | 13% | 2% | |
| B.R5 | B5: AST provides more confidence in the quality of the product and increases the ability to meet schedules | 25 | 51 | 19 | 16 | 4 | 4 |
| | | 22% | 44% | 17% | 14% | 3% | |
| B.R6 | B7: The investment in application-specific test infrastructure can significantly reduce the effort that automation requires from testers | 13 | 63 | 28 | 11 | 0 | 4 |
| | | 11% | 55% | 24% | 10% | 0% | |
| B.R7 | B7/B8: By having complete automation the cost of AST is dramatically reduced and facilitates continuous testing | 22 | 52 | 17 | 10 | 4 | 4 |
| | | 19% | 45% | 16% | 17% | 3% | |
| B.R8 | B4/B7: AST requires less effort on the developers side, but cannot find complex bugs as manual testing does | 19 | 49 | 19 | 24 | 4 | 4 |
| | | 17% | 43% | 16% | 21% | 3% | |
| B.R9 | B9: Automated testing facilitates high fault detection | 9 | 38 | 35 | 29 | 4 | 3 |
| | | 9% | 33% | 30% | 26% | 3% | |

[1]5=completely agree, 4=agree, 3=neutral, 2=disagree, 1=completely disagree

**Figure 3.4:** Questions related to benefits of AST[22]

| Rank | Questions related to limitations | Answers on Scale[1] | | | | | Median |
|---|---|---|---|---|---|---|---|
| | | 5 | 4 | 3 | 2 | 1 | |
| L.R1 | L4: Compared with manual testing, the cost of AS is higher, especially in the beginning of the automation process. However, automated software testing can be more productive after a period of time | 42<br>37% | 60<br>52% | 8<br>7% | 4<br>3% | 1<br>1% | 4 |
| L.R2 | L3/L4: Automated testing needs extra effort for designing and maintaining test scripts. | 37<br>32% | 64<br>56% | 7<br>6% | 6<br>5% | 1<br>1% | 4 |
| L.R3 | L6/L7: Testers should have enough technical skills to build successful automation | 40<br>35% | 53<br>46% | 12<br>10% | 9<br>8% | 3<br>3% | 4 |
| L.R4 | L5: Compared with manual testing, AST requires a high investment to buy tools and train staff to use the tools | 32<br>28% | 56<br>49% | 12<br>10% | 11<br>10% | 6<br>5% | 4 |
| L.R5 | L5: AST requires less effort on the developers side, but cannot find complex bugs as manual testing does | 19<br>17% | 49<br>43% | 19<br>17% | 24<br>21% | 4<br>3% | 4 |
| L.R6 | L3: Most of the testing tools available in the market are incompatible and do not provide what you need or fits in your environment. | 11<br>10% | 40<br>35% | 30<br>26% | 24<br>21% | 10<br>9% | 3 |
| L.R7 | L1: Automated testing fully replaces manual testing. | 1<br>1% | 6<br>5% | 16<br>14% | 49<br>43% | 43<br>37% | 2 |

[1]5=completely agree, 4=agree, 3=neutral, 2=disagree, 1=completely disagree

**Figure 3.5:** Questions related to limitations of AST[22]

To summarize what we have found, automated software testing has the benefits of reusability and repeatability, improves product quality, and has a lower cost in the long-range view. Its primary limitations are high initial costs, its higher demand for the technological skill of the staff, and its inability to find more complex bugs.

## 3.5 Factors Affecting Test Effectiveness

There are many factors that can influence the effectiveness of software testing. By considering and optimizing these factors, software development teams can improve the effectiveness of their testing efforts, ultimately delivering higher-quality software products.

- *Test coverage: The degree to which a given test or set of tests addresses all specified requirements for a given system or component* [1]. Higher test coverage generally results in more effective testing by identifying a broader range of defects.

- *Test data quality:* The quality and the relevance of the input data used for testing. Test data fully representing the whole range of possible inputs improves the likelihood of uncovering defects.

- *Testing environment:* The environment in which the testing is conducted, including the software, hardware, and network configurations. Ensuring consistency between testing and production environments enhances the effectiveness of testing.

- *Testing Techniques:* The methodologies and techniques used during testing, such as unit testing, integration testing, and system testing. Utilizing testing techniques appropriate to the software's characteristics enhances test effectiveness.

- *Testing tools and frameworks:* The usage of relevant tools can complement the different types of testing, resulting in a faster generation of tests of better quality, which can be simpler to update and expand.
- *Skill and Experience of Testers:* The proficiency and experience of individuals performing the testing activities can greatly impact the test effectiveness. Skilled testers can better design test cases, detect and describe found defects more effectively, and interpret test results.
- *Defect Management Process:* The process for logging, tracking, and describing defects identified during testing can greatly improve the simplicity of identifying and fixing the faulty code. An efficient defect management process ensures that defects are addressed promptly, further improving the effectiveness of testing.

## 3.6 Metrics for Evaluating Test Effectiveness

Test effectiveness metrics are essential for assessing the quality and efficiency of the testing process. They help the testing team evaluate the testing strategy in different views and find possible shortcomings. All following metrics were discussed in [5].

- ***Test Improvement (TI)*** *of Product Quality:* $\frac{W_{TTP}}{KCSI}$

  The Test Improvement (TI) metric demonstrates the relation between the number of weighted defects detected by the test team during testing ($W_{TTP}$) and the size of the product release ($KCSI$). It evaluates the impact of testing efforts on enhancing product quality. A higher TI value suggests more defects or more significant defects were identified during testing, leading to improved product quality attributed to the test teams.

- ***Test Effectiveness (TE)*** *to Drive Out Defects:* $\frac{W_T}{W_{BF}+W_F} * 100\%$

  The Test Effectiveness (TE) metric measures the relation between the number of weighted defects detected during the product cycle ($WT$) and the total number of weighted defects in the product, which is the combined value of number of defects found during the test cycle ($W_{TP}$, and the number of weighted defects found in the product after release. It gauges the efficiency of the test organization in identifying defects before product release. A higher TE value indicates a greater proportion of defects or important defects were identified before release, reflecting the effectiveness of the test organization in driving out defects.

- ***Test Time (TT)*** *Needed Normalized to Size of Product:* $\frac{T_t}{KCSI}$

  Test Time (TT) Normalized to the Size of Product measures the time spent on testing ($T_T$) relative to the size of the product release ($KCSI$). It indicates the testing process's efficiency by quantifying the time required by the test teams to test the product. A lower value suggests that the test teams need less time to complete testing, reflecting potentially more efficient testing practices.

- ***Test Time Over Development Time (TD):*** $\frac{T_T}{T_D} * 100\%$

  Test Time Over Development Time (TD) assesses the ratio of time spent on testing ($T_T$) to the time spent on product development ($T_D$). It demonstrates the efficiency of the testing process relative to the development process. A lower TD value indicates that the test teams require less time to test the product compared to the time spent by the development team on product development, potentially indicating effective testing practices and shorter testing cycles.

- ***Test Cost Normalized to Product Size (TCS):*** $\frac{C_T}{KCSI}$

  Test Cost Normalized to Product Size (TCS) assesses the cost of testing ($C_T$) relative to the size of the product release ($KCSI$). It quantifies the efficiency of resource or monetary investment in testing per thousand lines of code. A lower TCS value indicates a lower cost required to test each thousand lines of code, reflecting potentially more cost-effective testing practices.

- ***Test Cost as a Ratio of Development Cost (TCD):*** $\frac{C_T}{C_D} * 100\%$

  Test Cost as a Ratio of Development Cost (TCD) evaluates the ratio of testing cost ($C_T$) to the development cost ($C_D$) of the product. It highlights the relationship between testing expenditure and product development investment. A lower TCD value indicates that the test teams require less cost to test the product compared to the cost incurred by the development team, suggesting potential cost savings in testing.

- ***Cost per Weighted Defect Unit (CWD):*** $\frac{C_T}{W_T}$

  Cost per Weighted Defect Unit (CWD) measures the relationship between the total cost of testing the product ($C_T$) and the number of weighted defects found by the test team during the product cycle ($W_T$). It quantifies the cost-effectiveness of the testing process in identifying defects. A lower CWD value indicates a lower cost of finding one defect unit, implying a more efficient and cost-effective testing process.

These metrics can help the development and testing team determine the effectiveness of the testing process and possibly identify parts of the testing process that can be improved. They can also be used to track the performance of said teams.

## 3.7 Validation and Reliability Measures

The validation of automated tests and the measures to ensure high reliability of the testing are one of the core parts of the effectiveness of automated testing. Without sufficient attention to this part, automated testing can become ineffective or even fall behind manual testing in various ways[5].

### 3.7.1 False Positive and False Negative Rates

Despite its benefits, automated testing is prone to false positives and negatives, undermining confidence in test results and impeding the development process. False positives occur when automated tests report issues that do not actually exist in the software, leading to wasted time and effort in investigating non-existent faults. On the opposite side, false negatives occur when automated tests fail to detect genuine defects, allowing potentially harmful errors to go undetected. To mitigate false positives and false negatives, testers must create a balance between test coverage and precision, ensuring that tests are neither overly strict nor overly lenient. Additionally, continuous refinement of test cases based on feedback from test executions can help reduce false positives and false negatives over time, enhancing the reliability of automated testing.

### 3.7.2 Test Suite Stability and Consistency

The stability and consistency of automated test suites are crucial factors in ensuring reliable test results across different executions and environments. Test suites that are prone to frequent failures or inconsistencies can disrupt the development workflow, leading to delays and uncertainty in the software delivery process. This behavior can either suggest an unstable implementation or a test quality often described as "flakiness", which means that the test fails or passes almost randomly with no change to the code. This poses a threat to the test reliability since a possibly critical defect can be undetected, or the test can incorrectly fail, which can waste the resources of the testing and development team by searching for a nonexistent defect.

Several factors can contribute to test suite instability, including changes in the application codebase, updates to third-party dependencies, and environmental variations. To enhance test suite stability and consistency, practitioners adopt practices such as version control for test assets, isolation of test environments, and periodic maintenance of test cases to address evolving requirements and conditions. By prioritizing stability and consistency in test suite design and execution, teams can minimize disruptions and maintain confidence in the reliability of automated testing.

[5] https://hr-guide.com/Testing_and_Assessment/Reliability_and_Validity.html

### 3.7.3 Reproducibility and Replicability

Reproducibility and replicability are one of the essential characteristics of automated testing that ensure the consistency and reliability of test results over time. Reproducibility refers to the ability to rerun automated tests and obtain consistent outcomes, even manually if needed, regardless of when or where the tests are executed. Replicability, on the other hand, involves the ability to reproduce test results across different environments or configurations, validating the robustness of the testing approach. Achieving reproducibility and replicability requires careful management of test environments, dependencies, and configurations to minimize variations that could affect test outcomes. Moreover, documenting test procedures, including setup instructions and test data requirements, facilitates the replication of test results by other team members or external testers.

### 3.7.4 Comparative Analysis with Manual Testing

When comparing automated testing with manual testing, it is important to consider various factors affecting validation and reliability measures. While manual testing offers flexibility and adaptability in exploring edge cases and user interactions, it is inherently time-consuming and error-prone, especially for repetitive or complex test scenarios. In contrast, automated testing excels in scalability, repeatability, and coverage, enabling complex validation of software functionality and performance across varying environments and configurations. However, automated testing may struggle with certain types of testing, such as exploratory testing or subjective evaluation of user experience. By combining both automated and manual testing strategies thoughtfully, teams can leverage the strengths of each approach to achieve a balanced and effective testing regimen, maximizing the validation and reliability of the software under development.

## 3.8 Previous Studies on Automated Test Effectiveness

In this section, we will cover similar studies conducted on the topic of Automated Test Effectiveness. Understanding the landscape of previous studies is essential for contextualizing our own research. We will point out relevant questions, discuss observed results and possible challenges regarding automated testing.

### 3.8.1 Comparing the effort and effectiveness of automated and manual tests [7]

The paper highlights the significance of automated testing in software development, which aims to improve efficiency and reliability by automating test design and execution. However, it also acknowledges the potential drawbacks, such as the initial investment and ongoing maintenance required for automated tests. The decision to automate tests should be accompanied by a thorough cost-benefit analysis, considering factors like organizational culture, staff training, and available tools. The study aims to compare the effort and effectiveness of automated versus manual testing in a multinational organization's Java web applications, emphasizing the need for context-specific analysis in determining the most suitable testing approach.

Three case studies were conducted to compare the effort and effectiveness of automated versus manual testing approaches within a multinational organization, focusing on a set of Java web applications. The first case study evaluated the effort required for both testing methods using predefined test cases and data, while the second determined the effectiveness in terms of defect detection within a fixed time period. The third case study assessed the effort and effectiveness of both approaches in the context of incremental changes to the system after its release. Results indicated that although automated testing initially demands more effort, it proves to be more effective at defect detection, especially when considering regression testing over time.

### 3.8.2 Increasing the Effectiveness of Automated Testing [23]

This paper presents techniques aimed at decreasing the execution time and maintenance expenses associated with automated regression test suites used in eXtreme Programming (XP) development. These techniques are crucial as they facilitate developers' engagement with testing by minimizing associated burdens. Strategies outlined include leveraging in-memory databases to mitigate latency induced by disk-based databases or file systems, as well as employing frameworks that streamline the setup and teardown processes of test fixtures, thereby reducing the effort required for test development.

The conclusion of this paper is that the efficiency of a test-first development approach correlates inversely with test execution time, which can be significantly reduced by eliminating disk-based I/O latency. Replacing

disk-based databases with in-memory databases achieves this optimization, particularly if the application logic operates solely with objects instead of utilizing SQL for database interactions. Additionally, enhancing the testing framework can substantially decrease the effort required for test development and maintenance, although various challenges may arise, all of which are possible to overcome.

### 3.8.3 On the Effectiveness of Unit Test Automation at Microsoft [26]

Implementing automated unit testing across a large software development team can pose technical challenges and consume significant time. However, a case study involving a team of 32 developers at Microsoft demonstrated the benefits of transitioning to NUnit automated unit testing framework, with tests typically written after coding functionality every two to three days. After a year of utilizing this practice on Version 2 of a product, the team observed a 20.9% decrease in test defects, albeit at the cost of approximately 30% more development time compared to Version 1. While other industrial teams have achieved greater defect reductions through iterative test-driven development practices, these results underscore the value of automated unit testing, suggesting that even more significant quality improvements may be realized with iterative test-writing approaches.

# Chapter 4

# Methodology

## 4.1 Research Design and Approach

This work mainly focuses on comparing automated and manual testing in a wide range of aspects. Manual and automated testing will be conducted in each of the studies, while time requirements will be tracked. Since automated testing has a greater time investment, which is one of the points of the comparison that will be discussed, a fraction of the generated tests will be programmed. Their time requirements will be used to calculate an average time to design and program one test. In addition, the time consumed to set up the testing environment will be tracked to estimate the time required to test the chosen system under test as precisely as possible. The same procedure will be done for manual testing, with the change that the test cases will be designed primitively without the knowledge of combinatorial testing.

Then, for each of the studies, a set of defects will be evaluated against the conducted tests to evaluate how successful the tests were. The set will consist of historical, currently present, and artificial defects. Since all of the systems are public releases already exhaustively tested by their developers, artificial defects will be the prevalent type, with historical defects as the second most common occurrence. With artificial and historical defects that are not present in the system, each will be compared carefully with the conducted tests to determine if it would have been discovered. Various metrics will be computed and discussed after estimating each testing technique's time consumption and success rate.

### 4.1.1 Combinatorial test cases

To consistently produce effective test cases through combinatorial testing, each of the parameters of chosen parts of the applications has to be analyzed to determine different values that would influence the behavior of the application. The most straightforward are checkboxes, radio boxes, and select elements that have no further functionality. To correctly represent these, their values are put as the possible values for the parameter. There can also be situations where the values have a deeper effect on the application, which the tester must consider. A great example is in the new issue form in Trac, where each

value of milestone parameters can have an assigned due date. By taking this into account, the tests can be more precise, test functionalities that could otherwise be omitted, and, in case of a defect, help understand the reason for the error better.

#### ACTS

ACTS was used as the tool for generating the combinatorial test cases, as it is a known standard in software testing. The tool supports n-way and mixed-strength combinatorial test case generation. For the generating, IPOG algorithm was used as it is the default and recommended option [27].

### 4.1.2 Comparison of manual and combinatorial generated test cases

A comparison between manual and combinatorial generated test cases points to the contrasting approaches to software testing and their respective benefits and limitations. Manual test case creation involves testers creating test scenarios based on their application knowledge, intuition, and experience. While manual testing allows for flexibility in exploring complex interactions and edge cases, it is inherently subjective, time-consuming, and prone to human error. On the other hand, combinatorial test case generation uses mathematical algorithms to systematically generate test cases by considering various combinations of input parameters and their values. This approach offers scalability, repeatability, and coverage, ensuring comprehensive validation of software functionality with minimal human intervention. However, combinatorial testing may overlook certain nuanced scenarios that require human insight, and its effectiveness heavily relies on the quality of input parameters and the accuracy of the combinatorial algorithms employed. One of the most prevalent advantages of manual testing is the quality control of the user interface. While executing manual test cases, the tester can also check the application layout and can identify, for example, misaligned elements or wrong coloring. By combining these two methodologies, organizations can develop a testing strategy that combines the strengths of manual testing's flexibility with combinatorial testing's efficiency, thereby enhancing their software products' overall quality and reliability.

## 4.2 Selection of Test Subjects and Case Studies

The selection of test subjects was primarily on the following criteria: (1) The System Under Test should be web browser-based since these user interfaces are the easiest to test; (2) The application should have at least one reasonably sized input form or configuration form since combinatorial testing is the most applicable approach in them; (3) The application should be open-source, or at least with a tracking system with historical and currently occurring defects so that they could be used with the addition of artificial defects in following

experiments. In relation to these criteria, the following three applications were chosen:

### 4.2.1 Redmine

Redmine[1] is an open-source project management web application that has been a solid option in team collaboration and project tracking since its inception in 2006 by Jean-Philippe Lang. Built on Ruby on Rails framework, Redmine helps teams with a comprehensive suite of tools to streamline project planning, execution, and monitoring. Its features include defect and ticket tracking, time tracking management, and wiki creation to codebase management.

In Redmine, two different forms were chosen as test subjects. The first subject is a form for creating a new Ticket. This form was chosen because of its many parameters and possibly interesting interaction of different parameters. The original manual test set created without the knowledge of CIT consisted of 18 test cases. Out of those, 14 of them were replaced by 21 test cases, which were generated using CIT, adding to a total amount of 25 test cases. The form had 10 parameters with a total of 25 different values, which can all be seen in table 4.1.

| **Parameter Name** | **Number of values** |
|---|---|
| Subject | 2 |
| Assignee | 3 |
| Category | 3 |
| Target Version | 3 |
| File | 2 |
| Parent Task | 2 |
| Start Date | 3 |
| End Date | 3 |
| Estimated Time | 4 |
| **Total** | 25 |

**Table 4.1:** Parameters in New Issue Form in Redmine

The second one is a form for managing time tracking. It's the second largest form in the application, with many historically documented defects that could be possibly used in this experiment. The approach with test cases was very similar. 12 manual test cases were created, and 6 were replaced by 9 CIT-generated test cases, totaling 15. This form is smaller, only with 4 parameters and 10 total values, displayed in table 4.2.

[1] https://www.redmine.org/

| Parameter Name | Number of values |
|---|---|
| Issue | 2 |
| Date | 3 |
| Hours | 3 |
| Project | 2 |
| **Total** | 10 |

**Table 4.2:** Parameters in Time Tracking Form in Redmine

### 4.2.2 Trac

Trac[2] is an open-source, web-based project management, issue tracking and wiki application written in Python. It is primarily used for managing software development projects but can also be used for other types of projects. Trac provides a range of features designed to make collaboration among team members easier and track project progress.

In Trac, a form for creating a new ticket was chosen as the study target. The choice was made because of the same qualities as in the previous application, and the form has many parameters with varying value types. As this form is slightly larger than the forms in Redmine, a total of 30 manual test cases were created. In this study, 25 of the manual cases were replaced by CIT-generated ones, of which there were 17. This form has 13 parameters with a total of 34 different values, all described in table 4.3.

| Parameter Name | Number of values |
|---|---|
| Summary | 2 |
| Reporter | 3 |
| Description | 2 |
| Type | 3 |
| Milestone | 3 |
| Version | 3 |
| Priority | 5 |
| Component | 2 |
| File Attached | 2 |
| Keywords | 2 |
| CC | 2 |
| Owner | 3 |
| Severity | 2 |
| **Total** | 34 |

**Table 4.3:** Parameters in New Issue Form in Trac

[2]https://trac.edgewall.org/

### 4.2.3 Tracks

Tracks[3] is an open-source productivity web-based application designed to help individuals manage tasks, projects, and goals efficiently. It provides features to organize tasks, set priorities, and track progress toward achieving objectives. It is written using Ruby with Rails framework.

As in previous applications, a form for creating a new task was chosen as the first experiment target. The second experiment subject is a configuration form for the whole application. In the first form, 8 parameters with a total of 23 values were used in combinatorial testing, which can be seen in the table 4.4.

| Parameter Name | Number of values |
|---|---|
| Description | 2 |
| Notes | 2 |
| Project | 3 |
| Context | 3 |
| Tags | 3 |
| Due Date | 4 |
| Show from Date | 3 |
| Depends On | 3 |
| **Total** | 23 |

**Table 4.4:** Parameters in New Action form in Tracks

A second target for an experiment in this application was its configuration. Sadly, not every parameter could be tested, as some are time-dependent. In the end, 6 parameters were chosen to be used in CIT with a total of 14 values, which are displayed in table 4.5.

| Parameter Name | Number of values |
|---|---|
| DueStyle | 2 |
| Show Completed Projects | 2 |
| Show Hidden Projects | 2 |
| Show Hidden Contexts | 2 |
| Go To Project | 2 |
| Show Number Of Completed Actions | 4 |
| **Total** | 14 |

**Table 4.5:** Parameters in Configuration of Tracks

[3] https://www.getontracks.org/

## 4.3 Criteria for Selecting Automated Testing Tools and Frameworks

Selecting the right automated testing tools and frameworks is critical for ensuring the efficiency and effectiveness of the testing processes. The considered criteria for experiments in this work were:

- *Compatibility and Support:* Ensure that the tool supports the technologies and platforms on which the SUT was built. If the tool is not outdated and is frequently updated, it is also a huge plus, mainly in the long-term view. An additional positive is a possible integration with CI/CD pipelines.
- *Ease of Use and Learning Curve:* Choosing a testing tool that is easy to use and learn is one of the most crucial criteria. It can have a huge impact on the effectiveness of automated testing.
- *Scripting Language Support:* Depending on the experience and preference of the testing team, choosing a tool with a programming language that the team is familiar with can streamline test script development and maintenance.
- *Reporting and Analysis:* Use of tools that offer reporting and analysis features to help identify issues, track test coverage, and measure the effectiveness of your tests.

Cypress[4] was selected as the testing environment because it is the most suitable framework for this type of experiment. Cypress is a popular testing framework commonly used for end-to-end testing of web applications. It provides a set of tools and libraries for automating web application testing across different browsers and platforms.

## 4.4 Data Collection Methods

In the realm of software testing, collecting and analyzing data is one of the most important processes for assessing the quality, performance, and reliability of software products. In this section, we will describe the parameters of the different studies conducted, such as the manual and automatic test cases with examples, and the defects introduced to the experiments.

### 4.4.1 Data collection

Various data had to be collected while conducting all of the following experiments. To reliably maintain and analyze the data, google spreadsheets were utilized. All of the data can be viewed at this google spreadsheets

[4]https://www.cypress.io/

document[5]. GitHub has been utilized for version control. All of the testing scripts used in the studies can be seen at this GitHub repository[6]. For each of the experiments, one automated test case is presented as an example in the appendix. The code is truncated by whitespaces and unimportant comments to make it more readable.

### 4.4.2 Redmine

The study on Redmine was split into two different parts of the application. The first experiment revolves around creating a new ticket and editing it. A total of 18 manual test cases were designed and executed. 14 of them were replaced by 15 automated test cases designed using CIT. The set of manual cases can be seen in the table 4.6.

| Test case | Name | Description |
|---|---|---|
| Iss1 | NewIssueWithMandatory | Create new issue with only mandatory fields |
| Iss2 | NewIssueAssignToMe | Mandatory, and assign to me |
| Iss3 | NewIssueWithAttachment | Mandatory, and add an attachment |
| Iss4 | NewIssueWithoutSubject | Try to create an issue without subject and check failure |
| Iss5 | NewIssueWithParentIssue | Mandatory, and add parent issue |
| Iss6 | EditIssueAddParentIssue | Edit an issue by adding parent issue |
| Iss7 | EditIssueAddChildIssue | Edit an issue by adding child issue |
| Iss8 | NewIssueWithDueDateInPast | Choose due date in past, while start date is today, check failure |
| Iss9 | NewIssueWithNewCategory | Mandatory, create a new category |
| Iss10 | NewIssueWithNewTargetVersion | Mandatory, create a new target version |
| Iss11 | EditIssueStatus | Edit status of a issue |
| Iss12 | EditIssueCloseWithoutChildClosed | Close an parent issue without child issue closed |
| Iss13 | NewIssueDuplicateSubject | Mandatory, create a issue with already existing subject |
| Iss14 | NewIssueInvalidEstimatedTime | Input invalid estimated time, check failure |
| Iss15 | NewIssueStartDatePast | Mandatory, choose start date in past |
| Iss16 | NewIssueStartDateFuture | Mandatory, choose start date in future |
| Iss17 | NewIssueTooBigAttachment | Add attachment larger than 5 mb, check failure |
| Iss18 | NewIssueAnyAssignee | Mandatory, choose assignee other than the user |

**Table 4.6:** Manual test cases for New Issue in Redmine

The creation and programming of the automated CIT test cases were pretty straightforward, with the exception of the insertion of dates. Even though the element taking care of the date is a standard input, because of the application's design, the date could not be simply entered, so we had to come up with a special syntax using invoke() and trigger() functions. This can be seen in the example code below. Otherwise, the application was easy to use from the tester's perspective. Automated test case number 14 is provided as an example in the appendix C.1. This test is supposed to create an error since the ticket's due date cannot be before the start date, which, in this case, is the current date. The code has been truncated by comments that are not important to the work.

The last part of this experiment are the defects. In total 10 defects were introduced to this experiment, 8 of them were artificial: *DueDateBeforeStartDateUnchecked, AssignToMeButtonNotWorking, FileUploadNotWorking, NewCategoryNotWorking, ParentTaskWhileOtherAssigneeFails, Start-*

[5]https://docs.google.com/spreadsheets/d/1yZ5_PwGjLzA5JKZTzbSxKBv9970iL3BdzJylEEXzREY/edit?usp=sharing

[6]https://github.com/Syor/Thesis

*DateAndDueDateInPastFail, EmptyAssigneeFails and ParentTaskWhileEmptyEstimatedTimeFails* , and two of them were reused historical defects: *EstimatedHourStringInvalid and DefaultAssigneeIsNotSet*.

The second experiment was conducted on the part of the application that is used to track time spent on different tickets. This experiment was the most straightforward of all. There was the same problem with dates as in the previous study, but it had already been solved earlier. This application form is smaller, as can be seen in table 4.2. The 12 manual test cases are displayed in table 4.7.

| Test case | Name | Description |
|---|---|---|
| Time1 | BasicTimeLog | Log time with only mandatory fields |
| Time2 | LogZeroHours | Mandatory, log 0 hours |
| Time3 | LogDecimalHours | Mandatory, log hours with decimal point |
| Time4 | TimeLoggedToSubCountsToParent | Check if time logged adds to parent issue |
| Time5 | TryLogNegativeHours | Input negative hours, check failure |
| Time6 | LogTimeInFuture | Mandatory, choose date in future |
| Time7 | LogTimeInPast | Mandatory, choose date in past |
| Time8 | FindByHoursLessEqual | Filter existing time logs by less equal |
| Time9 | FindByHoursMoreEqual | Filter existing time logs by more equal |
| Time10 | EditTime | Edit time of a time log |
| Time11 | EditIssue | Edit issue of a time log |
| Time12 | EditDate | Edit date of a time log |

**Table 4.7:** Manual test cases for Time Logging in Redmine

Out of them, 6 were replaced by 9 automated test cases. As an example, automated test case number 7 is provided in the appendix C.2.

Lastly, 4 artificial defects were created, and unfortunately, no historical defects that could be reintroduced were found. The artificial defects are: *LoggingTimeInPastFails, LoggingNegativeHoursDoesNotFail, LoggingForOtherAssigneeFails and LoggingWithoutIssueFails*

### 4.4.3 Trac

The second study was conducted on Trac. In total, 30 manual test cases were created. Most of them focused on different values of most of the parameters and few of their combinations, which ought to be explored. The last few were focused on editing the tickets, such as closing or accepting them. The names of the test cases were primarily chosen to describe what the test case does differently than the default and mandatory values, or what the tested process does. The manual test cases can be seen in the table 4.8.

After analyzing the form and creating the CIT test cases, 17 new test cases were created that replaced 25 of the manual test cases. The script programming was reasonably fast since the form was created very well with regard to testability. The only problem we encountered was already at the start of the programming. The application login process is handled very unusually, as opposed to a standard approach, with a small login screen with two inputs for username and password, the login is handled as an alert popup, which is really hard to handle with cypress, since the popup is not a part

| Test case | Name | Description |
|---|---|---|
| Tic1 | MandatoryAndDefault | New ticket with mandatory and default values |
| Tic2 | NoSummaryShouldFail | Create Mandatory and default with no summary, check failure |
| Tic3 | DefaultAndMilestoneEmpty | Mandatory and default, set milestone to empty |
| Tic4 | DefaultAndVersionEmpty | Mandatory and default, set version to empty |
| Tic5 | DefaultAndMilestoneOther | Mandatory and default, set milestone to other than default |
| Tic6 | DefaultAndVersionOther | Mandatory and default, set verison to other than default |
| Tic7 | MilestoneAndVersionEmpty | Mandatory and default, milestore and version empty |
| Tic8 | MilestoneAndVersionOther | Mandatory and default, version and milestone other than default |
| Tic9 | MilestoneWithDueDatePast | Mandatory and default, milestone with due date in past |
| Tic10 | MilestoneWithDueDateFuture | Mandatory and default, milestone with due date in future |
| Tic11 | ComponentOther | Mandatory and default, set component to other than default |
| Tic12 | FileAttached | Mandatory and default, attach a file |
| Tic13 | OwnerEmpty | Mandatory and default, set owner to empty |
| Tic14 | ReporterEmpty | Mandatory and default, set reporter to empty |
| Tic15 | VersionInPast | Mandatory and default, set version with past release |
| Tic16 | VersionInFuture | Mandatory and default, set version with future release |
| Tic17 | PriorityMajor | Mandatory and default, set priority to major |
| Tic18 | PriorityBlocker | Mandatory and default, set priority to blocker |
| Tic19 | PriorityCritical | Mandatory and default, set priority to critical |
| Tic20 | PriorityMinor | Mandatory and default, set priority to minor |
| Tic21 | PriorityTrivial | Mandatory and default, set priority to trivial |
| Tic22 | HighSeverity | Mandatory and default, set severity to high |
| Tic23 | LowSeverity | Mandatory and default, set severity to low |
| Tic24 | NonEmptyKeywords | Mandatory and default, add any keywords |
| Tic25 | NonEmptyCC | Mandatory and default, add any CC |
| Tic26 | AddCommentToTicket | Edit ticket by adding a comment |
| Tic27 | ResolveAsFixed | Edit ticket by resolving as fixed |
| Tic28 | ResolveAsInvalid | Edit ticket by resolving as invalid |
| Tic29 | ReassignTicket | Edit ticket by reassigning to different owner |
| Tic30 | AcceptTicket | Edit ticket by accepting it |

**Table 4.8:** Manual test cases for New Ticket in Trac

of the web Document Object Model. This was handled by inputting the credentials into the web address itself, since it is the way that the popup actually handles the process. The example code, which was also added as a command into cypress for easier testing, can be seen below.

```
Cypress.Commands.add('trac_login', (user) => {
cy.visit('http://' + user.username + ':' + user.password +
    '@localhost:8123/');
})
```

The automated testing was conducted using 17 test cases generated with CIT. Automated test case number 4, provided as an example, can be found in the appendix C.3.

The last part of the setup for the experiment is the defects. In total 10 defects were used in this experiment, 7 of them were artificial: *EmptySummaryPasses, EmptyMilestoneFails, AttachmentWithEmptyOwnerFails, PastVersionFails, KeywordsAndCCTogetherFail, MajorPriorityFails and HighSeverityWithTrivialFails*, and three were based on historical defects: *StringValueNotResetOnErase, TicketCreationFails and CannotAttachFile.*

### 4.4.4 Tracks

In the last application, two different experiments were conducted. The first experiment was very similar to previous applications conducted on a form for creating a new action. For this form, 23 manual test cases were created, and 17 of them were replaced by 15 automated CIT-generated cases. The manual test cases are displayed in table 4.9.

| Test case | Name | Description |
|---|---|---|
| Act1 | OnlyMandatoryAndDefault | Create new action with only mandatory and default values |
| Act2 | NoDescription | Create new action without description, check falilure |
| Act3 | ActiveProject | New action with active project |
| Act4 | HiddenProject | New action with hidden project |
| Act5 | ActiveContext | New action with active context |
| Act6 | HiddenContext | New action with hidden context |
| Act7 | ClosedContext | New action with closed context |
| Act8 | DuePast | New action with due date in past |
| Act9 | DueToday | New action with due date today |
| Act10 | DueFuture | New action with due date in future |
| Act11 | ShowFromPastShouldFail | Create new action with show from date in past, should fail |
| Act12 | ShowFromFuture | New action with show from date in future |
| Act13 | ShowFromTodayShouldFail | Create new action with show from date today, should fail |
| Act14 | OneTag | New action with one tag |
| Act15 | MultipleTags | New action with multiple tags |
| Act16 | DependsOnExisting | New action, add dependency on existing action |
| Act17 | DependsOnNonexistent | New action, try add dependency on nonexistent action |
| Act18 | EditDescription | Edit description of action |
| Act19 | EditDue | Edit due of a action |
| Act20 | EditFrom | Edits show from of a action |
| Act21 | EditContext | Edit context of a action |
| Act22 | EditProject | Edit project of a action |
| Act23 | DeleteAction | Delete an action |

**Table 4.9:** Manual test cases for New Action in Tracks

This form was the easiest to work on, mainly because of the fact, that most of the input elements had id attributes, which contributed to easier element querying, and the date pickers were easy to work with. An example of the automated scripts is the test case number 13 provided in the appendix C.4.

In this study, 6 defects were compared against the tests, 5 defects were artificial: *ShowFromPastWorksIfDueIsPast, MultipleTagsDontSeparate, HiddenContextAndProjectFail, EmptyDescriptionWorksIfDependent and NotesDoNotWork* , and one historical defect was reintroduced: *ShowFromFails*.

The second experiment was very different from the others since it was conducted on a configuration form of the application. Only 9 test cases were created for manual testing, which can be observed in the table 4.10. All of them were replaced by 8 automated test cases.

The programming of the tests was very straightforward. Since most of the options required some data to be tested on, a lot of test preparation had to be done before checking the correctness of the processes. Test case number 13 is provided in the appendix C.5 as an example. In this experiment, 3 artificial defects were introduced: *ShowHiddenContextAndProjectFails, DueInFails, and GoToProjectsTrueFails*.

| Test Case | Name | Description |
|---|---|---|
| Conf1 | DefaultSetting | Default values: due in, all true, positive completed actions |
| Conf2 | DueOn | Default values, but due style is due on |
| Conf3 | CompletedProjectsFalse | Default values, but show completed projects false |
| Conf4 | HiddenProjectsFalse | Default values, but show hidden projects false |
| Conf5 | HiddenContextsFalse | Default values, but show hidden contexts false |
| Conf6 | GoToProjectPageFalse | Default values, but go to project page false |
| Conf7 | NumberOfActionsZero | Default values, but number of actions shown zero |
| Conf8 | NumberOfActionsNegative | Default values, but number of actions shown negative |
| Conf9 | NumberOfActionsInvalid | Default values, but number of actions set to a string |

**Table 4.10:** Manual test cases for Configuration of Tracks

# Chapter 5

# Results

In this chapter, we will compare the results of automated testing with the usage of CIT against manual testing. Each of the studies was conducted on a different application, which was described in section 4.2. The first two studies were conducted on input forms of different sizes, which were purposed to create a particular record in the said application. The third study was conducted on a configuration form of the main part of the application and a form for creating a new activity.

In all three studies, we compare the efficiency of automated testing with CIT against manual testing without CIT. The results point to a significant increase in effectiveness regarding the number of detected defects and also in the time saved by implementing automated testing. In the following sections, the term *2-way* means a 2-way strength combinatorial array was used when designing the automated test cases using CIT, and *mixed* refers to a combinatorial array with mixed strength. This means that while some chosen groups of parameters are tested together at a higher interaction level, others may be tested at a lower interaction level. These terms will also be used in later sections to discuss the results.

## 5.1 Experiment 1 - Redmine

This study was conducted for a different yet unreleased publication named *Effectiveness of Combinatorial Interaction Testing in Test Automation - An Industrial Case Study* written by Feras Daoud, Miroslav Bures, Zdenek David and Petr Syrovatka.

As mentioned in the previous sections, two different forms were chosen as the targets for the experiment. Summary of the time requirement and test case counts are provided in the table 5.1. When testing the form for creating a new issue, the number of test cases increased by 61%, but the hours spent testing were reduced by 36%. The average time to detect a single defect reduced significantly from 2.18 hours to 0.56 hours. The smaller form for logging time tracking had the number of test cases increased by 25%, and the total number of hours was reduced from 7.25 to 5.35 hours. The average time to detect a defect in this experiment went from 3.6 hours to 1.3 hours, which is an almost 65% reduction. Overall, in this study, the average time to

detect a defect has decreased by more than 70%, from 2.66 to 0.78 hours. In this study, CIT with mixed strength showed the same results as with 2-way strength. The results of the experiments can be seen in the table 5.2.

| | Manual cases | CIT - 2-way | CIT - Mixed |
|---|---|---|---|
| **Time Spent on testing in hours** | | | |
| Total time spent on New Issue form | 8.75 | 5.6 | 5.6 |
| Total time spent on Time Tracking form | 7.25 | 5.35 | 5.35 |
| **Total** | 16 | 10.95 | 10.95 |
| **Test case count** | | | |
| New Issue form | 18 | 29 | 29 |
| Time Tracking form | 12 | 15 | 15 |

**Table 5.1:** Study 1: Summary of time requirement and test case count of experiments on Redmine

| | Manual cases | CIT - 2-way | CIT - Mixed |
|---|---|---|---|
| **Defects Reintroduced or Present in SUT** | | | |
| Historical defects in New Issue form | 2 | | |
| Artificial defects in New Issue for | 8 | | |
| Artificial defects in Time Tracking form | 4 | | |
| **Total** | 14 | | |
| **Count of Detected Defects** | | | |
| Defects detected in New Issue form | 4 | 10 | 10 |
| Defects detected in Time Tracking form | 2 | 4 | 4 |
| **Total** | 6 | 14 | 14 |
| **Defect detection rate related to time spent** | | | |
| New Issue | 2.18 | 0.56 | 0.56 |
| Time Tracking | 3.6 | 1.3 | 1.3 |
| Overall average time to detect one defect | 2.66 | 0.78 | 0.78 |

**Table 5.2:** Study 1: Comparison of defect detection rate and defect detection effectiveness of experiments on Redmine

## 5.2 Experiment 2 - Trac

One of the main advantages of automated testing combined with CIT prevailed in this study. To sufficiently manually test an application with many parameters, multiple similar test cases with only a minor difference have to be executed, while automated testing with CIT tests all of the parameters and their combinations in fewer cases and with a capable testing team with a lesser time investment. As it can be seen in the table 5.3, the number of automated test cases was reduced to almost half, and the total time investment decreased by more than a third compared to manual testing. The average time needed to detect a defect has decreased by 56% while using automated testing compared to manual testing.

| | Manual cases | CIT - 2-way |
|---|---|---|
| **Time Spent (hours)** | | |
| Total time spent on New Ticket form | 9.1 | 5.75 |
| **Test case count** | | |
| New Ticket form | 30 | 17 |
| **Defects Reintroduced or Present in SUT** | | |
| Historical defects in New Ticket form | 3 | |
| Artificial defects in New Ticket form | 7 | |
| **Total** | 10 | |
| **Count of Detected Defects** | | |
| Defects detected in New Ticket form | 7 | 10 |
| **Defect Detection rate related to time spent** | | |
| Overall average time to detect one defect | 1.3 | 0.57 |

**Table 5.3:** Study 2: Comparison of defect detection rate and defect detection effectiveness of experiments on Trac

## 5.3 Experiment 3 - Tacks

In this study, one of the disadvantages of automated testing prevailed. In smaller systems under test with fewer parameters, the initial time investment can be higher than that of manual testing. Although there were fewer automated test cases than manual ones, the total time spent on automated testing was slightly higher, by a mere 8%. Yet the automated testing with CIT has shown its advantage in its effectiveness in detecting more complex defects. It detected half as many defects, and its average time to detect one defect decreased by 27% compared to manual testing. A summary of the data can be seen in the table 5.4

An interesting event also happened during this experiment since an unexpected behavior of the application in one of its configurations was discovered, which could be considered a defect. When configuring the application, one of the options sets how many already completed actions will be shown on the main page. While testing a positive value and zero, the application performed as expected. But when the value was set to a negative integer, the application unexpectedly accepted the value without a failure or a warning and showed all of the existing completed actions on the main page. Since no record of this behavior is a feature found in the application's documentation, this behavior will be considered a defect and submitted as a discovered bug to the issues management system of the application. Similarly, the same parameter accepted a string value, in which case the application sets the value to zero. As in the previous case, this is undefined behavior without the application's reaction.

Both of these defects were detected by manual and automated testing. In this case, it could be discussed that, without the proper technical analysis of the SUT, manual testers might not test certain parameters with intentionally wrong values relative to the application. Because of that, the defects could have been undiscovered while manually being tested.

| | **Without CIT** | **CIT - 2-way** |
|---|---|---|
| **Time Spent (hours)** | | |
| Total time spent on New Action form | 6.3 | 6.6 |
| Total time spent on Configuration | 1.8 | 2.15 |
| **Total** | 8.1 | 8.75 |
| **Test case count** | | |
| New Issue form | 23 | 15 |
| Configuration | 9 | 8 |
| **Total** | 32 | 23 |
| **Defects Reintroduced or Present in SUT** | | |
| Historical defects in New Action form | 2 | |
| Artificial defects in New Action form | 5 | |
| Artificial defects in Configuration | 3 | |
| **Total** | 10 | |
| **Count of Detected Defects** | | |
| Defects detected in New Action form | 4 | 7 |
| Defects detected in Configuration | 4 | 5 |
| **Total** | 8 | 12 |
| **Defect detection rate related to time spent** | | |
| New Action | 1.58 | 0.94 |
| Configuration | 0.45 | 0.43 |
| Overall average time to detect one defect | 1 | 0.73 |

**Table 5.4:** Study 3: Comparison of defect detection rate and defect detection effectiveness of experiments on Tracks

# Chapter 6

# Results Analysis

In this chapter, we will evaluate the results of the three experiments and analyze the differences between automated testing using combinatorial interaction testing (CIT) and manual testing. The purpose is to assess the outcomes based on the efficiency, benefits, and limitations encountered during the experiments.

## 6.1 Effectiveness of Automated Testing with CIT

The experiments have revealed the positive effect of the use of CIT for automated testing on the number of defects detected. In the case of the Redmine experiment, the average time to detect a single defect was reduced from 2.66 hours in manual testing to 0.78 hours with CIT, showing a significant gain in efficiency. Similarly, the Trac experiment's average time to detect one defect decreased from 1.3 hours manually to 0.57 hours with CIT. This pattern in various applications suggests the strength of CIT in automation testing by detecting more defects in less time. Additionally, the number of defects found can be used to argue that automated testing with CIT provides better coverage of the software under test. This comprehensive defect detection can be attributed to the systematic and exhaustive nature of combinatorial testing, which covers a wide range of input combinations that manual testing might not have considered.

However, these findings should be interpreted cautiously since the results may differ from the actual usage of automated testing in real practice. Although the defects introduced in the experiments aimed to be as similar to real-world scenarios as possible, even with the inclusion of historical defects, they might be influenced by the writer's experience, which might not fully represent the actual situations in software testing.

## 6.2 Efficiency and Resource Utilization

The first and one of the most important benefits of the automated testing, that is mentioned in the experiments, is the time and human resource savings. For instance, the total time spent on testing the New Issue form in Redmine

decreased from 8.75 hours manually to 5.6 hours with automated testing. This time efficiency is particularly important in big software projects that involve frequent releases. It enables the development team to run more tests in shorter periods, making it an ideal tool for continuous integration and continuous deployment (CI/CD) practices.

In addition, a crucial long-term benefit of automated tests is that they can be performed repeatedly with little extra work, unlike manual tests, which need to be conducted repeatedly by the tester. This reusability helps save time and ensures that tests are executed consistently, helps eliminate the different results that may be obtained from different testers, and prevents possible code regression.

## 6.3 Limitations and Challenges

Despite the many advantages, the experiments also identified some disadvantages and difficulties associated with using automated CIT testing. One notable weakness is the time and resources used to develop the automated test cases and the required infrastructure. For instance, the amount of time spent on programming and recording the automated tests for the New Issue form in the Trac experiment was relatively high, indicating the larger initial time investment needed.

Nevertheless, it is clear that automated testing will not be able to substitute manual testing entirely. It is important to note that human intelligence and intuition are invaluable in certain situations, especially when dealing with difficult edge cases that may not be easily detected by automation testing. Manual testing is also important in terms of testing the UI and UX of the app, as automated testing might not be able to address some issues. One of the additional advantages of manual testing is to simulate an inexperienced user who uses the application incorrectly, possibly inducing other defects that are undetectable by automated testing.

## 6.4 Practical Implications and Recommendations

The outcomes of the experiments have several possible implications for the software development and testing teams. Firstly, incorporating CIT-based automated testing into the development lifecycle can greatly improve the detection rate of defects and testing productivity. However, it is suggested that both approaches be used in parallel to take the best of both approaches. Automated tests are useful when it comes to running large and frequently repeated tests, while manual tests can be more effective when it comes to exploratory testing, finding corner cases, and UI/UX reviews.

Second, it is vital to invest in training and tools for the automation of testing. To get the maximum benefits and minimize the problems that might be encountered in the initial stages of the testing process, it is necessary to make sure that the testing team has the necessary skills to build and maintain automated tests. Finally, constant reassessment and updating of the testing process are critical to maintaining a relevant and effective approach to software testing.

# Chapter 7

# Conclusion

This thesis investigated the effectiveness of automated software testing using Combinatorial Interaction Testing (CIT) compared to manual testing methods. The studies aimed to assess various aspects such as defect detection rates, testing efficiency, and practical challenges involved in implementing CIT. The study was conducted through a series of experiments on three open-source software systems: Redmine, Trac, and Tracks. The results clearly demonstrated the advantages of automated testing over manual testing in several key areas.

Automated testing showed a notably higher defect detection rate than manual testing. This outcome underscores the ability of automated tests, with the addition of suitable testing methodologies, to identify more defects within the same timeframe, enhancing overall software quality. The systematic nature of automated testing allows for broader and more thorough coverage of test cases, reducing the likelihood of undetected defects.

The efficiency of the testing process was also significantly improved with automation. Automated tests executed faster than manual efforts, enabling more frequent and comprehensive testing cycles. This efficiency reduces the time-to-market for software products, allows the testing teams to assess the software quality more frequently, and helps developers identify and address defects earlier in the development process.

Despite these advantages, implementing automated testing comes with challenges, such as the initial setup complexity and the need for robust test management practices. It can be concluded that the best practice for testing software development projects is a combination of automated and manual testing techniques. In this way, the testing teams can provide extensive test coverage, increased reliability, and higher software quality.

# Appendix A

## Bibliography


[1] Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84, 1990.

[2] Proceedings the eighth international symposium on software reliability engineering. In *Proceedings The Eighth International Symposium on Software Reliability Engineering*, Los Alamitos, CA, USA, nov 1997. IEEE Computer Society.

[3] Richard Bache and Monika Mullerburg. Measures of testability as a basis for quality assurance. *Softw. Eng. J.*, 5(2):86–92, apr 1990.

[4] V. R. Basili and B. T. Perricone. Software errors and complexity: an empirical investigation. *Communications of the ACM*, 27(1):42–52, January 1984.

[5] Yanping Chen, Robert L Probert, and Kyle Robeson. Effective test metrics for test strategy evolution. In *Proceedings of the 2004 conference of the Centre for Advanced Studies on Collaborative research*, pages 111–123. Citeseer, 2004.

[6] Ermira Daka and Gordon Fraser. A survey on unit testing practices and problems. In *2014 IEEE 25th International Symposium on Software Reliability Engineering*, pages 201–211, 2014.

[7] Ignacio Dobles, Alexandra Martínez, and Christian Quesada-López. Comparing the effort and effectiveness of automated and manual tests. In *2019 14th Iberian Conference on Information Systems and Technologies (CISTI)*, pages 1–6, 2019.

[8] Felix Dobslaw, Robert Feldt, David Michaëlsson, Patrik Haar, Francisco Gomes de Oliveira Neto, and Richard Torkar. Estimating return on investment for gui test automation frameworks. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pages 271–282, 2019.

[9] Elfriede Dustin, Jeff Rashka, and John Paul. *Automated software testing: Introduction, management, and performance: Introduction, management, and performance*. Addison-Wesley Professional, 1999.

[10] Badr El Khalyly, Abdessamad Belangour, Mouad Banane, and Allae Erraissi. A new metamodel approach of ci/cd applied to internet of things ecosystem. In *2020 IEEE 2nd International Conference on Electronics, Control, Optimization and Computer Science (ICECOCS)*, pages 1–6, 2020.

[11] Donald Firesmith. Four types of shift left testing. Carnegie Mellon University, Software Engineering Institute's Insights (blog), Mar 2015. Accessed: 2024-Feb-28.

[12] Adam Kolawa and Dorota Huizinga. *Automated Defect Prevention: Best Practices in Software Management*. Wiley-IEEE Computer Society Press, 2007.

[13] D Richard Kuhn, Raghu N Kacker, and Yu Lei. *Introduction to combinatorial testing*. CRC press, 2013.

[14] Rick Kuhn, Raghu Kacker, Yu Lei, and Justin Hunter. Combinatorial software testing. *Computer*, 42(8):94–96, 2009.

[15] H.K.N. Leung and L. White. A study of integration testing and software regression at the integration level. In *Proceedings. Conference on Software Maintenance 1990*, pages 290–301, 1990.

[16] Dr Madhavi. A white box testing technique in softwre testing: Basis path testing. *Journal for Research*, 1(04), 2016.

[17] Sonali Mathur and Shaily Malik. Advancements in the v-model. *International Journal of Computer Applications*, 1(12):29–34, 2010.

[18] E.M. Maximilien and L. Williams. Assessing test-driven development at ibm. In *25th International Conference on Software Engineering, 2003. Proceedings.*, pages 564–569, 2003.

[19] Kai Petersen, Claes Wohlin, and Dejan Baca. The waterfall model in large-scale development. In Frank Bomarius, Markku Oivo, Päivi Jaring, and Pekka Abrahamsson, editors, *Product-Focused Software Process Improvement*, pages 386–400, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[20] Kai Petersen, Claes Wohlin, and Dejan Baca. The waterfall model in large-scale development. In *Product-Focused Software Process Improvement: 10th International Conference, PROFES 2009, Oulu, Finland, June 15-17, 2009. Proceedings 10*, pages 386–400. Springer, 2009.

[21] Du Qingfeng and Dong Xiao. An improved algorithm for basis path testing. In *2011 International Conference on Business Management and Electronic Information*, volume 3, pages 175–178, 2011.

[22] Dudekula Mohammad Rafi, Katam Reddy Kiran Moses, Kai Petersen, and Mika V. Mäntylä. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In *2012 7th International Workshop on Automation of Software Test (AST)*, pages 36–42, 2012.

[23] Shaun Smith and Gerard Meszaros. Increasing the effectiveness of automated testing. In *Proceedings of the Third XP and Second Agile Universe Conference*, pages 88–91, 2001.

[24] Andreas Spillner and H Bremenn. The w-model. strengthening the bond between development and test. In *Int. Conf. on Software Testing, Analysis and Review*, pages 15–17, 2002.

[25] Amjed Tahir. A study on software testability and the quality of testing in object-oriented systems. 2016.

[26] Laurie Williams, Gunnar Kudrjavets, and Nachiappan Nagappan. On the effectiveness of unit test automation at microsoft. In *2009 20th International Symposium on Software Reliability Engineering*, pages 81–89, 2009.

[27] Linbin Yu, Yu Lei, Raghu N. Kacker, and D. Richard Kuhn. Acts: A combinatorial test generation tool. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*, pages 370–375, 2013.

[28] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, dec 1997.

# Appendix B

## Used Software

As advised in the *Methodological guideline 5/2023*[1], the following software was used while writing this thesis:

- Grammarly - style and grammar checking[2]
- Writeful - style and grammar checking[3]
- ChatGpt OpenAi - rephrasing and style suggestions[4]

[1] https://www.cvut.cz/sites/default/files/content/d1dc93cd-5894-4521-b799-c7e715d3c59e/en/20231003-methodological-guideline-no-52023.pdf
[2] https://www.grammarly.com/
[3] https://www.writefull.com/
[4] https://chat.openai.com/

# Appendix C

## Code Examples

**Listing C.1:** Redmine - New Issue automated test case

```
it('test case no. 14', () => {
    const currentDate = new Date();
    const currentDateString = currentDate.toLocaleDateString('cs-CZ')
        .replace(' ', '') + " " + currentDate.toLocaleTimeString('cs-CZ');
    //subject - Valid
    const subject = "TestCaseNum14 " + currentDateString
    cy.get('input[id="issue_subject"]').type(subject);
    //assignee - Empty (default)
    //category - new
    const category = "Category " + currentDateString
    cy.get('a[title="New category"]').click();
    cy.get('input[id="issue_category_name"]').type(category);
    cy.xpath('//p/input[@value="Create"]').click();
    //Target version - New
    const version = "1.1 " + currentDateString
    cy.get('a[title="New version"]').click();
    cy.get('input[id="version_name"]').type(version);
    cy.xpath('//p/input[@value="Create"]').click();
    //file - not chosen
    //parent task - not chosen
    //end date - less than start
    const endDate = new Date()
    endDate.setDate(currentDate.getDate() - 10)
    const dayEnd = endDate.getDate().toString().padStart(2, '0');
    const monthEnd = (endDate.getMonth() + 1).toString().padStart(2, '0');
    const yearEnd = endDate.getFullYear();
    const formattedEndDate = `${monthEnd}/${dayEnd}/${yearEnd}`;
    cy.get('input[id="issue_due_date"]')
        .invoke('val', endDate.toISOString().split('T')[0]).trigger('input');
    //estimated time - valid
    const estimatedTime = 10
    cy.get('input[id="issue_estimated_hours"]').type(estimatedTime);

    //----evaluation----
    cy.xpath('//form/input[@value="Create"]').click();
    cy.contains("Due date must be greater than start date").should('exist');
})
```

**Listing C.2:** Redmine - Time Tracking automated test case

```
it('test case no. 7', () => {
    const currentDate = new Date();
    const currentDateString = currentDate.toLocaleDateString('cs-CZ')
        .replace(' ', '') + " " + currentDate.toLocaleTimeString('cs-CZ');

    cy.get('input[id="time_entry_comments"]')
        .type(currentDateString + " test case no. 7");
    //project - entered
    const project = 'Test Project'
    cy.get('select[id="time_entry_project_id"]').select(project);
    //issue - entered
    cy.wait(500)
    cy.get('input[id="time_entry_issue_id"]').type('13');
    //date - future
    const date = new Date()
    date.setDate(currentDate.getDate() + 10)
    const day = date.getDate().toString().padStart(2, '0');
    const month = (date.getMonth() + 1).toString().padStart(2, '0');
    const year = date.getFullYear();
    const formattedDate = `${month}/${day}/${year}`;
    cy.get('input[id="time_entry_spent_on"]')
        .invoke('val', date.toISOString().split('T')[0]).trigger('input');
    //hours - valid
    const hours = 10
    cy.get('input[id="time_entry_hours"]').type(hours);

    cy.get('input[value="Create"]').click();

    //----validation----
    cy.get('div[id="flash_notice"]').should('exist');
    cy.contains(currentDateString + " test case no. 7").should('exist');
    cy.contains(currentDateString + " test case no. 7")
        .parent().contains(hours + ":00").should('exist')
    cy.contains(formattedDate).should('exist')
  })
```

**Listing C.3:** Trac - New ticket automated test case

```
it('testcase4', () => {
    const currentDate = new Date();
    const currentDateString = currentDate.toLocaleDateString('cs-CZ')
        .replace(' ', '') + " " + currentDate.toLocaleTimeString('cs-CZ');
    //summary valid
    const summary = "TestCase4 " + currentDateString
    cy.get('input[id="field-summary"]').type(summary);
    //reporter default
    //description nonempty
    const description = "description";
    cy.get('textarea[id="field-description"]').type(description);
    //type task
    const type = "task";
    cy.get('select[id="field-type"]').select(type);
```

```
//milestone WithoutDueDate
const milestone = "milestone1";
cy.get('select[id="field-milestone"]').select(milestone);
//version PastVersion
const version = "1.1past";
cy.get('select[id="field-version"]').select(version);
//priority blocker
const priority = "blocker";
cy.get('select[id="field-priority"]').select(priority);
//component component1
const component = "component1";
cy.get('select[id="field-component"]').select(component);
//keywords empty
//cc nonempty
const cc = "any_cc";
cy.get('input[id="field-cc"]').type(cc);
//owner other
const owner = "any_owner"
cy.get('input[id="field-owner"]').clear().type(owner);
//severity Low
const severity = "Low";
cy.get('select[id="field-severity"]').select(severity);
//fileattached false

cy.get('input[value="Create ticket"]').click();

//--evaluation--
//summary exists
cy.contains(summary).should('exist');
//reporter default
cy.get('td[headers="h_reporter"]').find('a')
.contains('trac_admin').should('exist');
//owner other
cy.get('td[headers="h_owner"]').find('a').contains(owner).should('exist');
//priority blocker
cy.get('td[headers="h_priority"]').find('a')
    .contains(priority).should('exist');
//milestone WithoutDueDate
cy.get('td[headers="h_milestone"]').find('a')
    .contains(milestone).should('exist');
//component component1
cy.get('td[headers="h_component"]').find('a')
    .contains(component).should('exist');
//version PastVersion
cy.get('td[headers="h_version"]').find('a').contains(version).should('exist');
//severity Low
cy.get('td[headers="h_severity"]').find('a')
    .contains(severity).should('exist');
//keywords empty
//cc nonempty
cy.get('td[headers="h_cc"]').find('a').contains(cc).should('exist');
//description nonempty
```

```
    cy.xpath('//div[@class="searchable"]/p')
        .contains(description).should('exist');
    //type task
    cy.get('span[class="trac-type"]').find('a').contains(type).should('exist');
})
```

**Listing C.4:** Tracks - New action automated test case

```
it('test case no. 13', () => {
    const currentDate = new Date();
    const currentDateString = currentDate.toLocaleDateString('cs-CZ')
        .replace(' ', '') + " " + currentDate.toLocaleTimeString('cs-CZ');
    //description valid
    const description = "TestCase13 " + currentDateString
    cy.get('input[id="todo_description"]').type(description);
    //notes nonempty
    const notes = "Nonempty notes"
    cy.get('textarea[id="todo_notes"]').type(notes)
    //project empty
    //context HiddenContext
    const context = "HiddenContext"
    cy.get('input[id="todo_context_name"]').clear().type(context)
    //tags mone tag
    const tags = "tag1"
    cy.get('input[id="tag_list"]').type(tags)
    //duedate Future
    const dueDate = new Date()
    dueDate.setDate(currentDate.getDate() + 10)
    const dayDue = dueDate.getDate().toString().padStart(2, '0');
    const monthDue = (dueDate.getMonth() + 1).toString().padStart(2, '0');
    const yearDue = dueDate.getFullYear();
    const formatteddueDate = `${dayDue}/${monthDue}/${yearDue}`;
    cy.get('input[id="todo_due"]').type(formatteddueDate)
    cy.xpath('//*[@id="ui-datepicker-div"]/div[2]/button[2]').click()
    //showfrom empty
    //dependsOn empty
    cy.get('button[id="todo_new_action_submit"]').click();

    //--evaluation--
    cy.visit('http://localhost/search')
    cy.get('input[id="search"]').type(description)
    cy.xpath('//*[@id="search-form"]/input[2]').click()
    cy.contains(description).should('exist')
    cy.contains("Due in 10 days").should('exist')
    cy.contains('tag1').should('exist')
    cy.get('a[title="View context: ' + context +  '"]').should('exist')
    cy.xpath('//div/a[3]/img').click()
    cy.contains(notes).should('exist')
})
```

**Listing C.5:** Tracks - Configuration automated test case

```
it.skip('test case n. 1', () => {
    const due = 'Due in _____ days'
    cy.get('select[id="prefs_due_style"]').select(due)
    //ShowCompletedProjects false
    cy.get('select[id="prefs_show_completed_projects_in_sidebar"]')
        .select('false')
    //ShowHiddenProjects false
    cy.get('select[id="prefs_show_hidden_projects_in_sidebar"]').select('false')
    //ShowHiddenContexts false
    cy.get('select[id="prefs_show_hidden_contexts_in_sidebar"]').select('false')
    //GoToProject false
    cy.get('select[id="prefs_show_project_on_todo_done"]').select('false')
    //ShowNumberOfCompleted positive
    const NumberOfCompleted = 5
    cy.get('input[id="prefs_show_number_completed"]').clear()
        .type(NumberOfCompleted)
    cy.get('button[id="prefs_submit"]').click()

    //----evaluation----
    cy.visit('http://localhost/')
    cy.contains('Hidden context').should('not.exist');
    cy.contains('Hidden projects').should('not.exist');
    cy.contains('Completed projects').should('not.exist');
    //+1 because there is an extra element
    cy.get('div[id="completed_container_items"]').children()
        .should('have.length.lte', NumberOfCompleted + 1);

    const currentDate = new Date();
    const currentDateString = currentDate.toLocaleDateString('cs-CZ')
        .replace(' ', '') + " " + currentDate.toLocaleTimeString('cs-CZ');
    const description = "TestCaseConf1 " + currentDateString
    cy.get('input[id="todo_description"]').type(description);
    cy.get('input[id="todo_project_name"]').type("ActiveProject")
    const dueDate = new Date()
    dueDate.setDate(currentDate.getDate() + 6)
    const dayDue = dueDate.getDate().toString().padStart(2, '0');
    const monthDue = (dueDate.getMonth() + 1).toString().padStart(2, '0');
    const yearDue = dueDate.getFullYear();
    const formatteddueDate = `${dayDue}/${monthDue}/${yearDue}`;
    cy.get('input[id="todo_due"]').type(formatteddueDate)
    cy.xpath('//*[@id="ui-datepicker-div"]/div[2]/button[2]').click()
    cy.get('button[id="todo_new_action_submit"]').click();

    cy.contains(description).parent().parent().as('action');
    cy.get('@action').contains('Due in').should('exist')
    cy.get('@action').find('input[class="item-checkbox"]').click()
    cy.url().should('eq', 'http://localhost/')
})
```