Assignment of bachelor's thesis

Title:	Web application quality assurance and test case design
Student:	Marie Kalousková
Supervisor:	Ing. Oldřich Malec
Study program:	Informatics
Branch / specialization:	Web and Software Engineering, specialization Software Engineering
Department:	Department of Software Engineering
Validity:	until the end of summer semester 2024/2025

Instructions

- Analyze and describe different types of tests that can and should be used for web application quality assurance.
- Analyze and describe different techniques for creating test cases.
- Compare manual testing and test automation. Contemplate if test automation is always better, or even possible.
- Describe in depth the purpose, usage, and benefits of E2E testing, and also the problems and often made mistakes.
- Analyze Atlantis - a warehouse management system - design the E2E test cases, decide which are suited for automation, and implement these using Playwright. Describe the testing environment and setup. In the end, they should be runnable from a local machine with a summary of which test cases passed/failed.

Electronically approved by Ing. Michal Valenta, Ph.D. on 23 January 2024 in Prague.

Bachelor's thesis

WEB APPLICATION QUALITY ASSURANCE AND TEST CASE DESIGN

Marie Kalousková

Faculty of Information Technology
Department of Software Engineering
Supervisor: Ing. Oldřich Malec
May 16, 2024

Czech Technical University in Prague
Faculty of Information Technology
© 2024 Marie Kalousková. All rights reserved.
This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author's permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis: Kalousková Marie. *Web application quality assurance and test case design.* Bachelor's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

Contents

Acknowledgments **v**

Declaration **vi**

Abstract **vii**

List of abbreviations **viii**

Introduction **1**

1 Quality Assurance **3**
1.1 Web application QA 4
1.2 Testing approches 4
1.2.1 Static testing 4
1.2.2 Dynamic testing 5

2 Test types **6**
2.1 Functional tests 6
2.2 Non-functional tests 6
2.2.1 Efficiency tests 7
2.2.2 Compatibility tests 7
2.2.3 User experience tests 7
2.2.4 Reliability tests 8
2.2.5 Security tests 8
2.2.6 Maintainability tests 9
2.2.7 Portability tests 9

3 Test desing techniques **10**
3.1 Black-box test techniques 11
3.1.1 Equivalence partitioning 11
3.1.2 Boundary value analysis 12
3.1.3 Decision table testing 13
3.1.4 State transition testing 14
3.2 White-box test techniques 14
3.2.1 Statement testing 15
3.2.2 Branch testing 15
3.2.3 Condition testing 15
3.2.4 Decision condition testing 15
3.2.5 MC/DC 16

3.2.6 All paths coverage . . . 16
3.3 Experience-based test techniques . . . 17
3.3.1 Error guessing . . . 17
3.3.2 Exploratory testing . . . 17
3.3.3 Checklist-based testing . . . 17

4 Test automation 18
4.1 Benefits of test automation . . . 18
4.2 Risks of test automation, i.e., what speaks for manual testing . . . 19
4.2.1 SEARCH of automation testing . . . 19
4.2.2 Depending on the people . . . 20
4.3 Test automation of GUI interactions . . . 20
4.4 Conclusion . . . 20

5 End-to-End Testing 22
5.1 Test automation pyramid . . . 23
5.2 Benefits of E2E testing . . . 25
5.3 Pitfalls of E2E testing . . . 25
5.4 E2E tests automation . . . 25

6 Atlantis 27
6.1 About . . . 27
6.2 How it all started . . . 28
6.2.1 Examples of found bugs . . . 28
6.2.2 User manual and diagrams . . . 29
6.3 E2E test case design . . . 30
6.4 Database . . . 32
6.4.1 Init state and getting data into database . . . 32
6.4.2 The importance of cleaning up . . . 33
6.5 Playwright . . . 33
6.5.1 About . . . 34
6.5.2 Codegen . . . 34
6.5.3 Locator strategies . . . 35
6.5.4 Page Object Models . . . 35
6.5.5 Fixtures . . . 35
6.6 Setup . . . 35
6.7 Implementation . . . 38
6.8 Execution . . . 40

Conclusion 43

Attachments 48

List of Figures

5.1 Martin Fowler's test pyramid [11] 23
5.2 Scope of different tests in pyramid from semaphore web [13] 24
5.3 Kent Dodds' JS test trophy [14] 24

6.1 Activity diagram of scanning while item picking for personal collection . . 30
6.2 Part of a test report 41
6.3 Part of a test case in report 42

List of code listings

3.1 Statement vs. branch coverage example 15
3.2 Decision condition coverage example 16

6.1 Fragment of playwright.config.js 36
6.2 Define projects 37
6.3 Defining Zebra device 38
6.4 Part of an automated test case 39

I would like to thank my supervisor, Ing. Oldřich Malec, for his guidance and valuable advice, and for letting me apply my theoretical quality assurance knowledge to an existing application. He and Ing. Jiří Hunka sparked my interest in testing a few years back during my studies, and I will be forever grateful for that. I also want to thank my family and friends for their unending support.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Section 2373(2) of Act No. 89/2012 Coll., the Civil Code, as amended, I hereby grant a non-exclusive authorization (licence) to utilize this thesis, including all computer programs that are part of it or attached to it and all documentation thereof (hereinafter collectively referred to as the "Work"), to any and all persons who wish to use the Work. Such persons are entitled to use the Work in any manner that does not diminish the value of the Work and for any purpose (including use for profit). This authorisation is unlimited in time, territory and quantity.

In Prague on May 16, 2024

Abstract

This bachelor thesis dives into the topic of quality assurance. It contains an overview of different test types and introduces various test techniques used to design test cases, e.g., black-box, white-box, and experience-based techniques. The work contemplates the idea of test automation, describes its benefits and potential risks, and explains the usage of E2E testing. The practical part focuses on E2E testing on a web application named Atlantis, a warehouse management system. It describes the first steps that helped to build a base for the automation of the E2E test cases, such as gaining domain knowledge and familiarizing with Atlantis. It introduces a testing framework named Playwright and its features and explains the test environment setup and possible execution of the E2E test suite. The last part contains information about how the designing part was approached and shows an example of an automated test case. In the end, manual testing helped detect various defects, and the created automated test suite serves as a defense against regression defects coming from the key application features.

Keywords web application, quality assurance, test case design, E2E testing, test techniques, test automation, Playwright

Abstrakt

Tato bakalářská práce se zabývá tématem zajištění kvality. Obsahuje přehled různých typů testů a představuje různé testovací techniky používané k navrhování testovacích případů, např. techniky černé skříňky, bílé skříňky a ty založené na zkušenostech. Práce rozebírá myšlenku automatizace testování, popisuje její výhody a možná rizika, a vysvětluje E2E testování. Praktická část je zaměřena na E2E testování webové aplikace Atlantis, systému sloužícímu ke správě skladů. Popisuje první kroky, které pomohly vybudovat základ pro automatizaci testovacích E2E případů, jako je získání doménových znalostí a seznámení se s Atlantisem. Představuje Playwright, platformu na vývoj testů, a jeho funkce, a vysvětluje nastavení testovacího prostředí a možné spuštění E2E testovací sady. Poslední část obsahuje informace o tom, jak bylo přistoupeno k návrhové části, a ukazuje příklad automatizovaného testovacího případu. Ruční testování nakonec pomohlo objevit různé defekty a vytvořená sada automatizovaných testů slouží jako obrana proti regresním defektům pocházejícím z klíčových funkcí aplikace.

Klíčová slova webová aplikace, zajištění kvality, návrh testovacích případů, E2E testování, techniky testování, automatizace testování, Playwright

List of abbreviations

API	Application programming Interface
BB	Black-box
BE	Back end (or Back-end)
BVA	Boundary value analysis
CFG	Control flow graph
CI	Continuous integration
CSS	Cascading Style Sheets
CTFL	Certified Tester Foundation Level
DCC	Decision condition coverage
DLL	Dynamic-link library
DOM	Document Object Model
EP	Equivalence partitioning
E2E	End-to-End
FE	Front end (or Front-end)
GUI	Graphical User Interface
HTML	Hypertext Markup Language
ISTQB	International Software Testing Qualifications Board
JS	JavaScript
JSON	JavaScript Object Notation
MC/DC	Modified condition/decision coverage
OWASP	Open Web/Worldwide Application Security Project
POM	Page Object Model
QA	Quality Assurance
QC	Quality Control
SEARCH	setup, execution, analysis, reporting, cleanup, help
SDLC	Software Development Lifecycle
SP1	Software Project 1
SP2	Software Project 2
UI	User Interface
UX	User Experience
WB	White-box
WSL	Windows Subsystem for Linux
XPath	XML Path Language

Introduction

In today's digital world, more and more people use web applications. They help with online shopping and getting in touch with friends on chats or during multiplayer games. They can be helpful tools in everyday life, but also beneficial for the work life, helping employees to get the stuff done.

Because many web applications are developed every day, the demands on them are becoming higher, focusing on fast performance, reliability, and functionality. If the application is buggy or lacks desired functionality, customers might switch to an alternative. That leaves developers in a competitive environment, pushing them to quickly release and add new functionality. Unfortunately, this stress often leads to the release of a faulty web application—the developer teams that can avoid that end up on top. So, how can the teams create a high-quality product and set the right processes to prevent a faulty release? What is missing is quality assurance and proper testing.

The thesis aims to encapsulate the most interesting and essential practices and thoughts about testing and quality assurance based on books and articles written by experts. Texts written by the International Software Quality Board or Microsoft test professionals provide input to the theory behind testing and also to the practical approach to it.

The theoretical part explains why quality assurance and testing are critical when developing a web application. Describes different test types typically used for quality assurance. Describes the best practices and techniques for designing test cases, such as black-box, white-box, and experience-based techniques. Compares manual testing and test automation and discusses if test automation is always better or even possible. And finally, it describes the purpose, usage, and benefits of End-to-End (E2E) testing and the pitfalls of this kind of testing.

The thesis's practical goal is to analyze Atlantis—a warehouse management system—running as a web application on computers and Zebra mobile devices. The student performs extensive manual testing first, then designs the E2E test cases and identifies which are best suited for automation. She implements these using the Playwright testing framework. Describes the testing environment and the setup of the framework. In the end, the implemented test cases are runnable from a local machine, and after execution,

a report with which test cases passed/failed is created. This all contributes to a better quality of Atlantis, and the possibility to run the test suite every time before a release can help catch failures before they reach the customer.

Chapter 1
Quality Assurance

> *"Quality in a product or service is not what the supplier puts in. Is it what the customer gets out and is willing to pay for. A product is not quality because it is hard to make and costs a lot of money, as manufacturers typically believe. This is incompetence. Customers pay only for what is of use to them and gives them value. Nothing else constitutes quality." -Peter F. Drucker [1]*

Pursuing a high-quality product is a primary objective in web development, encapsulating seamless user experience and robust functionality. But what means a high-quality product and how can we produce it consistently? These are the questions that drive the domain of quality management.

Quality Assurance (QA) and *Quality Control (QC)* are essential quality management processes. QA embodies a proactive, preventive approach dedicated to improving processes to prevent defects from surfacing during development. It is the oversight strategy, which is what methods were applied to each process in our company to comply with customer expectations—the assurances—and is based on the principle that following good processes creates a high-quality product. Its goal is to build a product that meets customer expectations regarding functionality, usability, reliability, performance and design. [2]

While some texts use QC or *testing* interchangeably with QA, they carry distinct meanings. Testing falls under QC and involves examining the product to identify defects and evaluate its quality. As a corrective approach, QC takes a reactive stance, focusing on detecting and fixing defects before the product reaches the customer.

Testing is crucial for controlling quality. Its objectives are verification of requirements, product validation against users' expectations, and building confidence in the quality. The test process consists of activities such as test planning, test monitoring and control, test analysis, test design, test implementation, test execution, and test completion. These activities are an integral part of the software development lifecycle (SDLC). [2]

QA uses test results to improve the test and development processes, while QC uses them to fix defects. That said, choosing the correct type of testing or test design technique falls not only into QC but also into QA. And that's what causes a lot of confusion in these two terms, not only in job posts but also in books and academic texts. Where is the line between what we do for QA and what we do for QC? [2, 3]

According to *ISO 9000:2015*, QC falls into QA. QA is a "part of quality management focused on providing confidence that quality requirements will be fulfilled". And QC is "part of quality management focused on fulfilling quality requirements". [3]

1.1 Web application QA

Web Application QA plays a crucial role in delivering a reliable, user-friendly, and secure web application that meets the needs and expectations of its users. The accessibility of web applications via web browsers and the internet presents significant challenges, particularly in protecting against potential cyber attacks. But it's essential to use all kinds of test types, not only security testing, to uncover the remaining defects and deliver a high-quality application.

1.2 Testing approches

There are two approaches for verifying and validating the application and ensuring the quality of it: static testing and dynamic testing.

1.2.1 Static testing

Static testing does not rely on code execution and can be applied to any work product that is easily understandable by humans, e.g., source code, user documentation, specification documentation, test cases, and models. This form of testing is instrumental in identifying inconsistencies in code formatting using tools like parsers and linters. Additionally, static testing can uncover potential risks, such as incorrect number of parameters and undefined variables, as well as detect unreachable code through control flow analysis. Analyzing data flow is another part of static testing, designed to uncover unwarranted side-effects of variable conversion or access to a variable that is out of scope or already destroyed.

One form of static testing is reviews, ranging from informal to formal. *Informal reviews* focus on detecting anomalies without following to a structured process. *Walkthroughs*, led by the author of the code, aim to educate the reviewers and gain consensus. *Technical reviews*, moderated by a facilitator, evaluate the technical side of the work product. *Inspections* follow a very formal process through which they collect metrics to improve the SDLC. [1, 2]

1.2.2 Dynamic testing

Dynamic testing is restricted to the executable code, focusing on evaluating the software's behavior during runtime. While dynamic testing may seem limited in scope compared to its static counterpart, it is not inferior. In fact, these two testing strategies are complementary, each offering unique insights into the software's quality.

Dynamic testing excels in measuring quality characteristics *dependent upon executing code*, such as performance metrics, runtime failures, wild pointers, memory leaks, and bugs in external libraries. It provides invaluable feedback on the software's functionality and reliability under varying conditions by simulating real-world scenarios and interactions. [1, 2]

Chapter 2

Test types

This chapter will explore the different test types, allowing developers and QA engineers to systematically assess various aspects of a software application. Each test type serves a unique purpose and employs specific methodologies to uncover defects. By strategically applying these tests throughout SDLC, teams can identify defects early, leading to more reliable software releases. Whether it's verifying the features, validating compatibility across platforms, or fortifying against potential security threats.

The ISTQB® Foundation Level Syllabus [2] addresses four different test types: functional testing, non-functional testing, black-box testing, and white-box testing. Chapter 3 will describe black-box and white-box testing as test techniques, as it better grasps the idea behind these test types.

2.1 Functional tests

Functional testing is about testing functionality, hence the name. It tests what the application should and shouldn't do, e.g., "Application should accept shipping order from a logged-in user."

2.2 Non-functional tests

Non-functional testing is merely everything else that is not functional testing. These tests are sometimes called quality tests, as they evaluate how well the application behaves in performance, reliability, maintainability, etc. [4] It is still connected to the user requirements and expectations as these requirements can be: "The application should process a shipping order in less than a minute."

2.2.1 Efficiency tests

Efficiency testing encompasses various aspects to evaluate an application's performance and resource utilization. It includes measuring load and processing time and assessing the application's behavior under stress conditions.

Performance testing, a subset of efficiency testing, measures the system's responsiveness. It involves analyzing processing, response, throughput, and turnaround times under normal operating conditions to ensure optimal performance.

Additionally, efficiency testing evaluates the application's ability to handle stress and heavy workloads. *Stress* and *scalability tests* simulate extreme conditions by subjecting the application to reduced resource availability or an enormous number of connected users. These tests help identify potential bottlenecks and weaknesses in the system, such as memory leaks that, over time, can gradually deplete available memory, causing the application to crash. These failures can be found more effectively using stress testing instead of running the application under normal conditions for a long period. [1]

2.2.2 Compatibility tests

Compatibility testing aims to ensure the application works as intended across different operating systems, devices, and browsers. It also includes evaluating the ability to interact with older and newer versions of related software components such as libraries, plugins, and external APIs.

With many browsers on the market, the compatibility testing checks if the application renders correctly on the supported browsers and is responsive to different screen sizes and resolutions, allowing users a seamless user experience. [5]

2.2.3 User experience tests

User experience (UX) goes beyond usability; it is also about usefulness, meaningfulness, and emotional impact. UX evaluation is usually set in a lab where the test user is not distracted by other external influences. Lab environment has other benefits, such as installed cameras that can track everything from eye movement to create heatmaps to how many times the user switched from keyboard to mouse. Users follow written instructions and complete predefined tasks.

UX is almost always evaluated qualitatively. The most known technique is called *think-aloud*: The test user is encouraged to share everything that goes through her mind when completing the tasks, e.g., what she is looking for on the screen, why she is looking for something, etc. It's important to remember that we are evaluating the application, not the test user, so any comments or adverse reactions to her performance are not in place.

There are also quantitative measures. Task performance can be timed, user errors can be counted, or the test users can fill out various questionnaires. The most known

questionnaires are the *standardized User Experience Questionnaire (UEQ)*, the Questionnaire for User Interface Satisfaction (QUIS), the System Usability Scale Questionnaire (SUS), and one for Usefulness, Satisfaction, and Ease of Use (USE). Each one focuses on a slightly different area of UX. [6]

Usability testing is more about how easily users can understand and operate the application to complete their tasks. [4]

2.2.4 Reliability tests

When talking about reliability, we are talking about recoverability, robustness, and fault tolerance. Reliability testing evaluates if the application reacts to invalid inputs and incorrect data without failure, tests error tolerance and exception handling, and judges the application's ability to reestablish performance and recover the affected data. It looks for immediate detection of failures followed by reliable failover. [1, 7]

2.2.5 Security tests

"Security issues often have no symptoms, right up until the time a hacker breaks in and torches the system. Or, maybe worse, the hacker breaks in, steals critical data, and then leaves without leaving a trace. Ensuring that people can't see what they should not have access to is a major task of security testing." [1]

The question is how to defend our application against these attacks; an SQL injection that can lead to unauthorized access, a buffer overflow that allows a hacker to rewrite a return address to jump where she wants to, breaking encryption that can lead to leakage of sensitive data, or being overwhelmed by requests in a distributed denial-of-service attack (DDoS). [1]

There are a few interesting websites that might serve as guidelines for security testing: Common Vulnerabilities and Exposure site (CVE), which lists and identifies vulnerabilities and exposures; Common Weakness Enumeration (CWE) website that categorizes known weaknesses; and Common Attacks Pattern Enumeration and Classification (CAPEC), which is a list of security attack patterns. [8]

Probably the most known "checklist of threats" for web applications used by developers and penetration testers is *OWASP Top Ten*, compiled by the Open Web/Worldwide Application Security Project (OWASP), which lists the most critical web application security risks. OWASP keeps updating it based on actual data gathered from organizations. [9]

The job of a *penetration tester* is to try to break into the web application by simulating real-world attack scenarios. The penetration tester documents the findings, used exploit techniques, and identified vulnerabilities.

Organizations usually employ an external penetration testing agency that tries to breach the system with some limited knowledge to simulate a real-world attack. Explicit

consent and cooperation of the organization are a must. Otherwise, it is illegal and can be sanctioned appropriately.

Phishing campaigns can be part of penetration testing but are usually done as a separate activity as they focus on educating employees about potential threats rather than testing the application's security. The goal is to assess how employees respond to suspicious emails and messages—whether they click on suspicious links, download attachments, or disclose sensitive information such as their username and password.

2.2.6 Maintainability tests

Maintainability is about the ease of modifying and updating code and *testability*. Writing testable code with high coupling and low cohesion is the heart of good software design.

Part of the maintainability testing falls under static testing. That part assesses the system's analyzability. Poor analyzability is caused by spaghetti code, lack of documentation, and poor or nonexistent standards and guidelines (indentation, naming conventions). [1]

2.2.7 Portability tests

Portability testing examines the installation and deployment process of the software application on different systems, evaluating how hard the transfer is and if it is even possible. It looks for problems like apps consuming each other's resources, undefined dependencies after an update, and DLL hell (Dynamic-link libraries hell)—shared resources are incompatible. [1]

Chapter 3

Test desing techniques

"Good testers seem to understand that no matter how absurd clicking a button 50 times in a row might sound, somewhere, a customer will do exactly that." [4] This chapter is about test case design and will go through different test design techniques that aim to identify (critical) scenarios to be tested.

One of the main testing principles states that "*exhaustive testing* is impossible" [2]. Exhaustive testing means testing every possible combination of inputs and every possible scenario. Let's look at a trivial example of buying beer.

Imagine a simple web application with a form with one input field for the user's age and a submit button. After submitting, the application checks the age, and a message appears on the screen showing if the user can buy alcohol in Czechia or not. In Czechia, the legal age for buying alcohol is 18 years old. Even if we only test reasonable numbers between 0 and 122—the oldest person who ever lived died as 122-year-old. The manual tester would need to try all the different values, i.e., submit the form 123 times and check the corresponding results. That does not sound that impossible, right? Now, imagine instead of inputting the age, the users will be met with an input field for their date of birth. That means *around 44,926 possible inputs*, even though the logic of who can buy beer is the same. Even automation won't help. The test analyst will still need to prepare the combination of corresponding inputs and outputs.

That is why we need test techniques, systematic approaches to design test cases. It is important to understand that there is *no silver bullet.* There is no universal approach to how to test an application. There is no ultimate test technique. Each test technique has its benefits and drawbacks. Most of the techniques complement each other.

Test techniques help develop a small set of test cases that envelop all possible inputs and outputs without actually checking all the combinations. Because it is not exhaustive testing, it can miss some defects, but the systematic approach should help find most of them if implemented correctly. [2]

3.1 Black-box test techniques

Black-box (BB) test techniques, also called *specification-based techniques*, derive test cases based on documentation of specified behavior, i.e., application specification, without knowing an actual implementation.

Test cases are completely independent of how the software is implemented. That means tests can be reused if the code is updated while the specification remains unchanged. Further, tests can be developed before the code is written, as only the existing specification is required.

The drawback of these techniques is the error of commission; the extra functionality that is not in the specification cannot be covered by the definition. Also, only a few tools provide coverage measurement for these techniques. [1, 2, 10]

3.1.1 Equivalence partitioning

Equivalence partitioning (EP) is the simplest BB technique. It divides data into partitions that do not overlap based on the expectation that they behave and are processed similarly. For each type of processing, at least one representative input value is used, and at least one representative output value is produced.

One value for each EP is sufficient as the values should behave equivalently, so if the test case detects a defect, any other value from that EP should have caught the defect as well. It doesn't matter which value is used for the test case if it's not the boundary value of the partition (more about boundary values in the following BB technique) because these values are special edge values, so they are not the proper representative of the whole partition.

Coverage is expressed as partitions exercised by test cases divided by identified partitions. [2, 10]

Let's get back to the beer example. Specification can state: "A user fills her age into the age field and submits it. If a negative number is inputted, the application should alert the user that it is not a valid age. If the age is between 0 and 17 inclusive, the user should be told she cannot drink beer in Czechia. Being above 18 years old and less than 122 years old inclusive, she is allowed to drink beer. Above 122 years old exclusive, she should be alerted that this is likely not her age, but she can drink. It is impossible to input anything other than a number by default."

Let's break it into the EPs sentence by sentence. The first one leads us to the fact that we will look at the input and the corresponding output for the age field. The second one is more interesting as it describes one EP.

```
I. age < 0 => "Not a valid age."
```

The third one does not yield the same result as the previous one, so it will be another partition.

```
II. age 0-17 inclusive => "You cannot drink beer."
```

The fourth one does not match either of the previously recognized partitions, leaving it as another partition.

```
III. age 18-122 inclusive => "You can drink beer."
```

The same goes for the fifth one.

```
IV. age > 122 => "That is very unlikely your age,
                  but it is possible to drink beer at that age."
```

The last sentence rules out testing of invalid values (other than the negative numbers), as the field does not permit the user to input anything but numbers. That means we do not need to add another partition for that.

We identified four EPs that do not overlap each other. From each test partition, we randomly select a value, e.g., `-29, 14, 28, 159`. Ending with *four test cases in total*, we can see how faster it will be to check than 123 values that were not even covering the invalid partitions. The change to the calendar field would only mean changing the ages to dates of birth; otherwise, it would leave the test cases untouched, as only the first and last sentences in the specification would be changed, leaving the logic of the drinking age unchanged.

3.1.2 Boundary value analysis

Boundary value analysis (BVA) is another BB technique. It builds on identified EPs, exercising the boundary (minimum and maximum) values of the range. Therefore, this technique can be used only on ordered partitions. The theory behind it is that developers are more likely to make errors on the boundaries of the partitions, e.g., off-by-one errors or complete omission of the boundary.

There is a *2-value BVA* and a *3-value BVA*, the difference being what other values need to be covered in addition to the boundary value. For each boundary value in 2-value BVA, the closest value from the adjacent partition must also be covered. That means exercising all the boundary values because the closest one to the adjacent partition is the boundary value of the adjacent partition.

Coverage is measured as the number of boundary values and its neighbor(s) exercised divided by the total number of identified values. [2, 10]

With 3-value BVA, both neighbors of the boundary value are exercised, allowing for the catching of defects that 2-value BVA overlooks. If

```
if (age >= 18) "You can drink beer."
```

is incorrectly implemented as

```
if (age == 18) "You can drink beer."
```

leads to allowing beer to an 18-year-old but not a 19-year-old or older.

Let's see it implemented using the beer example.

```
I. EP is from negative infinity to -1. -> boundary: -1

II. EP ranges from 0 to 17.            -> boundaries: 0, 17

III. EP from 18 to 122.                -> boundaries: 18, 122

IV. EP from 123 to infinity.           -> boundary: 123
```

Without infinities, there are six boundary values: `-1, 0, 17, 18, 122, 123`. With 2-value BVA, these six values will need to be exercised.

With 3-value BVA: -1 has neighbors -2 and 0. 0 has neighbors -1 and 1, etc. Therefore, 3-value BVA identifies 12 values to be exercised: `-2, -1, 0, 1, 16, 17, 18, 19, 121, 122, 123, 124`.

3.1.3 Decision table testing

Decision table testing, a BB technique, considers different combinations of conditions, i.e., input values based on equivalence partitions. The rows of the table are the conditions and the possible results. Each column represents a unique combination of conditions along with the resulting action. Each condition cell holds a Boolean value: true or false, or "-" if the condition is irrelevant to the outcome. Action cell contains an "X" if the action should occur; it is blank if it should not happen.

A table containing every combination of conditions would be too large and complex. It is common practice to delete the columns containing infeasible combinations of conditions. The table can also be reduced by merging columns if the condition does not affect the outcome ("-", remember?) or by using minimization algorithms. Creating a decision table is also helpful in finding contradictions in a specification.

To achieve 100% decision table coverage, all columns of the decision table must be exercised. [2, 10]

3.1.4 State transition testing

State transition testing is a BB technique. The system's behavior can be modeled as states and transitions between the states. It can be written as a state transition diagram or a state table. A state transition diagram is a graph representation of the states as the nodes and the transitions as the edges. "The common transition labeling syntax is as follows: event [guard condition] / action. Guard conditions and actions can be omitted if they do not exist or are irrelevant for the tester." [2] The diagram usually does not explicitly define invalid transitions.

A state table's rows represent states, and its columns represent events and guard conditions. Each cell either contains the target state (and the action) or is left empty. An empty cell represents an invalid transition.

A test case usually covers several transitions between states. Depending on what we want to measure, we can choose *all states coverage*, *valid transitions coverage*, and *all transitions coverage*. All states coverage is the number of visited states divided by the total number of states. Valid transition coverage is the number of exercised valid transitions divided by the total number of valid transitions. All transition coverage depends on both valid and invalid transitions; therefore, a state transition table is more suitable as a base for computing all transition coverage. [2]

3.2 White-box test techniques

"Performing only black-box testing does not provide a measure of actual code coverage. White-box coverage measures provide an objective measurement of coverage and provide the necessary information to allow additional tests to be generated to increase this coverage, and subsequently increase confidence in the code." [2]

White-box (WB) test techniques, also known as *structure-based* or code-based techniques, are based on examining the internal structure of a system, exercising the code implementation, and deriving test cases from it to achieve the highest possible coverage. How the coverage is defined and counted depends on the specific WB technique. The important thing is that although test cases—the input values and steps to take—are derived from the implementation, the expected results for each test case must come from the specification. If not, the test cases are pointless as they only verify that the tester understands how the code works.

The benefit is that with the implementation knowledge, the tester can write specific test cases to test boundary values (the boundary values in the code, not the specification) and even to check edge cases of the used programming language.

The WB techniques are usually used after BB techniques, as the code must be already written and executable. They also depend on both the implementation and the specification, so changes in either can invalidate the test cases—code changes usually do. Another drawback is that WB techniques are prone to *defects of omission*; if the implementation lacks functionality, test cases cannot be derived from it. [1, 2, 10]

3.2.1 Statement testing

A statement is a line of executable code. The aim is to design test cases that lead to exercising each statement at least once. This technique is usually used during unit testing, the smallest scope of testing, so the tester doesn't need to come up with complex combinations of input values to execute some missing statement deep in the code.

Statement testing helps identify unreachable code but is not demanding of the compound decisions - that a complex Boolean decision has been executed doesn't mean all possible combinations of the atomic conditions were. Using some programming languages with a packed syntax can mean that even the decision wasn't resolved both ways (true/false). [1, 2, 10]

If we use an input value age = 17 for the following code fragment, we achieve 100% statement coverage (the line is executed). However, it never exercises the if branch "You can drink beer."

```
if (age >= 18) "You can drink beer." else "You cannot drink beer."
```

■ **Code listing 3.1** Statement vs. branch coverage example

3.2.2 Branch testing

Branch testing or decision testing looks at the decisions, bearing in mind that every decision can be resolved as either true or false. The coverage is measured as exercised branches in code divided by the total sum of branches. 100% branch coverage guarantees 100% statement coverage, as executing a branch means executing all statements in that branch. Compared to statement testing, the input data for 100% coverage is more challenging to generate. [1, 2, 10]

For the previously mentioned code fragment 3.1, there will need to be two test cases to achieve 100% branch coverage. One value greater or equal to 18 exercises the if branch, and another one that is less than 18 executes the else branch.

3.2.3 Condition testing

A condition in this context means the atomic condition in a complex Boolean decision. Condition testing is based on the principle that defects can be hidden in the condition evaluation. This technique aims to ensure that each condition is resolved both ways. [1, 2, 4, 10]

3.2.4 Decision condition testing

Decision condition coverage (DCC) is focused on exercising not only every condition both ways but also every decision both ways because 100% condition coverage doesn't

necessarily mean 100% branch coverage. 100% DCC coverage guarantees 100% condition coverage and 100% decision coverage. [10] To show that, let's extend the beer example with one simple condition: "If the person is a man, he cannot drink beer at all; his age doesn't matter." This change can be incorporated in the code fragment like this:

```
if (gender == "woman" && age >= 18) {
    "You can drink beer."
} else {
    "You cannot drink beer."
}
```

■ **Code listing 3.2** Decision condition coverage example

For 100% condition coverage, we need to exercise inputs for gender: woman and man, and for age: 17-year-old and 23-year-old. The first test case can be a 17-year-old woman; the second one covers a 23-year-old man. 100% condition coverage was achieved, but only 50% decision coverage was because both these test cases resolved to "You cannot drink beer." For 100% DCC coverage, we need to add one more test case that resolves to "You can drink beer." For example, testing a 23-year-old woman.

3.2.5 MC/DC

Modified condition/decision coverage (MC/DC) goes further than DCC. The previously mentioned techniques don't consider possible combinations of conditions in the sense that each condition can affect the decision outcome. The independent effect is verified by varying the condition while holding the other conditions in that decision fixed. It still does not test all possible combinations of conditions in a decision, testing that would lead to a large number of test cases—that technique is called multiple condition coverage. MC/DC achieves stronger coverage than DCC; 100% MC/DC coverage ensures 100% DCC coverage. [10]

3.2.6 All paths coverage

All paths coverage, another WB technique, relies on creating *control flow graphs (CFGs)*. A path is an executable sequence of statements. CFG or control flow diagram is an abstract representation of the program in the form of a directed graph depicting statements (or indivisible blocks) as nodes and branches in the flow as edges. The CFG represents the code flow structure, including different branches, outcomes, and decisions the code makes. Creating CFGs is complex and time-consuming, so this technique is usually employed only for critical software. [10]

3.3 Experience-based test techniques

Experience-based techniques are the third approach to testing, complementary to the BB and WH techniques. They rely heavily on the skill of the tester and her domain expertise: “The tests may be based on a range of experiences, including end-user experience, operator experience, business or risk experience, legal experience, maintenance experience, programming experience and general problem domain experience, and so on.” [10]

When there is no time to implement the BB and WB techniques, the experience-based techniques can quickly but effectively check the application’s core. Because of their unstructured nature, they do not and cannot ensure any degree of completeness of the testing, so it’s essential to use the WB and BB techniques, too.

3.3.1 Error guessing

The error guessing technique is guided by the tester’s experience of what defects occurred in the past in the application or other similar applications and what mistakes developers usually make. [2, 10]

3.3.2 Exploratory testing

In this technique, the tester familiarizes with the application while going through it and designs the test cases simultaneously. It is used to test the general usability of software and to find untested application areas. [2, 4]

3.3.3 Checklist-based testing

“In checklist-based testing, a tester designs, implements, and executes tests to cover test conditions from a checklist. Checklists can be built based on experience, knowledge about what is important for the user, or an understanding of why and how software fails. Checklists should not contain items that can be checked automatically, items better suited as entry/exit criteria, or items that are too general.” [2] For example, the graphical interface includes big enough buttons, non-disruptive colors, flexible layout, etc.

Chapter 4

Test automation

"To Automate or Not to Automate, That Is the Question: Why should you write automated tests? Why or when should you choose manual testing over automated testing? Choosing whether to write automation and determining the extent of test automation are issues that nearly every tester must contend with at some point. If you are going to run a test only once, it doesn't make sense to automate it. However, just because you are going to run it twice doesn't mean you should automate it either." [4]

This chapter will talk about the differences between manual and automated testing. Manual testing is done by human testers by manually executing every step of a test case. In test automation, the test cases are executed by an automation software that needs to be programmed to perform these steps. There are many things to consider when choosing between manual and automated testing: the effort and the cost of automating the test cases, the execution time, the accuracy with which they report correct results, etc.

4.1 Benefits of test automation

Test automation saves time and test effort of test cases that must be repeated once in a while. It is a suitable approach for *regression tests* used in *release testing* or *smoke testing* suites. Regression tests check that a change in code somewhere did not change the functionality in the unchanged part of the application somewhere else. The release test suite consists of test cases that are repeated every release of the application. Smoke test suite groups tests checking the core functionality and metrics. These test cases would be impossible to cover manually every update. The test suite execution time is reduced, allowing earlier feedback and defect detection.

Manual testing is slow and prone to human errors caused by loss of concentration. Automated test cases, therefore, pose a lower risk by greater consistency and repeatability.

Some test cases are almost impossible to execute without automation; code convention checks, performance tests, and stress tests fall into that category. For web applications, it can be scanning a website for broken links, measuring the speed of downloads, or counting the hits. [2, 4]

4.2 Risks of test automation, i.e., what speaks for manual testing

When testing is introduced too late in the SDLC, there is sometimes no option to do anything besides manual testing. There is just no time, and the initial costs of automation are too high. It's important to consider the time spent on selecting the right tool—check if there is sufficient support, if it complies with the safety standards, how much it costs, and how hard it is to integrate the tool—or developing a tool. The problem is that the testers must also learn to use the tool. Tools can also come out of date, so maintenance is needed.

The automation tool is not the only thing that needs maintenance. Test cases develop as a specification or implementation change, so the automated test case code needs to be changed accordingly. [2, 4]

4.2.1 SEARCH of automation testing

Another problem is the automation setup. Because the test cases are executed automagically by automation software, the product under test must be in a correct state before running the test case. It's important to remember that manual testers also deploy the application or prepare the database before running the test case. Test automation is even more strict. In most cases, the automation tool counts on whether the application is in some particular state—that it contains or does not contain some data. The preparation of these can take a lot of time, too.

If a step should result in failure, the manual tester can consciously decide that the test failed, but what about automation software? It can wait for some data to appear, and the data never appears; what would happen next? That is why it is essential to add timeouts to ensure the tests will finish and run not too long.

The components of automation testing can be summed as *SEARCH*: setup, execution, analysis, reporting, cleanup, and help. Every step is essential. We talked about the importance of setup and execution. The analysis consists of determining the result of execution—pass or fail. Reporting collects and displays the results (typically for QA people). A cleanup of artifacts and database is needed to proceed to the next test case. The most neglected part—the help phase—is about the maintainability of the tests, that they can be used later on, and not just once, if there is some significant modification change in the application. [4]

4.2.2 Depending on the people

It's also essential not to rely on automation too much; manual testing is sometimes more appropriate because some defects can only be detected by a human being—a flashing screen, unexpected notifications (that do not make the execution result in failure), wrong font, etc.

Last but not least: "Microsoft test developers have coding skills equal to their development peers, but there is one big difference between test code and product code: Product code is tested." Even if the test case is designed correctly, that does not say anything about whether it was correctly automated. Because the code of automated test case, such as every code, is not safeguarded against human errors. For that, it needs to be carefully written and reviewed. [4]

4.3 Test automation of GUI interactions

GUI (graphical user interface) interactions are the most difficult testing that can be automated. This is because of the ever-changing look of the application GUI and the instability of the simulation of the clickings and manipulation with the GUI.

A capture/replay tool can help quickly write the test automation code. The manual tester turns on the tool and executes the test case normally, and after she finishes, she stops the tool. The tool captures the manual tester's interactions with the GUI and generates a code accordingly. The code can be executed and will perform the actions just as the manual tester did. "It is our experience and that of every automator that we have ever spoken to, that the capture/replay architecture is completely worthless as a long-term testing solution." The problem is that this solution is not stable and does not check everything it should. The locators—the identifiers the automation tool uses to identify buttons and elements on the screen—generated by the capture/replay tool are sometimes complex or prone to change. In addition, the capture/replay tool does not capture what the human checks only by looking at the screen because it does not see the interaction happening. [1, 11]

4.4 Conclusion

It's important to stress that on the whole project level, manual vs. automated is *not an either-or question*. Sometimes, it is not even an either-or question on the test case level. When a new test case is introduced, and it is a "simulating the user" test case, it can make sense to test it both ways—manually and by using the automation tool—for the first few releases because the user interactions with GUI can be difficult to automate due to instability and are time-consuming to write.

Test automation is not a replacement for manual testing. When used effectively, it can save time and money, but some tasks are better suited for human testers, and some cannot be performed by the automation tool because the tool is not thinking. "A

manual tester—at least one that knows what they are doing—adds important elements to the abstract list of steps we call a test procedure. We can narrow it down to two important characteristics added by the human tester to every line of any test: context and reasonableness." [1] In the future, there may be tools using artificial intelligence that will come closer to thinking human beings, but as for now, there is no such option.

Chapter 5

End-to-End Testing

> *"End-to-end testing (E2E testing) is a testing method that evaluates the entire application flow, from start to finish. It ensures that all components work as expected and the software application functions correctly in real-world scenarios. In E2E testing, the software is tested from the end user's perspective, simulating a real user scenario, including the user interface, backend services, databases, and network communication. The purpose of E2E testing is to validate the application's overall behavior, including its functionality, reliability, performance, and security." [12]*

E2E testing encompasses the entire application, verifying and validating that the application works correctly as a whole. This is done through UI (user interface) and should be done in a production-like environment to simulate the real user scenarios as closely as possible. It tests the application from the user's point of view and doesn't care about what's underneath. Its purpose is to check if the user interaction is correctly received, interpreted, and displayed, e.g., clicking on the button "Show picture" results in rendering a picture on a screen.

The E2E test cases are user-oriented. One of the first steps is to get into the user's mindset, and the other is to talk to a real user or a potential future user. The next step is to understand the system—how it works and what's possible to do in it—by using it and studying the specification and user documentation. An exploratory test technique can be very helpful in this endeavour. [12]

The test cases are complex, and executing them takes a lot of time. Therefore, E2E testing should be only used to test common and critical scenarios. E2E testing is not a silver bullet. And there is no place to test boundary values or exceptional behavior in this kind of testing, which should be covered by unit and integration tests mentioned in section 5.1.

5.1 Test automation pyramid

Test Automation Pyramid is a model popularized in the book *Succeeding with Agile* by Mike Cohn. The author describes three kinds of tests based on their scope and granularity: unit, service, and UI. In some articles, they are called unit, integration, and E2E, but the idea behind it stays the same. *Unit tests* check a small piece of code—one unit—be it a function, method in a class, or a component (in CTFL v4.0 Syllabus [2], they are called component tests). Because of the small scope, many are needed to achieve reasonable coverage. Luckily, they are fast; one should run in milliseconds or even faster. *E2E tests*, on the other hand, are more extensive in scope, running slower (in seconds), and complex and costly to make. However, as they cover larger pieces of functionality, they can catch defects arising from the combination of components or running the app on different devices or browsers. *Integration tests* are only a little bit slower than the unit tests, still usually running in milliseconds, looking for defects in the integration of the components—how well the components work together. [13]

"The test pyramid is a way of thinking about how different kinds of automated tests should be used to create a balanced portfolio." [11] Looking at Martin Fowler's representation of the test pyramid 5.1, we can see that the test automation should build on the many unit tests because they catch the defects as close as possible to the root cause. They also run fast and are relatively cheap to make, so the return on investment is high as they give invaluable feedback early. The next floor of the pyramid is the service tests. Not so many are needed; they should not cover every edge case as the unit tests. These tests focus on the integration of the component and are faster and easier to make than E2E tests as they avoid the complexities of dealing with the application's UI.

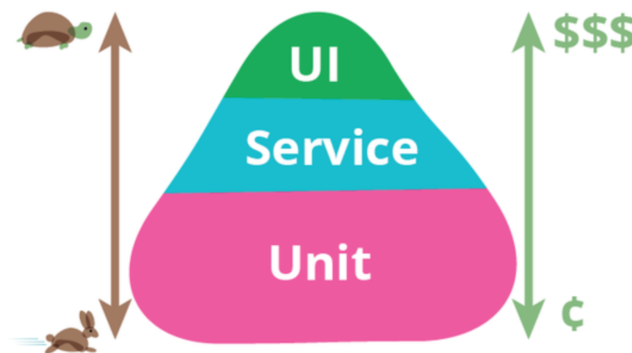


Figure 5.1 Martin Fowler's test pyramid [11]

The following figure 5.2 shows what the different layers of the pyramid cover. Unit tests are focused only on the code. The logic of extracting data from a database or communicating with other components is mocked or stubbed to allow tests to run faster and eliminate dependency on other functions that are not under test and can cause failure. The integration tests evaluate how components work together, for example, saving and extracting data from a database or communicating with an API (application programming interface). E2E tests add interaction with the UI to the stack, mimicking the user interactions with it.

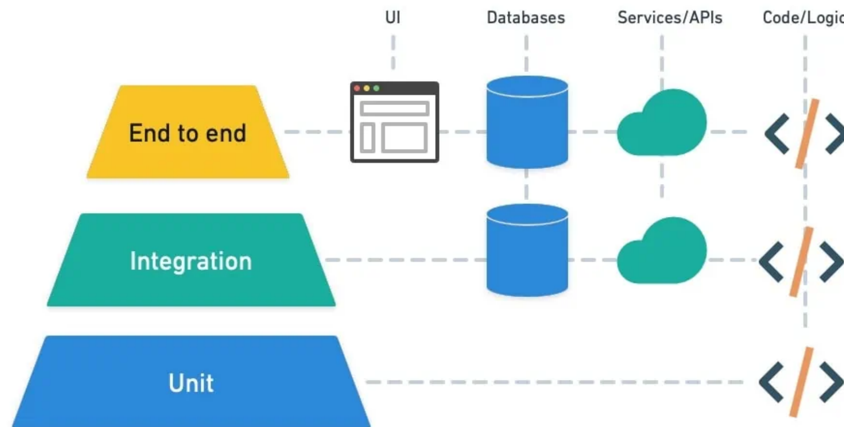


Figure 5.2 Scope of different tests in pyramid from semaphore web [13]

The “balanced portfolio” is needed to ensure that the suite containing all these automated tests is fast and reliable so that it can be run every release/daily build/push (depending on the criticality of the product) and produce quick and valuable feedback to catch the failures early in the development process.

With the constant technological change and development of new technologies, Kent C. Dodds proposes a new alternative model focused on testing JS (JavaScript) applications, naming it *The Testing Trophy* 5.3. “The Testing Trophy reorders priorities. Integration tests are king as most modern UIs rely on backend components and are difficult to test in isolation.” [13] Some part of unit testing is replaced by static testing—using parsers and linters—to catch basic errors. E2E tests still take the same small part in the overall number of tests.

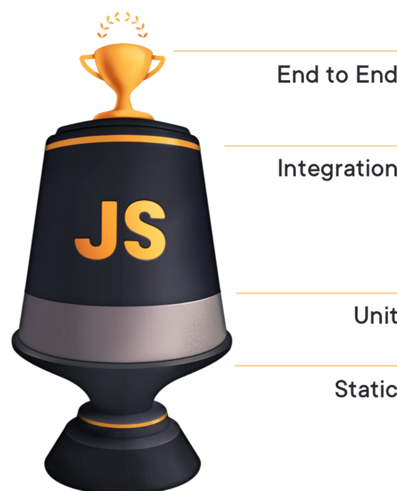


Figure 5.3 Kent Dodds’ JS test trophy [14]

5.2 Benefits of E2E testing

The E2E testing increases confidence in the application by ensuring that all components work correctly together and that the application meets its business and user requirements. It complements the other test kinds by offering the user's perspective, leading to a more reliable and defect-free application. It allows the testers to create more complex scenarios, test a broad scope, and ensure the components are integrated correctly and the data flow is correct. In addition, it ensures that the application is running and usable, testing it in a production-like environment, as running it in actual production is not a good idea. Failing tests show how the failure can affect the user. These tests can be helpful to test cross-browser and cross-device compatibility, even more so if they are automated, and permissions—one role should see something, the other should not. [12]

5.3 Pitfalls of E2E testing

E2E depends on the entire application being runnable and having UI; therefore, these tests can be developed only later on in SDLC. The design of an E2E test case is challenging as the tester first needs to understand the workflow and the user's goals; only then can she identify the critical scenarios and use cases. The execution is slow and time-consuming, simulating real user interactions, such as clicking a button, waiting for a render, etc.

5.4 E2E tests automation

E2E test cases can be automated. Many frameworks, such as Selenium, Cypress, and Playwright, can help automate GUI interactions, emulating a real user—clicking on a button, navigating on a screen, and inputting text. The pitfalls of the use of capture/replayed tools have already been discussed in section 4.3. Although, the benefits of test automation 4.1 are still there. It makes sense to automate the common scenarios that will be otherwise repeated many times manually, considering the return on investment. If it's too hard or impossible to automate, it doesn't make sense to do it. It still all starts with the manual execution of the test cases. When designing these complex test cases, the tester must be careful to understand the user workflow properly. It makes sense to execute the test cases (also) manually for the first few times to ensure that the test cases are designed soundly and don't overlook something important.

Automating E2E test cases poses challenges mentioned in the previous chapter 4, among many more. It's essential to identify possible screens, web elements—input fields, buttons—and additionaly the notifications and toast messages that could disrupt the automated test case execution. Still, it did not cause a problem for a manual tester because she is a thinking human being. They tend to be flaky. They are difficult to maintain. They depend on GUI; they use locators to identify the web elements. Although XPath (XML Path Language) is probably the most well-known locator strategy, it is a bad idea to use it in E2E tests—the page layout is the thing that changes perhaps the

most. Some frameworks allow text selectors or even the creation of a custom selector. The soundest and, in practice, the most used technique is to have the FE developers team add a unique test ID selector to each critical element. That helps a little with the stability and maintainability. [15]

Chapter 6
Atlantis

This chapter is about Atlantis that is neither a myth nor an island. Nonetheless, it is about exploring this application full of wonders, about preparation and setup of the hunt for potentially poisonous bugs that can be dangerous for the whole ecosystem. Using Playwright as a tool to set up in place a net, woven carefully using various test techniques and experience, will hopefully stop the most dangerous bugs. Unfortunately, the net will always have holes through which some bugs can slip, but the chance at least will be minimalized as it's always with proper testing.

6.1 About

Atlantis is one of the commercial applications developed by Jagu s.r.o. It is a warehouse management system aiming to help small and medium-sized warehouses organize their stock, manage orders, track reservations, deliveries, the actual inventory, and the packing process.

It is designed for four roles, each having a slightly different objective:

- *Warehouse worker* stocks the delivered goods and does inventories.
- *Warehouse packer* packs orders ready for shipment.
- *Warehouse organizer* manages stock and its location in warehouses. She can create tasks regarding stock transfer.
- *Warehouse manager* manages warehouses, products, users, and orders. She divides tasks and checks their fulfillment. In addition to that, she has the same rights and can do everything someone in the warehouse organizer role can do.

The roles use Atlantis, a web application, on devices with various screen sizes:

- Warehouse manager and warehouse organizer typically uses the application on a *computer with a big screen.*
- Warehouse packer uses a *tablet.*
- Warehouse worker goes around the warehouse only with a *Zebra device*—a mobile-sized device capable of scanning barcodes.

6.2 How it all started

In the summer semester of 2022, a bold team of five people signed up to test the FE of Atlantis. And I was one of them. The others are Jakub Vondráček, Jan Hlaváč, Jan Jeníček, and Ivo Kořínek. Next semester, Jiří Heller joined the fold. It was a project marketed under subjects Software Project 1 (SP1) and Software Project 2 (SP2), with supervisors Ing. Jiří Hunka, CEO of Jagu s.r.o., and Ing. Oldřich Malec, Project Manager of Jagu s.r.o. [1]

The testing should have been focused on the manual testing of GUI, with the possibility of testing automation. That mainly meant concentrating on UX and functional E2E tests. As it was known that I would like to focus on this topic in my bachelor thesis, I was put as the leader of the testing part of the project. Although the project was focused on testing, the SP1 and SP2 subjects' demands must have been satisfied, too, slowing the whole testing process down.

The first step was to familiarize ourselves with the web application and gain warehouse domain knowledge. There was no user manual at that time, so the only possibility, other than a brief explanation from the two supervisors, was going through the application to figure out what it does—create a warehouse, assign and complete tasks, add a barcode to a product, etc.

Manual testing was a tedious process, but it helped exploit many bugs that were not on the usual and most common paths through Atlantis and pinpoint places where visual enhancements were needed.

6.2.1 Examples of found bugs

One of the problems I encountered was that there was no format validation for codes and barcodes, so it was possible to input almost anything into that field. That is both a security issue and, from the user's point of view, can be an unanticipated behavior—mistype of one character is not pointed out, so the user can submit a non-existent barcode.

In the configuration list for the home screen, one of the filters remained with the i18n (internationalization) text because no translation was defined for it.

[1] For the convinience of the reader the student changed this section to be written in first person.

Jakub and I also found a bug: if an item-picking task without a specified target location was assigned to the warehouse worker and she started working on it, she could specify a non-existent barcode in the target location field, and the web application let her do it. That leaves the task with an unspecified location and, therefore, in an invalid state.

All these issues are fixed in the February 2024 version of Atlantis.

Functional and security unit tests should have found the first one. The second one is a type that is easier to find during exploratory manual testing. The third one is a functional test suitable both for the unit—the field component check—and for the E2E test—the defined target location writes itself to the task assignment.

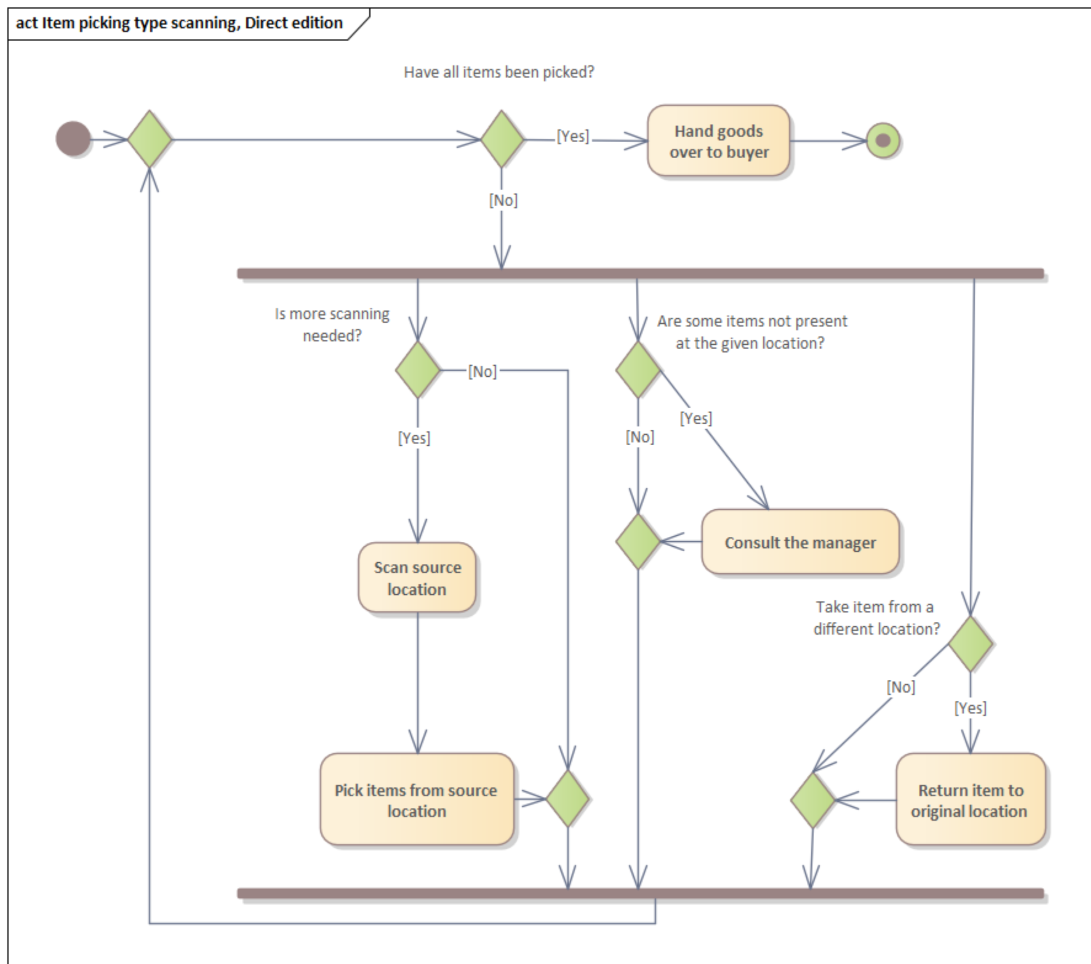
6.2.2 User manual and diagrams

As part of the SP1 and SP2, we were tasked to write the missing user manual. During the extensive manual testing and consulting of different possibilities of the applications and flows through the application with our supervisors, we became domain experts and experts on how the application works. It also helped that one of the course requirements was to create use case diagrams, which forced us to relate to the user and understand their motivations, as these diagrams illustrate different user interactions with the application. Or that we needed to create activity diagrams, "used to illustrate the flow of control in a system and refer to the steps involved in the execution of a use case" [16], and that helped us understand many possible user flows in the app. We, of course, were consulting not only the creation of these diagrams but also the creation of the user manual with our supervisors every week, and they consulted it with the warehouse managers and workers to see if it made sense and corresponded with how they use the application in a real warehouse environment.

The process of scanning during the tasks can be complicated, as the order of scanning barcodes by itself can define which item is put where, and it is slightly different for each task. I was put in charge of modeling that. An example of the scanning control flow, one of the easiest, can be seen in figure 6.1. Activity diagrams are sort of CFGs but on a different level, a much higher level of abstraction, but they can be useful for creating test cases. Particularly for E2E test cases, as they depict various valid and invalid paths through the software to achieve something. They can also help show the users what they can do, so we put them into the user manual.

Figure 6.1 depicts the control flow of the scanning part of an item-picking task that results in the personal collection of the goods. It all starts and ends with the question, "Have all items been picked?". Then, there are three parallel flows that can be executed in any order. One is about picking items from the source location, the second is about reporting problems (missing items), and the third is about the possibility of returning previously picked items if the warehouse worker comes upon a better candidate.

The other activity diagrams of different types of scanning are much more complicated, consisting of many flows and branching a lot, which makes them even more beneficial as a tool to use when designing test cases.



■ **Figure 6.1** Activity diagram of scanning while item picking for personal collection

6.3 E2E test case design

E2E test case design relies heavily on the tester's domain knowledge because of their nature: the simulation of real user interactions and use cases. It is also the inability to test all scenarios and the fact that edge cases and possible input values are thoroughly tested on unit and integration levels. The best approach to designing an E2E test case is either using experience-based techniques to cover key and critical scenarios or seeing the software in use, seeing the real users interact with it in a natural environment—that way, the most traveled paths in software can be noted and are the best candidates for E2E smoke testing.

During SP1 and SP2, a visit to a warehouse where they use Atlantis was arranged, and a warehouse manager and worker were observed. Combining it with manual testing and the following exploratory testing performed by the student, led to noting more than 180 ideas about what to test that lay on the critical paths.

Atlantis is a web application consisting of more than 80 different pages, each containing many UI components with which to interact. During the exploratory testing, each page was visited at least once, but that does not mean all different interactions and combinations of interactions were carried out. E2E testing is the last layer of tests; finding bugs at this level is tedious. The exploratory testing was performed with the user manual and activity diagrams at hand to confirm the app's behavior against the intended functionality. Other than the created test cases, the exploratory testing led to the reporting of more than 40 failures in just April and May of 2024. The failures ranged from missing translation to incorrectly set user rights and non-deterministic behavior where there should have been one.

Key scenarios were identified as ones about creating and completing tasks, the core being the manipulation with an external order. On the other hand, such use cases as creating the warehouse or carrier are not usually carried out daily, so these were decided unsuitable for the E2E test level. Some use cases, such as filtering tables, are better suited for UI component testing and not to be implemented as E2E test cases.

Most of the 180 ideas, the most important ones, were compiled into 52 E2E test cases. Some test cases are only intended for manual execution because they cannot be done with the testing framework. These cover mainly checking the printing and scanning functionality and need a Zebra mobile device connected, such as printing the labels for packages as the last part of the packing process. The others are suitable for test automation as they are critical E2E test cases and can be implemented using Playwright. All test cases are written as checklists to be carried out by manual testers and to serve as a template for the automated test cases that will be almost step by step the same as these, only including more checks that are required by the framework—waiting for some page or component to appear. Each test case consists of the preconditions for the case and the steps to be executed, including actions and verification checks. Here is a part of an actual test case performed by a warehouse worker, the numbers in the brackets are the ideas this test case covers:

Complete item picking for local collection (01.03, 01.04, 01.24, 01.27, 02.28)

- ☐ precondition: #53211 (work in progress item picking for local collection with one item to be picked)
- ☐ go to the work in progress list
- ☐ select a task by clicking on it
- ☐ *input + last scanned location barcode*
  - ☐ fill in the location code in the scan box
  - ☐ the location is highlighted
  - ☐ the location is written below the scan field as last scanned

- ☐ *input + last scanned item barcode*
  - ☐ fill in the item barcode in the scan box
  - ☐ last scanned item is 1x item
  - ☐ item has moved out of to be stocked column
  - ☐ to direct handover to customer column
  - ☐ highlighted is direct handover to customer

6.4 Database

An important part of setup and teardown is how to manage test data. Atlantis uses a Postgres database, which leads to the following questions: What should be the initial state of the database? How should it be achieved? And to which state will the database be reverted after the suite's execution?

6.4.1 Init state and getting data into database

The database could start *empty*, and the data generated during the test case execution. It is "the natural way of getting data into the database—using the user interface of the application." It does not violate direct access to the database, no fixture[2] maintenance is needed, and the fixtures[2] "are always correct since they use the application's natural entry points for entering the data in the first place." However, it would lead to a lot of execution time being consumed on generating data without much time being focused on the key and critical scenarios. So, in reality, this approach is, in most cases, not applicable. [17]

Another option is to populate the database with data also by using the application's user interface, but do it before the execution, save it as a snapshot, and then load this snapshot each time before testing (either before the test suite or each test case). This approach keeps the advantages of the previous option, but the execution will be much faster. However, it can take a lot of work to prepare the database to the point that it will contain data to cover each test case.

The next approach is manually writing fixtures[2]. It takes time to write them, and they are not as natural as the data created by using the application. However, the set of fixtures is usually small, which means quicker load times than when importing a database. [17]

Atlantis has a deployed testing environment where every developer can manually test the new features and check the application is working as intended. The student writing this thesis also used this testing environment during her manual testing. That means there are a lot of production-like data without actually being the production data that will fall under GDPR (General Data Protection Regulation). The dump of this testing

[2]The term *fixtures* is used either for sample data or for a mechanism to load a set of data into a database.

database can serve as a perfect basis for the final database dump. Additional data are also created through GUI to provide as much natural data as possible. The final database dump is imported only before the suite execution to allow parallel execution of the test cases without worrying about a corrupt database state. It is also much faster to import it only once than before every test case. The thing to be careful about is that the database has to allow running test cases in any order, so it should cover enough data so the test cases can work with different tasks and not affect each other.

6.4.2 The importance of cleaning up

The teardown part of the test environment is as essential as the setup part. “Let’s say you’re in the kitchen and you want to do some cooking. You’re making tomato soup in a pot, you cook the soup, you put it in a bowl, and you eat the soup. But now you want some ice cream to top off the meal, but you’ve only got one bowl. You probably wouldn’t just put the ice cream in the bowl that just had soup in it, first you’d clean the bowl. You would want to completely clean out your bowl of soup before adding any ice cream. The same principle applies to automated testing.” [18]

The problem is that the leftover data can overwhelm the testing environment, break other tests, or impact manual testers that will use the same testing environment thinking it’s in the same state they left it—that is mostly a problem on deployed testing environments shared for manual and automation testing. “There should be no evidence that we were in the system at all once our tests are finished executing.” [18]

The teardown can be done on a test case level, test suite level, or even all test suites execution level—but mostly it’s set up on both test case level and test suite level, allowing for things like deleting only part of the database on the test case level that can cause problems to other tests, but leaving the cleanup of the whole database to the teardown of the suite.

The Atlantis test environment for automation has the database in a Docker container, and the initial state of this database is empty. So, after the suite’s execution, it should be reverted to empty. What about teardown after each test case? Because of the setup decision (the database loaded only once before the suite execution), the teardown cannot include the cleanup of the entire database. Deleting part of the database (the data created by the test case) could lead to a corrupt database for one of the test cases because of the parallel execution. The only teardown is, therefore, the one after the suite execution.

6.5 Playwright

“Playwright enables reliable end-to-end testing for modern web apps.” [19] At least, that is what their website says. It is a relatively new tool developed by Microsoft and released in 2020. In 2022, it was already recommended in Vue.js documentation. That was relevant because the FE part of Atlantis “utilizes a Javascript framework Vue.js and a graphical library Vuetify” [20]. So, Playwright and JS were chosen, as they appeared

to be most suitable for the needs and corresponded with the already existing technology stack.

6.5.1 About

This tool supports multiple rendering engines—like Chromium and Firefox—and various operating systems. The monitor's screen size can be set, and the usage of mobile devices is simulated using native emulation of Google Chrome. The scenarios can span multiple tabs and multiple users, enabling saving the authentication state to bypass tedious and time-consuming user login for each test case.

Like many other E2E tools, Playwright can auto-wait: it waits for elements to become visible or interactive prior to performing a click or any other action. These actions and checks of a state are limited with a timeout that can be either globally set or set for the particular action.

It can capture screenshots and videos of what is happening during the execution. The screenshot taken when a failure occurs can be highly valuable to speed up the process of finding the root cause; the picture can say a lot if taken at the right moment; sometimes, it is everything the tester needs for assessing the current situation. Sometimes, it's clear from the screenshot that the test case is just flaky and needs to be retried once more; the retry strategy is also configurable in Playwright. And for CI (Continuous integration) jobs, it's very useful that Playwright can record a trace of the retried test, allowing therefore even better insight into what was happening. [19]

Playwright can also be used to test the application's REST API. That does not fall under E2E testing as it bypasses the UI. Still, it can be helpful in combination with some UI interaction, for example, "validating server post-conditions after running some actions in the browser." [21]

6.5.2 Codegen

Section 4.3 supposes that the capture/replay tool is worthless. The Playwright's capture/replay shows that a lot can change in 13 years and that the technologies can evolve and become helpful and sophisticated.

Playwright has a test generator in the form of a capture/replay tool with something extra. It records the user interactions and generates the code directly into Visual Studio Code or Playwright Inspector—depending on how the test generator was started. It uses many different locator strategies and tries to choose the right one for the element. It also enables the generation of assertions right from the tool. The assertion that the element is visible, that it contains specific text, or has a specific value. There is also the possibility to "record at cursor," which means start recording from a specific point in the test code. The locators can be picked using this tool and then placed in code.

There is also the possibility to "use the test generator to generate tests using emulation so as to generate a test for a specific viewport, device, color scheme, as well as emulate the geolocation, language or timezone". [22]

After the generation, it is still necessary to go through the code if it makes sense and the locators seem sensible. It is impossible to use it to check everything, so manual code improvements are needed, but compared to what the book talked about, this tool is certainly not worthless.

6.5.3 Locator strategies

"Locators are the central piece of Playwright's auto-waiting and retry-ability. In a nutshell, locators represent a way to find element(s) on the page at any moment." Playwright focuses on the user-facing locators: getting an element by its role—button, checkbox, alert—label, placeholder, or even text. It also supports the classic locators, e.g., test id, CSS (Cascading Style Sheets) selector, and XPath (XML Path Language) locator, but these are recommended to be used only when necessary. [15]

6.5.4 Page Object Models

"Large test suites can be structured to optimize ease of authoring and maintenance. Page object models (POM) are one such approach to structure your test suite... Page objects simplify authoring by creating a higher-level API which suits your application and simplify maintenance by capturing element selectors in one place and create reusable code to avoid repetition." [23]

6.5.5 Fixtures

"Test fixtures are used to establish the environment for each test, giving the test everything it needs and nothing else. Test fixtures are isolated between tests." One of the predefined fixtures is `page`, providing an isolated page for the test case. A fixture can encapsulate setup and teardown for the page and is reusable between the test cases and on-demand, meaning only the needed ones for the test case are set up. They are easily combined with the POMs, making the test cases cleaner without unnecessary page setups and locator redefinitions. [24]

6.6 Setup

Automated test cases are run using Playwright and Node.js. *Prerequisities* are that FE and BE are set up using the instructions on Jagu Gitlab.

The setup for running E2E tests locally consists of:

1. Changing the `db_scripts.config` to match the project structure

2. Installing Playwright browser dependencies with command:

```
npx playwright install --with-deps chromium
```

3. Starting BE docker environment using command:

```
docker-compose up -d
```

4. Starting FE test environment using command:

```
pnpm serve:test
```

Test configuration can be defined in the `playwright.config.js` file. It can specify the directory where to look for tests, timeout for a test case run, and timeout for assertions—how long it should wait for some condition to be met. [25]

The first keyword in the fragment of `playwright.config.js` file (code listing 6.1) sets retries to 2 for the CI job and 0 retries for the local execution. It is because if the test case fails on the local machine, there is always the person who executed it and, therefore, can make the decision to run the specific test once more or perform the scenario manually instead.

```
retries: process.env.CI ? 2 : 0,
fullyParallel: true,
workers: process.env.CI ? 1 : undefined,
use: { browserName: 'chromium' },
```

■ **Code listing 6.1** Fragment of playwright.config.js

The `fullyParallel` keyword specifies whether the tests in one test file should run in parallel or not. For Atlantis E2E testing, each file just assembles test cases focused on the same topic, so the parallel execution of test cases in one file is a desired behavior, as they do not depend on each other. If it's set as false, the tests in one file run in the order of declaration. Parallel execution means the total run time is reduced, and it also follows a good practice to have all tests as independent as possible. If it's impossible and some test cases depend on each other, Playwright allows annotating tests for serial execution:.

```
test.describe.configure({ mode: 'serial' });
```

`workers` keyword limits the maximum number of parallel worker processes. Limiting the workers on CI to 1 turns off parallel execution, but it should lead to a little better stability, ensuring each test gets full system resources, with a trade-off of slower execution. The `undefined` means that for local execution, Playwright uses the default formula that sets the maximum number of workers to 50 % of the machine's available cores.

The suite is run using Chromium (code listing 6.1), therefore it is tested against Chromium-based browsers such as Google Chrome and Microsoft Edge. That gives a

headstart if the application starts breaking to fix it because the Chromium releases are ahead of Google Chrome releases.

"A project is logical group of tests running with the same configuration." It is useful if it's desired to run the tests on different browsers or different devices. [26]

```
projects: [
  {
    name: 'manager',
    testMatch: '*.@(manager|organizer).spec.js',
    use: { ...manager }
  },
  {
    name: 'worker',
    testMatch: '*.worker.spec.js',
    use: { ...worker }
  },
  {
    name: 'organizer',
    testMatch: '*.organizer.spec.js',
    use: { ...organizer }
  },
  {
    name: 'packer',
    testMatch: '*.packer.spec.js',
    use: { ...packer }
  },
  {
    name: 'settings',
    testMatch: '*/settings.spec.js',
    fullyParallel: false,
    use: { ...serial }
  },
],
```

■ **Code listing 6.2** Define projects

This presents a perfect opportunity to divide the suite into projects (code listing 6.2) based on different roles and their devices. The 'worker' project uses `worker` login and device configuration for all tests in files that match the expression `'*.worker.spec.js'`. There is a special project named 'settings' that groups tests focused on checking settings features, such as language change or time tracking toggle on and off. These tests do not depend on each other but affect each other and the other tests—because changing the user settings changes the settings for the same user across all the browsers—e.g., switching the language to Slovak causes the text locators in Czech not to be found.

The devices for the specific roles can be defined in JSON (JavaScript Object Notation) files and then used in the configuration. For example, the Zebra device for the

warehouse worker can be set like in code listing 6.3, based on Galaxy S5 Playwright definition.

```
{
    "viewport": {
      "width": 360,
      "height": 640
    },
    "deviceScaleFactor": 2,
    "isMobile": true,
    "hasTouch": true
}
```

■ **Code listing 6.3** Defining Zebra device

6.7 Implementation

The core part of converting manual test cases to automated test cases is to locate the elements with which the interactions are. The codegen tool can help with that: it shows the locators of the elements when hovering over them, which is true for all the elements on the highest layer. Moving the cursor around is, therefore, one of the strategies that can be used to locate the elements. It can be tricky as, for example, a button can be partially covered with some div element, and this codegen tool will show the locator to this div, not the button. That means the locators need to be carefully chosen. Sometimes, more than the codegen tool is necessary. Look directly into HTML, derive a locator from that, and trial and error if it works. The second strategy is essential for locating elements that are not interactable and are deep into the HTML page structure; they usually don't have roles.

The locators and interactions with the pages were then coded using the POM pattern, and the POMs were employed as fixtures in each test case. Manual test cases can be converted step by step, but the automated equivalent of some steps needs to be taken more strictly or less. Imagine such a simple step as "open item-picking task". The manual tester clicks on the task and then implicitly waits for the page to load. The automation tool clicks on the task and waits for the page to load. Where is the difference? Playwright waits for the page load event to be fired, which is a problem with dynamic web applications like Atlantis. The load event is fired at the beginning, so not all components are loaded. So if the load of the whole page is not fast enough, the next expectation that some element is visible can fail on the timeout. These expectation checks are, therefore, carefully ordered to lower the chance that this happens; first are the ones of the first loaded components on the screen. The solution, of course, will be to (somehow) wait for the whole page to load, but that's a) almost impossible to do, b) slows the whole execution, and c) does not simulate the real user: she will not wait for all components to load if she doesn't need it. Another solution can be to expand the timeouts, but it will mean that if some test fails, like really fails, it would make it

fail slower, slowing the whole execution down. The pages are loading smoothly under normal circumstances, and the only reason why it occasionally takes longer time for the page to load is overloaded BE. The correct approach to the loading problem is limiting the number of test cases that run in parallel.

Just as for the manual test cases, the preconditions must be satisfied, but it's necessary to be even more specific in automation. In Atlantis, every test case has defined specific tasks this test case touches. Because the database is set up only once, it's essential that the automated test cases don't interact with each other and don't change some preconditions that other test cases build on, e.g., changing a sub-warehouse setting about merging tasks or moving an item from a location that other test case assumes it there.

The following example 6.4 shows part of a test case performed by a warehouse packer. There is a title of that test case *complete packing - allow selection for each*, and the following line shows which fixtures are used. The next lines specify the tasks this test case interacts with, and the rest shows the automation tool's steps. The `test.step` syntax is used to encapsulate part of code that is logically together and show it as a step in the report; don't worry, the report steps can be expanded.

```
test('complete packing - allow selection for each',
  async ({ common, packing, assignment }) => {
    const taskPacking = '53268';
    const taskOrder1 = '53265';
    const taskOrder2 = '53277';
    await common.goToTaskAndWaitForTitle(taskPacking, taskType.packing);
    await common.startWorkingOnTaskButton.click();
    await expect(packing
      .itemInToBePacked(`300 Kus ${item.preclik.name}`)).toBeVisible();

    await test.step('scan and confirm product, first order',
      async () => {
        await common.scanBarcode(item.preclik.code);
        await expect(packing.areAllInstancesPackedDialog).toBeVisible();
        await packing.confirmPackedButton(`300 Kus`).click();
        await expect(
          packing.itemInToBePacked(`300 Kus ${item.preclik.name}`))
          .toBeHidden();
        await expect(
          packing.itemInPacked(`300/300 ${item.preclik.name}`))
          .toBeVisible();
        await expect(assignment.notYetShipmentCheckAlert).toBeVisible();
        await assignment
          .expectTextToBeVisible(assignment.order, `#${taskOrder1}`);
    });
```

■ **Code listing 6.4** Part of an automated test case

6.8 Execution

Commands for database setup and database teardown are part of the execution command. Database setup and teardown were discussed in section 6.4. BE and FE of Atlantis don't need to be restarted for repeated execution.

To execute a full suite of E2E tests, run command:

```
pnpm test
```

To run a specific project, test file, or test, run the corresponding command from the following commands:

```
pnpm run test:debug --project=<project_name>
pnpm run test:debug <file_name>.spec.js
pnpm run test:debug -g <test_titles_to_grep>
```

The list of existing projects is defined in `playwright.config.js` file (code listing 6.2). `test:debug` command contains only database setup as it is enough for debugging purposes and it saves time—the presumption is these commands are run repeatedly and in quick succession.

These four execution commands are for WSL (Windows Subsystem for Linux)/Linux. The author does not promise anything for Windows users but it should be possible (bash is required) to run the suite using `pnpm test:win` and the debug commands using `pnpm run test:win:debug`.

Tests, by default, run in headless mode, with the GUI hidden from the human tester. That is very useful when using automated software because it enables faster execution due to less resource use but, at the same time, behaves in the same way as with the GUI, navigating the page and clicking buttons by accessing the web page. However, sometimes, it can be helpful for a human tester to have the opportunity to see what steps the automation tool takes in the browser. If the screenshots of failures from the headless mode run are insufficient to figure out the problem, two Playwright flags come with a helping hand. The first one is `--headed`; it will open the browser at the start so that all interactions will be visible on the GUI.

```
pnpm run test:debug --headed
```

The other possible mode is UI mode, and it can be started with a `--ui` flag. That opens a UI mode window, where the tester can run the tests and see what steps happened and what actions the software took. It also enables the inspection of the DOM (Document Object Model) and picking locators on the different screens the test case ran by. It also has the Log, Console, and Network tabs. This mode is handy for debugging.

```
pnpm run test:debug --ui
```

After execution, Playwright produces a report. There are many possible formats of the report, to be viewed here: [27], but for its visual and easy navigation, the HTML (Hypertext Markup Language) report was chosen. The report can be accessed using the following command, which opens it in the default browser. It shows not only which test cases passed or failed (figure 6.2), but after clicking on the test, it shows the steps and time each step took. And, eventually, at which step and on which line the error happened, and screenshot at the moment of the failure 6.3.

```
pnpm test-report
```

key-scenarios/create.organizer.spec.js
× create stock transfer manager 31.1s
key-scenarios/create.organizer.spec.js:5
✓ create location transfer manager 12.2s
key-scenarios/create.organizer.spec.js:44
✓ create stock transfer organizer 10.4s
key-scenarios/create.organizer.spec.js:5
✓ create location transfer organizer 7.7s
key-scenarios/create.organizer.spec.js:44

key-scenarios/create.manager.spec.js
✓ create stocking manager 17.2s
key-scenarios/create.manager.spec.js:5
✓ create subwarehouse transfer - completed automatically manager 17.0s
key-scenarios/create.manager.spec.js:47

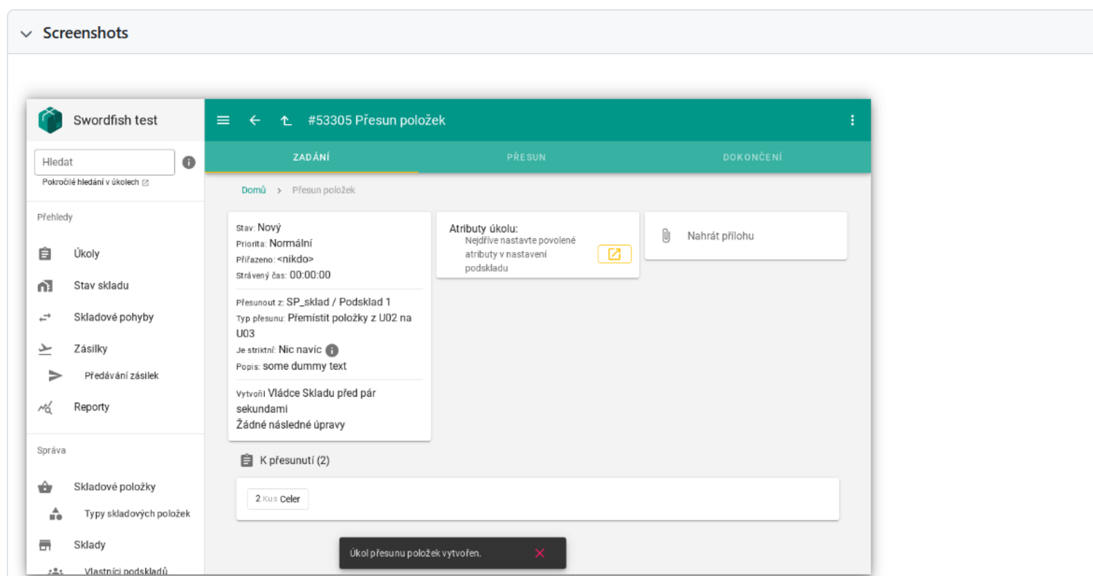
Figure 6.2 Part of a test report

On the student's notebook, which has eight logical processors and uses WSL for the execution, Playwright spawns four workers. The full suite consisting of 47 logical test cases runs in 4 minutes. The number 52 shown in the report is because test cases consisting of switching between different users have to be implemented like separate tests, and that they run serially and are in one described group doesn't make a difference to Playwright.

Using just one worker slows the execution to 7 minutes. This confirms that BE (or the device running it) can become overloaded, which causes pages to load slowly. If many tests fail, the right approach is to close all other applications or run it with fewer workers using command:

```
pnpm run test:debug --workers=1
```

Another problem that can cause many tests to fail is that the person executing it plays with the settings for the users on the locally deployed Atlantis, and she leaves the window with it open during the execution. Settings, such as language and time tracking, are saved in the database, but they are also loaded from the browser's local storage. This means that the changed settings in the open windows would override the settings in the database. The easiest solution is to avoid playing with the settings or at least close all browsers with Atlantis.



■ **Figure 6.3** Part of a test case in report

Conclusion

The main goal of this thesis was to introduce the reader to the topic of quality assurance and testing, explain why it is necessary, and show the complexity of delivering a high-quality product. It described various types of tests, such as static, functional, and user experience. The next part analyzed test design techniques, covering black-box, white-box, and experience-based techniques. Some of them backed with an example of an application on an imaginary web application that helped to point out their value but also potential blind spots.

The benefits and risks of test automation were discussed, and the conclusion that it's still necessary to perform manual testing as some defects are impossible or near impossible to detect using automated tests was reached. The E2E testing chapter introduced the reader to the test automation pyramid and showed how E2E testing fits in with the other kinds of tests. It also described the benefits and pitfalls of that testing and discussed the potential automation of E2E test cases, including all the problems that can arise during the implementation. The theoretical part was then concluded, and its goals were met.

The work then continued to a more practical part. Atlantis, a warehouse management system, was analyzed using manual testing. With the cooperation of colleagues from Jagu s.r.o. and warehouse workers, a user manual was created, including activity diagrams detailing, e.g., the possible paths of the scanning process. Thanks to the exploratory testing over than 40 failures were reported, including visual, functional, and security ones. Most of them are already assigned to specific colleagues from Jagu s.r.o. to fix. Also, many recommendations on the visual appearance or application logic were discussed with the supervisor, approved, and later assigned to the developers to be implemented.

The test automation environment was set up, including the database and Playwright testing framework. Commands to prepare the setup and run the E2E test suite were written for local execution on Linux/WSL machines. Another command was written to show the report of passed and failed tests in a browser instead of a terminal. The work also mentioned the Playwright's many possibilities, such as code generating, running tests in parallel, or simulating devices of various types with various screen sizes.

The actual test cases were designed primarily using the exploratory test technique combined with black-box test techniques and not-so-conventional approaches using an existing user manual and activity diagrams. Most of the designed test cases were suitable for test automation, but for example, testing the correct integration of the Zebra mobile device scanner with the web application is only possible with the actual device and someone scanning the code. On the other hand, automation was very suitable for checking if the toggle to show or not show product images in task works. The test cases were written in sentences in form of a checklists so the manual tester could easily understand and execute them and the ones suitable for automation were implemented using Playwright. There were 54 test cases designed, 47 of them were automated, and the created test suite runs circa 4 minutes using 4 workers or 7 minutes using 1 worker.

The practical part showed results of the potential applications of the theory explained in the first few chapters. The created E2E test suite will in future serve as a defense against regression defects. Therefore, every new Atlantis release that reaches the customers should work as intended, at least in key scenarios. All in all, the testing helped and will help make Atlantis better for users, which concluded the goal of this part.

The work includes only a part of the setup of a CI environment. It is set only on the Playwright's side of things, meaning how the execution of the test cases will go. That is the next sound extension of this thesis. Having it in a pipeline would improve the whole process, as developers would not need to set up the testing environment to run the E2E tests. It can be automated even more, e.g., to run after every push to the main branch to check if the application still works correctly and report the issues directly to some bug tracking manager.

Bibliography

1. BLACK, Rex; MITCHELL, Jamie L. *Advanced Software Testing: Guide to the ISTQB Advanced Certification as an Advanced Technical Test Analyst*. Vol. 3. 2nd. Ed. by BARABAS, Michael. United States of America: Rocky Nook, 2011. ISBN 1933952393.
2. INTERNATIONAL SOFTWARE TESTING QUALIFICATIONS BOARD. *Certified Tester Foundation Level Syllabus v4.0* [online]. 2023. [visited on 2024-03-01]. Available from: https://istqb-main-web-prod.s3.amazonaws.com/media/documents/ISTQB_CTFL_Syllabus-v4.0.pdf.
3. COLEMAN, Lance B. *The ASQ Certified Quality Auditor Handbook*. 5th. United States of America: ASQ Quality Press, 2020. ISBN 1951058097.
4. PAGE, Alan; JOHNSTON, Ken; ROLLISON, Bj. *How We Test Software at Microsoft*. United States of America: Microsoft Press, 2008. ISBN 0735624259.
5. AHMAD, Nazneen. *What is Compatibility Testing: Tutorial With Examples* [online]. 2024. [visited on 2024-04-03]. Available from: https://www.lambdatest.com/learning-hub/compatibility-testing.
6. HARTSON, Rex; PYLA, Pardha. *The UX Book: Agile UX Design for a Quality User Experience*. 2nd. United States of America: Morgan Kaufmann, 2018. ISBN 0128053429.
7. ISTQB® GLOSSARY WORKING GROUP. *ISTQB Glossary* [online]. 2024. Version 4.3 [visited on 2024-04-03]. Available from: https://glossary.istqb.org/en_US/search.
8. MEHTA, Umang. *Risk Scoring Systems: Understanding CVE, CVSS, CWE, CAPEC, and NVD* [online]. 2024. [visited on 2024-04-03]. Available from: https://www.linkedin.com/pulse/blog-29-risk-scoring-systems-understanding-cve-cvss-cwe-umang-mehta-1a4xf.
9. OWASP FOUNDATION, INC. *OWASP Top Ten* [online]. 2023. [visited on 2024-04-03]. Available from: https://owasp.org/www-project-top-ten/.

10. BIERIG, Ralf; BROWN, Stephen; GALVÁN, Edgar; TIMONEY, Joe. *Essentials of Software Testing: Guide to the ISTQB Advanced Certification as an Advanced Technical Test Analyst*. United Kingdom: Cambridge University Press, 2022. ISBN 1108833349.
11. FOWLER, Martin. *Test Pyramid* [online]. 2012. [visited on 2024-03-16]. Available from: https://martinfowler.com/bliki/TestPyramid.html.
12. BOSE, Shreya. *What is End To End Testing* [online]. 2023. [visited on 2024-03-16]. Available from: https://www.browserstack.com/guide/end-to-end-testing.
13. FERNANDEZ, Tomas; ACKERSON, Dan. *The Testing Pyramid: How to Structure Your Test Suite* [online]. 2022. [visited on 2024-03-16]. Available from: https://semaphoreci.com/blog/testing-pyramid.
14. DODDS, Kent C. *Testing JavaScript: Learn the smart, efficient way to test any JavaScript application.* [online]. 2024. [visited on 2024-03-16]. Available from: https://www.testingjavascript.com/.
15. MICROSOFT. *Locators* [online]. 2024. [visited on 2024-04-03]. Available from: https://playwright.dev/docs/locators.
16. GEEKSFORGEEKS. *Activity Diagrams* [online]. 2024. [visited on 2024-04-08]. Available from: https://www.geeksforgeeks.org/unified-modeling-language-uml-activity-diagrams/.
17. NOBACK, Matthias. *About fixtures* [online]. 2018. [visited on 2024-04-12]. Available from: https://matthiasnoback.nl/2018/07/about-fixtures/.
18. DIMAYACYAC, Keilah; WAGNER, Chris. *The Teardown and What I Learned About Test Environments From a Bowl of Soup* [online]. 2023. [visited on 2024-04-12]. Available from: https://platotech.com/2021/03/04/the-teardown-and-what-i-learned-about-test-environments-from-a-bowl-of-soup/.
19. MICROSOFT. *Playwright: Fast and reliable end-to-end testing for modern web apps* [online]. 2024. [visited on 2024-03-01]. Available from: https://playwright.dev/.
20. MALEC, Oldřich. *Frontend skladového systému*. 2020. Master's thesis. Faculty of Information Technology, Czech Technical University in Prague. Available from: https://dspace.cvut.cz/handle/10467/86593 or https://gitlab.fit.cvut.cz/malecold/master-thesis.
21. MICROSOFT. *API Testing* [online]. 2024. [visited on 2024-04-08]. Available from: https://playwright.dev/docs/api-testing.
22. MICROSOFT. *Test Generator* [online]. 2024. [visited on 2024-04-08]. Available from: https://playwright.dev/docs/codegen.
23. MICROSOFT. *Page object models* [online]. 2024. [visited on 2024-04-08]. Available from: https://playwright.dev/docs/pom.
24. MICROSOFT. *Fixtures* [online]. 2024. [visited on 2024-04-08]. Available from: https://playwright.dev/docs/test-fixtures.
25. MICROSOFT. *Test Configuration* [online]. 2024. [visited on 2024-04-08]. Available from: https://playwright.dev/docs/test-configuration.

26. MICROSOFT. *Projects* [online]. 2024. [visited on 2024-04-08]. Available from: `https://playwright.dev/docs/test-projects`.
27. MICROSOFT. *Reporters* [online]. 2024. [visited on 2024-04-08]. Available from: `https://playwright.dev/docs/test-reporters`.
28. KOVÁŘ, Pavel. *Backend skladového systému*. 2019. Master's thesis. Faculty of Information Technology, Czech Technical University in Prague. Available from: `https://dspace.cvut.cz/handle/10467/82591`.
29. GRAMMARLY INC. *Grammarly* [online software]. 2024. [visited on 2024-05-15]. Available from: `https://www.grammarly.com`. Used in the EDU version with suggestions but without Generative AI prompting.

Attachments

- `text.pdf` .... text of thesis in PDF format
- `link to GitLab` .... implementation source code
- `link to Notion` .... E2E testing materials
  - `Notes from exploratory testing`
  - `Ideas on what to test`
  - `Key test cases`