Technische Universität Berlin

Usability Evaluation of Acceptance Testing Frameworks

Bachelorarbeit
am Fachgebiet Software Engineering of Embedded Systems (SEES)
Prof. Dr. Sabine Glesner
Fakultät IV Elektrotechnik und Informatik
Technische Universität Berlin

vorgelegt von
Anton Bardishevs

Gutachter:	Prof. Dr. Sabine Glesner
	Prof. Dr.-Ing. Sebastian Möller
Betreuer:	Stefan Sydow, M.Sc.

Anton Bardishevs
Matrikelnummer: REDACTED
CONTACT INFORMATION REMOVED

Contents

Erklärung der Urheberschaft

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form in keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Riga, 10. August 2020 Anton Bardisevs (elektronisch)

Abbreviations

BDD	Behaviour-Driver Development
CLI	Command Line Interface
IEEE	Institute of Electrical and Electronics Engineers
OOPSLA	Object-Oriented Programming, Systems, Languages & Applications conference
UI	User Interface
TDD	Test-Driver Development
QA	Quality Assurance
HQ	Headquarters

List of Figures

List of Tables

Zusammenfassung

In diese Arbeit werden wir die Verwendbarkeit der populär Frameworks für Abnahmetests auswerten. Wir werden eine Umfrage in eine große Unternehmensorganisation durchführen, um die Verständlichbarkeit des Tests miteinander zu verglichen.

Dafür werden wir einige Abnahmetesten mithilfe von allem drei recherchierte Frameworks: JUnit, Cucumber und Robot Framework implementieren. Danach werden wir diese Implementationen zwischen den Umfrageteilnehmer verteilen. Das Ziel der Arbeit ist es zu verglichen, ob es die Unterschiede bei diesen Frameworks zwischen Verständlichkeit und Benutzbarkeit für den Benutzer sind.

Als Ergebnis sollen wir die Vor- und Nachteile der moderne Test-Framework verstehen können. Wir werden eher Schlussfolgern können, dass einer der Frameworks mehr verwendbar als die andere sind, oder dass die Frameworks eher gleich sind.

Chapter 1

Introduction

Acceptance tests in software development are a form of tests aimed at verifying that the tested system is fulfilling specifications. In contrast with unit tests, requirements for acceptance tests come from people with a deep understanding of the application's domain logic - e.g., business logic or possible legal nuances and issues.

Automation of acceptance tests, however, creates the following problem. Developers working on test implementation are not expected to understand the domain they are working in full. In contrast, people with domain knowledge typically lack programming experience to evaluate the quality of implemented tests. With changes to the requirements over time, this communication gap may result in acceptance tests becoming an obsolete black box that no one in an organization can fully understand.

Martin [2008] provides an example of a software company that went out of business because of bad code. Management rushed developers and demanded constant changes and bug fixes, and this went on until software developers could no longer maintain their product. Later in the book, Martin postulates, that the code must always be rigorous, accurate, formal, and detailed - otherwise, owning a "bad" code costs too high. The same requirements should apply for testing code as well - acceptance tests must be maintainable by software developers while staying clear and understandable for stakeholders or end-users.

This thesis aims to compare existing solutions and practices to determine which approach is best suited for implementing large sets of acceptance tests in domain logic-heavy systems.

An ideal solution would meet the following criteria: allow and encourage developers to write clean and understandable code; ensure that testing code is manageable for a long time; ensure that implemented tests are understandable for people without technical knowledge.

In the real world, achieving such feats proves to be a challenge. To make testing code easy to read by end-users, developers have to hide details underneath several levels of abstractions. However, the more abstract code becomes, the harder it gets to find workarounds in corner and exception cases.

There are several ways to implement acceptance tests that this study author has met in practice. Acceptance tests can be implemented like regular unit tests, using all tools that programming language/unit testing framework provides. Alternatively, developers could define a custom domain-specific language (DSL) and use it to implement tests so that users with domain knowledge can easily understand.

This work's scope should be limited to frameworks and methods, used internally by Swedbank across various teams. A long history of different teams working independently from each other resulted in various solutions for similar problems. Various development teams wrote their acceptance tests as regular unit tests with Java and JUnit [JUnit, 2018]; with human-readable DSL defined with Cucumber Framework and using glue code implemented in Java [Cucumber, 2019]; by using Robot Frameworks - an already defined DSL which combines human-readable language and pseudocode constructs [Robot Framework Foundation, 2019].

Additionally, this work's scope will be limited to the following question: is there a difference in how usable testing code can be, depending on the test framework? In other words: can we produce a clean and easily readable code using one of those frameworks? Do custom DSL of either Cucumber or Robot Framework make more sense than JUnit code for a stakeholder with no programming experience, or if there is no difference?

We will conduct a survey inside one large organization - namely, Swedbank, to find out how well the tests could be understood.

We will implement several acceptance tests using all three frameworks and hand those out to the survey participants. The goal is to compare how well were those tests understood and interpreted, and if there are noticeable differences between frameworks. The survey will be for everyone - from test automation engineers and software developers to team and product managers and domain experts.

In the conclusion of this thesis, it should be possible to understand the differences between modern testing frameworks, their advantages and disadvantages. The results will need careful assessment: it might be the case that the results will show the superiority of one method above all else; it might also be the case that end-user results will differ from software engineers. In the latter case, this work's results could either improve one of the frameworks to be more end user-oriented or to define development practices that would improve code maintainability.

Chapter 2

Background

In this chapter, we will go over the necessary background for doing this work, such as:

- Common terminology
- Applicable standards
- Brief description of psychophysics

2.1 Terms and definitions

For the duration of this work, the terms and definitions given in ISO/IEC/IEEE 24765 will apply.

dynamic testing testing that requires the execution of the program code

feature functional characteristic of a system of interest that end-users and other stakeholders can understand

feature set a logical subset of the test item(s) that could be treated independently of other feature sets in the subsequent test design activities

test case 1. a set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement;
2. documentation specifying inputs, predicted results, and a set of execution conditions for a test item;

3. set of test case preconditions, inputs (including actions, where applicable), and expected results, developed to drive the execution of a test item to meet test objectives, including correct implementation, error identification, checking quality, and other valued information

Note 1 to entry: A test case is the lowest level of test input (i.e., test cases are not made up of test cases) for the test subprocess for which it is intended

test case specification 1. document specifying inputs, predicted results, and a set of execution conditions for a test item;
2. documentation of a set of one or more test cases

test design specification a document specifying the features to be tested and their corresponding test conditions

test driver software module used to invoke a module under test and, often, provide test inputs, control, and monitor execution, and report test results

Note for the entry: for the duration of this work, we may interchange "test driver" and "test framework" terms as they seem to be synonymical for this study.

2.2 IEEE 29119-2

We refer a section 8 of Software testing - Test processes standard for processes required for implementing acceptance tests. In the scope of this work, we focus on TD4 and TD6 from figure 2.1, and comparing different approaches for deriving test cases and test procedures. We may assume that we are dealing with provided feature sets, but stakeholders have no preferences over the framework we use, so choosing a technology is up to us.

2.3 Quality Engineering

The basis of working with human respondents was researched by [Möller, 2010, chapter 2]. While working with respondents, we assume that a physical event (in our case - observing of test implementation) affects the answer of a respondent. However, we do

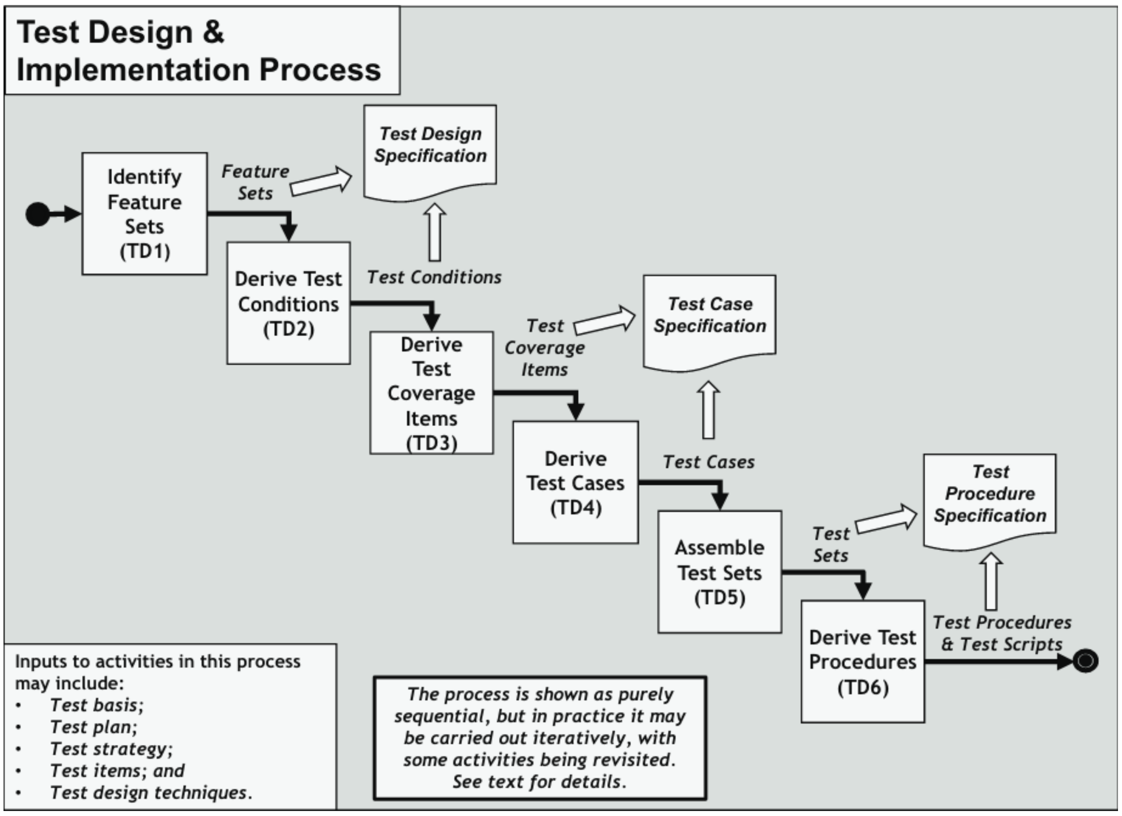

Test Design & Implementation Process
Identify Feature Sets (TD1)
Feature Sets
Test Design Specification
Derive Test Conditions (TD2)
Test Conditions
Derive Test Coverage Items (TD3)
Test Coverage Items
Test Case Specification
Derive Test Cases (TD4)
Test Cases
Assemble Test Sets (TD5)
Test Sets
Test Procedure Specification
Derive Test Procedures (TD6)
Test Procedures & Test Scripts
Inputs to activities in this process may include:
• Test basis;
• Test plan;
• Test strategy;
• Test items; and
• Test design techniques.
The process is shown as purely sequential, but in practice it may be carried out iteratively, with some activities being revisited. See text for details.

Figure 2.1: Test Design and Implementation Process
Source: ISO/IEC/IEEE 29119-2:2013 ©2013, IEEE

not know what the respondent really thought to himself.

A correct answer might be given by a wrong reason, or just be a lucky guess (especially if a survey has single or multiple choice questions). An incorrect answer might be caused by a misunderstanding of a question itself, not the test implementation.

Furthermore, we must consider the background of a participant. It is likely that someone who works with code routinely - e.g., software developer or a test automation engineer - will be able to analyze test implementation faster and more accurately than an end-user or a stakeholder. A test specification and test implementations, mimicking the style of internal bank code standards, could also give an advantage for those, that worked in a bank for a longer period of time.

2.4 Summary

In this chapter, we've listed the terms and definitions that are going to be used during this work. We overviewed the applicable IEEE standards for software testing and documentation and reviewed the general challenges of working with human respondents from the area of psychophysics.

Chapter 3

Related work

In this chapter, we will review the state of automated testing and go into a history of development and testing practices that lead us to the current moment.

3.1 Brief history of unit testing

JUnit is a unit testing framework that revolutionalized the software development field by introducing new development processes and practices.

JUnit history begins in 1994 when Kent Beck, working as a consultant, prepared a presentation to introduce a concept of automated tests for a software development company. According to Beck [2004], he wanted to present a concrete example. He wrote a tiny testing framework, consisting of three classes and twelve methods, using the Smalltalk programming language. Even though Beck did not think much of the usefulness of a project initially, its importance was confirmed later on by other senior developers. The project was accepted both in a company he made it for and in the Smalltalk community.

Three years later, Beck went on an OOPSLA conference together with his friend Erich Gamma. Beck wanted to learn Java, while Gamma was curious about his friends' testing framework, so together, they decided to port that framework to Java. They worked in a process that later got named Test-Driven Development (TDD): at first, they wrote tests (using a framework that did not even exist yet!), and only then they implemented the framework itself. Framework got its first fans before the conference even ended, and so upon returning home, friends agreed on a name JUnit and prepared their first public release.

Interestingly enough, in their early releases, JUnit developers also spent some time on several features that we could now evaluate as unnecessary. Among those features, we can name such as standalone GUI for test execution and on-the-fly class reloading - they were removed in following major releases, as JUnit got better integration with IDEs and build systems. Dependency Finder [2020]

JUnit also was ported to at least 30 other programming languages [Beck, 2002] under a common family name xUnit. According to Beck [2004], this shows that the problems solved by JUnit are universal. At the time of writing this work, there are three publicly accessible major versions of JUnit - 3, 4, and 5, where JUnit 3 is outdated, and JUnit 4 is in the process of replacement by JUnit 5. Throughout all versions [Beck, 2004, Tahchiev et al., 2010] and releases JUnit team followed the same goal of simplifying tests, and defines three discrete goals for the framework:

- The framework must help with writing useful tests
- The framework must help with the creation of tests that retain their value over time
- The framework must help with lowering of the costs of writing tests by reusing code

In retrospect, we can see that the xUnit family development family impacted software development so much because it introduced and standardized a new approach for software development. Before xUnit, a software developer would only write production-ready code and let manual testers worry about testing. Before xUnit, developers would only test their new functionality with small bits of throwaway code, that they would themselves delete once they made sure that the code is working. Even the idea of writing automated tests instead of focusing on a production code could seem like a counter-productive waste of time for many. Upon trying a new framework, however, even the most experienced developers realized that it was quite the opposite: having automated tests in place allowed them to find bugs sooner, modify and deliver new code faster, write more structured code, and save money for the business by not relying on expensive external consultants if something broke down.

A logical next step from automating tests was formalizing the rules of Test Driven Development. As per Martin [2008], "do not write production code until you have written a failing unit test; do not write more of a unit tests that is sufficient to fail (not compiling is failing); and do not write more production code that is sufficient to pass the currently failing test." This development process should allow writing clean, testable, and covered by test code from the very beginning, instead of trying to come up with a way to refactor and test new code later.

A recent data-mining study [Stefan et al., 2017, p. 405] revealed that JUnit is used in approx. 30% of all open-sourced Java projects. Another study on test framework usability by [Brunet et al., 2011, p. 149] demonstrates, that in-built in JUnit assertive decorations are preferred for software developers and are intuitively understandable.

3.2 Behavior Driven Development (BDD)

Over time, many developers began to notice problems with the TDD approach. While TDD describes clearly how new code should be written, it becomes less clear what to do when requirements change and old tests start failing. This problem has inspired many software engineers worldwide to think and try to solve or work around those issues. Their contribution has led sequentially to the formularization of a concept for BDD.

At first, Chris Stevenson created a simple documentation utility TestDox [Stevenson, 2003], which printed test case names from JUnit as a sentence by splitting words: method "testFindsCustomerById()" would turn into "finds customer by id." Just having this documentation utility encouraged developers to write test methods as real sentences in a language of a business domain, which, in turn, allowed domain experts, analysts, and testers to understand those methods more easily.

North [2006] categorized test failures in three categories: a test may fail because the developer introduced a bug, a test may fail because some parts of functionality were moved elsewhere, or a test could fail because the tested behavior is no longer correct. A test either needs to be fixed, moved, and probably changed, or removed. By that time, there already was a convention to start test method names with the word "should." North observes that this convention implicitly helps developers challenge the premise of a test: "Should it really do that now?", making deletion of obsolete tests easier.

With his next steps, North reasoned, that the word "test" itself is confusing, and suggested to replace it with a more useful word "behavior". That way, it solved the naming problem - developers could describe the behavior they are interested in for a test name. A single sentence can not fit the too large description, which naturally limits a test's scope.

At about the same time, Eric Evans published a book on Domain-Driven Design, where he addressed problems with a common language and sharing knowledge between colleagues. As per Evans [2004] "A project faces serious problems when its language is fractured. Domain experts use their jargon while technical team members have their language tuned for discussing the domain in terms of design...Across this linguistic divide, the domain experts vaguely describe what they want. Developers, struggling to understand a domain new to them, vaguely understand."

This book has made a significant impact on both Java and Ruby developer communities. To discover and facilitate a ubiquitous language - a universal language that would be understood both by domain experts and technical team members, for Java was created a JBehave framework and for Ruby - RSpec testing. While JBehave drafted the first approach for imposing a "Given - When - Then" structure for test specification (by extending Given, When and Then Java classes), RSpec allowed developers to describe expected behavior with plain strings. Later on, JBehave got ported to Ruby (as RBehave) and became part of an RSpec project. Experiments for combining those two approaches in a "story-based description" led to the creation of a Cucumber framework.

In Cucumber, the system behavior is specified in a set of feature files describing a user story and written in Gherkin language. Cucumber runner executes those feature files by matching the human-readable text against annotated code. Feature files are intended to be the definitive source of truth as to what the system does. With this approach, development teams can save time by keeping requirements, documents, tests, and code all in sync, and where every team member can contribute towards improving system correctness [Hellesoy et al., 2015].

A paper by Oliveira Bertholdo et al. [2011] also illustrates that mapping of goals (like usability goals) into a development process by using a BDD framework plays an important role in agile development, and strengthens the understanding of a typical user, their needs, and the context of their use.

3.3 Robot Framework

Independently from Java and Ruby communities, Klärck [2006] worked on problems of test automation for his master thesis. While xUnit family of unit testing frameworks was mentioned, his work was based on academic research and scientific standards and described steps needed for creating a data- and keyword-driven test automation framework. The framework's high-level requirements were: full automation of test execution, ease of use, and high maintainability. This framework would also explore data-driven development: a test development process, where input data is separated and kept apart from the code in tabular format. The next logical step from data-driven testing, as pointed by Dorothy Graham [1999], is keyword-driven testing. Similarly to Cucumber feature files, test specifications could also be written in a separate file, but this framework format for such files could be any tabular one, like CSV, TSV, or even Excel file - allowing users to modify specifications without any special software.

The thesis confirmed the feasibility of implementing every single concept and functionality and suggested using high-level scripting languages such as Perl, Python, or Ruby. The first version of a framework was implemented the same year in Python within Nokia Siemens Networks, and per Klärck [2020], PyPi [2008], the project was open-sourced under the name of Robot Framework in 2008.

3.4 Usability research

The usability of development practices and processed has been researched in various papers. [Stefan et al., 2017] researched the usability of a custom JUnit-based test driver by observing how experienced programmers would implement given test design specifications. They used a Think Aloud Protocol as described by Lewis [1982] - participants were invited to keep up the running commentary on their actions, to see if there are any questions or confusion. The framework was evaluated against cognitive dimensions defined by Clarke [2006] such as "Abstraction level" and "Learning style," and the results of this survey led to improvements to the API.

3.5 Summary

In this chapter, we reviewed the current state and history of all three frameworks that we will focus on in this work.

Each framework had its historical context for creation and was written to address some challenges in software development.

JUnit was one of the first modern unit testing tools and has become a de-facto standard, known to most Java developers.

Cucumber framework was born out of various development communities' efforts towards bringing domain and technical experts together by writing specifications in a common language that both developers and "business people" can understand. Robot Framework is a practical implementation of academic research on Data- and Keyword-driven development concepts that found its place in regressions and UI tests.

In the following chapters, we will introduce methods for comparing those frameworks against each other and survey to determine how well those frameworks can help with quality assurance in their current state.

Chapter 4

Methods

In this chapter, we will go over the methods we will use for conducting a survey. Till the end of the chapter, we should answer the following questions:

- How are we going to conduct our study?
- What questions are we going to ask the participants?
- What are we going to show them?
- How are we going to evaluate answers?

4.1 Preconditions for a survey

We are going to conduct a survey within a bank where the author is employed, and the target audience consists of four groups of participants:

- Software developers - people with Java/Python knowledge (requirements for the position often include JUnit knowledge and familiarity with other testing concepts)
- Manual testers - People who should have a good overall knowledge of some of the banking systems and are familiar with working on test plans and scenarios
- End users - customer managers and bank tellers. People with no programming experience working with the product on a daily basis.
- Stakeholders - product owners, product support managers, team managers. Depending on the background, they may or may not have programming or testing

experience; however, it is their formal duty and responsibility to ensure that products are working up to specifications

Further preconditions for the survey are:

- Limited time and budget - expensive technologies such as eye-tracking cameras were unavailable
- Distributed teams across several countries, located in various cities and buildings
- Security measures requiring risk assessment and approvals for any new service or software executed internally
- Only a number of whitelisted websites are allowed for internal usage

4.2 Survey

As per Geisen and Bergstrom [2017]: "No questionnaire can be regarded as ideal for soliciting all the information deemed necessary for a study. . . . The researcher must use experience and professional judgment in constructing a series of questions that maximizes the advantages and minimizes the potential drawbacks". From the authors' experience, every test suite requires some context for understanding. While reading through test case specifications, it is best to interact with the target system in some safe (e.g., testing or staging) environment for better understanding. Sometimes one may want to manually repeat some steps (if possible) or sometimes merely remembering how the system looks like is enough.

To give all participants an equal chance for understanding test specifications, we should provide them access to the system we test. For this survey, we create a mockup for a real-life system, that would allow participants to interact with it as much as they would want to. Functionality of this system is described in section 4.3. Constructing our sample system has further advantages for this survey: we provide every participant a new system they have not yet worked with before; having a sample system allows us to validate test implementations; and lastly, even though this survey was approved by authors' manager, creating a sample system just for the research reduces legal risks of breaching an NDA, and requires less work for final approval for the publishing of this work.

Having a sample system in place, we can start with the survey implementation. With limitations from 4.1, the best form for our survey is in an online, web-based format - that way, we will be able to distribute questionnaires among colleagues from all the cities fast and without the need for an author to travel anywhere.

The survey is structured in the following way: a participant is asked to analyze several test case specifications and answer questions related to the understanding of target system features. We follow the advice of Rea and Parker [2014] "potential respondents are more likely to participate when they perceive that the study's findings will directly impact their well-being" by telling participants that the outcome of this survey is important for their work. Motivation is simple: if we find the best framework for their work, the job should get more comfortable.

There are 3 test specifications for the survey, implemented in all three frameworks. There are three tasks in the survey - to analyze each specification and answer probe questions or perform certain steps on a sample system. Questions are listed in table 4.1.

4.3 Sample system

We create a sample system that was imitating work with term sheets for the survey. Term sheets are used for managing and negotiating non-binding agreements with bank customers and for contract preparations. The author created a site that allows users to view a list of offered term sheets grouped by products offered to one or several parties. A user can add a new product, add new term sheets for any product, change agreed contract limits for a term sheet, and adjust risk class for parties.

This functionality mimics a slice of real-world banking applications. However, it is not connected to any real system and only used fake data to avoid any possible legal breaches.

4.4 Test implementations

Here we should briefly overview how we wrote test specifications for each framework and why we choose this way. Full listings of code are in Appendix B.

Table 4.1: Survey questions and expected answers

Question ID	Question	Expected answer
age	What is your age?	Radio button for one of the age groups
experience	How many years have you worked in a bank?	Working experience in a bank
role	Regarding products and specifications, what describes your work the best way?	Radio button for one of the following: Customer Developer Tester Someone who defines test specifications Test automator Other
other	Other (please specify)	Other role
	First test case with following actions is shown: User opens term sheet nr. 155, clicks button "Edit term sheets", enters "1000" and saves changes; It is explicitly verified that value "Change of limit" is changed and properly formatted.	
termSheetOpened (1)	Which term sheet will be opened?	155
tc1 (2)	Could you sum up in one sentence what this test does?	Should mention change of limit and validations
tc1rerunnable (3)	Can you re-run this test and expect it to pass?	Yes
tc1fix	If no - what would you change/suggest developers to change to make this test repeatable?	*blank
	Second test case with following actions is shown: Within same term sheet user presses "Edit Term Sheets" button, adds new product and contract to it, tries to save changes; It is verified that this won't work and the user will see an error; Then the user fills required fields and presses "Save" button again; It is verified, that value "Change of limit" is changed and properly formatted for a row from the first test case, modified and summary rows;	
tc2validations (4)	How many validations does this test have?	5
tc2checked (5)	What is checked in this test?	Should mention following checks: user sees error message data is saved 3 rows contain proper values
tc2dependant (6)	Does this test rely on the success of the previous test?	Yes
tc2changed (7)	Can you sum up in a few sentences what will be changed in the system if we run this test?	New product and contract rows are created
	Third test case with following actions is shown: User opens term sheet nr. 0; User clicks "Submit form" but sees an error; User assesses risk class for all companies as "Low" User clicks "Save changes" then "Submit form".	
	User is given a link to the term sheet and is asked to repeat those actions manually	
tc3properly	Do you feel you were able to execute this test properly?	"Yes/No/Maybe"
tc3help	Would you need a help from colleague (software engineer or domain expert) to better understand what this test does?	"Yes/No/Other"
tc3other	My answer	-
tc3form (8)	JSON was automatically validated	Are all rows correct? Were risk classes assigned properly? Were changes submitted?
satisfaction	How satisfied are you with the approach that was used for test specifications?	1-10
acceptability	Would it be acceptable for you to work with acceptance tests if they were all defined the way you saw in this survey?	"Yes/No"
feedback	Do you have any more comments or suggestions?	Free input

4.4.1 Supporting code

We create two helper Java classes for JUnit - FormPage and TermSheetRow encapsulating logic for accessing page elements.

Cucumber specification cannot work without glue code in some programming language. For this task, we construct a DSL which enables us to work with data rows in the Term Sheet by their ordinal position and row type. In test we might access or edit rows and their corresponding columns by specifying something like "Input <text to input> in [first/second/third/last] [customer/product/contract] row 'column name' column". We would also have to specify row type and rows number to create a child row for a customer and product rows.

For the Robot framework, we take advantage of the framework support for variables. We define keywords that return map objects describing row coordinates. Using those, we can create new child rows or edit columns of the selected row while making a tree-like structure of elements more transparent for the end-user. Furthermore, this should minimize potential efforts for maintaining the testing codebase.

For all frameworks, supporting code is not shown for the participant: the focus is on test cases and scenarios that we built on top of that.

4.4.2 First test specification

At first, we introduce participants to the framework they will analyze, so we will not lead with complicated cases. In this scenario, the user opens a web page and changes a limit for a first contract. We store a page that should be opened is stored in a global variable in the Robot test, in a global static field for the JUnit test suite and defined it in a Background section of the Cucumber feature file.

4.4.3 Second test specification

For this test, we take the most straightforward approach with the Cucumber framework, as mentioned in 4.4.1. In the test, we take advantage of the fact that new entities (product and contract rows) are added on top, right after their parent nodes, enabling us to implement test scenarios without going into details of how the rows are related to one another. We create a new product for the first company, and then we create a new contract for the first product. This approach has several drawbacks - to understand, which contracts for which products should get modified, a user might have to repeat each step manually. Another problem is if the target system would be changed - for example, by adding new rows at the bottom, right before parent's next sibling node - then all the tests would need to be rewritten. Such limitations can be acceptable if test specifications are getting re-negotiated every time serious design changes occur or if we know with some degree of certainty that the design of the system will not be changed. Otherwise, updating large parts of test codebase will slow the development speed down.

As mentioned before, we use variables in the Robot Framework to manipulate variables. Thus, test specifications are more resilient against changes, but it comes at the price of needing extra lines of code for variable assignment. Reading the code also gets more complicated, because now users have to wander with the eyes on the code to find which variable was defined where. On the positive side, this code is easier to manage in the long run - should table logic change, the developers would only need to update the definition of a few keywords and not rewrite whole scenarios.

A test specification states that a user should try saving term sheet with incorrect data at first, get a validation error, and then fix the data to have the term sheet saved. Both Cucumber and Robot framework tests are split into two smaller parts - scenarios and test cases - so that each part will follow the "Given - When - Then" structure.

The general recommended approach for clean Java code is to keep each method's size to the minimum and let each method do only one clearly stated purpose. Those practices typically help and encourage software developers to create testable and maintainable code. However, they stop working so well when we have to implement a specific flow of actions dictated by an outside party. Surely it would still be possible to separate test case somehow - by splitting it similarly as we did it with JUnit and Cucumber, or by creating even smaller methods. However, even that would require to solve several challenges, like "Can we construct a method that will create new products and contract rows?", or "How should we pass newly created rows for other methods to use - in some global fields, or by returning a map object or a data class?".

While all those questions could be solved, it is often a good idea to start with a heads-on solution, and see where it leads. For JUnit tests, we define created FormPage and TermSheetRow classes to convert test specifications from human language into a Java code "as is."

4.4.4 Third test specification

For the final test, we have a condition to update the risk classes of "all" the customers. In Java, this is done by getting a list of rows and iterating over them. In the Robot framework, we have a ":FOR" directive for loops as well.

For Cucumber, the closest we can get for cycles is a Scenario Outline (or Scenario Template) keywords, that allow us to repeat one scenario several times with different input parameters. This only works if all the parameters are known at the time of creating a test. Since there is no information about what kind - or how many companies there will be, we have to resort to creating a magic "set every risk class as <Low>" step, which works in the same way as Java code in its glue code part.

4.4.5 Overview of specifications

Customer-faced implementation of specifications has 39 lines of code in Cucumber, 73 lines of code in Robot framework (from those seven lines are configuration or section separation parts), and 61 lines of JUnit Java code (from those 10 have only annotations or braces).

4.5 Preparation

The author created a web service using Java and Spring Framework. Service distinguishes users by assigning a random "session-id" HTTP cookie to each user. Users can navigate to the sample system located on /memo URL path, or to the survey on /survey path. Each user has his separate cache for Term Sheets - that way, each user can interact with the sample system without affecting the work of others.

"Ease of use is especially important for self-administered surveys since there is no interviewer present to explain how to use the survey. If the survey is too complicated, people will simply not respond" [Geisen and Bergstrom, 2017]. We keep that in mind and follow agile practices to perform tests more often, but in smaller groups and shorter "fix issues - test - discover further issues" working cycles (as opposed to waterfall model). The survey website was tested in several iterations on small focus groups. At first, only basic functionality and correctness of navigation logic was tested. On the second step, various design layouts were tested. Among others, there were tested vertical layout - test specification on top, questions are listed below, horizontal layout - test specification on the left, questions on the right, as well as several others. It was confirmed that usability testing participants were the most comfortable with a vertical layout, which reduced risks of some missed questions, even though it might require some extra scrolling if some questions were too long.

During the survey, a participant is allowed to navigate back and forward. No questions were marked as mandatory, and users were encouraged to skip the question they could not - or would waste too much time understanding.

We log each user interaction within the survey. Upon completion of the questionnaire, all answers are written to another log. While constructing the survey, it was unclear what kind of infrastructure the author would have, so the service has no database and only log interactions in a text file.

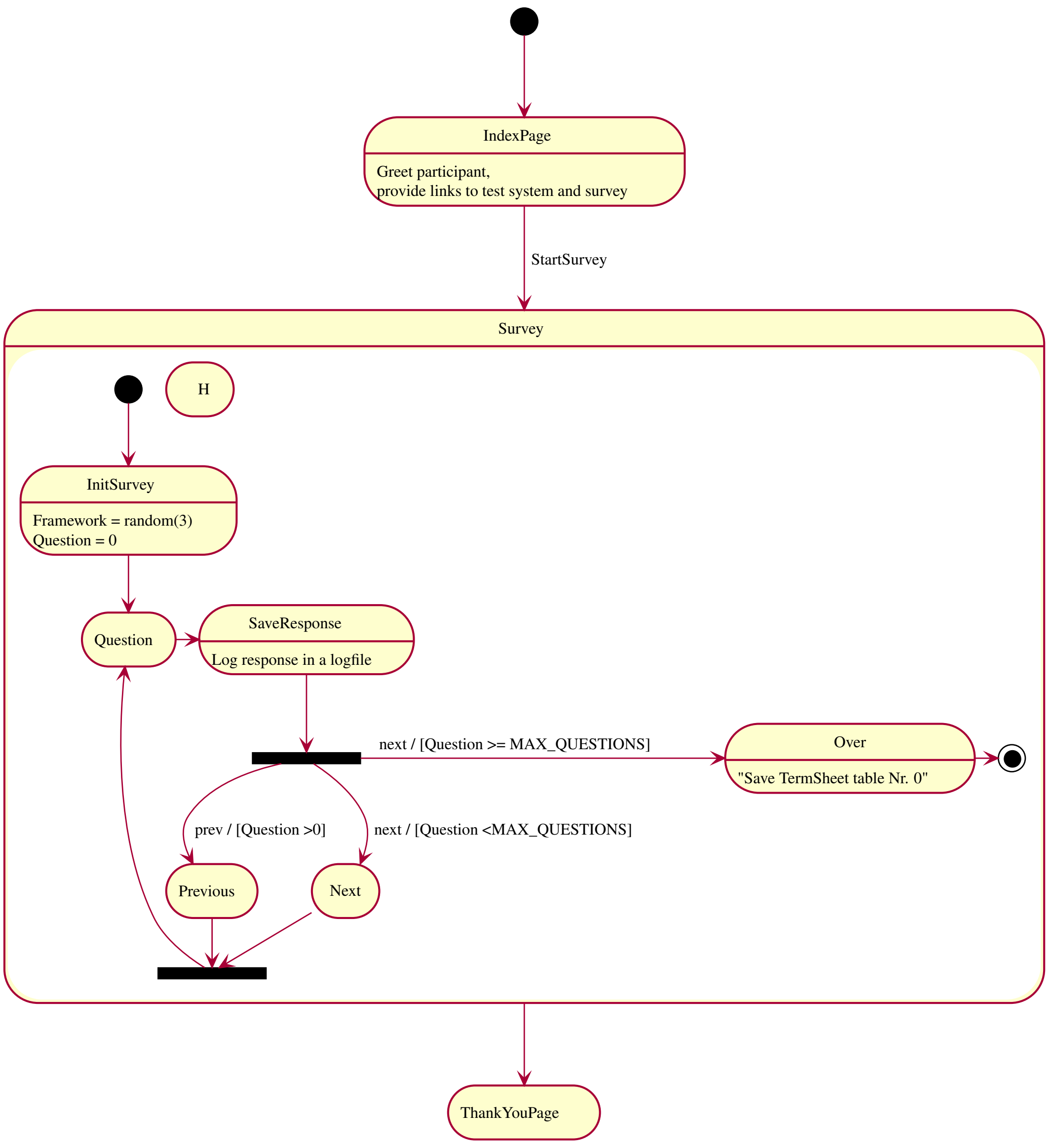
Originally author intended to choose frameworks at random during the survey for each question. However, early usability tests revealed that the participants find this confusing, so in the remaining versions of a service, we start to choose the testing framework at random whenever a survey begins to see the same framework during the whole session.

In a final step of survey preparation, survey service was deployed to a local hoster's virtual private server. A security team reviewed and whitelisted the website for a duration of a survey - under the circumstances, it was safer and easier than trying to get an approval for running a custom service on the internal network.

4.6 Evaluation of answers

A custom-made CLI utility in Kotlin programming language is used to evaluate answers. This utility uploads logged responses to a MongoDB database and enabled evaluating answers one by one. For the evaluation, we can mark "+" or "-" (meaning "correct" or "wrong" and stored as 1 or 0 respectively), or add a text remark comment that should have started with colon symbol. The order of answers was shuffled randomly each time

Figure 4.1: Survey service state diagram for participant


IndexPage
Greet participant, provide links to test system and survey
StartSurvey
Survey
H
InitSurvey
Framework = random(3)
Question = 0
Question
SaveResponse
Log response in a logfile
next / [Question >= MAX_QUESTIONS]
Over
"Save TermSheet table Nr. 0"
prev / [Question >0]
next / [Question <MAX_QUESTIONS]
Previous
Next
ThankYouPage

the utility was started to prevent any bias from the evaluator's side.

We also count all those who abandoned survey or skipped a question in a separate group as seen on the following figures 5.3, 5.2 and 5.4.

4.7 Summary

In this chapter, we defined a list of questions for the participants. We decided to use web format for conducting our survey, and selected technologies for the survey implementation. We implemented selected tests for all three frameworks, to present to the participants. Finally, we created a utility and defined a method for evaluating user responses as impartially possible.

Chapter 5

Conducting an experiment

In this chapter, we will conduct a survey for several groups of participants and provide overall results.

5.1 First survey

The first survey was organized within the authors' team among the closest colleagues.

Despite usability testing preparations, the first real-world survey still yielded several problems. In the original design of the survey website, pressing the "Previous" button on the form resulted in discarding the participant's currently entered data. Analysis of logs showed that one of the participants wanted to check their previous answers and went back to the first question upon reaching the final question. Since the survey does not allow navigating to a specific page, the participant had to click "Next" several times. This, supposedly, resulted in them paying no more attention, and the fact that the input from the last question is lost was overlooked. This behavior was fixed by improving the page navigation logic, as seen in the diagram 4.1.

Another problem with the initial survey was that it did not store unfinished questions, and I had to extract those answers from response logs manually.

As seen in Figure 5.1, participants were not equally distributed in all three groups for this survey to draw any conclusions from it. Seven participants started the survey with cucumber framework, two with Robots framework and two with plain Java implementation, and six, one and two respectively completed it.

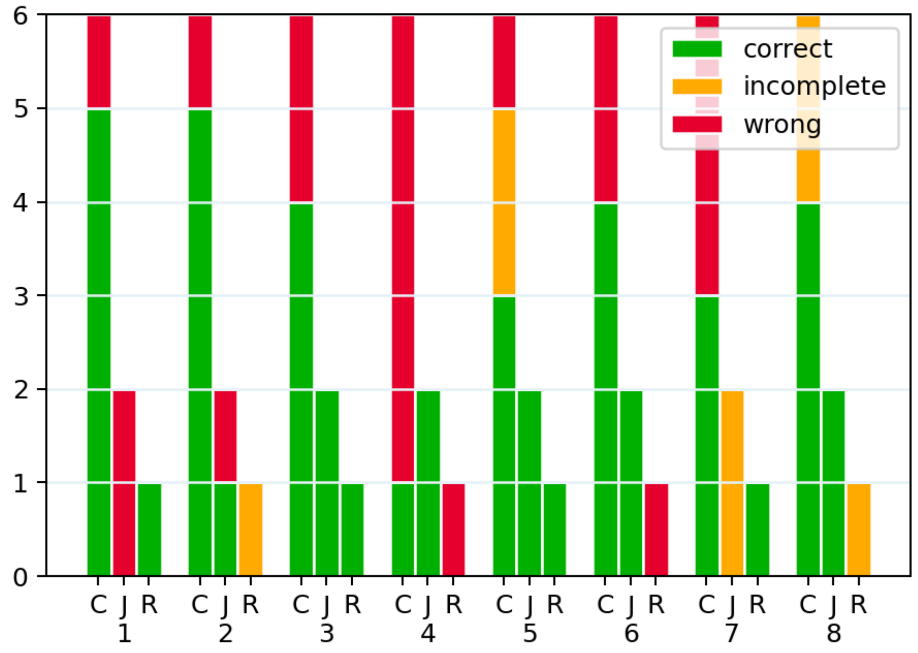


Figure 5.1: First survey: one team;
Results for (C)ucumber, (J)Unit, (R)obot Framework, for questions 1-8

While analyzing the results of a first survey, we can see that question 4 has a higher number of wrong responses - five out of six responses are wrong. However, in a follow-up question 5, we see three correct answers two incomplete, and only one is wrong. This seems strange: in question 4, we ask "how many checks are there," in question 5 - "what is checked." One could assume that the participant should either passes or fails both questions. After all - if you understand that a line contains some validation - you should be able to count it. However, it is not the case here.

Furthermore, unlike JUnit, both Cucumber and Robot frameworks encourage implicit error checking with HTML report format. In Cucumber, a user can see each line's execution status as it is marked with either green, red, or gray color. The same applies to a Robot framework. For JUnit, the source code of a test specification is not a part of a report, and the only thing user gets is whether a whole test case has passed or failed (with a stack trace). This difference could also lead to different interpretations among survey participants.

Finally, in JUnit, every "check" is defined with assertions and starts with the same "assert" (-Equals/-NotEquals/-True/-False) method. Cucumber guidelines recommend providing starting conditions with "Given," test steps with "When" and validations with "Then" keywords. However, our study showed that some responders do not understand the difference between "When" and "Then" steps. It might be the case that participants counted those rows as validations: there are 7 "When" steps and 5 "Then" checks, and some answers ranged from 9 to 12.

Unfortunately, even though the results of a first survey were analyzed, no overall diagram was made. As a result, problems with Questions 4 and 5 were not discovered until the completion of a second survey. Otherwise, the author would adjust those questions to include more precise wording or potentially change those.

5.2 Second survey

The second survey was conducted for a broader audience. It was distributed among all scrum masters of Swedbank, all those interested in the QA topic, and a few stakeholders involved in maintaining services. The invitation text is in Appendix C. The invitation letter also permitted forwarding it to any number of colleagues.

In total, 35 colleagues got past the first survey question.

From figures 5.2, 5.3 and 5.4 we see that participants with JUnit skipped the least questions, while participants with Robot Framework skipped the most. Among those who answered, most of the answers are correct. The only exception is questions 4 and 5 - for the reasons discussed in 5.1. If we exclude those questions, then there are a total of three incorrect and six incomplete answers for both JUnit and Cucumber framework, and four incorrect with ten incomplete answers for Robot framework.

Upon reviewing results, it became apparent that within the authors' team, an understanding of what is a "test" matched authors', and so a few imprecisions with question wordings were overlooked. For example, in question 3, with the follow-up for a question, the intended answer would be "Yes" and blank input field, respectively, since the test might be restarted as many times as the user wants. However, a certain number of participants observed that re-running this test a second time the way it is written will not change the system. There are no checks in the beginning that the initial "Change of

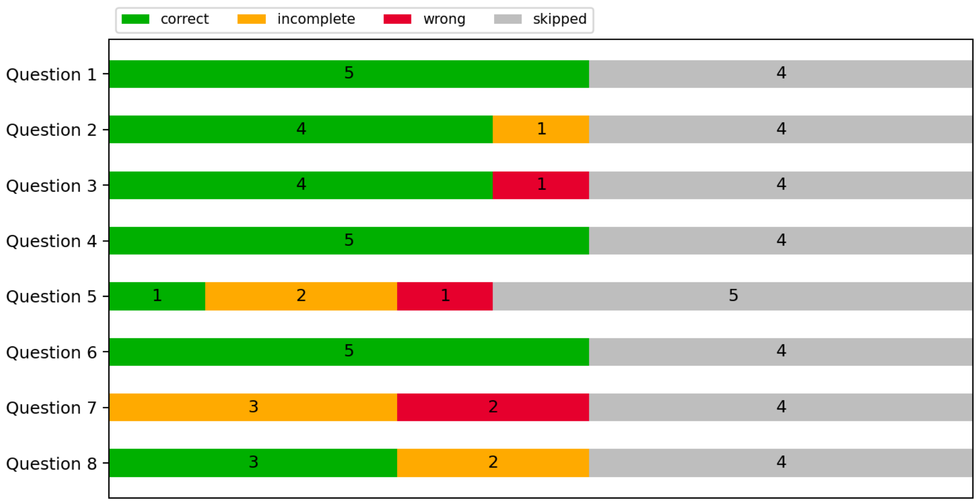


Figure 5.2: Second survey.
Results for JUnit

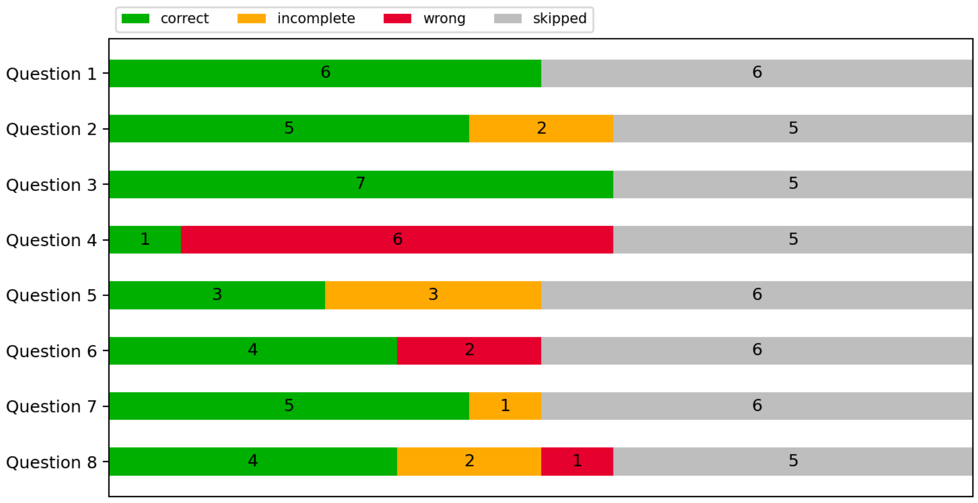


Figure 5.3: Second survey.
Results for Cucumber

limit" field is not "0", so the test will overwrite the same value. Depending on a system, this might be a "silent failure" or a false positive for the system under tests. Answers that noted this kind of failure were marked separately, and we can see the differences in the table 5.1.

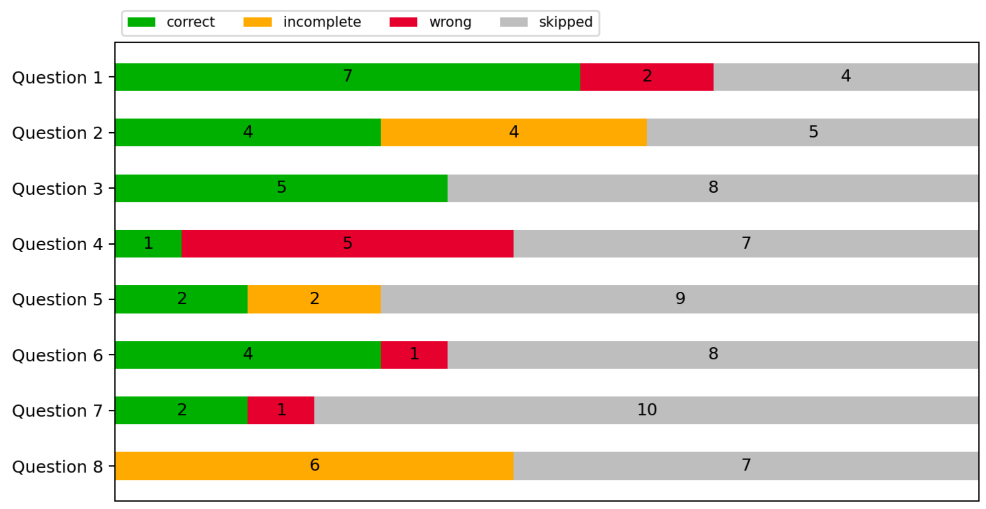


Figure 5.4: Second survey.
Results for Robot Framework

Table 5.1: Different understandings of test being re-runnable in question 3

Framework	Technically correct	Noticed a flaw
JUnit	1	3
Cucumber	5	2
Robot	4	1

5.3 Summary

In this chapter, we discussed the initial findings of our experiment. Raw results of the survey may be found in the data archive attachment for this work or in a project repository. A more detailed qualitative analysis for the second survey will be provided in the next chapter.

Chapter 6

Evaluation

In this chapter, we will go into more detail and analyze the results of a second survey, conducted on the whole organization and draw conclusions from our research.

6.1 JUnit

Among all three frameworks, JUnit has the least skipped questions: with only one exception, everyone who started the test finished it without skipping a single item.

In question 3, a single participant got distracted with a new feature of JUnit 5: advised using a "new RepeatedTest annotation" - and missed the essence of a question completely. It was asked whether the test would pass or fail after running several times in a row, not if we have a means for re-running tests. For the remaining JUnit respondents, three out of four colleagues noticed that test is flawed and may give a false-positive result on a second run. This number is higher than with any other framework - only two Cucumber and one Robot framework respondents could notice the same issue on that question. One of the JUnit participants who noticed that even had no prior programming experience: that participant chose a "customer with no interest on how the tests are implemented" role, and repeated in the question that he or she has no prior programming experience.

As discussed in a previous chapter, question 4 was answered correctly by all JUnit participants. For question 5, we see worse results - there is only one entirely correct response. The rest either gave a vague explanation ("check for adding product"), which was marked as wrong and two more incomplete answers where both the participants missed some parts of functionality.

For question 7, we have two wrong and three partly correct answers, and it can be considered the worst result among other frameworks on that question. From specification, this scenario should have tested the user's ability to save a new product and contract rows after correcting errors from a previous failed attempt. The whole test consists of a chain of actions - the user creates new data, attempts to save it, receives validation error, corrects those, and finally saves the data. To implement this scenario without creating additional helper methods, we had to go against clean code practices from Martin [2008] - and write the test case as one method. Judging from the answers, we may conclude that the users found it difficult to track what changes did affect the system.

It is possible to rewrite the test case to be more readable - by introducing the setup methods and moving parts of functionality to small, clearly named helper functions. Nonetheless, this process would require a programmer's mental effort since rewritten code would deviate from the specifications in plain human language.

Finally, in a question 8 - where users were asked to manually perform actions from the test specification, one participant (identified as a developer) didn't correctly assign all the requested risk classes, but still estimated his response as correct. Another participant (defined as a test automation engineer) refused to perform the actions but did acknowledge that it will be incorrect.

We skip questions 1 and 6 since there were no incorrect answers for JUnit and only one vague response in question 2.

6.2 Cucumber

For the Cucumber framework, some participants skipped first, fifth, sixth, and seventh questions - we have no way of confirming if it was the same person.

For question 2, two participants were too general in their responses and stated only that the test is for "modification for existing contract" or "changing and saving limit." In the next one, two manual testers observed that the test is flawed and that the data needs to be restored, while the remaining five persons gave a technically correct answer.

An example test case for questions 4 and 5 was split into two parts for Cucumber example. From chart 5.3, we see that this caused difficulties for participants in estimating the correct number of different validations. Only one person gave entirely accurate and expected answers both in questions 4 and 5 - by counting "Then" lines and then copying them to the answer field. Remaining participants estimated from one till seven validations. Question 5 sheds some light on why it happened: two participants who gave a higher answer regarded each test line as a validation of some sort. Their expectation: if a user does some action, it should be implicitly checked that it was actually performed. This expectation differs from JUnit answers, where respondents only expected from assertions to check for something. Those participants that estimated the amount of checks as one or two have counted the amount of Cucumber test scenarios.

For question 7, we have only one incomplete answer where a participant misunderstood the nature of the relation between rows. The remaining five answers are correct, which is the best result for this question among all frameworks.

For the final question, two participants estimated their actions as incorrect and marked that they would need help to understand better what a test case does. Another participant abandoned the survey on this question, while the rest modified term sheet appropriately and were confident in their answers.

6.3 Robot Framework

At first glance at the results of the Robot framework in 5.4, we see that it has the most skipped responses.

Starting from question 1, we have two incorrect answers due to users not understanding how variables in Robot Framework work; in a second question, half of the respondents gave only vague answers as to what the test does. As mentioned before, only one answer addresses the technical flaw of the test case in question three.

By question 4, only five out of nine participants continue answering the survey. One respondent answered as expected for both questions four and five. Out of remaining, we have one response for question 4 says that there are only three checks, which is then explained through the following question: a participant counted three final checks as one. The remaining responses either miss some parts of validation or simply skipped question 5, leaving us no way to analyze the answer's reasoning. For question 7, two participants were still able to give a correct answer, which is better than JUnits' similar question.

The last question of a survey for the Robot Framework (to authors' great shame) revealed a problem with the code - Robot Framework specification did not include a "click <Submit form> button" command despite what the author stated himself in a table 4.1. This mistake invalidates this question; however, we can still try to draw some conclusions from it. One more participant abandoned the survey on this question; three participants haven't done anything, weren't sure about what they did, and marked that they would probably need external help to understand test case more. Finally, two participants did everything correctly (minus the form submission) and were confident in their results. Even if we assume their answers as correct, Robot Framework will still have the lowest score on that question.

6.4 Feedback

After the last questions, we asked participants to rate their satisfaction from working with the chosen framework, whenever or not they would say that it's acceptable to use the framework they worked with inside Swedbank, and if they have any more feedback of any kind.

Responses for their satisfaction are seen in the table 6.1. We can see that average weights were 7, 5.8, and 5.6 for JUnit, Cucumber, and Robot Framework.

Three participants responded that Cucumber would be acceptable, and three - that it would not. For JUnit, there were five responses for "acceptable" and Robot framework - five "acceptable" to one "not acceptable."

Only 7 participants in total gave any feedback for the survey, and we will overview every single response here.

A person who gave 1 for Robot Framework satisfaction appears to be very annoyed at the end of the survey and comments on test specification being "written for the machine" and "not understandable for a person who sees them for the first time." From the background, that person is an experienced developer, and looking at their results, we see correct answers for most questions in the survey. Even though they could understand everything, working with the Robot Framework was too annoying for them.

A person who evaluated 3 for satisfaction for Cucumber commented that a tester would have to understand a system more to know "if a test is good or bad" and if "the tests should be changed or complemented." They complained about implementation and remarked that "a specific language that requires a FAQ to decipher is not ideal." Finally, there was a complaint about the incorrect implementation of the test specifications for the survey. From the other responses, we can see that this person is a manual tester who also answered correctly for most of the questions, and even noticed test inaccuracy in Question 3.

There is an "it would be acceptable [to use Robot Framework] but with some onboarding and [more] practice" response from a manual tester, who rated satisfaction from working with Robot framework at 4.

One test automation engineer/manual tester answered "no" for Cucumber acceptability but clarified that such format would be limiting for exploratory testing while being acceptable for test automation. Given this clarification, we count their "acceptability" answer a vote for "acceptable."

A developer who has to maintain quality of tests commented on a Robot Framework that "it has a learning curve, but is a very human-like language on overall" and rated their satisfaction at 6.

A QA engineer who awarded the Robot framework with an 8 commented that the tests could be improved by inviting everyone on the team to contribute "with their thoughts and feedback." It can uncover more issues in the earlier stages of development.

Finally, a customer with no programming experience graded the JUnit approach at eight and commented on tests being understandable even for him. This person also answered most of the questions correctly, noticed a problem with question 3, and considered a user perspective in question 5.

Table 6.1: Participants satisfaction

Framework	**Evaluation**					
JUnit	6	6	7	8	8	
Cucumber	3	4	5	7	8	8
Robot	1	4	6	7	8	8

In conclusion we can see that the satisfaction of working with the JUnit approach was, on average, rated as the highest among all other frameworks, with Cucumber and Robot frameworks failing behind with a similar level of satisfaction. The Robot framework's lowest score was given by somebody who (apparently) was disappointed with the framework. For Cucumber, the lowest score was from someone criticizing how the survey went instead.

From the point of survey organization, we gained the most insights from questions 3, 4, and 5, where participants were invited to motivate their answers, or the questions were linked to each other. Even without any formal request or demand from our side, several participants went to a great length to explain their answers.

Questions 2 and 7, where we asked participants to re-phrase contents of a test in their own words, provided useful metrics for comparing frameworks; however, it presented challenges for assessment. If an answer contains almost everything except for a line - can we consider it a correct one, or should it still be marked as partially correct? Does it matter if an incomplete answer has more correctly mentioned items? Would we get more reliable metrics if we could replace those questions with a larger quantity of shorter multiple-choice questions?

While the first question was only a warm-up one, the sixth one held no value to us. A user choosing between "yes" or "no" already has a 50% chance of getting a correct answer. Without any follow-up question on reasoning or motivation, we can't draw any conclusions from it.

Finally, we gave users the possibility to edit a "live" demo system for the last question. It gave users more interactivity during the survey but demanded an additional development effort: to store the form after a survey and to validate it against a set of checks. An aspect could be improved here: a term sheet was generated using a random number generator, initialized with a number of a term sheet. That meant that for different users, a term sheet form for the same number would look the same even at a different time. However, the term sheet included java LocalDate fields generated by adding random numbers to a LocalDate.now(). This resulted in no two saved forms being the same and complicated validation logic.

6.5 Summary

In this chapter, we have analyzed the responses for the second survey.

Surprisingly enough, the JUnit framework has shown great overall results in terms of participant satisfaction, the correctness of the answers, and the amount of skipped questions, despite not being created with acceptance tests in mind in the first place. The results related to a second test case could still be improved by refactoring a test case and moving actions to smaller but well-named helper methods. Despite JUnit being written in a Java programming language, colleagues with no programming experience could understand what it is doing.

Cucumber framework has shown close results to JUnit, with some aspects being better and some - worse. Some questions have shown us that users do not differentiate between how "When" and "Then" steps should work, and treat every line of a test specification as a validation of some sort. This expectation can help design shorter and cleaner tests, but it can also be misleading in some cases - for example, if an action "press every red button on a page" does nothing if there are no red buttons on a page, the user could still expect it to fail instead.

The Robot Framework has shown the worst results, and most respondents were skipping their questions. This could be explained by a length of a code: JUnit has 40 lines of code (excluding blank lines, annotations, class, and method declarations); Cucumber has 30 lines of code (excluding empty lines and scenario descriptions); Robot Framework 53 lines of code and each example required providing keywords definitions, increasing the size of code on the page. Robot framework results could also be improved by offering users an Excel file with implementation instead of an HTML page - a more familiar environment could give us better results.

Robot Framework (along with Cucumber) has still shown better results in questions 5 and 7 - the ones where users' understanding of a test case was tested the most. Then again, more JUnit respondents have noticed a flaw of implementation in a first test case for question 3, which was initially overlooked even by the author.

All in all, it seems like there are no significant differences between testing frameworks. Each framework has its ways of writing a test; each framework has some advantages and disadvantages. Re-phrasing survey participants: to improve test suites' quality, we need more collaboration within the team and all involved parties. This collaboration comes prior to the choice of a framework.

Chapter 7

Future work

Practical experiments at a larger scale revealed several technical issues with a survey organization that could be improved.

Logging survey answers to a database instead of working with log files would simplify service creation and validation of results. A standalone tool from 4.6 would be unnecessary, if a service would have restricted administrator panel, allowing organizers to evaluate responses while the survey is still ongoing. We could do a survey for a more quantitative approach, containing only "select an option" variants with automatic validation. This way, a survey could become even more interactive for users.

Another problem with this survey was that it attempted mimicking real-life working conditions by presenting participants with extensive test cases (inspired by examples from authors' practice) and a series of questions, spanning the whole test case. This approach allowed us to test various ideas for question format, like free text inputs, "select one"-questions and interactive forms in demo systems, and see how those types of questions work. However, that format could stress the respondents and cause some to skip questions or abandon the survey altogether. A small number of questions covering complex topics presented challenges for evaluation as well. For the following studies and improved reliability, it would be better to prepare a survey with a larger number of more refined questions of the same format.

A frontend of a survey service could also be improved. The latest survey server access logs revealed that several users were submitting duplicating answers for the same question repeatedly. Users were probably annoyed by server response delays, which we could avoid if there were a better indication that the next question is loading. Another possible fix would be preloading all survey questions and test specifications at the start of a survey on the client-side. That way, we would enable users to navigate between questions without waiting for a server response.

Fixing those issues could improve the survey's accuracy and reliability. We could potentially improve user engagement and get a higher response rate, but there is no way to confirm that for this study. With future research, it would be possible.

There are also several potential research topics for each framework. For JUnit, we could study how well acceptance tests can be decomposed and refactored in a "clean code" way, while still being understood by people with no technical background. For Cucumber framework, we could focus on user expectations while working with DSL and if there is a way to assure that those expectations will be met. For the Robot framework, we could experiment with different test specifications file formats. Could we have improved engagement results by using Excel spreadsheet instead of TSV-file for test specifications, while mowing keyword definitions away from the file?

Chapter 8

Conclusion

The purpose of this study was to evaluate the usability of popular acceptance testing frameworks within one large enterprise organization.

We addressed the challenges of distributed teams and office branches by creating a custom online survey service and demo system, which served us well in this work context. The survey reached colleagues from several cities and countries. On top of the survey responses, we received constructive feedback from multiple participants. No participants were bound by any time restrictions and were free to advance through the question without being distracted from their regular work. We can also safely assume that it would have taken way more effort, approvals, and budget, for an author to travel across several HQs in different countries to interview representatives from every country.

In the scope of this survey, participants were less engaged while working with the Robot Framework than with Cucumber and JUnit frameworks. Cucumber and Robot framework showed the same overall number of incorrect responses - nine, compared to only four in JUnit. However, in several comprehension questions, participants working with Cucumber and Robot Framework were able to give more precise answers than with JUnit. Hence, we may conclude that a framework's choice plays little role in the usability of tests.

This survey has also demonstrated that even people with no prior programming experience can understand clearly structured JUnit based Java test cases. At the same time, experienced developers can work with test specifications from previously unfamiliar "user-friendly" tools even if they personally do not like them.

Summarizing the above, we say that the usability of a mentioned acceptance testing frameworks should not be an issue while choosing a tool to use. Nevertheless, this work has reviewed the current state of acceptance testing frameworks and revealed several potentially important research topics that could be the subject of further, more profound research.

Bibliography

Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall PTR, USA, 1 edition, 2008. ISBN 0132350882. 2, 11, 31

JUnit. Junit - about. `https://junit.org/junit4/`, 2018. URL `https://junit.org/junit4/`. [Online; Accessed 11-June-2019]. 3

Cucumber. Cucumber reference. https://cucumber.io/docs/cucumber/api/, 2019. URL `https://cucumber.io/docs/cucumber/api/`. [Online; Accessed 11-June-2019]. 3

Robot Framework Foundation. Robot framework. `https://robotframework.org/`, June 2019. URL `https://robotframework.org/`. [Online; Accessed 11-June-2019]. 3

Sebastian Möller. *Quality Engineering*. Springer, 2010. Chapter 2. 6

Kent Beck. *JUnit Pocket Guide*. O'Reilly Media, Inc., 2004. Accessed online. 9, 10

Dependency Finder. Junit - api change history, jul 2020. URL `https://depfind.sourceforge.io/Samples/junit.html#4.1`. [Online; Accessed 19-July-2020]. 10

Kent Beck. *Test Driven Development: By Example*. Addison-Wesley Professional, 2002. Chapter 24. 10

Petar Tahchiev, Felipe Leme, Vincent Massol, and Gary Gregory. *JUnit in Action, Second Edition*. Manning Publications Co., USA, 2nd edition, 2010. ISBN 1935182021. 10

Petr Stefan, Vojtech Horky, Lubomir Bulej, and Petr Tuma. Unit testing performance in java projects: Are we there yet? In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, ICPE 17, pages 401–412. Association for Computing Machinery, 2017. ISBN 9781450344043. 11, 13

J. Brunet, D. Serey, and J. Figueiredo. Structural conformance checking with design tests: An evaluation of usability and scalability. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 143–152, 2011. 11

Chris Stevenson. Testdox, 2003. URL `http://agiledox.sourceforge.net/`. 11

Dan North. Introducing bdd, mar 2006. URL `https://dannorth.net/introducing-bdd/`. 11

Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2004. 12

Aslak Hellesoy, Matt Wynne, and Seb Rose. *The Cucumber for Java Book*. Pragmatic Bookshelf, 2015. 12

Ana Oliveira Bertholdo, Dos Santos, and Fabio Kon. Applying usability and user experience goals in agile software development. 01 2011. 12

Pekka Klärck. Data-driven and keyword-driven test automation frameworks. Master's thesis, HELSINKI UNIVERSITY OF TECHNOLOGY, feb 2006. 13

Mark Fewster Dorothy Graham. *Software Test Automation*. Addison-Wesley, 1999. 13

Pekka Klärck. Experience, 2020. URL `http://eliga.fi/experience.html`. 13

PyPi. robotframework pypi, 2008. URL `https://pypi.org/project/robotframework/2.0/#history`. 13

Charles Roscoe Lewis. Using the thinking aloud method in cognitive interface design. 1982. 13

Steven Clarke. Describing and measuring api usability with the cognitive dimensions. 01 2006. 13

Emily Geisen and Jennifer Romano Bergstrom. *Usability Testing for Survey Research*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2017. ISBN 0128036567. 16, 21

Louis M. Rea and Richard A. Parker. *Designing and Conducting Survey Research: A Comprehensive Guide, 4th Edition*. Jossey-Bass, 4th edition, 2014. 17

Appendices

Appendices

A Choice of literature

Author chose several academic research articles and papers to establish a language framework and reference fundamental concepts and methods for this work. Those articles were found by using Google Scholar service or in the university library of TU Berlin.

Furthermore, this work references several peer-reviewed or widely known books from well-trusted publishers. Those books were accessed with the OŔeily Learning platform in an electronic format, which prevents author from sourcing an exact page, and forces citing and referencing by chapter name or number. Those books were used to learn more about specific topics by getting a historical context, or by trusting interpretation of various research papers by book authors.

Finally, some online sources were referenced to provide an example of actual documentation or API changelogs. Those sources serve a mostly illustrative purpose.

B Source Code

JUnitTest.java

Test specifications are provided in normal java test cases, annotated with corresponding Order tag. Global constants are defined in lines 2, 3 and 4. Helper classes and methods were not provided to the survey participants.

```
1  public class TermSheetTest {
2      private static Integer formId = 155;
3      private static final Integer FIRST = 0;
4      private static final Integer SECOND = 1;
5
6      @Test @Order(1)
7      @DisplayName("Test␣Case␣1:␣Modify␣limit␣for␣existing␣contract")
8      public void userCanEdit() {
9          FormPage formPage = new FormPage(formId);
10         formPage.open();
11
12         formPage.editButton().click();
13
14         TermSheetRow firstContract = formPage.rowOfType(RowType.CONTRACT_ROW, FIRST);
15
16         firstContract.editThisRow();
17         firstContract.setChangeOfLimit(1000);
```

```

        formPage.saveChangesButton().click();

        assertThat("Changes␣should␣be␣saved", formPage.changesSaved(), equalTo(true));

        assertThat("Should␣contain␣valid␣change␣of␣limit␣for␣the␣first␣product",
                formPage.rowOfType(RowType.PRODUCT_ROW, FIRST).getChangeOfLimit(), equalTo("1,000.00"));
    }

    @Test @Order(2)
    @DisplayName("Test␣Case␣2:␣Add␣new␣contract")
    public void userCanCreateNewContract() {
        FormPage formPage = new FormPage(formId);
        formPage.open();

        formPage.editButton().click();

        TermSheetRow mainCompany = formPage.rowOfType(RowType.COMPANY_ROW, FIRST);

        TermSheetRow productRow = mainCompany.newChildrenRow();
        productRow.selectProduct("Corporate␣small␣loan");

        TermSheetRow contractRow = productRow.newChildrenRow();
        contractRow.setChangeOfLimit(500);
        contractRow.setContractId("11-222222-AA");

        formPage.saveChangesButton().click();

        assertThat("Must␣contain␣'maturity␣date␣is␣not␣provided'␣error", formPage.errorMessage(),
                containsString("maturity␣date␣is␣not␣provided"));

        contractRow.setMaturityDate("08-2022");
        productRow.setMaturityDate("08-2022");

        formPage.saveChangesButton().click();

        assertThat("Changes␣should␣be␣saved", formPage.changesSaved(), equalTo(true));

        assertThat("Should␣contain␣newly␣changed␣limit",
                formPage.rowOfType(RowType.PRODUCT_ROW, FIRST).getChangeOfLimit(), equalTo("500.00"));

        assertThat("Should␣retain␣change␣of␣limit␣from␣the␣first␣test",
                formPage.rowOfType(RowType.PRODUCT_ROW, SECOND).getChangeOfLimit(), equalTo("1,000.00"));

        TermSheetRow summaryRow = formPage.rowOfType(RowType.SUMMARY_ROW, SECOND);
        assertThat("Should␣have␣change␣of␣limit␣from␣the␣both␣tests", summaryRow.getChangeOfLimit(),
                equalTo("1,500.00"));
    }

    @Test
    @Order(3)
    @DisplayName("Test␣Case␣3:␣Assess␣risk␣for␣all␣companies␣and␣submit␣form")
    public void userCanSubmitForm() {
        FormPage formPage = new FormPage(0);
        formPage.open();

        formPage.submitFormButton().click();

        assertThat("Not␣all␣risks␣are␣assessed␣error", formPage.errorMessage(),
                containsString("You␣must␣assign␣all␣risk␣classes␣for␣product␣rows"));

        formPage.assesRiskButton().click();

        formPage.companyRows().forEach(row -> row.setRiskClass("Low"));

        formPage.saveChangesButton().click();

        formPage.submitFormButton().click();
        assertThat("Changes␣should␣be␣saved", formPage.changesSaved(), equalTo(true));
    }

}
```

CucumberTest.feature

Line 3 uses custom keyword to define a global constant "form page". First test specification is described in lines 6 to 13. Second test specification is split into two scenarios and is described in lines 16 to 32. Final test specification is listed in lines 35 to 43.

For Cucumber tests, Java Gluecode is defined separately and was not presented to the survey participants.

```
1  Feature: Test Term Sheet Form page
2    Background:
3      Given <form page> is 155
4
5    @Test Case 1
6    Scenario: Modify limit for existing contract
7      Given open Term Sheet Form for <form page>
8      And click "Edit␣Term␣Sheets"
9      When edit first contract row
10     And input <1000> in first contract row "Change␣of␣Limit" column
11     And click "Save␣Changes"
12     Then changes should be saved
13     And "Change␣of␣Limit" in first contract row is <1,000.00>
14
15   @Test Case 2
16   Scenario: Add new contract: getting error
```

```
    Given open Term Sheet Form for <form page>
    And   click "Edit␣Term␣Sheets"
    When add <Corporate small loan> product for first company
    And  add <11-222222-AA> contract for first product
    And  set "Change␣of␣Limit" to <500> in first contract
    But click "Save␣Changes"
    Then should get error message containing <maturity date is not provided>

  Scenario: Add new contract: fix error
    When set "Maturity" to <08-2022> in first contract
    And  set "Maturity" to <08-2022> in first product
    And  click "Save␣Changes"
    Then changes should be saved
    And "Change␣of␣Limit" in first product row is <500.00>
    And "Change␣of␣Limit" in second product row is <1,000.00>
    And "Change␣of␣Limit" in summary row is <1,500.00>

  @Test Case 3
  Scenario: Assess risk for all companies and submit form
    Given open Term Sheet Form for <0>
    When click "Submit␣form"
    Then should get error message containing <You must assign all risk classes for product rows>
    When click "Assess␣risk␣classes"
    And set every risk class as <Low>
    And click "Save␣Changes"
    And click "Submit␣form"
    Then changes should be saved
```

RobotTests.robot

Lines 1 to 5 provide global configuration. Line 8 defines a global variable. First test specification is described in lines 12 to 23. Second test specification is split into two test cases and is described in lines 26 to 54. Final test specification is listed in lines 57 to 70. Custom keywords are defined after a line 71.

Only relevant keywords were presented to the survey participant under each test specification. More basic keywords were defined in additional resoruce file helper-methods.robot, those contents were not shown to the participants.

```
1  {*** Settings ***
2  Documentation     Suite for testing term sheet functionality
3  ...
4  Resource          helper-methods.robot
5  Test Teardown     Close Browser
6
7  *** Variables ***
8  ${FORM ID}  155
9
10
```

```
*** Test Cases ***
Edit Existing Contract
    Open Term Sheet Form    ${FORM ID}
    Click   "Edit␣Term␣Sheets"

    ${FIRST CONTRACT} =    Get Contract Row  first
    Edit    ${FIRST CONTRACT}   "Change␣of␣Limit"     1000

    Click   "Save␣Changes"
    Check Changes Were Saved

    ${FIRST PRODUCT} =  Get Product Row     first
    Check Column  ${FIRST PRODUCT}  "Change␣of␣Limit"   "1,000.00"


Add new product and contract and get error
    Open Term Sheet Form    155
    Click   "Edit␣Term␣Sheets"

    ${MAIN COMPANY} =    Get Company Row  first
    ${PRODUCT ROW} =     Create Child Row    ${MAIN COMPANY}
    ${CONTRACT ROW} =    Create Child Row    ${PRODUCT ROW}

    Edit    ${CONTRACT ROW} "Change␣of␣Limit" 500
    Edit    ${CONTRACT ROW}  "-" 11-222222-AA
    Edit    ${PRODUCT ROW}  "-" Corporate small loan

    Click   "Save␣Changes"
    Displays Error  "maturity␣date␣is␣not␣provided"

Add new product and contract and get error
    Edit  ${CONTRACT ROW}   "Maturity"  "08-2022"
    Edit  ${PRODUCT ROW}   "Maturity"  "08-2022"

    Click   "Save␣Changes"
    Check Changes Were Saved

    ${NEW PRODUCT ROW} =     Get Product Row  first
    ${OLD PRODUCT ROW} =     Get Product Row  second
    ${SUMMARY ROW} =         Get Summary Row

    Check Column  ${NEW PRODUCT ROW}  "Change␣of␣Limit"   "500.00"
    Check Column  ${OLD PRODUCT ROW}  "Change␣of␣Limit"   "1,000.00"
    Check Column  ${SUMMARY ROW}      "Change␣of␣Limit"   "1,500.00"


Submit Form
    Open Term Sheet Form    0

    Click   "Submit␣form"
    Displays Error  "You␣must␣assign␣all␣risk␣classes␣for␣product␣rows"

    Click   "Assess␣risk␣classes"

    :FOR    ${COMPANY ROW}  IN  All Company Rows
    \   Set Risk Class  ${COMPANY ROW}  "Low"

    Click   "Save␣Changes"
    Check Changes Were Saved

*** Keywords ***
Check Column
    [Arguments]  ${row} ${column} ${expected value}
    ${actual value} =   Find Column     ${row} ${column}
    Value Equals    ${expected value} ${actual value}

Create Child Row
    [Arguments]  ${row}
    Click In  ${row}  "+"
    [Return]  Next Sibling Row  ${row}

Displays Error
    [Arguments]  ${expectedError}
    ${error} =  Page Error Message
    Should Contain  ${expectedError}    ${error}

Edit
    [Arguments]  ${row} ${column} ${value}
    Click In  ${row}  "+"
    Input In    ${row} ${column} ${value}

Open Term Sheet Form
    [Arguments]  ${FORM ID}
    Open Browser To Page    memo/${FORM ID}

Set Risk Class
    [Arguments]  ${row} ${value}
    ${dropdown} =   Dropdown In     ${row}
    Select Value In     ${dropdown} ${value}
```

C Information for participants

Invitation letter

> Hi colleagues,
>
> I am working on my bachelor thesis, and I would ask for your help by participating in a survey for it.
>
> You can find the survey by following this external link. On that website, you will also find a demo system (which is not related to any real existing banking system and was only built with common terminology you can find on Investopedia). You can try getting familiar with it or start straight away with the survey.
>
> During the survey, you will be asked to analyze test specifications for that system and answer a few related questions. This survey is for everyone, and it is not intended to test programming knowledge. The point is to research whether or not any interested person can understand what the tests are about. Ideally, both business and technical people should understand those and work together if needed. So, if you cannot answer any question within a few minutes at most, or if you do not want to answer any more questions - it is ok to skip them.
>
> Feel free to contact me if there are any problems with the survey. You are also free to forward this survey to anyone who might be interested - the more participants - the better.
>
> Many thanks in advance,

Introduction text

This text was displayed at the beginning of the survey.

> This is a survey for the bachelor thesis of Anton Bardishev. I am conducting this survey to assess and compare the advantages and drawbacks of different approaches to automation of acceptance tests. Acceptance tests in software development are a form of tests aimed at verifying that the tested system is fulfilling specifications. This survey may help us with finding an approach that would allow organizing acceptance tests in a manner that is understandable for everyone and can be easily maintained by developers.

For this survey you will have a a demo system for managing term sheets. A link to this system will be available during the survey as well. This demo system was designed for this survey only, does not affect the real world, and is not connected to anything. You are free to interact with the system as much as you want, and it will not break anything.

The system allows users to view a term sheet form. One term sheet form may have info records for one or several customers. Each customer may have a list of products that they use. Each product may store records for several contracts. Within the form, a user may add new products or contract rows. The user also has to assess risk classes (Low, Medium, High) for each customer. After assigning all risk classes, the user may choose to submit the form. After submission term sheet form will be locked from any future modifications. You may also open FAQ at any time to get more information on demo system.

During the survey, you will be presented with several acceptance tests implemented in one of the researched frameworks. You will get asked to analyze the content of the test and answer the related questions.

Before we start, I also want to tell you that you cannot make a mistake or do anything wrong here. Difficulties you may run into are reflecting the problems of the selected approach for test specification, not your skills or abilities.

By proceeding, you agree with this site using an anonymous cookie to identify you during the survey. The survey itself stores no personal information, and once you complete it, any identifying information will be removed from the dataset. If you disagree with cookies policy or do not want to proceed - you can remove cookies from this website and leave it by following this link