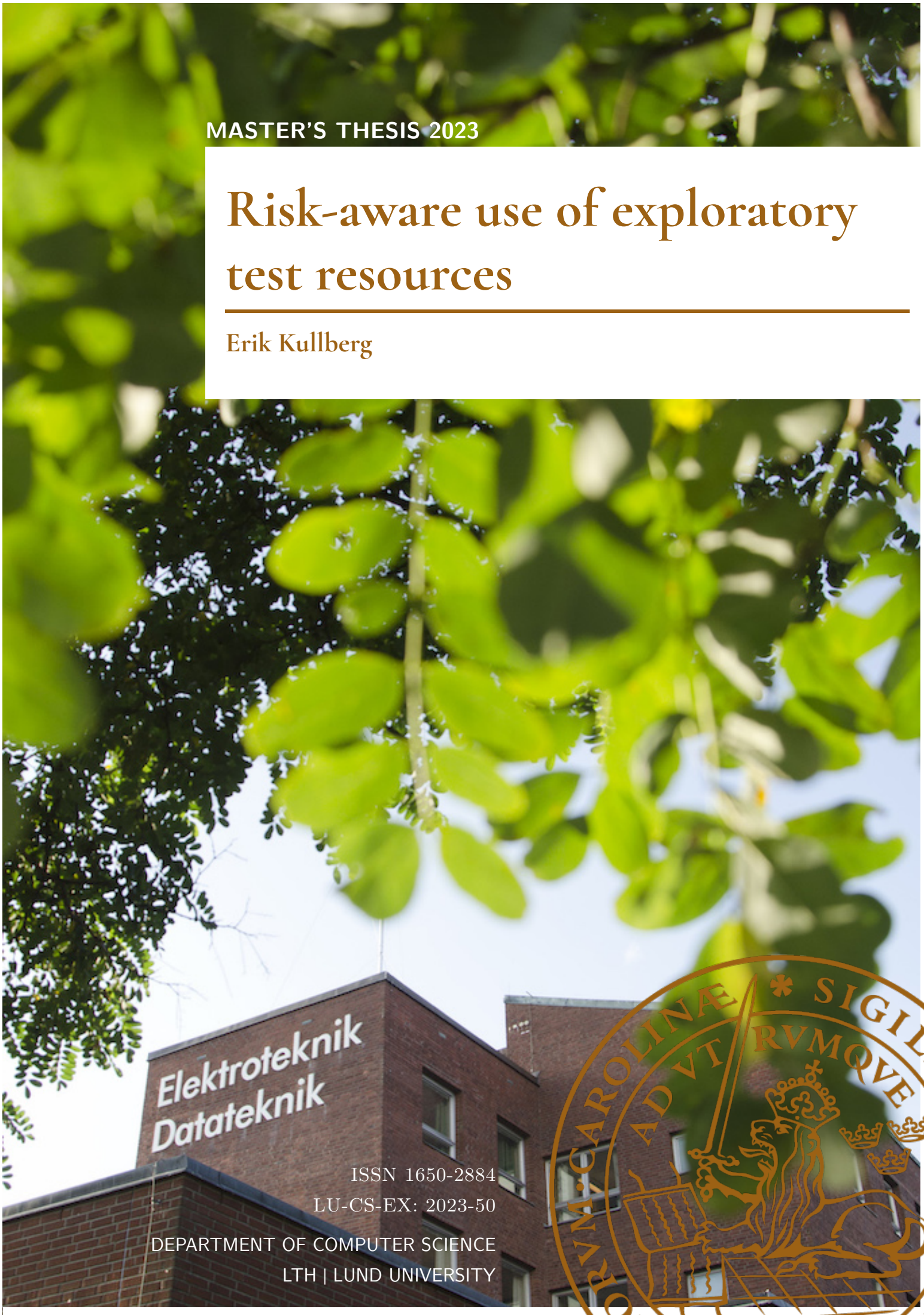


MASTER'S THESIS 2023

Risk-aware use of exploratory test resources

Erik Kullberg

ISSN 1650-2884
LU-CS-EX: 2023-50

DEPARTMENT OF COMPUTER SCIENCE
LTH | LUND UNIVERSITY

EXAMENSARBETE
Datavetenskap

LU-CS-EX: 2023-50

Risk-aware use of exploratory test resources

Riskmedveten användning av utforskande testresurser

Erik Kullberg

Risk-aware use of exploratory test resources

Erik Kullberg
erik.kullberg.2867@student.lu.se

December 7, 2023

Master's thesis work carried out at Qlik Technologies Inc..

Supervisors: Markus Borg, markus.borg@cs.lth.se
José Díaz López, jose.diazlopez@qlik.com

Examiner: Emelie Engström, emelie.engstrom@cs.lth.se

Abstract

Exploratory testing (ET) is widely used for testing of software products. Previous studies show that ET is efficient at finding defects of varying types, severity and difficulty levels. This thesis project studies the use of ET practices and how they can be used in a risk-aware sense. Specifically the use of ET on the software product Qlik Sense Enterprise on Windows, which is a server-based data analysis and visualization software. We measure the code coverage of ET on the software product as well as gather the metrics code quality, code churn and the number of bug fixes for each source code file. This data is used to determine how the files' risk-profiles correlate with the ET practices. We found that the ET does not cover the source code very extensively and that it has no correlation with a file's risk-profile. We also saw that a file's code quality is an indicator of the prevalence of bugs, where a higher code quality yields fewer bugs. Furthermore, the level of churn is another indicator of risk for bugs, where the risk only pertains up to a certain change frequency. With these results, we have corroborated previous work on the subject in regards to code quality and churn, and proposed metrics Qlik can use to further improve their ET practices.

Keywords: exploratory testing, manual testing, code quality, code churn, bug fix distribution, code coverage

Acknowledgements

I would like to thank my supervisors Markus Borg and José Díaz López for their help during this thesis project, Qlik for providing resources and the case I studied, and Magnus Ohlin for helping me get started and providing the initial idea for the project.

Contents

Chapter 1
Introduction

This chapter gives the background of the topic, presents a description of the proposed project as well as related work that have previously been done.

1.1 Background

Software testing relies on a combination of manual and automated approaches. While there are a lot of tools to measure the test coverage of automatic tests, the same can not be said for manual tests [1]. Automatic and manual testing are often done to test different aspects of the software. Automatic testing in the form of unit tests makes sure individual components are working as intended while manual testing often is done as a functional test of the software as a whole from the perspective of a user.

This thesis project focuses on exploratory manual testing, specifically the testing of Qlik Sense Enterprise on Windows performed at Qlik. Exploratory testing (ET) integrate learning, designing tests and execution of the tests [2]. To achieve success in ET the testers must use their creativity and imagination to provide a variety of inputs to the system that is being tested. Due to the nature of ET, in contrast to test case-based testing, the testers are not exercising the system in the same way every time. Despite this, it can still be valuable to evaluate which parts of the system that are tested by a test case, to provide rough guidance to the testers on where to focus their testing resources.

To perform this evaluation one option is to measure code coverage. However, it is often not ideal to rely exclusively on the coverage metric when designing test cases. Setting an arbitrary coverage goal, such as 80% statement coverage, without considering the implications might provide poor guidance to the testers [3]. You can increase coverage through shortcuts in the testing design, e.g. implementing changes to a test case that does not actually improve its efficiency at detecting faults, but still raises the coverage by increasing the amount of code that it executes. Also, reaching 100% code coverage does not guarantee fault-free software [4] [5]. Not even stricter coverage criteria, such as modified decision/decision coverage (MC/DC)

[6] [7].

Coverage measure can also serve the purpose of verifying that certain high-risk areas of the source code are adequately covered by test activities. A powerful predictor of such areas, i.e. areas with the potential of being defect, is code churn [8]. Code churn is a measurement of the amount of changes made to a component over time [9] and most bugs are found in files that have recently undergone substantial change. There is also a correlation between defect density and source code quality [10] [11]. The objective of this thesis is to study how code churn and code quality can aid in prioritizing exploratory test efforts. To summarize, code that has remained unchanged recently demands less exhaustive testing, whereas code with inferior quality necessitates greater attention during testing. We refer to this as risk-aware use of exploratory test resources. Our work will build on academic research on risk-based software testing [12].

1.2 Project Description

Qlik provides a number of business solutions. This thesis project focuses on the development of the data analysis and visualization software Qlik Sense Enterprise on Windows, which is the Windows server version of the software. The manual testing of the software is divided into flows that are done weekly on the latest build as well as every 3 months on official releases. The tests are used to detect regression, i.e. to verify that the latest build still works as intended. The test effort in this activity is a process Qlik is looking to improve.

When looking at test coverage you often want to have it as high as possible, but for a software the size of Qlik Sense it is not feasible to just increase the test coverage without considering if it is necessary on all components of the code. That is why we also will be looking at other metrics such as the number of bug fixes, the source code quality, and the code churn to see on what parts of the code the test coverage needs to be improved.

The goal of the project is to help Qlik investigate how they can take steps toward a risk-aware use of their exploratory test resources. Our approach is to introduce new metrics, which we propose Qlik can use to achieve this goal. This means we want to measure what parts of the source code the exploratory tests currently cover and analyze what parts are the most important to cover. We also aim to analyze the correlation between these new metrics and the exploratory tests' code coverage to look for patterns that could be used to reach a more efficient and effective use of exploratory test resources.

In short, Qlik is looking to achieve a more risk-aware use of their resources for ET, for which we propose they use the new metrics code churn, code quality and bug fix distribution, to determine the code's risk profile. The risk profile can then be used to focus their testing efforts on risk-prone areas of the source code. To study how to come closer to a risk-aware use of exploratory test resources, we seek answers to three research questions:

RQ1 How do current exploratory test activities cover the source code?

RQ2 How does the file-level bug fix distribution correlate with the exploratory code coverage, code churn, and source code quality?

RQ3 How does the source code's risk profile align with the exploratory test coverage?

The goal of **RQ1** is to determine how much and what parts of the source code the current test activities cover, in order to locate areas that are lacking coverage. Deciding if the coverage needs to be increased in areas where it is lacking is then done by analysis of the code churn, code quality and bug fix distribution, which we use to locate risk-prone areas. Due to the nature of ET, the hypothesis for the ET code coverage is that it will be low. But if the current ET practices already extensively cover the source code, including risk-prone areas, the use of ET resources could already be deemed risk-oriented.

With **RQ2** the idea is to use the correlation between the bug fix distribution, code churn and code quality to direct the focus of the ET. If certain metrics are particularly correlated, it can provide guidance for more efficient and effective use of test resources. But if the metrics code churn and code quality currently have no or minimal association with the source code's bug fix distribution, another approach, with other metrics, will have to be selected and evaluated instead. Answering the question will also give us an indication of if the ET already matches the bug fix distribution, which would indicate the effectiveness of ET at finding bugs.

The final research question, **RQ3**, means to go into further detail about the link between the exploratory tests and the code's risk profile, which is defined by the code quality, code churn and bug fix distribution of each source code file. Performing this analysis in further depth can provide an example of how to use the code's risk profile to reach a more risk-aware use of ET resources.

All gathered metrics will remain static as long as the code and the exploratory tests remain unchanged. If they were to change, new data would need to be gathered and the research questions reevaluated. The result of this project is mainly in the context of Qlik Sense at Qlik, but we believe that the use of these metrics to come closer to a risk-aware use of ET resources can be applied outside of the context as well.

The Qlik Sense source code consists mainly of C#, JavaScript and C++. To limit the scope of the project, we will only measure the C# parts of the source code.

1.3 Related Work

This section presents related work on A) exploratory testing, B) test case prioritization, C) code coverage testing, D) code quality metrics, and E) code churn.

Exploratory testing (ET) is testing of software with minimal or no guidance of how the test should be performed, often in the form of black-box testing, i.e. the tester knows little or nothing of the underlying source code. It is widely used in the industry but is not always clearly defined [13]. In this thesis, we use the following definition: "Exploratory testing is testing where the test case is not fully scripted. The test case can have non-descriptive steps to follow, or the tester can be completely free to perform the test case in whatever way they please."

Ghazi et al. performed a study on the levels of exploration in ET, from freestyle testing to completely scripted testing [13]. They determined that different levels of exploration had different benefits, such as defect detection, ease of reproducing defects and ease of learning. But, while ET was efficient at finding otherwise hard-to-find defects, it lacked in verifying conformance to requirements, unlike scripted testing.

From a survey by Pfahl et al., it was found that ET is perceived as a testing practice that supports creativity in the testing phase [2]. The survey also showed that ET is effective and efficient but that it is hard to use and has little tool support. Although the tool support seems to have increased in the years since this survey [14].

Afzal et al. performed four experiments where testers and students performed testing using ET and test case-based testing [15]. The results showed that ET was more efficient at finding defects and also found more defects of varying types, severity and difficulty levels.

Test case prioritization (TCP) is often done when there is a large number of tests that needs to be completed in a relatively short amount of time. For ET, TCP becomes more relevant the lower level of exploration that the test cases have. While most TCP is based on code coverage, Hemmati et al. showed that history-based and diversity-based techniques can be modified and applied on manual black-box test cases in rapid-release environments [1].

Felderer and Schieferdecker performed a taxonomy of risk-based testing to understand, categorize, assess and compare its approaches to support selection for specific purposes [12]. The result was a taxonomy with the top-level classes risk drivers, risk assessment, and risk-based test process. This taxonomy can be used to allocate testing resources making the test phase more risk-aware.

Code coverage is used to determine to what extent test cases are executing the source code and can differ depending on what type of coverage is used, e.g. statement- or branch coverage. It can also be used to direct ET efforts to specific high-risk parts of the code, i.e. functionality that is poorly covered by existing test cases.

Marick describes different aspects of code coverage that are important to be aware of when using code coverage to design test cases and the dangers of blindly relying on the metric [3]. For instance, when creating tests you should not focus on increasing test coverage but instead focus on how to improve the test quality. Marick also states that coverage tools "are only helpful if they are used to enhance thought, not replace it".

Hemmati performed a study on the effectiveness of fault detection of different types of code coverage [4]. They found that a basic criteria such as statement coverage is the least effective at fault detection and reaching 100% code coverage does not guarantee that the software is fault free.

Kochhar et al. showed the effectiveness of using code coverage to develop and improve test suits for testing in large systems and showed that there is a statistical correlation between code coverage and test suits' fault detection effectiveness [5].

The concept of code quality is used to show how maintainable a software solution is and subsequently how costly continued maintenance and evolution would be [10]. Numerous metrics have been proposed for source code quality.

Baggen et al. give a summary of an approach created by the Software Improvement Group for code analysis and quality consulting where the focus is software maintainability [16]. The methodology is based on a standardized measurement model using the ISO/IEC 9126 definition of maintainability and source code metrics, with procedural standardization enhancing result comparability.

Tornhill and Borg performed an analysis on 39 proprietary production codebases using the CodeScene tool to highlight the importance of the code quality concept [10]. They demonstrated that code quality has a considerable impact on the product's time to market and its external quality, which is manifested through software defects.

A systematic review of software maintainability prediction and metrics was performed

by Riaz et al. to determine which maintainability prediction metrics are the most effective [17]. The review showed that the most commonly used maintainability prediction metrics were size, complexity and coupling and that the most effective prediction models were those that used multiple metrics.

Code churn can be described as the number of changes made to a component over a period of time [9] and can be used to predict components that are defect-prone.

Munson and Elbaum designed and conducted an evaluation experiment to compare some process measures to code churn and showed that software complexity metrics along with relative software complexity metric and code churn and code deltas are good measures of code quality [18].

Liu et al. developed a code churn-based unsupervised defect prediction model and investigated its effectiveness against existing supervised and unsupervised defect prediction models under the three prediction settings: cross-validation, time-wise cross-validation and cross-project prediction [8]. They showed that code churn is an important metric for effort-aware just-in-time defect prediction models.

Nagappan and Ball conducted a case study that introduced a method for predicting system defect density at an early stage, which employs a set of relative code churn measures [9]. This showed that using relative code churn measures is a better way of predicting defect density compared to absolute code churn measures.

Code coverage, code quality and code churn can be used to improve exploratory test practices by guiding the tester in creating test cases. Code coverage can be used to show what parts of the source code that have not been tested. Code quality can highlight areas with code of lower quality and code churn can highlight areas with code that are undergoing frequent changes, which both are areas likely to be more defect prone. While using these metrics to guide the creation of test cases, as long as they are not fully scripted, the testing activity still qualifies as ET, although with varying levels of exploration.

1.4 ET at Qlik

The exploratory tests performed at Qlik are called test flows and are divided into two categories, test flows that are done weekly on the latest internal build of the software and test flows that are done every 3 months before the latest public release. The test flows focus on testing different usage scenarios of the software product as well as usage on different platforms such as mobile and tablet versions. Many of the test flows are created to test the scenario of a certain role, e.g. administrator, developer, content creator, content handler and consumer. The test flows are done once in every test iteration and which tester that performs each flow are typically rotated between iterations. The test flows consist of non-descriptive tests divided into sections, often representing different sections of the software. An example excerpt of a test flow can bee seen in figure 1.1.

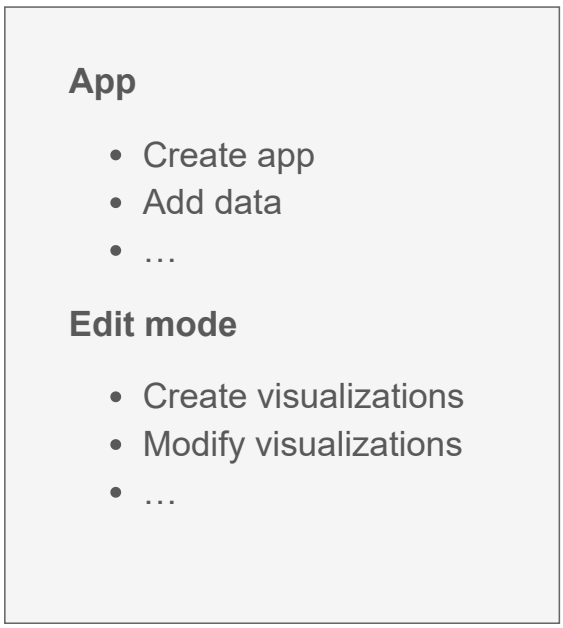


Figure 1.1: Example excerpt of a ET test flow at Qlik.

Chapter 2
Method

The primary approach that we use to address the research questions is case study research, that is, *"An empirical inquiry that investigates a contemporary phenomenon (the "case") in depth and within its real-world context, especially when the boundaries between phenomenon and context are unclear"* [19]. According to the ACM SIGSOFT Empirical Standards [20], this type of software engineering research:

- Presents a detailed account of a specific instance of a *phenomenon* at a *site*. The phenomenon can be virtually anything of interest (e.g. Unix, cohesion metrics, communication issues). The site can be a community, an organization, a team, a person, a process, an internet platform, etc.
- Features direct or indirect observation (e.g. interviews, focus groups, source code analysis).
- Is not an experience report or a series of shallow inquiries at many different sites.

In this study, the context is the development of Qlik Sense at Qlik and the phenomenon under study is ET. Our data collection methods are tool-based source code analysis and coverage measurements from the execution of exploratory test flows. The resulting data are analyzed using descriptive statistics and correlation analysis. With this data we will study what parts of the source code the exploratory tests cover, how they correlate with other code analysis measurement such as code quality, code churn and the bug fix distribution. Furthermore, we will study the correlation between code quality, code churn and bug fix distribution to determine if there is a correlation between these metrics that can be used to improve the efficiency and effectiveness of exploratory tests.

By applying the case study research method to our three research questions, we believe we can determine if our proposed metrics can be used to improve the efficiency and effectiveness of the ET at the case company. Finding what parts of the source code are lacking ET coverage in conjunction with determining the correlation between the bug fix distribution and ET

coverage, code churn and code quality, and analyzing the code's risk profile, the ET resources can be allocated to areas where they would be the most effectively utilized.

2.1 Code coverage of exploratory testing

To answer **RQ1**, i.e. how the exploratory test activities cover the source code, we measure the fraction of line coverage on a file-level that the ET achieves. This is performed by instrumenting the source code with the .NET code coverage tool dotCover (version 2022.3.3) [21] from JetBrains. We then performed the exploratory test flows on the instrumented build and collected coverage information from the sessions. Since ET is generally dependent on the tester, ideally we would measure the coverage from multiple testers performing the same test flows, but with the limited time we had to settle for one test session per test flow.

The result of the ET code coverage measurement in this study will be equivalent to code coverage measurement of any other type of testing method. Only the way we collect the data, the setup we use, will differ from measuring code coverage of other testing methods. Meaning that the theory applied in this study can be applied to all types of testing methods.

Due to the time constraints of this thesis as well as the limitations of the chosen code coverage tool, we limited the study to only look at the parts of the source code written in .NET (C#). Further, due to the size of the software under study, Qlik Sense Enterprise on Windows (QSEoW), as well as time constraints, we only measure the coverage on 3 of the 14 repositories of QSEoW that contain mainly C# code. Combined, the three chosen repositories consist of about 260 000 lines of code.

To measure the code coverage on these three components of QSEoW we need to run them in Visual Studio through the dotCover plugin alongside a full installation of QSEoW. This is required since we need a fully functioning installation of the software in order to perform the ET activities. The three components are Windows services which have the advantage of being able to be run independently from the rest of the software. The installed services can then be replaced with the instrumented versions run through Visual Studio while measuring the code coverage. The three Windows services are Qlik Sense Proxy Service (QPS), which handles site authentication, session handling, and load balancing, Qlik Sense Repository Service (QRS), which contains all data and configuration information for a Qlik Sense site and Qlik Sense Scheduler Service (QSS) which manages the scheduled reloads of apps, as well as other types of reload triggering based on task events [22]. Which parts of the source code for QSEoW to be included in this thesis project was not decided beforehand. Instead, we incrementally added repositories and source code files to the instrumentation in discussions with experts at Qlik until they perceived the volume was sufficient to generate convincing results.

We have included 14, out of 21, test flows that are done on QSEoW, 6 that are done weekly and 8 that are done quarterly. The goal was to include as many of the test flows as possible in this thesis project, only excluding those that test a different product, such as the desktop version, and those we are unable to sufficiently measure code coverage for with our testing environment, such as the test flows that test the installation of the software since we require a fully functional installation of the software to measure the code coverage.

Before starting this thesis project I was employed part-time by Qlik to perform these test flows, giving me experience of how the test flows are performed as well as experience using the Qlik Sense software. With that said, the test flows were done by myself and I was taught

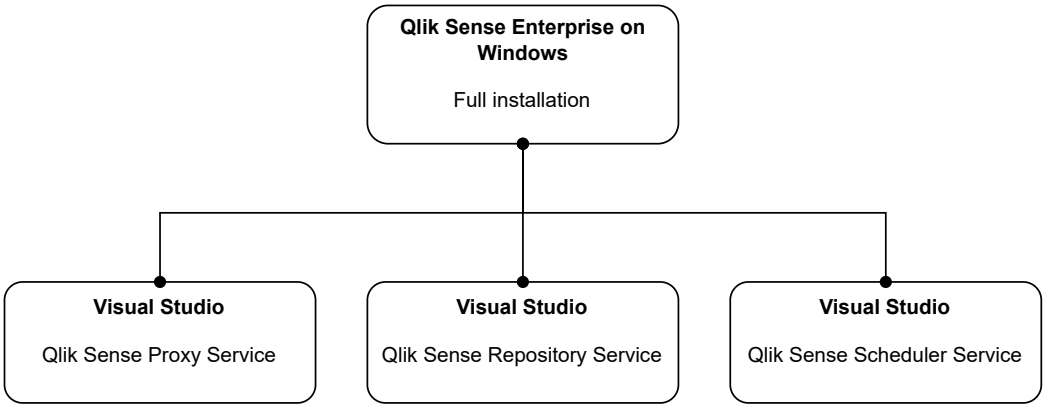


Figure 2.1: The coverage setup consists of a full installation of Qlik Sense Enterprise on Windows with the services QPS, QRS and QSS ran through separate instances of Visual Studio.

to perform them during my employment at Qlik.

2.2 Code coverage tool validation

To ensure that the selected coverage tool performs the measurements as expected, we created a simple validation test. The idea was to create a test with a known outcome, i.e. a test for which we knew exactly what code would be executed. We ended up looking at a network call that was done to the database when a certain button in the UI was pressed. The next step was to record the coverage that occurred when the UI button was pressed, which lets dotCover highlight the code in Visual Studio, green for covered and grey for not covered. Then we located the entry point for the call and placed a break-point in the source code, which allowed us to activate the debug mode and step through each statement after the break-point to verify all the statements executed were marked with green by dotCover. We considered the test successful.

2.3 Exporting code coverage data

The code coverage tool, dotCover, saves a completed coverage measurement as a 'snapshot'. These snapshots can be merged together into one file, which is what we will do with the snapshots from the three repositories for each test flow we measure. The snapshot can then be exported into different file formats, such as Extensible Markup Language (XML), JavaScript Object Notation (JSON) and HyperText Markup Language (HTML). We chose to export to XML, specifically what dotCover calls Detailed XML, since it is compatible with and was proven the easiest format to import into the Qlik Sense software, which is the software we will use to present and visualize and explore our results. See figure 2.2 for a visualization of the process.

To import the XML files with the code coverage data into Qlik Sense we need to write a data load script, see appendix A, which is done in Qlik Sense's Data Load Editor in the coding language QlikScript. When importing we can choose what parts of the XML file we

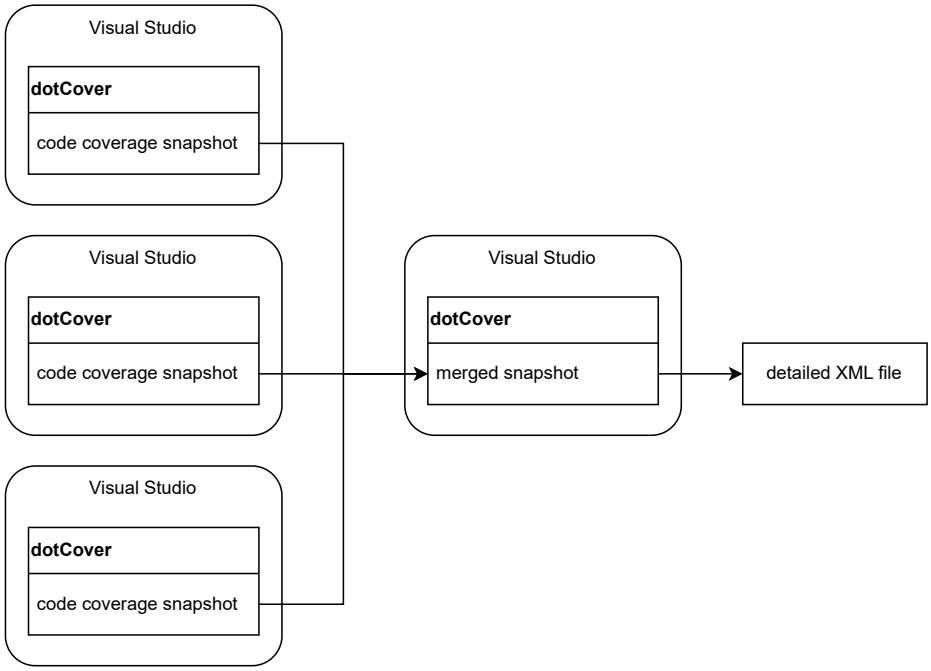


Figure 2.2: A visual representation of the creation, merging and exporting of dotCover snapshots.

want to import, e.g. for this study we are mainly interested in the statement (line) coverage on a file level. We will also use the filename of the XML file to indicate for what test flow coverage data is being imported, which is needed to sort the data by each test flow.

2.4 Bug reports and file-level fixes

When collecting the distribution of bugs on a file-level we want to look at issues posted to Qlik's Jira board for bug reports. However, since we want to know in what file a bug was located we need to look at the pull request's changed files in pull requests that are linked to a Jira issue in the bug reports board.

Qlik provides .qvd files, a Qlik Sense data file, containing different data exported from their Jira boards. Among them is a data file for all pull requests that have been merged in their repositories. With a load script we can import the data into Qlik Sense while also filtering out the specific information we want, i.e. only pull requests for the three repositories we are measuring code coverage on and pull requests that have been merged to the master branch since then the pull request has been reviewed and approved.

Unfortunately, the data file does not contain the pull request's changed files, but we can get that information from the GitHub API, since we have the pull request's repository and pull request number.

The importing of the pull request data file as well as getting the pull request's changed files from the GitHub API was done in the same load script, which can be seen in appendix A. In figure 2.3 you can see a visual presentation of the data collection.

The time window for our collection of reported bugs is the full life span of the source code, starting with the first bug report submitted in 2015 until the most recent report in

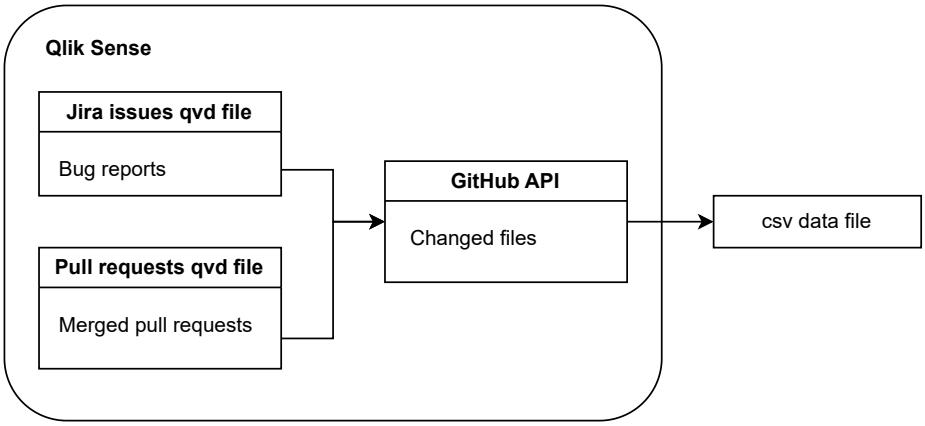


Figure 2.3: A visual representation of the bugs on a file level data collection.

August of 2023. The distribution of bug reports per year can be seen in figure 2.4. The resolutions to the reported bugs in total resulted in X changes to Y files, which we use in our later analysis.

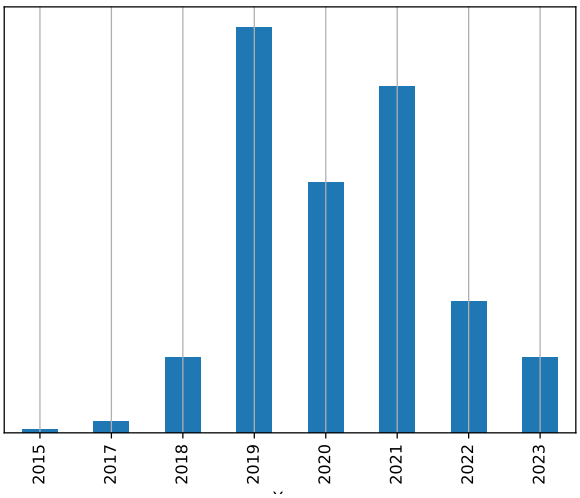


Figure 2.4: Distribution of the number of bug reports per year.

2.5 Source code quality and churn

CodeScene is a tool for code and repository analysis that can be used to visualize source code quality and development activity. It can show several different analysis metrics pertaining to your code, and the two metrics from CodeScene we are interested in are code quality and code churn. Code quality is a measure of how easy the code is to maintain and further develop, which in CodeScene is called code health and defined as a score from 1 to 10, were 10 is code of the highest quality. Code health is an aggregated metric based on 25+ factors scanned from the source code [10] [23]. Code churn is the code's change frequency, which

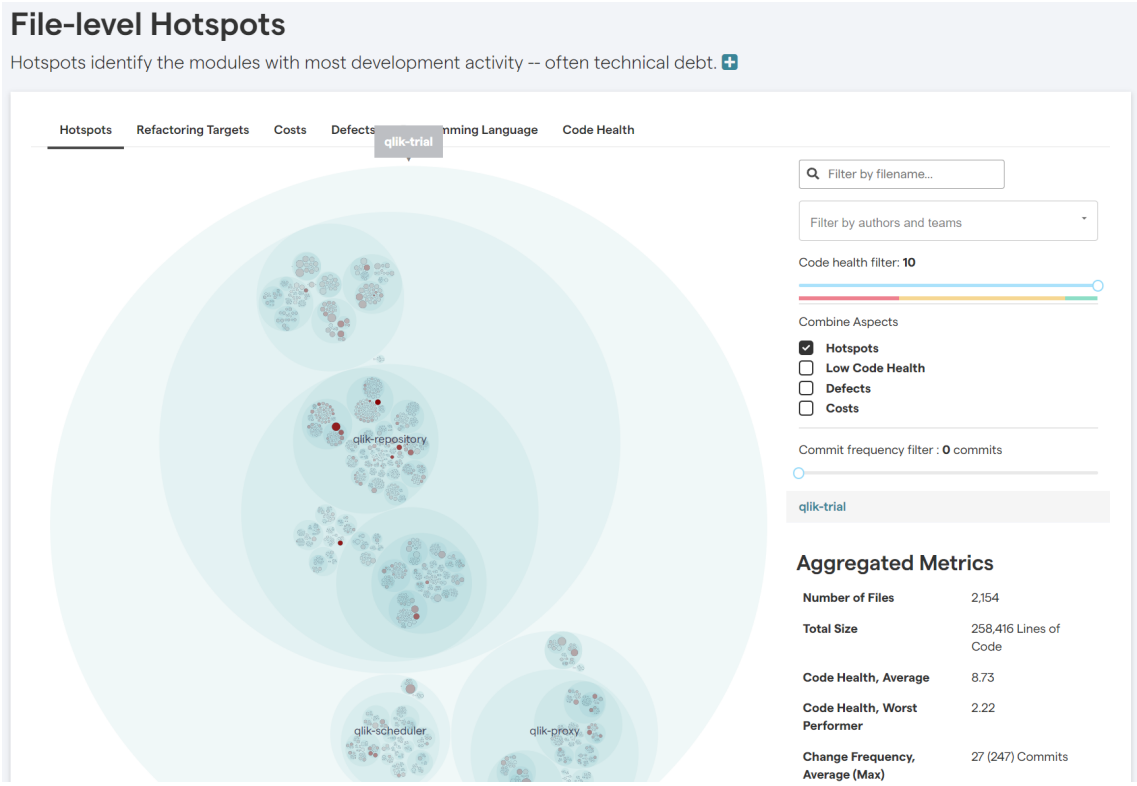


Figure 2.5: CodeScene Hotspots visualization.

in CodeScene is called Hotspots. CodeScene uses two types of churn, absolute and relative churn, which we are both interested in. Absolute churn is the number of times a file has been changed during the specified time period, while relative churn is the churned lines of code (LOC), lines added plus changed, divided by the files total LOC [9]. Each of these metrics are displayed on a file-level and in figure 2.5 you can see an example visualization of Hotspots in CodeScene.

Figure 2.6 shows the distribution of commits per year for the three repositories. The time window for the data collected from CodeScene is 2014-10-22 to 2023-08-25, corresponding to the entire available git history.

Since we want to integrate all data collected in this thesis project in one place for analysis and visualization we export the CodeScene data using a Python script, which can be seen in appendix A, and import it into Qlik Sense Desktop.

2.6 Code coverage aggregation

Since we measure code coverage per line per test flow, we want to aggregate the code coverage into one metric, total coverage. We have measured the coverage on the same code for all test flows, so the total coverage will simply be that if a statement in the code is covered by at least one of the test flows then that statement is considered covered in the total coverage. We wrote a Python script to aggregate the coverage, which can be seen in appendix A.

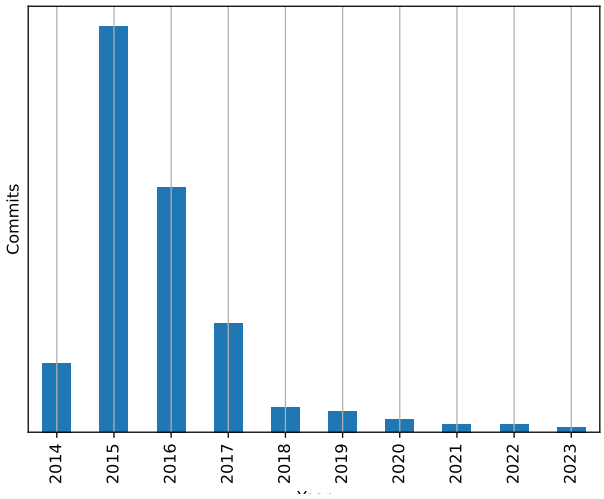


Figure 2.6: Distribution of the number of commits per year.

2.7 Data consolidation and preprocessing

We gathered all the collected data in a master table table in Qlik Sense listing all metrics per source code file. The table was exported to an Excel sheet so that it could be transformed to a Pandas [24] DataFrame for correlation analysis. We also used matplotlib to create graphs showing the distribution of each metric.

For the correlation analysis, we chose to use Spearman's since none of the data is normally distributed. The Python code for the Pandas DataFrame creation, correlation calculation and distribution graphs can be seen in appendix A.

When analyzing the data, we focus on code that changes. Previous work has shown that it is where changes are made that faults have a risk of manifesting [25]. In other words, few bugs are discovered in code that does not change. To explore this, we conduct the correlation analysis using different subsets of the data. We study the entire dataset, and then subsets reflecting files that have been changed at least 5, 10, 20, 40 and 80 times, respectively.

The correlation coefficient can be categorized into low, medium and high intervals. A high value (positive or negative) is between 0.5 and 1 which is said to be a strong correlation, a medium value lies between 0.30 and 0.49 which is a moderate correlation and a low value is below 0.29. A coefficient value of 0 shows no correlation.

The file-level source code's risk profile is based on the metrics code health, churn, and the number of bug fixes that have been committed to the file. We assume that a file exhibiting low code health, high churn or a high number of historical bug fixes has a higher risk of containing bugs. Furthermore, we consider combinations of these concerns as increasingly risky. When performing the analysis with respect to **RQ3** we look to see if there is any correlation between these conditions and the files' total code coverage. We also manually inspect the total code coverage of the 100 most risk-prone files.

Chapter 3
Results

In this chapter, we will report the results by presenting the data collected as well as the analysis performed on that data.

3.1 Data distribution

We gathered all collected data in a master table in preparation for further analysis. The table consists of metrics for 1,729 source code files.

Figure 3.1 shows the distribution of total coverage per file. We find that the distribution is heavily skewed. The majority of the files had 0 to 10 percent code coverage with the rest of the data relatively evenly spread across the 10 to 100 percent range. In total the test flows covered 14.4% of the source code.

Figure 3.2 presents the distribution of code health per file. We can see that the majority have high code health, i.e. code health of 9 to 10. There are few occurrences at code health range 6 to 9, and even fewer in the range 1 to 6. Files with code health below 1 do not contain logic, e.g. interface classes, and were excluded from the analysis. The average code health of the files included in the study is 9.64.

Figure 3.3 displays the distribution of bug fixes per file. The distribution is very skewed with most files having no bug fixes. The average number of bug fixes per file is 0.57.

Figures 3.4 and 3.5 show the distribution of relative and absolute churn per file, respectively. Both relative and absolute churn have high outliers compared to where the majority of the data lies. For the absolute churn, we can see that almost all the data is in the 0 to 50 range with only a single occurrence as high as 250. A similar pattern can be seen in the distribution for relative churn, where most of the data is in the 0 to 250 range and only a single occurrence of relative churn up to 2,500. The average absolute and relative churn per file is 12.78 and 153.24, respectively.

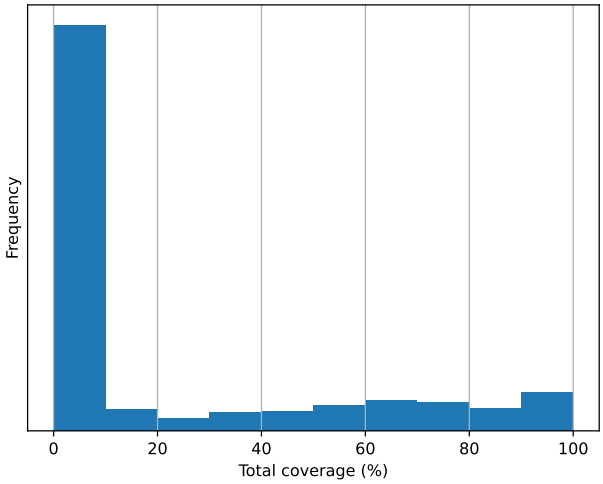


Figure 3.1: The distribution in percent of the amount of total code coverage per file achieved when performing the test flows.

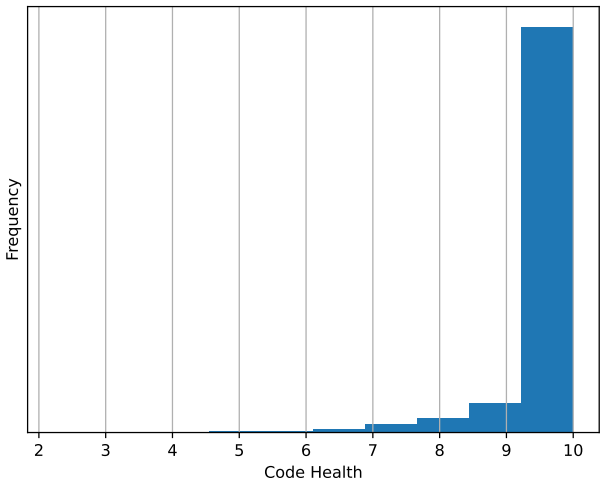


Figure 3.2: The distribution of the code health per file.

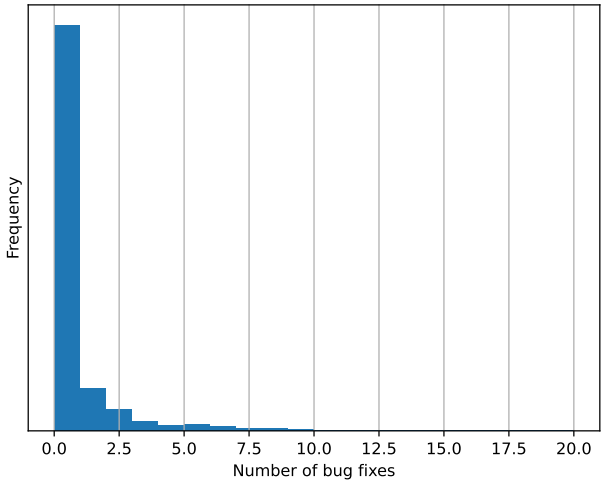


Figure 3.3: The distribution of the number of bug fixes per file recorded during the time period 2015-2023.

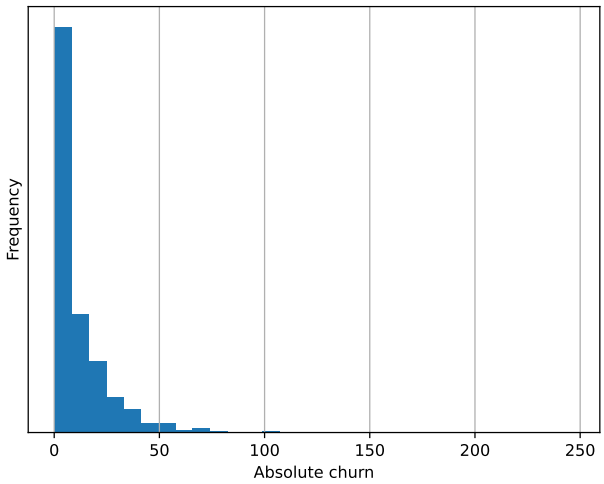


Figure 3.4: The distribution of the absolute churn per file.

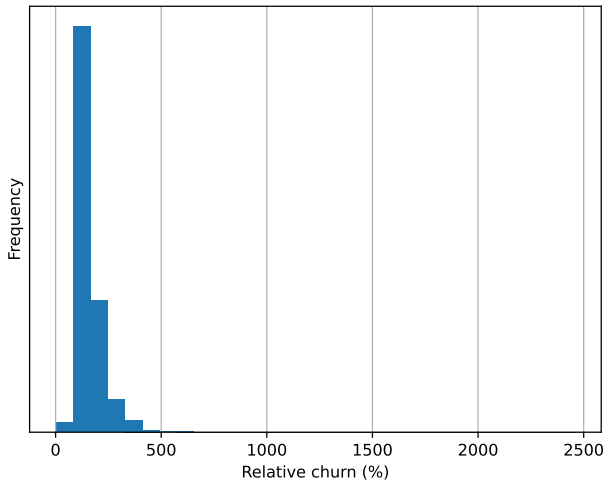


Figure 3.5: The distribution of the relative churn per file.

3.2 Data correlation

In table 3.1, we can see the Spearman correlation between our collected metrics. Firstly, we can see that there is a very low correlation between the total exploratory test coverage and any of the other metrics.

For code health, it has notable correlations with absolute churn, relative churn and the number of bug fixes. Its most notable correlation is with absolute churn which is a negative moderate correlation of -0.466, and its correlation with the number of bug fixes is also moderate at -0.382, while the correlation with relative churn is in the top end of a low correlation at -0.255.

Relative and absolute churn have a strong positive correlation with each other, which makes sense because of their close relationship.

Other than its negative correlation with code health, the number of bug fixes also has a positive correlation with relative and absolute churn, where the increase in absolute churn would increase the risk of more bugs at a higher rate than the increase of relative churn.

Table 3.1: The Spearman correlation matrix for the collected metrics. Values of 0.3 or higher are highlighted in bold.

	Total coverage (%)	Code health	Absolute churn	Relative churn	Nbr bug fixes
Total coverage (%)	**1.000**	-0.040	0.076	0.036	0.082
Code health	-0.040	**1.000**	**-0.466**	-0.255	**-0.382**
Absolute churn	0.076	**-0.466**	**1.000**	**0.787**	**0.382**
Relative churn	0.036	-0.255	**0.787**	**1.000**	0.238
Nbr bug fixes	0.082	**-0.382**	**0.382**	0.238	**1.000**

In table 3.2 and 3.3 we can see the correlation matrix for our metrics after we have filtered out files with a lower absolute churn than 20 and 40, respectively. The largest change we can see in these matrices compared to the correlation for all files is the decrease of the positive

correlation between absolute and relative churn. Another notable change is the decrease of correlation between code health and relative churn, which decreases as far as almost reaching zero. The correlation between code health and absolute churn and the correlation between code health and the number of bug fixes also decreased slightly. Furthermore, we see a large decrease in correlation between the number of bug fixes and relative churn which almost reaches zero for files with absolute churn of 20 or higher. Finally, we see an increase in correlation between total coverage and relative churn and between total coverage and the number of bug fixes, which both reach above 0.2 for files with absolute churn of 40 or higher.

Table 3.2: The Spearman correlation matrix for the collected metrics for files with an absolute churn of 20 or higher. Values of 0.3 or higher are highlighted in bold.

	Total coverage (%)	Code health	Absolute churn	Relative churn	Nbr bug fixes
Total coverage (%)	**1.000**	0.066	0.034	0.082	0.056
Code health	0.066	**1.000**	**-0.363**	0.153	**-0.322**
Absolute churn	0.034	**-0.363**	**1.000**	**0.306**	**0.311**
Relative churn	0.082	0.153	**0.306**	**1.000**	0.069
Nbr bug fixes	0.056	**-0.322**	**0.311**	0.069	**1.000**

Table 3.3: The Spearman correlation matrix for the collected metrics for files with an absolute churn of 40 or higher. Values of 0.3 or higher are highlighted in bold.

	Total coverage (%)	Code health	Absolute churn	Relative churn	Nbr bug fixes
Total coverage (%)	**1.000**	0.132	-0.067	0.206	0.202
Code health	0.132	**1.000**	**-0.371**	0.069	**-0.343**
Absolute churn	-0.067	**-0.371**	**1.000**	**0.412**	0.174
Relative churn	0.206	0.069	**0.412**	**1.000**	-0.063
Nbr bug fixes	0.202	**-0.343**	0.174	-0.063	**1.000**

Table 3.4 shows the correlation between code health and the number of bug fixes when we have filtered out files with lower absolute churn than 5, 10, 20, 40 and 80. The table also shows the number of files left after filtering. We can see that the negative correlation stays the same almost across the board, only increasing slightly for files with absolute churn of 80 or higher.

3.3 Risk-prone files

In table 3.5 and 3.6 we can see our metrics for the 100 most risk-prone source code files. We obtained these files by sorting our master table on code health, the number of bug fixes and absolute and relative churn (in that order). Code health is sorted in ascending order while the other metrics are sorted in descending order. We also added the lines of code (LOC) for each file, which we gathered manually from the repositories on GitHub.

Table 3.4: Correlation between Code Health and the number of bug fixes for different subsets of absolute churn.

Absolute churn	Correlation	Number of files
>=5	-0.351	1081
>=10	-0.322	664
>=20	-0.322	339
>=40	-0.343	101
>=80	-0.436	20

We can see that all the files have varied coverage, ranging from 0% to 90% but none have a coverage of 100%. The lowest code health we have is 2.2, but it increases rapidly when going down the table and only 11 files have a code health of below 6. Furthermore, we see that the file with the lowest code health is also one of the largest in terms of LOC.

Table 3.5: The 100 most risk-prone source code files (0-49).

	Total coverage (%)	Code health	Nbr bug fixes	Absolute churn	Relative churn	LOC
0	54.1	2.2	8	213	224	2135
1	50.0	4.9	1	39	122	1073
2	0.0	4.9	3	33	139	2404
3	0.0	5.2	2	65	193	2826
4	39.7	5.2	0	51	146	810
5	0.0	5.2	3	19	185	1399
6	0.0	5.2	2	23	153	1963
7	0.0	5.4	0	47	176	1862
8	0.0	5.6	5	56	175	1010
9	0.0	5.8	0	31	373	1095
10	0.0	5.8	0	56	154	1608
11	0.0	6.0	0	25	126	1159
12	0.0	6.0	0	79	421	1238
13	0.0	6.2	1	10	166	341
14	78.6	6.2	11	247	389	900
15	29.8	6.3	4	8	140	193
16	0.0	6.3	2	42	140	2390
17	0.0	6.4	3	83	228	1466
18	69.6	6.5	1	35	145	413
19	0.0	6.6	2	17	120	784
20	0.0	6.6	0	45	122	1632
21	62.3	6.6	20	89	242	667
22	0.0	6.7	0	22	116	930
23	0.0	6.7	2	69	184	1667
24	0.0	6.7	2	26	112	931
25	80.3	6.8	2	20	168	340
26	0.0	6.8	0	123	162	969
27	0.0	6.9	18	52	117	2942
28	0.0	6.9	0	27	208	632
29	0.0	6.9	2	25	121	1202
30	0.0	6.9	5	135	224	1113
31	0.0	7.0	12	224	347	705
32	0.0	7.0	2	43	296	728
33	0.0	7.0	15	67	295	1702
34	0.0	7.0	2	4	100	820
35	0.0	7.0	2	17	133	882
36	52.9	7.1	1	134	279	803
37	7.6	7.1	1	103	241	673
38	0.0	7.1	2	78	316	1476
39	0.0	7.2	1	4	112	723
40	0.0	7.2	0	52	359	768
41	36.8	7.3	8	44	167	286
42	60.0	7.3	8	41	158	302
43	0.0	7.3	0	21	135	1026
44	40.5	7.4	3	68	164	430
45	0.0	7.4	1	8	109	385
46	64.3	7.4	0	23	137	224
47	13.2	7.4	0	28	160	534
48	52.6	7.5	16	66	239	352
49	0.0	7.5	0	19	227	700

Table 3.6: The 100 most risk-prone source code files (50-99).

	Total coverage (%)	Code health	Nbr bug fixes	Absolute churn	Relative churn	LOC
50	60.8	7.5	0	27	169	298
51	0.0	7.5	1	7	106	780
52	0.0	7.5	1	15	146	327
53	0.0	7.6	5	8	103	1205
54	70.2	7.6	7	67	190	757
55	0.0	7.6	1	21	120	1747
56	0.0	7.6	0	17	151	341
57	0.0	7.6	2	36	274	563
58	30.3	7.6	0	33	69	482
59	0.0	7.7	1	28	301	503
60	0.0	7.7	2	21	134	442
61	0.0	7.7	0	31	226	862
62	0.0	7.7	0	6	100	588
63	71.8	7.8	2	71	219	541
64	0.0	7.8	7	22	108	249
65	0.0	7.8	2	21	131	659
66	0.0	7.8	0	18	215	346
67	34.1	7.8	5	74	439	314
68	61.0	7.8	11	35	164	248
69	0.0	7.8	7	22	125	1068
70	76.6	7.8	4	73	209	339
71	0.0	7.9	0	51	257	860
72	0.0	7.9	0	18	187	682
73	0.0	7.9	2	13	119	602
74	0.0	7.9	0	17	217	378
75	0.0	7.9	0	17	143	617
76	0.0	7.9	1	44	152	455
77	29.8	7.9	0	9	107	331
78	0.0	7.9	4	34	234	252
79	10.5	8.0	1	31	185	260
80	60.9	8.0	3	5	102	340
81	7.4	8.0	1	4	100	328
82	0.0	8.0	0	22	176	589
83	0.0	8.0	0	14	17	479
84	76.2	8.0	3	15	232	394
85	0.0	8.0	2	17	126	591
86	59.2	8.0	0	25	167	479
87	82.1	8.0	1	18	177	325
88	0.0	8.0	0	17	215	444
89	0.0	8.1	0	15	125	1345
90	44.0	8.1	5	50	200	286
91	52.6	8.1	1	6	133	211
92	0.0	8.1	1	6	133	211
93	0.0	8.2	2	26	179	779
94	0.0	8.2	8	77	226	474
95	67.4	8.2	1	80	314	235
96	15.7	8.2	0	35	255	213
97	0.0	8.2	2	29	127	741
98	90.1	8.2	0	8	137	385
99	0.0	8.2	2	21	138	715

Chapter 4
Discussion

In this chapter we will discuss the results and the main threats to the validity of this study.

4.1 RQ1: Code coverage of exploratory testing

As we can see from the results in figure 3.1, the exploratory test activities do not cover the source code very extensively. A majority of the files have no coverage at all. However, high coverage is seldom the goal of ET, and using ET it is difficult to achieve high coverage on a software product the size of Qlik Sense.

Studies have shown that varying levels of exploration, from freestyle to scripted, in ET have different benefits [13], and it can very well be that the level of exploration can determine the degree of code coverage. When performing scripted tests, e.g. unit tests, the goal is often to cover the code as extensively as possible. Traditional coverage testing is an iterative exercise where you continuously add test cases to increase the test coverage. This is structural testing (whitebox) while ET is functional testing (blackbox).

The test flows we have measured in this thesis project are high-level tests created to test functions in Qlik Sense a normal user will encounter. They are also relatively few, to achieve extensive coverage using ET one possibility could be to create more test flows. Another reason for the low coverage could be dead code, i.e. old code that is no longer used. And while dead code does not sound like a dangerous issue, studies have shown that it is harmful especially during the evolution and maintenance phases [26]. Furthermore, there could be code that is still under development that is hidden behind a feature flag to prevent execution until it is done, which is used instead of branching versions of a software product [27].

With this result, we have found that there is room for improvement regarding the code coverage of the ET practices. Furthermore, the coverage data can be used to identify source code files that are lacking coverage and to focus the ET on risk-prone areas with low coverage,

in order to improve the efficiency and effectiveness of the use of ET resources.

4.2 RQ2: Bug fix correlation

The correlation between the number of bug fixes and ET coverage, code churn and code health, as seen in table 3.1, gives us an idea of the relationship between source code and process metrics and the prevalence of bugs.

Looking at the correlation between the total coverage for the exploratory tests and the number of bug fixes we see a small positive correlation. This means a higher degree of test coverage is correlated with an increased ability to identify bugs. However, the correlation is almost negligible at 0.082. Previous studies have found that ET is perceived as effective and efficient [2], but that is not something we can corroborate with these results.

Our next observation is the negative correlation between code health and the number of bug fixes. This would indicate a prevalence of fewer bugs in files with higher code health. At a value of -0.382 the correlation between these variables is moderate and we believe it is a meaningful insight. This can be further corroborated when we look at the same correlation in table 3.4, which is the correlation after we have filtered out files with lower than 5, 10, 20, 40 and 80 absolute churn. Here we can see that the correlation stays the same across the board with a negative moderate correlation, only increasing slightly for files with an absolute churn of 80 or higher. When removing files with lower churn we only calculate the correlation for files that have undergone substantial change. Previous work has shown that it is where changes are made that bugs have a risk of manifesting [9], which is reflected in our result. The code health metric is an indicator of code quality, and code quality is used to show how maintainable software is and how costly continued maintenance and evolution would be [10]. From our results we can see an indication that ease of maintainability and evolution of the software lowers the risk of introducing bugs.

Since the correlation between the number of bug fixes and code health indicate areas with lower code health are areas with a higher risk of containing bugs, we can use this result to improve the ET by focusing the ET coverage to areas of the code with lower code health. This will improve the efficiency of the ET and reducing the amount of resources required to improve its effectiveness.

Looking at the correlation between the number of bug fixes and absolute and relative churn we see that these correlations differ slightly. With absolute churn, the number of bug fixes has a positive moderate correlation of 0.382. While with relative churn the correlation is a positive low correlation of 0.238. This is logical since, as we mentioned above, prior work has demonstrated that bugs are more likely to appear where changes are implemented [25]. But we can observe further that this correlation decreases if we only look at files that have been changed a certain number of times. In table 3.2 and 3.3 we can see that both the correlation with absolute and relative churn decreases quite significantly, with relative churn decreasing as far as almost reaching zero. For relative churn, this would indicate that the risk of bugs appearing is only correlated with churn up to a certain change frequency, when files change very frequently the pattern disappears. This is less apparent for absolute churn, although its decrease in correlation with the number of bug fixes still indicates a similar pattern. Future research could explore this phenomenon further.

Just like with the correlation between the number of bug fixes and code health we can use

the correlation between the number of bug fixes and both types of churn to focus the ET to improve its coverage of areas with high churn. Although to a lesser extent, due to the lower correlation and the fact that they are only correlated up to a certain change frequency.

4.3 RQ3: Risk profile and exploratory test coverage

The source code's risk profile is given by its code health, churn and bug fix distribution. Looking at table 3.1 we see a very low correlation between the exploratory test coverage and these metrics. After filtering on absolute churn we mostly see a notable change for the correlation between the ET coverage and relative churn and the ET coverage and the number of bug fixes, which for files of absolute churn of 40 or higher in table 3.3, increases to 0.206 and 0.202, respectively. We also see an increase in correlation between ET coverage and code health, which increased to 0.132. While these increases in correlation are quite high, note that the subset of files with absolute churn of 40 or higher is small, with only 101 out of 1,729 files.

When looking at the 100 most risk-prone files in table 3.5 and 3.6 we see that most of the files have no coverage from our ET. While some of the files have coverage, very few of them are very high. For example, of the 100 most risk-prone files only 16 have a coverage of above 60%.

While the risk profile at the moment does not align with the ET coverage it can be used to further improve it, increasing its effectiveness at locating bugs.

For instance, the test flows can be improved by focusing on areas with high risk. This can be done by improving the ET coverage of specific files. Such as file 0 in table 3.5 which has the lowest code health, high churn and moderately high bug fixes while only being covered at a total of 54.1% by the test flows. While 54.1% coverage is relatively high compared to the other risk-prone files, file 0 is also one of the largest files with 2,135 LOC, leaving about 980 LOC uncovered. But increasing ET coverage is not the only way to reduce a file's risk. Since we have seen a moderate correlation between code health and the number of bug fixes, and previous work shows that higher code quality leads to fewer bugs [10], increasing a file's code health could lower the risk of future bugs. Lowering the risk of bugs appearing in a file rather than increasing the chance of catching them would be the ideal solution, although it would probably require more resources.

Another risk-prone file worth focusing on is file 27, which has relatively low code health, moderately high churn and one of the highest number of bug fixes while having a total coverage of 0%. It is also the largest file among the most risk-prone files with 2,942 LOC. Compared to file 0, file 27 has relatively high code health, at 6.9, but it could still benefit from being increased.

Further down the list in table 3.5 we have file 31, which has high churn, moderately high bug fixes and relatively low code health with 0% total coverage. ET is often used for testing before each new release of a software, and when a file is changed as much as file 31 it certainly should be included in that testing phase.

In regards to churn, increasing the ET coverage of risk-prone files with high churn could be beneficial. Compared to code health churn is not something that can be easily improved.

Instead increased coverage could certainly be a solution for raising the chance of catching churn-induced bugs, which is shown by previous work [8].

4.4 Threats to Validity

There are several threats that affect the validity of this thesis project. Such as the results from the collected data regarding code coverage, code churn, code quality and bug fix distribution, the correlations between the data and the conclusions drawn from these results.

4.4.1 Internal Validity

Threats to internal validity refer to whether there exist other factors that could affect the achieved result. In the context of this study, it refers to the validity of the conclusions drawn from the correlation analysis of our collected data as well as the analysis of risk-prone files.

Using the correlation analysis of our data to draw causal conclusions would be invalid. We can only see correlation analysis, not causation.

The correlation we have seen between the bug fix distribution and code health as well as the correlation between the bug fix distribution and code churn could be caused by other factors. But what those factors are would require further study on the subject.

Similarly, the conclusions drawn from the analysis of risk-prone files could also be caused by other factors. Such as the fact that two of the largest files in terms of LOC are among the files deemed most risk-prone, which could have nothing to do with the size of the files.

4.4.2 External Validity

Threats to external validity refer to the generalizability of findings, which in the context of this study is generalizability of the ET code coverage, and use of code churn, code quality and bug fix distribution to reach a risk-aware use of ET resources.

While the amount of source code this thesis project studies is substantial, we have looked at a relatively small subset of the software product. The results may differ if the study was performed on the entirety of the product's source code. Further, this thesis project only looks at one software product at a single company written in one single programming language, and to generalize these results could be incorrect. But we believe that using the metrics code churn, code quality and bug fix distribution in conjunction with the code coverage of ET can help come closer to a risk-aware use of ET resources when applied outside of this thesis project's context. Still, additional studies in other contexts are needed in the future to corroborate this statement.

4.4.3 Construct Validity

Threats to construct validity refer to the suitability of evaluation metrics, which in the context of this study is the suitability of line coverage, code health, absolute and relative churn and the bug fix distribution.

Whether code coverage is a meaningful metric for measuring the performance of manual ET remains an open question. In ET, as the name suggests, you are encouraged to explore the software during execution. If you perform the same test multiple times, the idea is to vary in what way you check each step of the test flow.

To improve the code coverage metric for ET you would need to perform a test case multiple times, while varying each step as much as possible, and then join all covered statements from each execution into one total coverage metric. We also measure the code coverage as percentage covered per file, without considering the LOC of the file. Looking at the total LOC covered by the tests or somehow considering each file's LOC in the coverage metric could show a different result, since the other metrics may depend on the file's LOC.

Previous work has shown that code quality has a considerable impact on a product's time to market and external quality, manifested through bugs in the software [10]. But there are several ways of measuring code quality, and whether CodeScene's code health is a good measure of such could be further investigated.

Similar to assessing code quality, various metrics exist for measuring code churn. It would be beneficial to delve deeper into evaluating the effectiveness of CodeScene's absolute and relative code churn metrics in gauging the change frequency of the source code.

Chapter 5
Conclusion and Future Work

In this chapter we summarize the results of this thesis project into a conclusion and give suggestions for future work to be performed on the subject.

5.1 Conclusion

We have found that the exploratory testing (ET) does not cover the source code very extensively and that there is lacking coverage of risk-prone files. While previous studies have found that ET is perceived as effective and efficient [2], that is not something we can corroborate with our results. We also found that the ET coverage has no correlation with code quality, code churn or the number of historic bug fixes.

We saw a moderate negative correlation between the number of bug fixes and code health, indicating a prevalence of fewer bugs in files with higher code health. The correlation is also relatively unchanged when looking at our absolute churn subsets, reinforcing this result further. Code health, or code quality, is a metric to show maintainability and cost of continued maintenance and evolution [10] and from our result we see an indication that that leads to a lower risk of introducing bugs. For improvement of ET, this can be used to focus the ET coverage to areas of the code with lower code health, since the correlation between the bug fix distribution and code health indicate those are areas with a higher risk of containing bugs.

Furthermore, we saw a positive correlation between the number of bug fixes and our two churn metrics, absolute and relative churn. The correlation is somewhat weaker for relative churn, but still notable. This result suggests the risk of bugs being tied to how much a file is changed, which is in line with previous work [9]. We also observed that this correlation became weaker for our absolute churn subsets, indicating that the risk for bugs is only correlated with churn up to a certain change frequency. Since historic churn is not a metric that can be reduced, only its rate of future increase, increasing the ET coverage for files with high churn could be beneficial. Just like the previous correlation, the correlation between the number of bug fixes and both types of churn can be used to focus the ET, increasing coverage

in areas with higher churn, since this correlation would indicate that those are areas at higher risk of containing bugs.

The source code's risk profile is given by its code health, churn and bug fix distribution. From the correlation analysis we saw that the correlation between the risk profile and the ET coverage is very low. We also found that the ET coverage of the 100 most risk-prone files is low, where only 16 files had a coverage of above 60%. Looking further, we saw that the worst performer was also one of the largest files in terms of LOC, and with 54% coverage it has a substantial number of uncovered LOC.

Improving the ET coverage for it to align with the source code's risk profile could increase the chance of catching bugs in these high-risk areas. But since we have seen a moderate correlation between code health and the number of bug fixes – and previous work shows that higher code quality leads to fewer bugs [10] – one could also improve the risk profile by for instance increasing the files' code health. Thus decreasing the chance of bugs appearing in the first place.

With this thesis project, we have corroborated previous work in regards to the correlation between code quality, code churn and bug fix distribution and provided Qlik new metrics to further improve their ET practices.

5.2 Future work

This thesis project is an example of case study research on a subset of QSEoW at Qlik. To achieve a generalized result on this subject more studies need to be performed at different companies and different software products.

As discussed under threats to construct validity (see Section 4.4.3) we have not considered a file's LOC when measuring its code coverage and other metrics. This could bias our results as LOC might be a strong confounding factor. Previous work reports that LOC should be controlled when analyzing object-oriented languages [28]. Since we analyze C# code, future work should investigate how a file's LOC influences the correlation between code coverage, code health and churn.

Another future project could implement improvements of the ET coverage with these results as guidance. And furthermore, analyze if risk-aware use of ET could contribute to making Qlik's testing process more efficient and effective.

References

[1] H. Hemmati, Z. Fang, M. V. Mäntylä, and B. Adams, "Prioritizing manual test cases in rapid release environments," *Software Testing, Verification and Reliability*, vol. 27, no. 6, p. e1609, 2017. _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/stvr.1609.

[2] D. Pfahl, H. Yin, M. V. Mäntylä, and J. Münch, "How is exploratory testing used? A state-of-the-practice survey," in *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '14, (New York, NY, USA), pp. 1–10, Association for Computing Machinery, Sept. 2014.

[3] B. Marick, "How to Misuse Code Coverage," 1997.

[4] H. Hemmati, "How Effective Are Code Coverage Criteria?," in *2015 IEEE International Conference on Software Quality, Reliability and Security*, pp. 151–156, Aug. 2015.

[5] P. S. Kochhar, F. Thung, and D. Lo, "Code coverage and test suite effectiveness: Empirical study with real bugs in large systems," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pp. 560–564, Mar. 2015. ISSN: 1534-5351.

[6] S. Kandl and S. Chandrashekar, "Reasonability of MC/DC for safety-relevant software implemented in programming languages with short-circuit evaluation," *Computing*, vol. 97, pp. 261–279, Mar. 2015.

[7] G. Gay, M. Staats, M. Whalen, and M. P. E. Heimdahl, "The Risks of Coverage-Directed Test Case Generation," *IEEE Transactions on Software Engineering*, vol. 41, pp. 803–819, Aug. 2015. Conference Name: IEEE Transactions on Software Engineering.

[8] J. Liu, Y. Zhou, Y. Yang, H. Lu, and B. Xu, "Code Churn: A Neglected Metric in Effort-Aware Just-in-Time Defect Prediction," in *2017 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pp. 11–19, Nov. 2017.

[9] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, pp. 284–292, May 2005. ISSN: 1558-1225.

[10] A. Tornhill and M. Borg, "Code Red: The Business Impact of Code Quality – A Quantitative Study of 39 Proprietary Production Codebases," Mar. 2022. http://arxiv.org/abs/2203.04374.

[11] A. Okutan and O. T. Yıldız, "Software defect prediction using Bayesian networks," *Empirical Software Engineering*, vol. 19, pp. 154–181, Feb. 2014.

[12] M. Felderer and I. Schieferdecker, "A taxonomy of risk-based testing," *International Journal on Software Tools for Technology Transfer*, vol. 16, pp. 559–568, Oct. 2014.

[13] A. N. Ghazi, K. Petersen, E. Bjarnason, and P. Runeson, "Levels of Exploration in Exploratory Testing: From Freestyle to Fully Scripted," *IEEE Access*, vol. 6, pp. 26416–26423, 2018. Conference Name: IEEE Access.

[14] "17 Best Exploratory Testing Tools [2023 Ranking]," Apr. 2023. https://www.softwaretestinghelp.com/tools/top-17-exploratory-testing-tools/.

[15] W. Afzal, A. N. Ghazi, J. Itkonen, R. Torkar, A. Andrews, and K. Bhatti, "An experiment on the effectiveness and efficiency of exploratory testing," *Empirical Software Engineering*, vol. 20, pp. 844–878, June 2015.

[16] R. Baggen, J. P. Correia, K. Schill, and J. Visser, "Standardized code quality benchmarking for improving software maintainability," *Software Quality Journal*, vol. 20, pp. 1–21, June 2011.

[17] M. Riaz, E. Mendes, and E. Tempero, "A systematic review of software maintainability prediction and metrics," in *2009 3rd International Symposium on Empirical Software Engineering and Measurement*, (Lake Buena Vista, FL, USA), pp. 367–377, IEEE, Oct. 2009.

[18] J. Munson and S. Elbaum, "Code churn: a measure for estimating the impact of code change," in *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, (Bethesda, MD, USA), pp. 24–31, IEEE Comput. Soc, 1998.

[19] R. K. Yin, *Case Study Research and Applications: Design and Methods*. Sage publications, 2017.

[20] "Empirical Standards," May 2023. https://github.com/acmsigsoft/EmpiricalStandards.

[21] "dotCover: A Code Coverage Tool for .NET by JetBrains." https://www.jetbrains.com/dotcover/.

[22] "Services | Qlik Sense for administrators Help." https://help.qlik.com/en-US/sense-admin/May2023/Subsystems/DeployAdministerQSE/Content/Sense_DeployAdminister/QSEoW/Deploy_QSEoW/Services.htm.

[23] "Code Health - How Easy is Your Code to Maintain and Evolve." https://codescene.io/docs/guides/technical/code-health.html.

[24] "pandas - Python Data Analysis Library." https://pandas.pydata.org/.

[25] N. Nagappan and T. Ball, "Using Software Dependencies and Churn Metrics to Predict Field Failures: An Empirical Case Study," in *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, pp. 364–373, Sept. 2007. ISSN: 1949-3789.

[26] S. Romano, C. Vendome, G. Scanniello, and D. Poshyvanyk, "A Multi-Study Investigation into Dead Code," *IEEE Transactions on Software Engineering*, vol. 46, pp. 71–99, Jan. 2020. Conference Name: IEEE Transactions on Software Engineering.

[27] M. K. Ramanathan, L. Clapp, R. Barik, and M. Sridharan, "Piranha: reducing feature flag debt at uber," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: Software Engineering in Practice*, ICSE-SEIP '20, (New York, NY, USA), pp. 221–230, Association for Computing Machinery, Sept. 2020.

[28] Y. Zhou, H. Leung, and B. Xu, "Examining the Potentially Confounding Effect of Class Size on the Associations between Object-Oriented Metrics and Change-Proneness," *IEEE Transactions on Software Engineering*, vol. 35, pp. 607–623, Sept. 2009. Conference Name: IEEE Transactions on Software Engineering.

Appendices

Appendix A
Code listings

A.1 Qlik Sense load scripts

Below are the load scripts written in QlikScript used for importing data into Qlik Sense.

Load script for importing dotCover snapshots exported to XML as well as the aggregated total coverage.

```
let flow_index = 0;

for each file in FileList('lib://Testflows merged\*.xml')

    flow_index = flow_index + 1;

    TestFlows:
    Load
      '$(flow_index)' as TestFlowIndex,
      SubField(SubField('$(file)','/',-1), '-', 1) as TestFlow // file naming convention: TESTFLOW-Merged.xml
    FROM [$(file)];

    CoverageData:
    LOAD
      $(flow_index) & '-' & "Index" as FileIndex,
      SubField(Name, '\', 7) as Repository,
      Trim(Replace(SubField(Name, SubField(Name, '\', 6)&'\', -1), '\', '/')) as CoverageData.FileName, // removing unnecessary folder path by splitting before Repository, and changing all '\' to '/'
      lower(Trim(Replace(SubField(Name, SubField(Name, '\', 6)&'\', -1), '\', '/'))) as FileNameLowercase, // to avoid case sensitivity
      $(flow_index) as TestFlowIndex
    FROM [$(file)]
    (XmlSimple, table is [Root/FileIndices/File]);
```

```
22
23   Statements:
24     LOAD
25       $(flow_index) & '-' & FileIndex as FileIndex,
26       "Line" as Statement.Line,
27       Covered as Statement.Covered
28       FROM [$(file)]
29     (XmlSimple, table is [Root/Assembly/Namespace/Type/Method/
    Statement]);
30
31     LOAD
32       $(flow_index) & '-' & FileIndex as FileIndex,
33       "Line" as Statement.Line,
34       Covered as Statement.Covered
35       FROM [$(file)]
36     (XmlSimple, table is [Root/Assembly/Namespace/Type/Type/Method/
    Statement]);
37
38     LOAD
39       $(flow_index) & '-' & FileIndex as FileIndex,
40       "Line" as Statement.Line,
41       Covered as Statement.Covered
42       FROM [$(file)]
43     (XmlSimple, table is [Root/Assembly/Namespace/Type/Type/Type/
    Method/Statement]);
44
45 next file
46
47 TotalCoverage:
48 LOAD
49     FileNameLowercase,
50     Statement.Line as line,
51     Statement.Covered as totalcovered
52 FROM [lib://Testflows merged/totalcoverage.csv]
53 (txt, codepage is 28591, embedded labels, delimiter is ',', msq);
```

Load script for importing pull request data including pull request's changed files from the GitHub API.

```
1  JiraIssues:
2  LOAD
3    %ID,
4    SUMMARY,
5    issuenum,
6    PrettyNumID,
7    Resolution,
8    Status,
9    "type",
10   CREATED,
11   RESOLUTIONDATE
12 FROM [lib://JiraData/Issue_Now.qvd]
13 (qvd)
14   Where "type" = 'Bug'
15     and (Resolution = 'Fixed' or Resolution = 'Done');
16
17 PullRequests:
18 LOAD
```

```
  %Pull_Request_ID,
  Pull_Request_Name,
  Pull_Request_Destination_Branch,
  Pull_Request_Source_Branch,
  Pull_Request_Source_Repo,
  Pull_Request_Last_Status,
  Pull_Request_URL,
  SubField(Pull_Request_URL, '/', -1) as PR_Number,
  Pull_Request_Executed_By,
  Pull_Request_Author,
  Pull_Request_Updated_On,
  PR_TO_ISSUE_ID,
  PrettyNumID
FROM [lib://JiraData/Jira_Pull_Requests.qvd]
(qvd)
  Where Exists(PrettyNumID, PrettyNumID)
  and Match(Pull_Request_Source_Repo, 'qlik-trial/qlik-proxy', '
   qlik-trial/qlik-scheduler', 'qlik-trial/qlik-repository')
  and Pull_Request_Last_Status = 'MERGED';

//The PullRequestFiles response includes a maximum of 300 (3000?)
   files (API limit).

LIB CONNECT TO 'GitHub - ktm-erikkullberg';

for each repo in FieldValueList('Pull_Request_Source_Repo')

    for each url in FieldValueList('Pull_Request_URL')

      prnumber = SubField(url, '/', -1); //prnumber is at the end
   of the url (ex. https://github.com/qlik-trial/qlik-proxy/pull/7)

        if SubField(repo, '/', -1) = SubField(url, '/', -3) then //
   checks that current repo is the same as repo in current url

          ChangedFiles:
          LOAD
            '$(repo)' as [PullRequestsFiles.Source_Repo],
            '$(prnumber)' as PR_Number,
            Filename as [PullRequestsFiles.Filename],
            Status as [PullRequestsFiles.Status],
            File_Addition as [PullRequestsFiles.File_Addition],
            File_Deletion as [PullRequestsFiles.File_Deletion],
            Total_Changes as [PullRequestsFiles.Total_Changes]

            Where SubField(Filename, '.', -1) = 'cs' //load only .
   cs files
            and Status = 'modified'; //only modified files, i.e.
   bug-fixes(?)

              SELECT
                Filename,
                Status,
                File_Addition,
                File_Deletion,
                Total_Changes
```

```
                FROM PullRequestsFiles
                WITH PROPERTIES (
                repoName='$(repo)',
                PRNumber='$(prnumber)'
                );

    end if

    next url

    Commits:
    LOAD DateTime as [Commits.DateTime];

    SELECT DateTime
      FROM Commits
      WITH PROPERTIES (
      repoName='$(repo)',
      branchSha='master',
      sinceDate='01-01-2013'
    );

next repo

Store ChangedFiles into 'lib://Documents - OD/PullRequests/
   PullRequestsChangedFiles.qvd';
```

A.2 CodeScene data parser

Python script for parsing the data from the CodeScene analysis and writing it to a .csv file which can be imported into Qlik Sense.

```
# Script for parsing the systemmap.json file found in the CodeScene
    analysis folder.
# >python codescene-data-parser.py "input-file" "output-file"

import json
import sys

def parser(data_tree):
    if "children" in data_tree:
        for i in range(len(data_tree["children"])):
            parser(data_tree["children"][i])
    elif data_tree["path"][-1].split(".")[-1] == "cs":
        path = '/'.join(data_tree["path"])
        file_writer([path, data_tree["churn"], data_tree["high-
   resolution-score"], data_tree["revs"]])

def file_writer(data_values):
    f = open(sys.argv[2], "a")
    f.write(','.join(list(map(str,data_values)))+"\n")
    f.close()

data_file = open(sys.argv[1], "r")

```

```
json_string = data_file.read()
json_dict = json.loads(json_string)

f = open(sys.argv[2], "w")
f.write("filename,churn,high-resolution-score,revs\n")
f.close()

parser(json_dict)
```

A.3 Code coverage aggregation

Python script using pandas [24], a data analysis and manipulation tool, and the `reduce()` function from the functools library.

```
import pandas as pd
from functools import reduce

def totalcovered(seq):
    return reduce(lambda x, y: x if x=='True' else y, seq)

df = pd.read_excel(io="file-coverage.xlsx", sheet_name="Sheet1")
df_merged = df.groupby(['FileNameLowercase','Statement.Line']).agg
    ({'Statement.Covered': totalcovered}).reset_index()
df_merged.to_csv(path_or_buf='totalcoverage.csv')
```

A.4 Correlation calculation and 100 most risk-prone files

Python code for correlation calculation and extraction of the 100 most risk-prone files using pandas [24]. Originally written in a Jupyter Notebook.

```
import pandas as pd

df = pd.read_excel('mastertable.xlsx', na_values='-')
df = df.drop(df[df['FileNameLowercase'].str.contains('generated')].
    index)
df = df.where(df['high-resolution-score'] != 0).dropna(how='all')
df = df.dropna(subset='high-resolution-score')
df = df.set_index('FileNameLowercase')
df['nbr_fixes'].fillna(0, inplace=True)

df5 = df.where(df['revs'] >= 5).dropna(how='all')
df10 = df.where(df['revs'] >= 10).dropna(how='all')
df20 = df.where(df['revs'] >= 20).dropna(how='all')
df40 = df.where(df['revs'] >= 40).dropna(how='all')
df80 = df.where(df['revs'] >= 80).dropna(how='all')

corr = df.corr(method='spearman')
corr5 = df5.corr(method='spearman')
corr10 = df10.corr(method='spearman')
```

```
corr20 = df20.corr(method='spearman')
corr40 = df40.corr(method='spearman')
corr80 = df80.corr(method='spearman')

corr_chealth_nbrfix = pd.DataFrame(data={'\\textbf{{Absolute churn}}':['>=5','>=10','>=20','>=40','>=80'],
                            '\\textbf{{Correlation}}':[
                                corr5['high-resolution-score']['nbr_fixes'],
                                corr10['high-resolution-score']['nbr_fixes'],
                                corr20['high-resolution-score']['nbr_fixes'],
                                corr40['high-resolution-score']['nbr_fixes'],
                                corr80['high-resolution-score']['nbr_fixes']],
                             '\\textbf{{Number of files}}':[df5.shape[0],df10.shape[0],df20.shape[0],df40.shape[0],df80.shape[0]]})

chealth_nbrfix_latex = corr_chealth_nbrfix.to_latex(float_format='%.3f',
                                                    caption='Correlation between Code Health and the number of bug fixes.',
                                                    label='tab:chealth_nbrfix', decimal='.', index=False, position='h')
f = open('corr_chealth_nbrfix.tex', 'w')
f.write(chealth_nbrfix_latex)
f.close()

headers = ['Total coverage (\\%)', 'Code health', 'Relative churn', 'Absolute churn', 'Nbr bug fixes']
headers = list(map(lambda str: '\\textbf{{' + str + '}}', headers))
d = {list(corr.columns)[i]: headers[i] for i in range(len(headers))}
corr.rename(index=d, columns=d, inplace=True)
corr_latex = corr.to_latex(float_format='%.3f', caption='The Spearman correlation matrix for the collected metrics.',
                           label='tab:corr_table', decimal='.', position='h')
f = open('corr_matrix.tex', 'w')
f.write(corr_latex)
f.close()

d = {list(corr20.columns)[i]: headers[i] for i in range(len(headers))}
corr20.rename(index=d, columns=d, inplace=True)
corr20_latex = corr20.to_latex(float_format='%.3f', caption='The Spearman correlation matrix for the collected metrics for files with an absolute churn of 20 or higher.',
                               label='tab:corr20_table', decimal='.', position='h')
f = open('corr20_matrix.tex', 'w')
f.write(corr20_latex)
f.close()
```

```
56
57 d = {list(corr40.columns)[i]: headers[i] for i in range(len(headers
       ))}
58 corr40.rename(index=d, columns=d, inplace=True)
59 corr40_latex = corr40.to_latex(float_format='%.3f', caption='The
       Spearman correlation matrix for the collected metrics for files
       with an absolute churn of 40 or higher.',
60                                 label='tab:corr40_table', decimal='.
       ', position='h')
61 f = open('corr40_matrix.tex', 'w')
62 f.write(corr40_latex)
63 f.close()
64
65 def f0(x):
66     return '%.0f' % x
67
68 def f1(x):
69     return '%.1f' % x
70
71 df_sorted = df.iloc[:,[0,1,4,2,3]]
72 df_sorted = df_sorted.sort_values(by=['high-resolution-score','
       nbr_fixes','revs','churn'],ascending=[True,False,False,False])
73 df_sorted = df_sorted.head(100)
74
75 d = {list(df_sorted.columns)[i]: headers[i] for i in range(len(
       headers))}
76 df_sorted.rename(columns=d,inplace=True)
77 df_sorted_1 = df_sorted.head(50)
78 df_sorted_2 = df_sorted.tail(50)
79 df_sorted_1_latex = df_sorted_1.to_latex(formatters=[f1,f1,f0,f0,f0
       ,f0], caption='The 100 most risk-prone source code files (0-49).
       ',
80                                          label='tab:100_files_1', decimal='.', position='h
       ')
81 df_sorted_2_latex = df_sorted_2.to_latex(formatters=[f1,f1,f0,f0,f0
       ,f0], caption='The 100 most risk-prone source code files (50-99)
       .',
82                                          label='tab:100_files_2', decimal='.', position='h
       ')
83 f = open('100_files_1.tex', 'w')
84 f.write(df_sorted_1_latex)
85 f.close()
86 f = open('100_files_2.tex', 'w')
87 f.write(df_sorted_2_latex)
88 f.close()
89 df_sorted_1_latex
```

INSTITUTIONEN FÖR DATAVETENSKAP | LUNDS TEKNISKA HÖGSKOLA | PRESENTERAD 2023-10-26

EXAMENSARBETE Risk-aware use of exploratory test resources
STUDENT Erik Kullberg
HANDLEDARE Markus Borg (LTH), José Díaz López (Qlik)
EXAMINATOR Emelie Engström (LTH)

Riskmedveten användning av utforskande testresurser

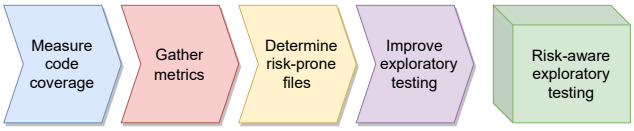
POPULÄRVETENSKAPLIG SAMMANFATTNING **Erik Kullberg**

Utforskande testning (UT) används i stor utsträckning för testning av mjukvara. Tidigare studier har visat att UT effektivt hittar buggar av varierande typ, allvarlighetsgrad och svårighetsgrad. Detta arbete är en fallstudie på hur företaget Qlik kan riskmedvetet använda UT för att testa sin programvara Qlik Sense Enterprise on Windows.

Programvarutestning bygger på en kombination av manuella och automatiserad metoder och testar vanligtvis olika aspekter av en mjukvara. Automatisk testning, ofta så kallade enhetstester, används för att verifiera att individuella komponenter fungerar som de ska, medan manuell testning ofta är funktionella tester av mjukvaran som helhet från perspektivet av en användare.

I detta arbete fokuserar vi på utforskande manuella tester, specifikt testningen av Qlik Sense Enterprise on Windows som utförs på Qlik. Utforskande tester (UT) integrerar inlärning, testdesign och testutförande och vi använder följande definition: "Utforskande testning är testning där testfallet inte är helt skriptat. Testfallet har ickebeskrivande steg att följa, eller testaren kan vara helt fri att utföra testfallet på vilket sätt de vill." För att framgångsrikt utföra UT måste testaren använda sin kreativitet och fantasi för att tillhandahålla en mängd olika input till systemet som testas.

I denna fallstudien mättes vilka delar av koden som de utforskande testerna exekverade och det samlades in data för att bestämma kodens riskprofil. Kodens riskprofil bestäms av parametrarna kodkvalitet, ett värde som baseras på hur enkel koden är att underhålla samt bygga vidare, och kod-churn, som säger hur mycket koden har ändrats under utvecklingens gång, samt antalet buggfixar som har utförts på koden. Dessa värden bestäms per fil och hjälper oss hitta vilka filer som är särskilt riskbenägna, det vill säga filer där det är hög risk att buggar förekommer.



Vårt resultat visar att de utforskande testerna inte täcker en särskilt stor del av koden och att det är en bristande kodtäckning av riskbenägna filer. Vi såg även att de utforskande testerna inte har någon korrelation med kodkvalitet, kod-churn eller antalet buggfixar. Däremot såg vi en korrelation mellan kodkvalitet och antalet buggfixar samt en svag korrelation mellan kod-churn och antalet buggfixar.

Med denna fallstudie har vi bekräftat tidigare studier kring korrelation mellan kodkvalitet, kod-churn och distributionen av buggfixar samt tillhandahållit Qlik med en process för att ytterligare förbättra sitt användande av utforskande tester.