DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

Grouping and Prioritization of Test Gaps

Michael Sailer

DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Master's Thesis in Informatics

Grouping and Prioritization of Test Gaps

Gruppierung und Priorisierung von Test Gaps

Author:	Michael Sailer
Supervisor:	Prof. Dr. Dr. h.c. Manfred Broy
Advisor:	Dr. Elmar Jürgens
	Daniel Veihelmann
Submission Date:	December 16, 2019

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, December 16, 2019 Michael Sailer

Acknowledgments

I want to express my gratitude to all the people who made this thesis possible and supported me during the process.

First of all, thanks to Dr. Elmar Jürgens. After I heard your talk about how to write a thesis at TUM a few years ago, this work has obviously been a piece of cake.

Thanks to Daniel Veihelmann who was always there to answer my questions and provided valuable feedback in all stages of this thesis. In particular, thanks for the introduction to the code base of Teamscale which I needed for the implementation of the prototype and thanks for the technical support in countless cases.

Thanks to Roman Haas who initially pitched the topic and joined me for the second half of this thesis as constant advisor. In particular, he supported me with the design and evaluation of the study, the technical setup for the study objects, and provided source code for the centrality calculation.

Furthermore, I would like to thank Prof. Dr. Dr. h.c. Manfred Broy for giving me the chance to work on this topic.

I would also like to thank all the employees of *CQSE GmbH* who took the time to participate in the survey. Without this data, the evaluation and the results would have turned out far less interesting.

Finally, thanks to my parents for always supporting me and believing in me.

Last but not least, thanks to my girlfriend Marina for the constant support, not only throughout this thesis, last-minute proof-reading and providing valuable feedback, and in general, thanks for being awesome!

Abstract

Test Gap analysis is an approach in software development to detect untested changes as these are known to be more defect-prone than other parts of the code. These test gaps have to be manually assessed to decide which of them are uncritical and which ones require testing. Especially for large sets, it is often not possible to analyze all of them. Consequently, there is a serious risk that critical test gaps are missed. This thesis proposes an automated approach to prioritize test gaps by estimated criticality to solve this problem.

A set of metrics, derived from related research in the field of defect prediction, serves as basis for the automated approach. In addition to that, a survey among developers is conducted to understand the reasons which are used for the decision about the criticality of test gaps in practice. The approach is evaluated on multiple study objects in different application areas by comparing the results of the automated prioritization to manual assessments.

The study shows, on the one hand, that manual criticality assessments are highly subjective. On the other hand, the assessment of the prioritization approach is in most cases similar to the manual one. Furthermore, it outperforms trivial strategies, such as random prioritization and single-metric strategies. Therefore, the automated prioritization will likely be a valuable extension of Test Gap analysis. The concept might also be transferred to rank methods or changes in general, and with a few modifications and further research it could also be applied for defect prediction tasks.

Contents

1. Introduction

This chapter introduces the topic of this thesis. It includes the motivation, the problem statement, the contribution of this work, and an outline of the remaining chapters.

1.1. Motivation

The pareto principle states that in many circumstances 80 % of the effects comes from 20 % of the causes. Ostrand et al. investigated the distribution of defects in large software systems and found that in fact 20 % of the files contained 80 % of the defects [OWB05]. If such information was available when planning the testing activities, the resources could be focused on the most defect-prone parts of the system. Despite significant contributions in the research field of defect prediction, the developed approaches are often not generalizable and cannot be applied in practice. Consequently, testing continues to make up a substantial part of the costs in the software lifecycle for many systems. The primary goal of testing is to make sure that the functionality of a shipped software product works as expected. Poor quality in software releases can greatly endanger their success. However, in many systems test execution is rather expensive. Often manual tests still make up the majority of tests or the automatic test suites may need days or even weeks to complete. In both cases a reasonable allocation of resources is crucial.

Typically, a quality assurance team is responsible to select the appropriate test cases and to decide when enough testing has been done. Experience has shown that changed files tend to be more defect-prone [OWB05], especially when they are untested [Ede+13]. However, studies have shown that in some cases, despite a structured testing process, whole components go into production untested. The reason is usually not a lack of discipline or commitment from the testers, but rather that without dedicated tools it can be extremely hard to catch these untested changes. Test Gap analysis (TGA) aims to solve this issue by enabling effective monitoring of test gaps, i.e. new or changed code which has not been tested in a specific timeframe. The difference to defect prediction is that TGA does not classify code into defect-prone and defect-free but rather points out parts of the code which are known to have a higher probability of defects than others.

1.2. Problem Statement

One use case of TGA is at the end of a development cycle or before a release to assess whether the tests have been updated appropriately to the new and changed functionality. Even though TGA serves as effective filter by pointing out the more defect-prone parts of the system, depending on the number of tests and changes, still a potentially large number of test gaps has to be analyzed. In these cases, the detailed manual assessment of all test gaps may require disproportionate effort so that only a part of them is actually analyzed. There is little indication to quickly determine which ones should be prioritized and which ones may be less critical. The quality assurance would profit from automated assistance which points out interesting test gaps. This thesis proposes an approach which takes the results of the TGA, i.e., the test gaps, as input and prioritizes them by estimated criticality. A valid prioritization of test gaps will reduce the time needed for the manual inspection and the risk to miss critical test gaps. Thus, the goal is to simplify the applicability of the TGA concept in practice.

1.3. Contribution

The contributions of this work are the following:

- Review of relevant literature in the field of defect prediction
- List of reasons for the criticality of test gaps from a survey among developers
- Proposal of an automated approach to rank test gaps by estimated criticality
- Evaluation of the automated approach on multiple study objects

1.4. Outline

The remainder of this thesis is organized as follows. Chapter 2 presents the fundamentals of TGA and gives definitions of terms which are used in the next chapters. Chapter 3 provides a detailed overview of defect prediction research. This is used as basis for the automated test gap prioritization approach which is described in Chapter 4. In Chapter 5 this approach is evaluated and the results are presented and discussed. Chapter 6 summarizes the insights and achievements of this work and gives an outlook into the goals of subsequent research on this topic.

2. Fundamentals of Test Gap Analysis

This chapter describes the general concept of TGA, including the application in practice and empirical evidence.

2.1. Concept

The necessary inputs to the analysis are a connection to the source code repository, which provides the source code for all points in time, and information on which parts of the code have been executed in a test case. TGA always compares two versions of a system. The version at the beginning of the interval is called baseline, the version at the end is the working copy. Depending on the use case, it may make sense to choose the code of the last major release or of the last development cycle as baseline. Static and dynamic analysis are combined to provide an overview of untested changes. In the static part the source code repository is used to determine which parts of the code have been added or changed in the respective timeframe. The dynamic part uncovers changes which have not been executed by any test case after their latest modification. The quality assurance team can react by triggering tests which provide additional coverage data to close the uncovered test gaps. This process is visualized in Figure 2.1.

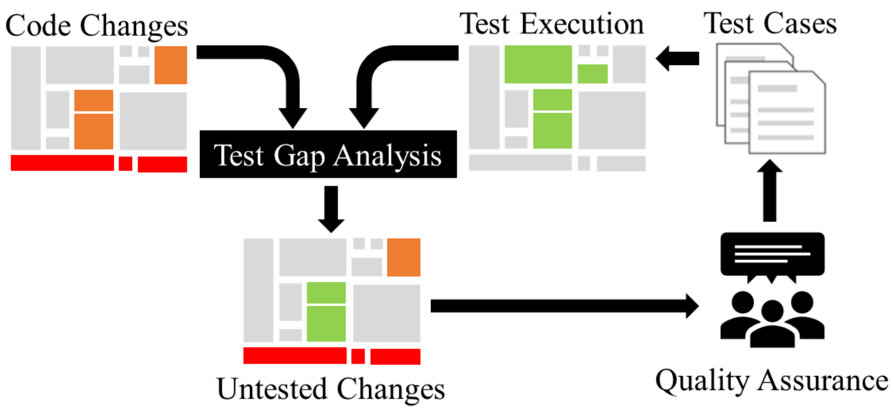


Figure 2.1.: TGA in the testing process (adjusted from [JP16]).

The implementation provides test gaps on the granularity of methods as they are known from programming languages such as Java and C#. A method is considered as changed if at least one line of code has been adjusted with a semantic change, i.e. the behavior is modified. Simple refactorings (e.g. renaming of the method, parameters and variables, changes to documentation) are not counted as actual change. Furthermore, simple getter and setter methods, which are used in object-oriented languages to access the attributes of a class, are ignored. This affects every method whose name starts with "get", "set", or "is", and only contains one statement. This kind of method can be considered as "too trivial to test" [NRW19] and is discarded from the analysis.

A method is marked as tested if it has been visited by at least one test case. Changed-tested describes a method which has been changed after the baseline and tested after its latest modification. Respectively, methods which have been changed and not executed by any test case afterwards are considered as changed-untested (definitions adapted from [Ede+13]).

There are two main limitations to the TGA approach. First, changes on configuration level which do not include actual code changes are not detected. Similarly, a prerequisite is that ideally all tests, manual and automatic ones, are recorded. Second, the analysis does not take into account how thoroughly the code was tested. In favor of a better overview of the system, simple method coverage is used. There is no guarantee that methods which are marked as tested do not contain any more defects. However, testing can in most cases only show the presence of defects, not their absence—even when more fine-grained metrics like statement, branch, or path coverage are used. TGA points out code changes which have not been tested at all and, thus, where no defects could have been found (adjusted from [JP16]).

2.2. Test Gap Analysis in Practice

The TGA results are visualized as treemap. An example can be found in Figure 2.2. Every rectangle represents a method, the area of the rectangle is proportional to its size. The color indicates the state: grey refers to unchanged code, green to changed-tested. For test gaps, i.e., changed-untested methods, TGA distinguishes whether they have existed at the baseline (orange) or have been added afterwards (red). Since the methods in the treemap are grouped by classes and components, it is easy for the quality assurance team to notice when whole parts of the system have been forgotten in the testing process by accident.

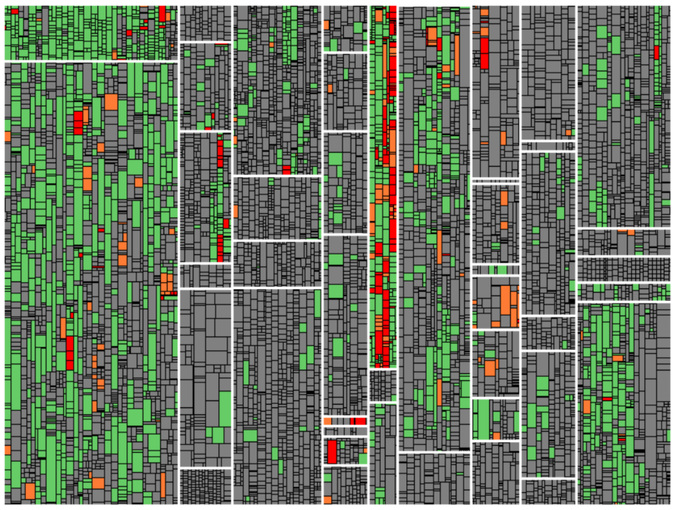


Figure 2.2.: Example results of TGA visualized as treemap.

TGA helps in the testing process to reliably catch untested changes. Especially interesting are cases where whole components have not been tested. In general, it is not necessary to aim for full change coverage. Test gaps which are not considered critical can be accepted to save testing resources. An example of an accepted test gap might be source code which is not yet available for the users. In any case, it is crucial that these decisions are well-founded and the consequences predictable.

Concrete applications scenarios are, for instance, release testing, hotfix testing, retrospectives, and monthly assessments. In release testing, TGA can be used to make sure that no critical test gaps are brought into production and, in particular, that no core components are completely untested. A hotfix is a release which repairs functionality which did not work correctly in a previous release. In this context, TGA helps to verify that the code changes of the hotfix have not introduced any critical test gaps which may have broken some other functionality. The terms retrospective and monthly assessment both refer to processes to assess the quality of a software project periodically. The difference is that the former are performed by the development team itself with an intrinsic motivation to uncover deficits and improve the quality of the software product, whereas for the latter usually an independent third party is tasked to provide an outside perspective on the system. The TGA monthly assessment uncovers untested changes for work items which have been closed in the previous month and evaluates their criticality. This focus is reasonable as issues are only closed once all testing activities have been finished.

2.3. Empirical Evidence

Already in 1985, Shen et al. found that new or modified modules are significantly more error-prone than unchanged ones [She+85]. Thus, they recommend to allocate more resources for these parts of the system. Similarly, Khoshgoftaar et al. found that unchanged code tends to be very stable and defects are introduced by changes [Kho+96]. These are fundamental ideas of TGA. These studies only used static analysis, i.e., code changes. Eder et al. extended these insights by adding coverage data from dynamic analysis [Ede+13]. They performed a study about the potential of TGA in practice by analyzing the number of untested changes which go into production and the number of defects in untested changes. The study object was a business information system at the reinsurance company *Munich Re* written in C# with a size of 340.000 Lines of Code (LOC). Two consecutive releases were analyzed over a total period of 14 months. One interesting finding is that, besides structured testing efforts, approximately half of the changes went into production untested. This confirms the need for tool support to reliably detect these test gaps. Another result is that untested changes are more likely to contain field defects (i.e. defects which occur in a released version of the software) than other parts of the code. The probability of a method to contain a defect for the two releases is shown in Figure 2.3.

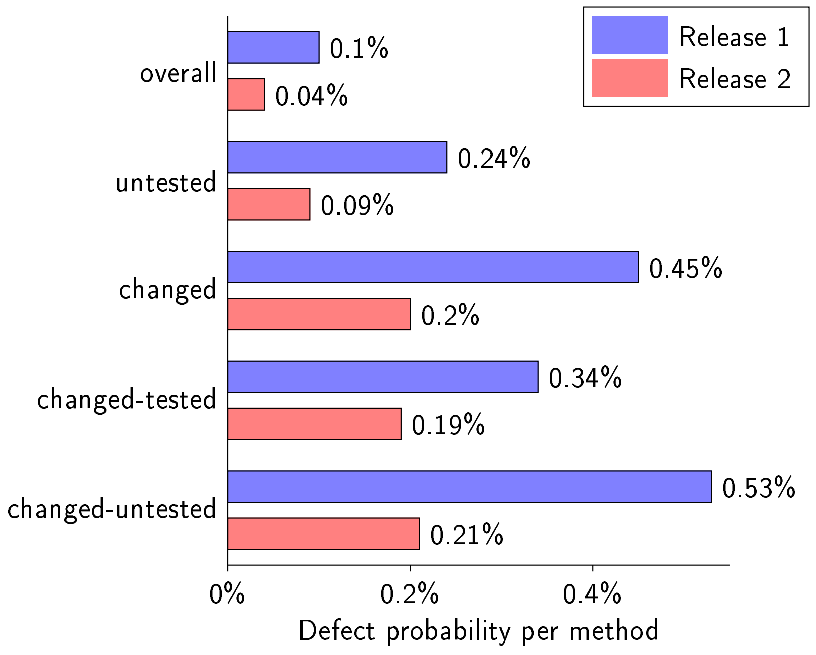


Figure 2.3.: Results of TGA study at Munich Re (adjusted from [Ede+13]).

In particular, the results show that for both releases the defect probability in changed-untested methods is about five times higher than the overall value (0.1 % to 0.53 % for release 1 and 0.04 % to 0.21 % for release 2). Changed code also contains about twice as many defects as untested methods. This implicates that using change coverage might be more effective than full coverage. Eder et al. report that TGA enabled the development team to close test gaps in features which would have gone into production untested otherwise. Thus, integrating TGA into the development process likely reduces the number of defects in the field.

3. Related Work

The last chapter has shown the concept and potential of TGA. The study by Eder et al. confirmed that untested changes are particularly defect-prone. TGA identifies changed-untested code with the expectation that it contains more defects. In that sense it is related to defect prediction. This chapter gives an overview of existing research in this field. The following sections explain the motivation and the development of defect prediction since the first attempts in the 1970s. Subsequently, the different dimensions which have been used for prediction models are presented. The insights are the basis for the test gap prioritization algorithm which is described in the next chapter.

3.1. Motivation of Defect Prediction

Writing source code introduces defects. Non-trivial, defect-free programs are either a rare exception or they only appear defect-free because the defects just have not been found yet. Defects in software can cause high costs with respect to the additional efforts which are needed to repair them or the risk of losing customers to a competitor's product. Solid testing is a way to assure the quality of software to a certain extent. However, testing can only show the existence of defects, but not their absence. Mathematical proof of the correctness of software according to a given specification is possible in theory, but not applicable for the majority of projects. Reasons for that in practice include that requirements in general cannot be fully formalized or are incomplete. For that reason researchers have been working on approaches to predict how many defects a program has and how these defects are distributed.

3.2. History of Defect Prediction

This section present an overview of the history of defect prediction. The goal is to show the overall development on the basis of selected papers rather than providing an exhaustive list of all research work in this field. This should especially show how the concepts have evolved and how the level of prediction granularity has changed over time.

Presumably the first study in the field of defect prediction was conducted by Akiyama in 1971 [Aki71]. The results showed that linear models of simple metrics like LOC provide reasonable estimates of the number o defects in a system.[1] A similar approach was proposed by Ferdinand in 1974 who used code segments instead of LOC [Fer74]. In 1977 Motley and Brooks aimed to answer the question "how programmers, program characteristics, management methods, and software testing and design factors influence and contribute to errors in programs" [MB77, p. 15]. They used variables which reflect programmer capability, program structure, and program complexity for their prediction models. Based on Halstead's formula for program length [Hal77], Lipow developed a function which could predict the total number of faults in the evaluated system rather accurately [Lip82]. Shen et al. investigated defect prediction in different phases of software development [She+85]. While previous research focused mostly on complexity metrics of the current version of the system, they already included metrics which are based on code changes. Basili and Perricone [BP82], Khoshgoftaar and Munson [KM90] and Munson and Khoshgoftaar [MK92] continued to analyze the connection of software defects and source code complexity. The latter in particular aimed to tackle the challenge of strongly non-normally distributed data, i.e., much more modules with a low than with a high number of defects. In contrast to most previous studies, which used the total number of defects as quantity of interest, Khoshgoftaar et al. [Kho+96] defined "defect-prone" as "exceeding a threshold of debug code churn, defined as the number of lines added or changed due to bug fixes" [Kho+96, p. 1]. The goal was to better reflect the necessary rework effort. This overview of early defect prediction research shows that some important concepts already exist for quite some time. For example, source code complexity has in many cases been the central prediction metric.

Even though these studies produced some valuable insights, Fenton and Neil [FN99] point out some serious methodological and theoretical flaws. They especially criticize the sole use of complexity metrics to deduce defects and argue that many more variables have to be considered to build an appropriate model. Another issue is the inconsistent use of the term "defect" which is used differently across studies or is not specified at all. This makes it hard to compare published results. Furthermore, they consider defect counts to be a poor predictor for reliability as defects may manifest in actual failures which are visible to the user in vastly different rates.

Additionally, caution is advised with the transfer of these results as programming concepts in today's languages and software engineering processes are not necessarily comparable to the ones used in the presented research.

[1]Summary from [FN99] as the original was not available online

3.3. Level of Granularity

After early research in this field mainly focused on predicting defects on module-level, especially since 2000 the tendency goes to more fine-grained granularity, such as files, classes, and methods. Even though Hall et al. found that this does not implicate better prediction performance [Hal+12], there are still multiple reasons why high granularity should be preferred. For instance, D'Ambros et al. [DLR12] decided for class-level instead of module-level prediction as statements on module-level can easily be derived by combining the prediction results for all of its classes, whereas the other direction is not possible. Another problem with coarse granularity levels is that studies often did not report the size of the respective modules. Even when the prediction finds 80 % of defects in 20 % of modules, these 20 % might in fact contain 80 % of the code. This effectively means that 80 % of defects have been found in 80 % of the code. D'Ambros et al. note that for this reason several recent work focuses on *effort-aware defect prediction* which aims to minimize the amount of effort which is necessary to locate the defects. In that sense, more fine-grained granularity reduces the necessary inspection time as it is easier to find the defects in single files or methods than in a whole component. Consequently, Hata et al. found that method-level prediction is more effective than package- and file-level prediction with respect to effort-based evaluation [HMK12]. They also concluded that the overhead for method-level prediction only concerns the computation, but there are no additional manual steps necessary compared to package- and file-level prediction. Thus, method-level defect prediction seems to be the most promising concept today. Additionally, TGA also operates on method-level. For these reasons, method-level defect prediction is the most interesting concept for this thesis. However, caution is advised when transferring results from module- or class-level to more fine-grained granularity as they are not necessarily transferable.

3.4. Dimensions

There are many different dimensions which can be used as basis for the prediction models. Most of these metrics can be assigned to one of the groups product metrics, process metrics, and organizational and developer-centered metrics. Some research focuses on one of these groups while others use a combination of some or all of them.

3.4.1. Product Metrics

Product metrics are all metrics which can be computed from a single snapshot of the software without the need to execute it, e.g. LOC and nesting [Gra+00]. In related work, product metrics are also called "code metrics" or "single-version approaches"

[DLR12]. One major benefit of product metrics is that no source code history is needed, but only the current version of code. This section provides an overview of product measures which have been used so far.

Complexity

Many metrics are connected to source code complexity. The basic idea is that complex components are harder to change, and, hence, more defect-prone.

Many studies use the metrics suite by Chidamber and Kemerer for object-oriented programming languages [CK94]. This suite includes, among others, the combined complexity of all methods of a class, the number of ancestors, and the number of direct descendants. The authors proposed to use their metrics suite to find areas in the code which might require more attention in the testing process or have potential for improvements of code quality, e.g., coupling and cohesion. Gyimothy et al. applied the metrics suite to open-source projects to predict the number of defects for classes [GFS05]. D'Ambros et al. extended this set by additional object-oriented metrics. Hall et al. found that studies which use object-oriented metrics tend to perform better with respect to defect prediction than those which only use complexity-based metrics, but not better than models based on LOC [Hal+12]. These metrics are meant to be computed on class-level. Thus, they are not directly applicable to method-level which is needed for TGA.

Many complexity metrics show a high correlation with LOC (e.g. [KM90], [MK92], [Gra+00]). The reason is that longer classes tend to be more complex than shorter ones. Thus, even though LOC is one of the simplest metrics, it proves to be a good predictor of defects on module- and file-level ([KM90], [GFS05], [OWB05], [Hal+12]). Despite these results, LOC might not be optimal with respect to effort-based defect prediction. It seems obvious that modules or files with more LOC contain potentially more defects. However, this also means a greater effort to find these in the large amount of code. For TGA, it should still be valid to prioritize longer methods over shorter ones. Thus, LOC is one of the metrics which is used in the prioritization approach.

Khoshgoftaar and Munson [KM90] recommend "volume" and "structure" as orthogonal complexity measures. The former is represented by LOC, for the latter McCabe's cyclomatic complexity [McC76], which is the number of linearly independent paths through a piece of source code, is used in this thesis.

Severity and Centrality

Ostrand et al. [OWB05] expected it to be beneficial to add a measure of defect severity to the prediction models as some defects might only have minor consequences, whereas others make the system unusable. However, they found that for most study objects information about the severity is not available. They were unsure about the usefulness of the priorities from the issue tracker as they experienced these to be highly subjective. Another possibility would be to add data about the usage frequency of the respective code in customer environment. The intuition behind this is that code which is used more often should be tested more thoroughly. However, that kind of data is not available for most systems and requires a dedicated setup. Hall et al. conducted a systematic literature review on defect prediction and only found very few studies which included severity measures [Hal+12]. They explain this by the fact that, although acknowledged to be important, severity is difficult to measure, especially as there is no widely agreed definition.

Another option is to consider code centrality measures as representation of severity. Metrics like *coupling between objects* from the metrics suite by Chidamber and Kemerer or *fanIn* and *fanOut* from D'Ambros et al. (i.e. number of other classes that reference the class or are referenced by the class) are related to the centrality of source code. Zimmermann and Nagappan found that the correlation of centrality and defects is worse than with complexity metrics [ZN08]. Thus, it should likely not be used for defect prediction. However, defects in central code are likely more severe. Steidl et al. and Sora propose an approach to identify key classes of a software system by modeling the system as a graph built by static analysis of dependencies ([SHJ12], [Sor15]).

An alternative or addition could be the manual selection of components of the system which are considered as particularly important or critical by the developers. Zimmermann and Nagappan describe such a concept which was applied in the development of Windows Server 2003. Whenever a programmer makes changes to a critical binary, the review and testing efforts are increased. For that purpose, a list of these binaries has to be created and maintained. This process could be transferred to the approach of this thesis so that test gaps which are part of critical components are ranked higher. Since a classification of the components into critical and non-critical ones was not available for the study objects, this is left for future work.

3.4.2. Process Metrics

Process metrics are computed using the change and defect history of a software system. They are also called "change log approaches" (e.g. by [DLR12]) or "change metrics". Models which only use product metrics have shown to perform relative poorly [Hal+12]. Di Nucci et al. conclude from a review of previous studies that process metrics tend to perform better than product metrics, even though the former might be superior in some contexts [Di +18]. For instance, Moser et al. still believe in a correlation between complexity metrics and defects as otherwise prediction models based on those would not work at all. They offer the following explanation for the superiority of process metrics: on the one hand, a skilled programmer can write complex programs with a low defect count. However, a prediction approach based on complexity metrics would classify it as defective. On the other hand, if a file has been changed many times in the past, then there is a high chance that one of these changes has introduced a defect, regardless of the file's overall complexity.

Here is a summary of existing approaches in the context of process metrics (adjusted form [DLR12]):

- Defects are caused by changes [MPS08].
- Complex changes are more defect-prone than simple ones [Has09].
- Past defects predict future defects ([KWZ08], [Gra+00]).
- Large change sets introduce less defects [MPS08].
- The number of defects grows with the revision number ([KWZ08],[MPS08]).
- Code which has received many refactorings contains less defects [MPS08].

All these studies worked on granularity of files or components. Thus, it is unclear how these results transfer to method-level. For the last point, it is not obvious how to determine the number of refactorings reliably. Moser et al. counted the number of commit messages which contain the word "refactor". This, however, may be inaccurate as developers likely have different understandings what counts as refactoring. If the development team has not agreed on a clear policy, they also might not include the term in every refactoring commit. More research might be necessary to generalize this approach. A valid refactoring detection is a prerequisite to include that metric for the prioritization approach in the future.

Nagappan et al. found that with a controlled change process, i.e., changes are only committed when they are considered ready and expected to keep the product stable, consecutive changes are a good predictor of defective components [Nag+10]. With their approach they reached remarkable prediction performance. Nevertheless, the approach is likely hard to transfer to method-level.

3.4.3. Organizational and Developer-Centered Metrics

The influence of the organizational structure on software has been described by Conway: "Organizations which design systems [...] are constrained to produce designs which are copies of the communication structures of these organizations." [Con67] Hence, it is likely that the organizational structure also influences the software quality. There are a couple of defect prediction metrics which are based on that assumption. For instance, Nagappan et al. were able to beat other common prediction strategies of that time with a model based on organizational measures, such as number of developers and level of code ownership [NMB08]. Bird et al. found that distributed development does not seem to affect software quality [Bir+09]. Contradicting results exist for the influence of the number of contributing developers. On the one hand, Pinzger et al. report that the more developers contribute to a binary, the more likely it will be affected by postrelease defects [PNM08]. On the other hand, Graves et al. did not find any evidence for this statement in their study [Gra+00]. For Weyuker et al. the addition of developer information at least contributed a slight improvement to their prediction results [WOB08].

Di Nucci et al. [Di +18] advocate developer-centered metrics with the rationale that source code is ultimately written by humans. Especially under stressing conditions and time pressure they are likely to produce more defects. Consequently, they propose an approach for class-level defect prediction based on the scattering of developer activities. The idea is that programmers who are not familiar with certain parts of the code are more likely to introduce bugs when they do adjustments there. Their results have shown that the scattering metrics are highly complementary to other metrics, i.e., they detect other kinds of defects. That means that adding the scattering metrics to an existing metrics set will likely improve the performance.

Due to the partly contradicting results in the different studies and the lack of respective information for the study objects in this thesis, this metrics group is not included in the approach, but left for future work.

3.5. Critique of Existing Approaches

One problem with study results in the field of defect prediction is that they are often not generalizable. Hall et al. notice in their literature review that most studies report bad performance when the approach is transferred [Hal+12]. D'Ambros et al. suggest to either gather more results from a variety of study objects to refine the observations or to make more use of domain knowledge to tailor the techniques to the prediction task [DLR12].

An indication for the poor generalizability are replication studies, e.g., by Pascarella et al. [PPB18] for the approach developed by Giger et al., which often yield worse prediction accuracy than the original publication when evaluated on new study objects. A common evaluation technique is *n-fold cross validation* (often used with n=10). Separate sets for learning and testing are necessary to appropriately benchmark the performance of a machine learning classifier. Arbitrarily splitting the available data into two parts could introduce a bias. For that purpose, n-fold cross validation has been developed. It divides the training data into n equal parts. The learning and testing process is performed n times, each time with a different part as testing set while the other n-1 parts are used for learning. The average performance of all n rounds is used as final result.

Pascarella et al. point out a serious flaw with this evaluation technique. For instance, in a scenario with two releases, both of which contain defects, the evaluation is performed with the data that is available after these releases. That means that defects are predicted with data that appeared after these have already happened. In a real-world scenario one would want to predict defects which will happen in the future with the currently available data. Consequently, Pascarella et al. propose *release-by-release validation* as more realistic evaluation technique. That means that the classifier is trained on the data that is available at the time of the first release to predict defects which occurred in the second release. This insight is not necessarily new, but it has not been adopted in the majority of the research community yet. Already in 1996, Khoshgoftaar et al. noticed that release-by-release validation is more meaningful [Kho+96]. Similarly, Moser et al. [MPS08] observed that obviously a model which is trained on one release produces less accurate prediction results for a subsequent release.

When Pascarella et al. evaluated the replicated approach with the release-by-release validation, the accuracy was drastically lower and no longer significantly better than a random classifier. Thus, they conclude that more research is necessary and method-level defect prediction is still an open challenge. This led to the decision to choose a different approach for this work which is described in the following chapter.

4. Approach

In this chapter, the prioritization approach is described. The concept is to develop a lightweight prioritization approach for test gaps which is based on a set of metrics derived from related research. After computation and normalization the metric values are combined into one criticality score for every test gap.

4.1. Concept

Method-level defect prediction appears to be still unsolved [PPB18]. Thus, instead of applying immature solutions to the problem at hand, the idea for the approach of this work is to tackle the challenge from a slightly different perspective. Rather than doing actual classification in defect-prone and defect-free test gaps, we propose an algorithm to rank them by estimated criticality. A valid prioritization is likely a good first step to support the manual assessment process in practice.

The majority of existing defect prediction strategies depend on machine learning. These usually require expensive setup and need to be trained for every project separately. The presented approach is rather lightweight as it works without machine learning and should require minimal configuration. Overall, the idea is to use TGA as preliminary filter, i.e., only consider untested changes which are known to be particularly defect-prone, and then prioritize the remaining list of test gaps by a set of metrics which reflect their criticality.

4.2. Selected Metrics

The basis for the prioritization is a set of metrics which is derived from related research. The metrics which are feasible to implement and evaluate with the available data and resources were selected.

The first two metrics are computed from the reference method. This is the version of the method before it became a test gap in the analyzed timeframe. Usually this is the state at the baseline. It may also refer to the last tested version if it has been tested afterwards. For methods which have been added after the baseline, the reference method is empty. The metrics Length of reference method (LEN) and Complexity of reference method (COM) describe the length and complexity of the method before it became a test gap. The method in the working copy is compared to the reference method for the computation of the change metrics Changed lines (CLI) and Complexity change (COC).

The term *finding* describes a potential problem which can be assigned to a specific code region and which is detected by static analysis. One dimension of differentiation is between non-critical findings, e.g., missing documentation, and critical findings, e.g., a potentially unhandled exception in the code.[1] The other dimension is whether the finding

- has been added in the change since the reference method, i.e., Added non-critical findings (ANF) and Added critical findings (ACF),
- removed with the change, i.e., Removed non-critical findings (RNF) and Removed critical findings (RCF), or
- has existed in the reference method and not been removed, i.e., Non-critical findings in changed code (NFC) and Critical findings in changed code (CFC).

The centrality of the method in the working copy is reflected in the metric Centrality (CEN), the number of previous defects in the metric Previous defects (PRD). For the last two metrics the birth issue is needed, i.e., the issue which introduced the test gap. Refactoring (REF) indicates whether the birth issue is considered a refactoring and High priority (HIP) reflects whether it has high priority in the issue tracker.

4.3. Preparation

The input of the algorithm is the result of the TGA, i.e., the test gap data which is not yet prioritized. This includes the path of the respective file in the software project, the region of the method in the file, the method name, and the timestamp of the last test of the method. Additionally, the beginning and end timestamps of the TGA are given.

On the one hand, the reference version of the method is necessary to compute the actual change. On the other hand, for the metrics REF and HIP, the birth issue of the test gap is needed. The first step for both is to retrieve a list of commits with respective timestamps in which the method was changed. The interesting threshold is the point in time before the method became as test gap. In most cases this is the baseline timestamp. If the method has been tested after the baseline, it is the timestamp of the last test

[1]Teamscale which is the basis of the implementation of the prioritization approach has definitions which findings are considered as critical. These have been chosen by the developers based on experience.

instead. For the reference method, the commit before or at the threshold is needed. It can be extracted through the snapshot of the repository at the desired time. The birth issue is the one which is associated to the commit after the threshold. It is a well-established policy in development teams to include a reference to the respective issue in every commit message. For example, the commit message "CR#123 Fix bug" indicates that this change is connected to the issue with the identification number 123 in the issue tracker.

An alternative to this interpretation of the reference method would be to use the empty method as reference for test gaps which have never been tested. However, in the author's opinion one goal of TGA is to assess the criticality of the current changes which happened since the baseline. Whether the version of the method at the baseline has been tested, is secondary according to this understanding. Of course, there is a probability that unchanged parts of the method contain defects, but these would likely already have been detected as they have been in the system for longer.

4.4. Metrics Computation

The value for LEN is the number of LOC, COM is the cyclomatic complexity which is computed as defined by McCabe [McC76]. COC is the difference of the complexity of the method in the working copy and the reference method. This metric may contain negative values if the method became less complex. The computation of CLI has to deal with ambiguous data. The problem is that, by default, the data in the repository only allows to retrieve changes in the *unified diff* format, i.e., the added and deleted lines between two versions. However, there is no clear indication which lines have been changed as these are represented by a combination of one deleted and one added line. Actually determining the number of changed lines would require additional computation which was not implemented for this thesis and might be added in the future. This leads to the situation that a change of one added and one deleted line can refer to one or two changed lines. The former is the case if an existing line has been adjusted, the latter if one line has been deleted and another one has been added. One possibility would be to use the the larger value of both. However, this is incorrect for the latter case which actually contains a change of two lines instead of one. Thus, Kim et al. compute the size of the change as sum of deleted lines d and added lines a [KWZ08]. In contrast to that, for this thesis CLI is defined as $d + 2a$. This is not based on empirical evidence, but rather on the intuition that adding lines may be more defect-prone than deleting lines. If that proves to be untrue, this computation should be adjusted.

One option for the refactoring detection in REF would be to check the issue type.

Many issue trackers allow to differentiate, for example, between *Maintenance*, *Feature* and *Bug*. In contrast to the other ones, *Maintenance* is usually not used to repair defects or implement new functionality, but rather for non-functional requirements, like maintainability or testability. Consequently, these issues are more likely to contain refactorings. However, an analysis of issues with large change sets for the first study object has shown that often also tickets of other types contain a rather large amount of refactorings. Thus, this option was discarded in favor of checking the size of the issue's change set, i.e., the number of adjusted methods. The assumption for this metric is, the larger this number, the more likely it is a refactoring. The challenge is to find a threshold which identifies refactoring issues with an appropriate accuracy.

CEN is a measure of the centrality of the file which contains the test gap. This should address the potential severity of a defect as more central defects tend to have more drastic consequences. The computation is based on the approach by Steidl et al. [SHJ12] and Sora [Sor15]. The algorithm to determine graph centrality which is used in this thesis is PageRank which was developed by Page et al. [Pag+99]. A link of a node, e.g., a class in a software system, to another one is interpreted as *recommendation*. Nodes which receive many recommendations are considered as central. The prerequisite is to define a representation of the system as a graph. In this case, the nodes are the files and the edges are the dependencies between them. The result of PageRank for every node is the probability to reach it when going through the graph. The graph can be traversed by following a link or by jumping to a random other node. For this thesis, the probability for the latter has has been defined as 0.001 as proposed by Steidl et al. A value close to 1 would yield a low variance of values between nodes. The links have less meaning as most of the time a random jump is performed instead. A random-jump-probability close to 0 is selected to reach a non-uniform distribution with high variance between the node values. Sora additionally suggests the use of *forward* and *back* recommendations. If a class A depends on a class B, the edge from A to B is a forward recommendation. This is motivated by the fact that a class which is part of the dependencies of many other classes is likely an important one. The back recommendation reflects the importance of a class which uses many other important classes. Without back recommendations, library classes would be ranked very high, whereas the classes which control the application are considered as unimportant. For this thesis, the weight of back recommendations is set to $\frac{1}{3}$ as compromise between the values $\frac{1}{4}$ and $\frac{1}{2}$ which Sora considers to yield the best overall results.

The value for HIP is defined as 1 if the birth issue of the test gap has high priority and as 0 otherwise. PRD is the number of previous defects which occurred in the method, i.e., the number of issues of type *Bug* which are associated to commits in which the method was changed. High priority issues are counted twice for this metric as they are expected to contain more important changes. In the current version of the approach, the issue which introduced the test gap is not counted as previous defect, but only all *Bug* issues before that.

4.5. Metrics Normalization

The metrics have vastly different value ranges. For instance, the REF metric which is based on the change set of the birth issue may in theory have arbitrarily large values, whereas HIP can only be 0 or 1. Thus, all metrics are normalized to the value range of HIP, i.e., [0,1] when only positive values are possible.[2] For metrics which can take negative values the value range is [-1,1]. The normalization is performed, for a set of metric values S, relative to the maximum value $max(S)$ and the minimum value $min(S)$ for this metric in the given set of test gaps. For most metrics this normalization is *linear*, i.e., the normalized metrics value v_n is computed from the metrics value v_m as

$$v_n = \begin{cases} 0, & max(S) = min(S) \\ \dfrac{v_m - min(S)}{max(S) - min(S)}, & otherwise. \end{cases}$$

For a metric which represents a difference, i.e., only COC in this case, the boundaries of the interval are defined by the highest absolute value for this metric in the current set of test gaps, i.e., $f(S) = max(abs(v_m)|v_m \epsilon S)$. That means that if the highest absolute value comes from a positive value, v_n will reach 1 but not -1 for this metric and vice versa. Hence, this strategy (*linear diff.*) preserves the sign of v_m by using the formula

$$v_n = \begin{cases} 0, & f(S) = 0 \\ \dfrac{v_m}{f(S)}, & otherwise. \end{cases}$$

As described earlier CLI is computed by combining the number of added and deleted lines. The current implementation determines the maximum for both of them separately, i.e., they may come from different test gaps. Thus, the maximum of 1 will only be reached for this metric if one test gap has both the most added and the most deleted lines. This may be considered as a flaw and will be addressed in the future.

The REF metric depends on a threshold t. The obvious implementation is a hard boundary where all test gaps with a change set which is only slightly smaller will receive the value 0. A more realistic representation might be the function

$$v_n = \begin{cases} 1, & v_m \geq t \\ (\dfrac{v_m}{t})^4, & otherwise. \end{cases}$$

[2]The common interval notation is used here, i.e., $[a, b] = \{x \epsilon \mathbb{R} | a \leq x \leq b\}$.

The exponent 4 is chosen arbitrarily to model that v_n grows rather slowly for low values and faster when v_m gets close to the threshold. The corresponding function plot is shown in Figure 4.1.

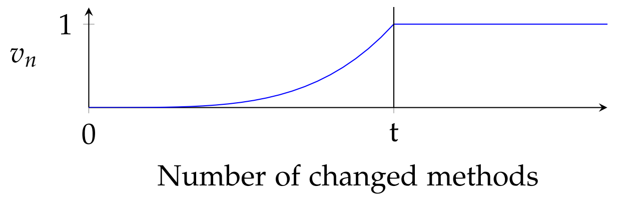


Figure 4.1.: Normalization function for REF.

Table 4.1 gives an overview of the metrics' absolute and normalized value ranges and the respective normalization strategies.

Set name	Range abs. (v_m)	Norm. strategy	Range norm. (v_n)
LEN	$[0, max(LEN)]$	linear	[0, 1]
COM	$[0, max(COM)]$	linear	[0, 1]
CLI	$[0, max(CLI)]$	linear	[0, 1]
COC	$[-f(COC), f(COC)]$	linear diff.	[-1, 1]
ANF	$[0, max(ANF)]$	linear	[0, 1]
NFC	$[0, max(NFC)]$	linear	[0, 1]
ACF	$[0, max(ACF)]$	linear	[0, 1]
CFC	$[0, max(CFC)]$	linear	[0, 1]
RCF	$[0, max(RCF)]$	linear	[0, 1]
RNF	$[0, max(RNF)]$	linear	[0, 1]
REF	$[0, max(REF)]$	soft threshold	[0, 1]
CEN	$[0, max(CEN)]$	linear	[0, 1]
PRD	$[0, max(PRD)]$	linear	[0, 1]
HIP	$\{0, 1\}$	none	{0, 1}

Table 4.1.: Overview of implemented metrics which are evaluated in the study.

4.6. Criticality Score Computation

Finally, the normalized values of all metrics are combined into a criticality score for every test gap which is used as basis for the prioritization. Every test gaps has a set of normalized metric values V_n. W is the set of corresponding weights. Each metric has exactly one weight which is the same for all test gaps. Thus, with $n = |W| = |V_m|$ the criticality score c for a test gap is computed using the formula

$$c = \sum_{i=0}^{n} V_n[i] * W[i].$$

The criticality score is not comparable across projects or different sets of test gaps. The reason is the normalization which is based on the maximum and minimum metric values of the current set. That means uncritical test gaps in an uncritical set may have the same criticality score as critical test gaps in a critical set. Thus, only statements about the relative criticality inside the current set are possible. Investigating possibilities to adjust the approach to allow comparability or classification into critical and uncritical test gaps remains for future work.

5. Evaluation

The following sections present the evaluation of the prioritization approach for test gaps which was described in the previous chapter. The answers to the Research Questions (RQs) are provided through a study where the results of the algorithm are compared to manual assessments. The chapter contains a description of the study design and study objects, the concrete steps which were necessary for the setup, and the presentation and discussion of results. Finally, some threats to validity are addressed.

5.1. Research Questions

In particular, this thesis aims to answer the following RQs.

RQ1: When is a test gap considered critical by developers? Related research has investigated a diverse set of metrics which are connected to the distribution of defects in software projects. However, it remains unclear what makes developers assess a test gap as critical. The reasons may overlap with metrics which are used in defect prediction, but might as well differ in certain parts. In particular, it is possible that there are criteria which make a test gap noteworthy which are not directly connected to expected defects. A formalization of these insights could be a good foundation for the development or refinement of an automated test gap prioritization approach.

RQ2: Do developers in general agree on the criticality of a test gap? The assessment of the criticality of a test gap may be highly subjective. In this RQ, we want to investigate how high this variance is in practice, where small variance would likely simplify the formalization. If the variance turns out to be rather high, the goal might be to find an appropriate compromise between the different opinions.

RQ3: Can an automatic prioritization of test gaps produce results similar to a manual assessment? The purpose of an automated prioritization is to remove the need to manually go through all test gaps. For this purpose, the automated approach should produce results which are similar to those of a developer.

RQ4: Does the developed prioritization algorithm outperform trivial prioritization strategies? There is no reference data of comparable automated prioritization approaches. However, to justify the sophisticated approach, it should in any case produce more accurate results than trivial prioritization strategies, e.g. prioritization based on single metrics or random prioritization.

5.2. Study Design

This section shortly describes the approach to answer the presented RQs. Details of the evaluation are later given in the respective results sections. The study basically consists of two parts which differ in study objects (for details, please refer to Section 5.3) and methodology.

The foundation of the first part is a survey among professional software developers. They were asked to assess the criticality of test gaps on a scale and state the rationale for their decision in a short text. These reasons give insights into what makes a test gap appear critical (RQ1). The variance of criticality assessments for a test gap shows the amount of deviation among multiple developers (RQ2). Besides the general statistical variance, a dedicated measure for rater agreement has been proposed by the name of Cohen's κ (kappa) [Coh60]. It was originally formulated for two raters, but can be generalized for three or more without major effort so that it can be used for the evaluation of this RQ. The average criticality assessments of the developers are compared to the results of the prototype to benchmark its performance (RQ3). In particular, these results are compared to trivial prioritization strategies to evaluate whether they are outperformed by the sophisticated approach (RQ4).

In the second part of the study, the prototype is tested on further study objects to receive insights about its performance in different systems. Further motivation for this separation is the work by Fenton and Neil [FN99]. They observed that many studies in defect production tend to rather do model fitting instead of actual prediction. That means the prediction models are optimized to give the best results for the study object, but are not evaluated on code which has not been used to create the models. In our case the approach was not actually optimized for the first study object, but, nevertheless, developed with it which might have introduced a bias. Thus, in the second part, the algorithm is applied to a series of study objects which have not influenced the development of the algorithm in any way. The reference data for the study objects in the second part comes from monthly assessments which point out critical test gaps.

The grouping of test gaps is not evaluated in this work as it would require a different study setup.

5.3. Study Objects

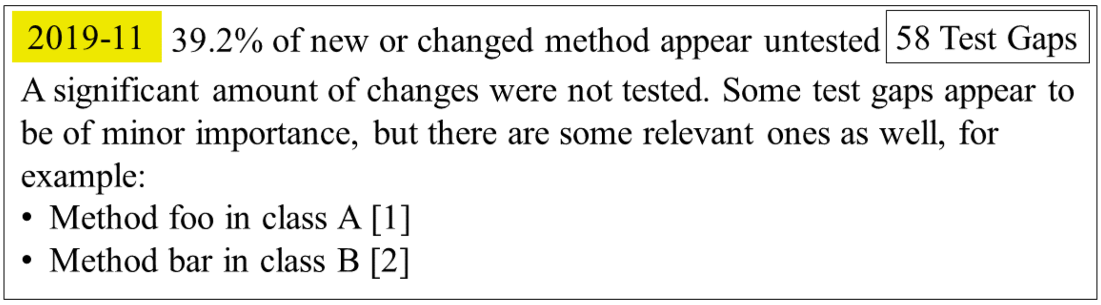
This section describes the software systems which are used in the evaluation.

The object of the first part of the study is the software quality analysis tool Teamscale.[1] Is is mainly written in the programming languages Java for the server- and Javascript

[1]Teamscale website: `http://teamscale.io/`

for the client-part. The server analyzes software projects for quality deficits, the results can be accessed through a web interface. Additionally, Teamscale offers plugins for developers to directly access the features in their IDE. A primary criterion for the choice of study objects was the availability of manual assessments of test gap criticality as reference data. The author has been a member of the development team of Teamscale for one month and, thus, has access to these resources.

In the second part, the algorithm is evaluated on several business information systems.[2] They have been in use successfully for multiple years and are still actively developed and maintained. A well-established issue tracking and testing process is in place which provides the necessary data for the evaluation. In particular, monthly assessments are conducted where the corresponding test gaps are assessed by criticality. That means that there is a rich archive available with reference data. An example of such an assessment can be found in Figure 5.1. The box in the top-left corner indicates the month which is analyzed. The state is given in traffic light colors: green stands for no problems, yellow for minor, and red for major ones. The top-right corner shows the total number of test gaps in that month and the subject line in the middle gives the percentage of untested changes. A short description summarizes the assessment and critical methods are mentioned.


2019-11 39.2% of new or changed method appear untested 58 Test Gaps

A significant amount of changes were not tested. Some test gaps appear to be of minor importance, but there are some relevant ones as well, for example:
- Method foo in class A [1]
- Method bar in class B [2]

Figure 5.1.: An example of a monthly assessment.

Many assessments do not differentiate between critical and non-critical test gaps. In some cases only vague formulations like "most" or "some" are used which do not uniquely identify the concrete test gaps and, therefore, were not suitable for the study. Furthermore, as the evaluated approach produces a ranking and not a classification, a set of mostly or only critical test gaps would unlikely yield interesting insights. Thus, from the portfolio we selected suitable projects where monthly assessments mention concrete critical test gaps and the set also contains uncritical ones. A total number of over 100 monthly assessments from 18 systems from the last twelve months was

[2]The names of the applications have been anonymized on request of the providing industry partner.

available, but only eleven of those from four systems fulfilled the prerequisites for the study.

An overview of all study objects is shown in Table 5.1.[3] The given contextual data aims to meet the criteria which were defined by Hall et al. [Hal+12] in an extensive literature review of defect prediction studies. These include the source of data studied (e.g. closed-source or open-source), the maturity, size in LOC, application area (e.g. telecoms, customer support), and programming language. All study objects used in this work are industrial, closed-source systems and have a history of multiple releases over several years.

Study Object	LOC	Application Area	Language	Source
Teamscale	971k	Software quality analysis	Java & JavaScript	closed
Study Object 2	1600k	Business information system	C#	
Study Object 3	49k		ABAP	
Study Object 4	195k		ABAP	
Study Object 5	1800k		ABAP	

Table 5.1.: Overview of study objects.

5.4. Study Setup

The preparation of the study required several steps, in particular the implementation of a prototype and the instantiation of the weights for the metrics, and the preparation of the survey.

5.4.1. Implementation and Weight Instantiation

A prototype of the prioritization approach using the selected metrics was developed as an extension of the quality analysis tool Teamscale. For most metrics this was straightforward. Some limitations are described in Section 5.6.

After the implementation, the second step was to instantiate the weights for the individual metrics. Here are some of the considerations which influenced the decision.

- The size and complexity of the change is more important for the criticality than the size and complexity of the reference method. (process metrics outperform product metrics)

[3]The size (LOC) refers to September 2019 for Teamscale, and December 2019 for the other projects.

- Anything which deteriorates the code quality or makes a change potentially more critical receives a positive weight. (e.g. adding findings or increasing the complexity)
- Anything which improves the code quality or makes a change potentially less critical receives a negative weight. (e.g. removing findings or reducing the complexity)
- Critical findings have more influence than normal findings.
- Added findings are more critical than existing findings.

An overview of all weights which are used in the study are shown in Table 5.2.

Set Name	Metric	Weight
ACF	Added critical findings	4
CLI	Changed lines	2
COC	Complexity change	2
ANF	Added non-critical findings	2
CFC	Critical findings in changed code	2
PRD	Previous defects	2
CEN	Centrality	2
HIP	High priority	1
LEN	Length of reference method	1
COM	Complexity of reference method	1
NFC	Non-critical findings in changed code	1
REF	Refactoring	-1
RNF	Removed non-critical findings	-1
RCF	Removed critical findings	-2

Table 5.2.: Overview of metric weights used in the evaluation.

The values for all metrics are normalized to the range [0,1] (resp. [-1,1] for COC) depending on the minimum and maximum values in the given set of test gaps as described in the previous chapter. The only metric which requires the selection of a certain threshold is REF which describes the probability that the test gap was introduced in a refactoring issue. The idea is to set this threshold depending on the average size of the change set. For that purpose the author of this thesis analyzed how many methods have been changed per completed issue in the Teamscale repository in the six weeks which were used for the first part of the study. In total, 38 issues have been closed in this time interval. The smallest change set consisted of one method, in the largest issue 725 methods were changed. This results in a mean of 99.81 and a median of 20 methods per issue. The average value turned out to be an appropriate threshold for this study. This means that all issues with a change set of at least 100 methods are considered a refactoring issue. As described in the previous chapter, this threshold is soft, i.e., the metrics value rises slowly from 0 to 1 for change sets which are smaller than 100 methods. The analyzed time interval contains 13 issues with a change set of at least 100 methods. Manual inspection showed that many of them in fact were actual refactorings, some others were bug fixes or features which, however, contain a significant part of refactorings. This refactoring threshold was also used for the remaining study objects. Project-specific thresholds might improve the results and should, thus, be subject to subsequent research.

5.4.2. Survey Preparation

The final step of the study setup was the preparation of the survey. The foundation was the selection of test gaps from a six-week interval, i.e., one release phase, in the Teamscale repository which are presented to the participants for a manual criticality assessment. In this timeframe a total number of 579 test gaps were observed. From the prioritized list, 33 test gaps have been selected. This includes eleven test gaps from each the top, the bottom, and the middle of the prioritized list. In a manual process the author removed test gaps which are either duplicates, i.e., semantically equivalent or closely related methods, or which did not contain a reference to a birth issue, e.g. merge commits. The rationale for the former was that closely related methods are unlikely to bring additional insights. The latter should increase the number of test gaps which can be used to evaluate the metrics which depend on the birth issue.

To decrease the effort per respondent, the 33 test gaps were divided into three surveys with eleven test gaps each. With an expected number of five responses per survey there should be enough assessments for a proper evaluation. In particular, the author considered this more valuable than 15 assessments of eleven test gaps. The survey process is visualized in Figure 5.2.

The survey starts with an introduction of the motivation and an explanation of the context. The eleven test gaps make up the first section of the survey. For each test gap, the method name and the path inside the project are shown. As already discussed, defects are introduced by changes. Thus, the criticality of the actual change since the method became a test gap in the analyzed time frame should be assessed. For that reason screenshots of the code before and after the change are provided. Additionally, the survey contains links to the birth issue in the issue tracker and to further information about the test gap in the Teamscale User Interface (UI). The criticality should be assessed

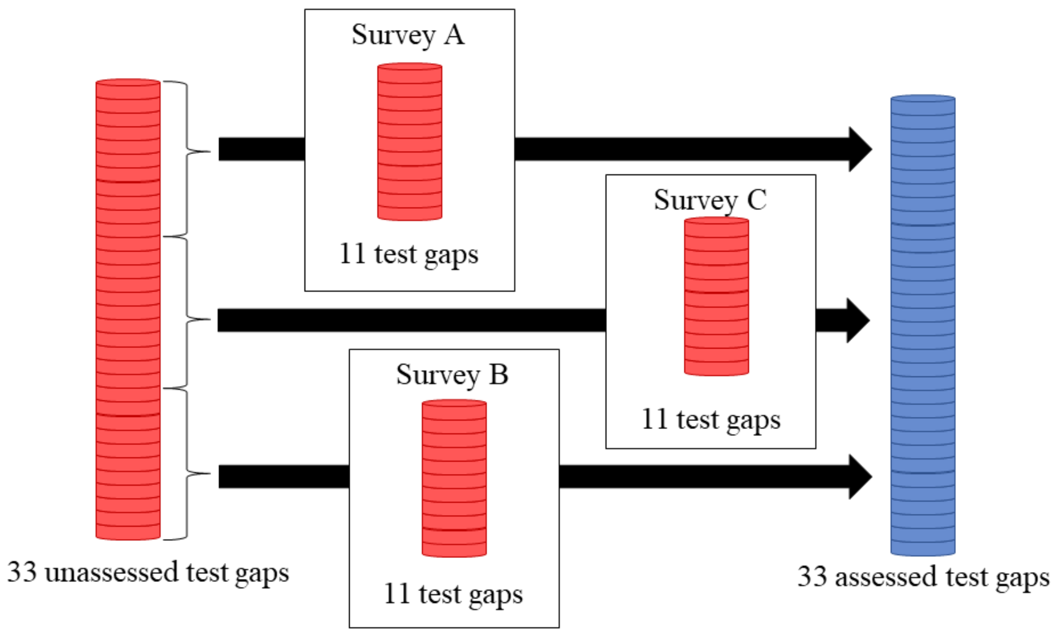


Figure 5.2.: Visualization of the survey process.

on a four-point Likert scale from “Not critical” to “Very critical”. The question allows the answer “Don’t know” to not force the respondents to do a random pick in case they cannot decide. The four-point scale has been chosen to allow more gradation in contrast to a binary classification into critical and non-critical. A larger scale was expected to be inappropriate as the manual assessment process likely does not differentiate between that many criticality levels. The participants are also asked to state the reasons for their choice in a short textual answer. This should provide answers for RQ1.

The second section includes questions about the personal experience of the participants as software developers, with Teamscale, and in the manual assessment of test gaps. In the last section, the respondents were asked to give their opinion about two metrics which were not included in the prototype, but might be of interest for further development. The first one is whether to rate test gaps as more critical when they have been introduced *shortly before a release*. The rationale is that these might only undergo a reduced quality assurance process before they are released. In the release process of Teamscale this could refer to every test gap which is added after the third Release Candidate (RC). The second question is how to handle code changes which are *not considered worth a review*. Structured development processes usually cannot be managed without some kind of issue tracker. Every relevant code change in the repository should ideally be connected to an issue in the issue tracker which explains the context of the change. Issues often include a review process to double check the validity of the changes. Small or quick changes are sometimes not thought to be worth the effort of the full review process. In these cases, developers might add a reference to a pseudo issue to the commit message to fulfill the policy configured for the repository. In the case of Teamscale this pseudo issue id has the identification number 0 and the commits are informally known as "CR#0 commits".

A short version of the survey can be found in Appendix A. There are some explanations which have been added after the start of the survey in response to feedback of participants to provide more clarity:

- Explanation of the term "CR#0-commit".
- More detailed explanation of the meaning of the criticality scale.
- Explanation that the whole method which is depicted is untested (TGA provides test gaps on method-level).
- Request to provide urgent feedback via email.

5.5. Results and Discussion

In this section the results of the study with respect to the RQs are presented and discussed. First of all, the experience value is introduced which is used to identify possible connections between the results and the experience level of participants.

5.5.1. Experience Value

The participants in the survey are all developers of Teamscale. In total 17 people completed the survey—six for two of the version, and 5 for the other one. Not all of them are full-time developers, but all have a strong background in computer science. The experience as software developer ranges from 1 to 20 years with an average of 8.94. This seems like a good representation of a common team consisting of both senior and junior developers. They have been involved in the development of Teamscale for between 1 and 7 years with an average of 3.76 years. 53 % of them have already done manual assessments of test gaps in the context of retrospectives, 47 % in monthly assessments, 31 % even in both.

To investigate the connection between the results of the study and the experience, the author defined an experience value for every participant. This value is the average of the scores for development, Teamscale, and test gap assessment experience. The median of the multiset M is written as $median(M)$, y_i denotes the years of experience for participant i, and Y refers to the multiset containing the experience years of all participants. Then, for the development experience and Teamscale experience the formula for the respective score s_i is

$$s_i = \begin{cases} 1, & y_i \geq median(Y) \\ \frac{y_i}{median(Y)}, & otherwise. \end{cases}$$

For the experience in the assessment of test gaps the value is 1 if the participant has at least experience in two of the categories retrospectives, monthly assessments, and others. Respectively, it is 0.5 for experience in one category, and 0 for no experience. The distribution of experience scores for this study can be found in Figure 5.3. This allows to compare the study results for all participants with results which only include answers of the more experienced participants. For this purpose, the author selected the threshold 0.6 which reduces the set of participants from 17 to the 13 most experienced ones.

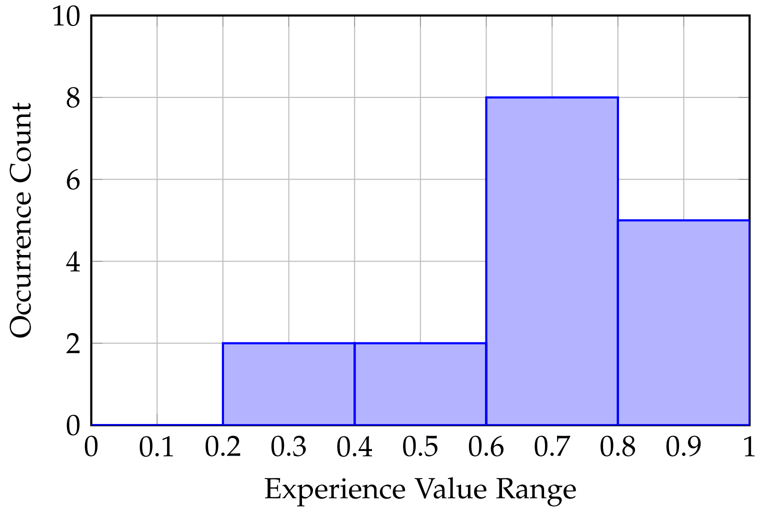


Figure 5.3.: Cumulative occurrence count for different experience value ranges.

5.5.2. RQ1: Reasons for the Criticality of Test Gaps

Results

For this RQ the reasons for the criticality assessments from the survey are evaluated. The author manually extracted the essence of every answer. The analysis yielded some rules for the mapping of certain keywords to the different categories which can be found in Table 5.3.

Category	Keywords
Change (length/complexity)	change, modification, method (for new method)
Method (length/complexity)	method (for modified method)
Length	short/long, little/significant amount of code/change
Complexity	few/many branches, simple/complicated control flow
Length and Complexity	minor, straightforward, (non-)trivial, (non-)critical, one-liner, no logic, simple
Feature importance	basic/critical/important/fundamental functionality, centrality, errors visible to user, util function, merely informational function, only admin-relevant, only one feature/service affected
Refactoring	renaming, parameter/method/code extraction
Boilerplate code	infrastructure code, generated code, done in a standard manner, default code
Testability	easy/hard to test, unsure how to test
New code	new
Visibility	private, protected, public
Avoid regression	bug fix, regression
Estimated defect-proneness	potential exceptions, bug-prone operations, many edge cases, bug-prone based on experience
Tested elsewhere	is/should be tested elsewhere
Unexpected test gap	surprised that not covered by tests
Error handling	different exceptions, error case

Table 5.3.: Mapping of keywords in answers to categories.

The categories for length and complexity exist for both the whole method and only the actual change. The differentiation also considers whether the test gap is a modified method which existed at the baseline or has been newly introduced. For modified methods, the keyword *method* counts for the respective method category, for new methods it counts for the change category as the whole method has been added or changed. The distinction between the terms *complexity* and *length* is not necessarily clear in all cases and the participants might have a different understanding of these.

Hence, some keywords are counted for both the length and complexity category.

Reason	Absolute	Relative			Direction (Reason $\implies$ Criticality)
	Count	Count	Count (Exp.)	Count (Exp. inv.)	
Centrality	33	0.15	**0.18**	0.12	+ $\implies$ +
Change complexity	27	0.13	0.11	**0.15**	+ $\implies$ +
Refactoring	21	0.10	**0.12**	0.07	True $\implies$ -
Change length	20	0.09	0.08	**0.12**	+ $\implies$ +
Boilerplate code	15	0.07	0.07	0.07	True $\implies$ -
Testability	10	0.05	**0.06**	0.03	+ $\implies$ +
New method	10	0.05	0.04	0.05	True $\implies$ +
Visibility	8	0.04	0.03	0.05	+ $\implies$ + (?)
Avoid regression	7	0.03	0.04	0.02	True $\implies$ +
Estimated defect-proneness	6	0.03	0.02	0.04	+ $\implies$ +
Tested elsewhere	6	0.03	**0.04**	0.01	True $\implies$ -
Method length	4	0.02	0.02	0.02	+ $\implies$ +
Unexpected test gap	3	0.01	0.02	0.01	True $\implies$ +
Method complexity	2	0.01	0.01	0.01	+ $\implies$ +
Error handling	2	0.01	0.01	0.01	?
Change length or complexity	34	0.16	0.14	**0.19**	+ $\implies$ +
Method length or complexity	5	0.02	0.02	0.03	+ $\implies$ +
Total	213	1	1	1	

Table 5.4.: Overview of dimensions for the criticality assessment mentioned in the study.

Table 5.4 shows the occurrence count of all categories as absolute and relative values. One answer can be assigned to more than one category. For example, the answer "Long method, comparative many branches. But not very critical because feature of limited importance." for a new method counts for *change length*, *change complexity*, and *centrality*. Due to their close relation *complexity* and *length* are counted both separately and in a combined value, i.e., number of answers in which either of them occurred. Additionally, the relative counts for the categories multiplied by the respective experience and inverse experience values are shown to investigate dependencies. If the column *Relative Count (Exp.)* has a higher value than the respective cell in *Relative Count (Exp. inv.)* the category has been picked more often by experienced participants and vice versa. The values which are at least 0.03 higher then in the opposite group are highlighted.

The most-mentioned single reason with 33 occurrences is *centrality*. This is only slightly surpassed by the combined category of *change length or complexity* with 34 occurrences. Refactoring has been mentioned in 21 cases either in answers like "Only a refactoring, thus not critical" or "Not a refactoring, thus more critical". Another recurring reason is that the test gap should be closed to *avoid regression* for test gaps which are connected to a bug fix. In most cases this answer was based on the fact that the type of the birth issue was *Bug*. Only one answer specifically pointed out that defects have appeared in the respective method repeatedly. Another reason which occurred a few times is that the respondent knows or assumes that the code in the test gap is essentially *tested elsewhere*, e.g. the test gap does not contain relevant logic, but only delegates to another method or class, which makes the respective test gap less critical. There were a couple of answers which did not really fit into any of the other categories. One noteworthy example is: "I wonder why there can be an InternalServiceException here. Test cases might help make this more clear." This hints at the possible benefit of test cases as additional documentation.

The last column in Table 5.4 shows in which direction the category influences the criticality. *More* (e.g. more complex, more important, more critical) and *less* are represented by the plus and minus sign. For instance, + $\Longrightarrow$ + for *centrality* means that a more central feature is more critical. For binary values, e.g. refactoring, *True* $\Longrightarrow$ - states that if the reason is fulfilled the test gap is less critical. + $\Longrightarrow$ + always implies - $\Longrightarrow$ - and *True* $\Longrightarrow$ - implies *False* $\Longrightarrow$ +. In most cases the direction was rather clear from the answers and the corresponding criticality ratings. For two of the categories the direction was not obvious. The first one is *visibility*. The intuition might suggest that public methods, i.e., methods which are part of the interface of a class and can be used by other classes, could be considered as more critical than private methods which are only valid in a limited scope. However, there is one answer in the survey which could contradict this assumption: "Critical, since there's a new private method". It is unclear whether the respondent based the criticality of the method on the fact that it is private or that it is new, or on both to some extent. The second unclear direction is for error handling. Some participants assessed changes to the error handling as more critical than other cases, others thought of them as less critical.

The option "Don't know" was selected in 10 out of the 187 cases (17 participants estimated the criticality of 11 test gaps, thus 17 * 11 = 187). Only four of them specified a reason. One participant chose this option as he knew about the bad testability of the respective code. Other reasons were missing context knowledge, temporarily broken reference links, and unclear connection to the given birth issue.

Another interesting question in this context is how often the refactoring value which is calculated by the prototype overlaps with the manual assessments. Twenty of the 33 test gaps in the study have a refactoring value of over 0.95. These are test gaps which were introduced in an issue with a change set of close to 100 methods or above. This rather high number can be explained by the fact that issues with a large change set likely contribute more test gaps than issues with a smaller change set (assuming a similar ratio of test gaps per issue). In fact, 46 % of the test gaps in the study have a refactoring value higher than 0.95. Eleven of these 20 test gaps are newly introduced methods. Obviously, none of these have been recognized as refactoring by the participants. However, for six out of the remaining eight test gaps at least some participants mentioned keywords of the refactoring category. This yields an accuracy of 75 %.

There are two metrics, *shortly before release* and *not considered worth a review*, where no evidence towards their validity or direction was available and which were not part of the evaluated prototype. In the survey the participants were asked about their opinion about these metrics. For the first one, 65 % of participants said that test gaps which have been introduced *shortly before release* should be considered as more critical. All participants who voted against a higher prioritization essentially said that the introduction date is in general not connected to criticality or defect-proneness. Additionally, one respondent said that "well-behaved teams" should know that this phase is more critical and, thus, may only be used for low-risk changes and bug fixes. The most-mentioned reason for a higher prioritization was that changes *shortly before release* only undergo a limited testing process. In particular, for Teamscale the release process includes three RCs which are tested manually by the developers. Every change after the third RC directly goes into the release without further manual testing. That means that test gaps in these changes are not tested as they are not covered by automated tests. The only tests are possibly manual tests of the developer or the reviewer themselves. However, this argumentation only applies to projects which use a similar testing and release strategy. An additional reason which has been mentioned by three participants is that these changes might be more defect-prone due to the increased time pressure. Another comment was that these test gaps are as critical as test gaps in a published version. In contrast, test gaps which only exist on unreleased development branches could be less critical. Another general remark is that recent changes tend to be a good predictor of defects. This would imply to consider newer changes as more critical than older ones. The idea is that defects in older code would have more likely already been detected.

For the second proposed metric, the participants were asked whether they consider test gaps from commits which are *not considered worth a review* as more or less critical than other test gaps on a scale from 1 (much less critical), through 3 (the same), to

5 (much more critical). The results are presented in Table 5.5. On the one side, the reason to rate them lower is that these kind of commits are expected to only contain less critical changes. On the other side, the fact that the review process is skipped could make them more dangerous. The participants who think that these test gaps do not require any special treatment say that the criticality should rather be based on the type of change and not on how it was introduced.

Criticality	Count	Relative	Average	Variance
1 (much less)	2	0.12		
2 (less)	5	0.29		
3 (same)	7	0.41		
4 (more)	3	0.18		
5 (much more)	0	0.00		
Total	17	1	2.56	0.87

Table 5.5.: Criticality ratings for test gaps from CR#0-commits.

The most-mentioned reasons for the criticality of a test gap are *centrality*, *change length and complexity*, and *refactoring*.

Discussion

One goal of this RQ is to find out whether the reasons from the survey are appropriately reflected in the metrics which are implemented in the prototype. The most-mentioned reason *change length or complexity* has its equivalent in CLI and COC. *Centrality* conceptionally corresponds to the CEN metric, *method length or complexity* to LEN and COM, and *refactoring* to REF.

The metric PRD is related to the reason *avoid regression*. The difference is that it does not consider the birth issue but all defects which have occurred in the respective method before that. The reason that previous defects were only mentioned one time in the study is that knowing about this requires good knowledge of the respective method history. The birth issue, however, is directly available in the survey, just as in the TGA implementation in Teamscale. This is where the automated computation can be used to augment the knowledge of the developers by including this information in the prioritization and displaying it to the user. In the current implementation the birth issue is not counted towards previous defects. According to the answers in the survey this behavior should be changed.

It is interesting that the priority of the birth issue (HIP) has not been mentioned in the survey at all. This would be plausible if the presented set of test gaps only contained birth issues with normal priority. However, the set consists of only approximately 50 % birth issues with normal priority, the other half are issues with the priority *high* or *urgent*. This distribution is the same for all three versions of the survey. Thus, apparently the priority of the birth issue did not influence the decision of the participants explicitly. This result questions the validity of the metric. If it cannot be justified by further studies or related research, it should likely be removed from the set of metrics.

The validity of all metrics connected to findings (i.e. ACF, ANF, CFC, NFC, RNF, RCF) cannot be evaluated as test gaps which were available in the survey contained a rather low amount of findings. This needs to be addressed by further studies in the future.

There are a few reasons from the study which have no direct counterpart in the implemented metrics so far. These are *unexpected test gap*, *error handling*, *boilerplate code*, *new method*, *visibility*, *tested elsewhere*, *estimated defect proneness*, and *testability*. The first two can probably be considered of minor importance as they both have very low occurrence counts. It is doubtful whether it makes sense to attempt a potentially complicated formalization of the reason *unexpected test gap*. In most cases the underlying reason probably should be a different one like *centrality* ("Why is this important feature untested?") or *testability* ("Why is this not tested? Could be done easily!"). The criticality direction for *error handling* is not known yet. This could be clarified through dedicated studies which target test gaps related to error handling explicitly. It is also worth checking if any related research has already investigated the defect-proneness or criticality of such changes.

Boilerplate code could probably be detected automatically, e.g. through the use of machine learning. However, as such code tends to be rather simple, it may correlate with other metrics like the complexity. Thus, implementing this metric can be treated with a lower priority. The reason *new method* is already indirectly reflected in the higher weights for CLI and COC. One motivation of this answer could be that development teams want to encourage that all new code is immediately covered by tests. Reaching an appropriate level of test coverage on the whole system likely requires huge effort. It could, therefore, help to introduce a best practice to develop tests on new code to at least reduce the growth of untested code. It is not practical to make this a strict policy as there are often parts of the system which require a disproportionate amount of effort to test. It may make more sense to check this in the manual review process.

The reason *change length or complexity* has been mentioned in the study significantly more often than *method length or complexity*. One reason for this may be the chosen layout of the survey which prominently displays the actual change in the method since it became a test gap in the respective time interval. However, the survey layout had unlikely such a strong influence on the occurrence count. Thus, for the prioritization this could still indicate that the relation between new/changed and existing code should reflect more drastically in the weights, i.e., the weights for LEN and COM could be even smaller in comparison to CLI and COC. This leaves the question to what extent the occurrence count should reflect in the metric weights in general. The presented set of test gaps is not comprehensive enough and the number of opinions may be to small too allow generalizations. On the one hand, such clear tendencies like the aforementioned should definitely reflect in the metric weights. On the other hand, exact statements like “the weight for *change length or complexity* should be 6 times higher than *method length or complexity* according to the occurrence count” are likely not valid. Also, for example, REF and CLI do not necessarily need to have the same weight even when they have approximately the same occurrence count. One reason for this is the different normalization. CLI is normalized to the interval [0,1]. If the set contains one test gap with a huge change, all other test gaps will have an comparatively small value. Thus, for flag-like metrics, which are basically 0 or 1, the weight has much more influence.

The reasons *visibility* and *tested elsewhere* need more detailed investigation. The former, first of all, requires a clear specification of the criticality direction, the latter, a clear set of rules. The reason *tested elsewhere* is mostly used in the study as “Should rather be tested elsewhere” (e.g. if the test gap only delegates to another method) or “Less critical if tested elsewhere” (e.g. a different method with quite similar logic is tested). However, the participants did not have this information in the survey and it is hard to know from experience, especially as coverage can change frequently. Thus, these statements may often be based on assumptions. An automated detection could be used to refine the automatic prioritization and provide more information for the manual assessment process.

Even though *estimated defect-proneness* has no direct counterpart in the evaluated metrics, the fact that most of the metrics are based on proven concepts from defect prediction research should address this point appropriately. Refining the implementation of these metrics and augmenting the set by additional metrics can improve the automatic prioritization in the future.

Testability might not be trivial to detect automatically. A semi-automatic strategy where components with good and bad testability are manually selected beforehand could work. But it needs to be investigated whether the benefits from this would justify the additional maintenance effort. The same strategy could be applied for critical components, i.e., components where a failure would have more drastic consequences. Both of these strategies would help to rank accepted test gaps lower, e.g. code that is known to be hard to test or that is part of a non-critical component.

The strategy to manually select critical components could alleviate the difference between the computed and perceived metric values. In the study, the importance or centrality of a test gap which is stated by participants is often not reflected in the CEN value which is calculated in the prototype. As described in the previous chapter, the

CEN metric is in the current implementation based on the RageRank centrality of the file inside the system. This may be refined in the future to better represent the actual importance. However, it has to be noted that the *centrality* perceived by participants may be highly subjective. Consequently, the manual assessment may not be the perfect oracle. The collective wisdom of experienced developers should, nevertheless, be a decent approximation. Further studies can be conducted and results from related work included to enable a better formalization of *centrality*. As a fully automated approach like this is likely not trivial, the manual selection of critical components could be an attractive compromise. The correlation between perceived and computed value should also be tested and refined for other metrics in the future. For instance, McCabe's cyclomatic complexity which is used for COM and COC is not necessarily equivalent to the participants' understanding of complexity.

One goal of this RQ was to find out whether there were metrics which contribute to the criticality of test gaps and are not directly connected to expected defects. The results show that for most reasons from the study this is not the case. However, they do not only target the expected number of defects, but also the potential severity especially in the often-mentioned reason *centrality*. The only reason which does not seem to directly refer to expected defects is *testability*.

The study results provide interesting ideas for the refinement of the REF metric. They indicate that using the refactoring threshold only on modified methods rather than on new methods could be a decent predictor of refactorings. This tendency should be evaluated in future studies. As the change set itself is a proven defect prediction metric, it might make sense to split the REF metric into one for the size of the change set (i.e. larger change sets have a lower defect density) and another which describes the probability that the change is a refactoring. If reliable refactoring detection is available, those test gaps can be ignored in the TGA completely as true refactorings do not change the functionality of the code and, thus, cannot introduce defects.

The combination of the occurrence count with the experience values also provides some valuable insights. There is a rather clear tendency that more experienced participants more often justify their rating with the importance or centrality of the feature. Developers with less experience may not have that kind of knowledge and rather base their decision on the metrics which can be derived from the test gap itself like length and complexity of change and do not require any context knowledge. This reason should also explain the higher values for *testability* and *tested elsewhere*.

Finally, the results for the newly proposed metrics *shortly before release* and *not considered worth a review* should be discussed. It is true that the commit date should in general not be connected to the criticality. However, due to the limited testing process for changes which are introduced *shortly before release* such test gaps should likely receive additional attention. One possibility would be a dedicated TGA session directly before the release to analyze all test gaps which have been introduced after the last RC. If this uncovers any critical untested changes, it could be considered to do additional testing. In case of the discovery of critical defects the release might be delayed until these are fixed. However, in practice a delay of the release might often not be possible. But even then the addition of this metric should be beneficial. An example is a scenario where two basically equivalent test gaps are discovered after the release. The only difference is that one of them has been introduced a few weeks before the release, the other one a few days. The first one has already been included in the test builds of the system for multiple weeks and has also been part of the dedicated release testing, the second one has not. Thus, it makes sense to view the second test gap as more critical. Critical defects in releases which are not fixed quickly have a bad influence on the company's image. Detecting these early should, therefore, be a priority to enable a quick reaction with hotfixes. The metric *shortly before release* can help to prioritize those test gaps which have a higher probability of defects. One counter-argument in the study was that development teams should know about this risky phase before the release and only merge non-critical changes. However, the proposed metric is still only one of many influences on the criticality score. A single metric does not have such a strong effect. Consequently, the addition should not have any significant disadvantages. The answers also hint towards adding another metric which rates recent changes as more critical. The rationale is that defects in older changes have already been detected with a higher probability as they have been tested for longer.

The results for the metric *not considered worth a review* are not really clear even though they show a slight tendency to treat them as less critical then other changes. It is probably true that on average these are not very complex. However, there does not seem to be the need to introduce a dedicated metric for this. Trivial changes will be ranked lower with the existing set of metrics already, regardless of how they have been introduced. It should be investigated whether those commits produce more or less defects than other commits. Those results might give more insights about this question. Nevertheless, with the current results an additional metric does not seem necessary.

5.5.3. RQ2: Variance of Manual Criticality Assessments

Results

The variance is a measure to describe the deviation of a random variable from its mean. For this RQ the random variable is the assessment of test gaps on the criticality scale in the survey. The minimum possible variance is 0, i.e all participants choose the same criticality value, the maximum possible variance is 2.7, i.e., half of the participant choose the maximum and the others the minimum value (assuming five or six answers per question). None of these extreme values occurred in any of the cases in the study. The actual range is between 0.17 and 1.77 with an average variance of 0.71. An overview of the cumulative distribution of variance values across ranges for all test gaps can be found in Figure 5.4.

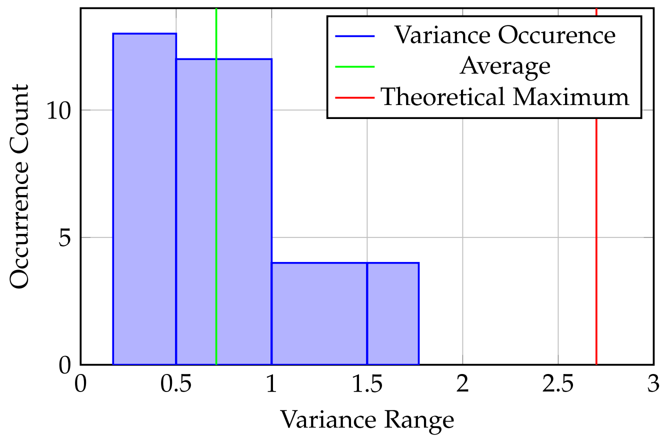


Figure 5.4.: Cumulative occurrence count for different criticality variance ranges.

Number of different assessments	Adjacent	Absolute	Relative
1	Yes	0	0.00
2	Yes	13	**0.42**
2	No	4	0.13
3	Yes	8	0.26
3	No	4	0.13
4	Yes	2	0.06
Total	-	31	1.00

Table 5.6.: The number of different criticality assessments per test gap.

Table 5.6 shows how many different values on the criticality scale were chosen by the participants per test gap. The value 1 means that all respondents agree on one criticality assessment. The value 4 means that every entry on the criticality scale has been chosen by at least one participant. The second column provides information whether the individual assessments are adjacent (e.g. 1 and 2) or not (e.g. 1 and 3). Adjacent values implicate a clearer tendency towards one part of the criticality scale. In 42 % of the cases, the answers were distributed about only two adjacent values of the scale. In the remaining cases, the answers were non-adjacent and/or distributed about more than two values.

In the survey 33 test gaps have been assessed. The number of assessments per test gap differs between three and six. There are two reasons for this. The first one is the division of questions into three separate surveys. Each version was filled out by a different group of people. Due to the varying response rates among the groups, there are six participants for two versions and five participants for the other one. The second reason is that the assessment allowed the option "Don't know". Thus, in cases where one or more participants chose this option the number of valid answers is smaller than the respective number of participants. However, Cohen's κ expects the same number of answers for each question. Therefore, it was necessary to divide the test gaps by the respective number of answers. Three and four answers only appeared in one case. As the formula is not defined for sets which only contain one test gap, these two questions are ignored in the following computations. Five answers appeared in 16 cases, six answers in 15. The average Cohen's κ of both sets weighted by the number of assessed test gaps is 0.18. An overview can be seen in Table 5.7.

Number of answers	Number of test gaps	κ
5	16	0.13
6	15	0.27
Total	31	**0.18**

Table 5.7.: Results of Cohen's κ for the assessments in the study.

The interpretation of this value is not clear and different propositions exist. Landis and Koch [LK77] divided the value range into six sections, Fleiss et al. [FLP13] only used three. The exact values can be found in Table 5.8. Thus, the value of 0.18 for this study can be interpreted as either good agreement according to Landis and Koch or poor agreement according to Fleiss et al.

Cohen's κ range	Agreement (Landis and Koch)	Agreement (Fleiss et al.)
< 0.00	Poor	**Poor**
0.00 - 0.20	**Good**	
0.21 - 0.40	Excellent	
0.41 - 0.60	Moderate	Good
0.61 - 0.75	Substantial	
0.76 - 0.80		Excellent
0.81 - 1.00	Almost perfect	

Table 5.8.: Different interpretations of Cohen's κ in literature.

The high variance is likely not caused by the different experience levels among the participants. The variance of the assessments of only participants with an experience value of 0.6 or higher is with 0.73 even slightly bigger than the overall variance.

The criticality estimations of test gaps show high subjective variations.

Discussion

The results indicate that it is not easy to assess the criticality of test gaps objectively. The previous RQ (Section 5.5.2) has shown that the decision may be based on a variety of factors. These might influence the final decision to a different extent for every individual. Metrics which are very important for one person may not play any significant role for the decision process of the next and vice versa. Hence, besides knowledge and experience, also personal preference can affect the assessment. On the one hand, this reflects in rather high variance values in some of the test gaps. However, on the other hand, there also exist enough cases where the participants showed a rather clear tendency towards certain criticality. This is a strong hint against pure randomness.

Cases with high variance seem to occur in particular when the participants focus on different properties of the source code. A common example is a rather trivial change in a method which however serves some potentially important role in the application. In these cases some participants assessed the criticality based on the whole method and not the criticality of the actual change. Recognizing the importance of functionality does not only require a certain knowledge and experience, but may also be partly subjective. Developers often specialize in some parts of the software. So even when they are experienced, they might not be able to evaluate the importance of methods in all parts of the code. Also they could tend to assess methods in their own components as more crucial. Participants who do not know the importance of the test gap can focus on metrics like complexity, which can be derived from the given code without further context knowledge, or need to guess, e.g. the method name sounds like it is connected to core functionality.

There are other examples for the subjectivity of the criticality assessment. One test gap in the study consisted of changes to the exception handling in a method. Some participants argue that error handling is crucial as the application should prevent the user from fatal errors under any circumstances. Other ones say that the priority should lie on the regular, non-error cases as these usually outweigh the cases which require error-handling in the user experience. Another test gap with high criticality variance was the replacement of a parameter by the value *null*, i.e., a representation of a reference which does not refer to any valid object, in an otherwise trivial method. Most respondents assessed the criticality as low due to the trivial change. Only one participant noted that the value *null* might lead to errors if the called method does not handle this special value properly.

Due to the different propositions for the interpretation of Cohen's κ there is no clear result. In general, a value of 0 equals the same agreement as random assessments. The result of 0.18 for this study indicates that the observed agreement is with high certainty more than mere coincidence. However, on both scales the value is rather low and far from the value 1 of optimal agreement.

Another factor which likely contributes to the variance is the size of the criticality scale. The agreement is expected to be inversely proportional to the size of the criticality scale as the probability that two participants select the same value on a 4-point scale is lower than on a binary scale. For this study, the 4-point scale has been chosen as compromise between high possible agreement and fine gradation which is needed to allow a ranking of test gaps.

The results for Cohen's κ and the variances reflect that manual criticality assessment is prone to subjective variations. The assessment may differ to a significant extent between multiple persons. This effect is expected to be proportional to the size of the system as it may be easier to consider and have knowledge about most relevant factors in smaller applications. For larger systems the use of collective wisdom may mitigate this issue. In practice, asking multiple people might not be possible due to limited resources and organizational overhead if it is not supported appropriately. Thus, the TGA results are often analyzed by single individuals and important test gaps might be missed. An automated approach which provides results similar to the average manual assessment could alleviate the issues of limited resources and missing critical test gaps.

5.5.4. RQ3: Comparison to Manual Assessments

Results

In this section the results of the prototype are compared to the manual assessments of the participants in the study. For the evaluation, these need to be normalized to the same range of numerical values. In the survey the participants had a 4-point Likert scale where 3 refers to high criticality, 2 and 1 to medium criticality with a tendency to high and low, and 0 refers to low criticality. This allows to compute one criticality score for every test gap from the average of manual assessments. Figure 5.5 shows how often the different criticality assessments have been chosen in the survey. *NA* refers to the answer "Don't know".

The chosen set contains test gaps with high, medium, and low criticality according to the prototype. An overview of the different mapping and normalization strategies from these categories to the range [0,3] is presented in Table 5.9. The first one is *discrete*. This

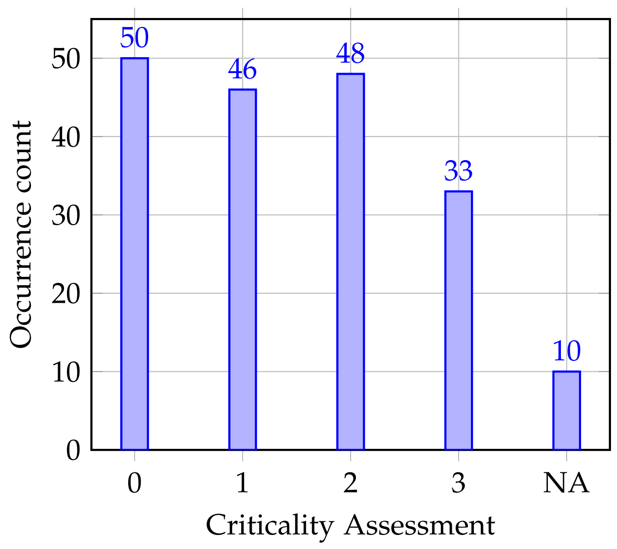


Figure 5.5.: Occurrence count for different criticality assessments in the survey.

assigns the value 3 for all test gaps in the high criticality category, value 1.5 for medium criticality, and value 0 for low criticality. The other two strategies are normalizing the *rank* or the *criticality score* to the range [0,3] using the formulas given in the table. The variable r refers to the rank in the prioritized list. In the first part of the study, the list contains 579 entries starting from rank 0. Thus, the test gap with the lowest score has rank 578. This equals $max(R)$ with R being the set of all ranks. The variable c is the criticality score. Accordingly, C is the set of all criticality scores. The highest score in the study is 4.6, the lowest one -0.86 (negative values are possible for some metrics).

Criticality		High	Medium		Low
Survey		3	2	1	0
Prototype	Discrete	3	1.5		0
	Rank	$3 * (1 - \frac{r}{max(R)})$			
	Criticality Score	$3 * \frac{c - min(C)}{max(C) - min(C)}$			

Table 5.9.: Different mapping strategies for the criticality score.

It has to be decided which of the three strategies is the most appropriate one for the evaluation of the prototype. Given the fact that in RQ4 the results will be compared to other prioritization strategies, the *discrete* strategy may not be the most suitable one. It is in general not possible to map the results of other strategies to the discrete values as the selected test gaps will likely fall somewhere in between of the categories of the prototype. Additionally, the test gaps have been selected by medium rank and not medium criticality score. The medium value for the *discrete* strategy would need to be adjusted to 0.84 which is the actual average criticality score of the test gaps from the medium criticality group. *Rank* or *criticality score* can be used regardless of the strategy. The problem with *rank* is that the criticality difference between the ranks is not equivalent. For instance, the criticality score difference between rank 0 and rank 1 is with 0.38 rather large, while other adjacent ranks often have the same criticality score or only a minimal difference. Thus, to facilitate a comparison with other prioritization strategies the normalization strategy of the *criticality score* is used.

For a first evaluation of the accuracy of the automatic prioritization, a threshold needs to be selected which indicates an appropriate fit for automatic and manual assessment. The initial goal is to beat the *random* strategy. This is a representation of a manual TGA session where no indication about the criticality is available initially. That means the assessor has to randomly pick an order to analyze the test gaps. The average deviation of a random assessment in relation to the criticality score is shown in Figure 5.6 (assuming a uniform distribution).

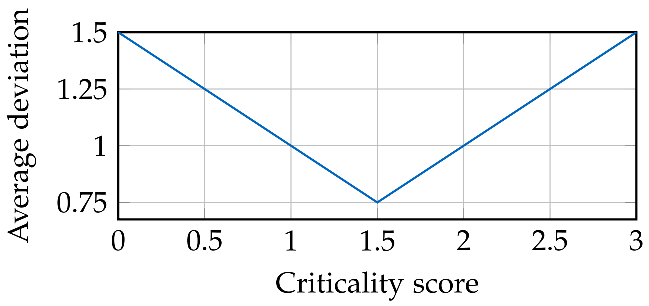


Figure 5.6.: Average deviation of random assessment for different criticality values.

Hereafter, deviations which are below the lower bound of the random deviation are considered as good fit. This lower bound is a fourth of the criticality score value range, i.e., $3/4 = 0.75$ in this case. The average deviation of the *random* strategy for the test gaps in the study is 1.08. A value which is lower than 1 is still considered an acceptable fit. This has been chosen instead of 1.08 for simplicity reasons as it does not make any difference for the test gaps in the study. Every deviation which is bigger than 1 (thus, not better than the average random assessment) is considered a bad fit. Another strategy which is used as additional benchmark is one which always returns 1.5 as criticality score. In the following this is called the *pseudo* strategy as assigning the same value to all test gaps is no real prioritization. The value 1.5 is selected as it has the lowest average deviation.

Figure 5.7 plots the assessments of the prototype against the average assessment from the study for all test gaps. Test gaps where the difference between the criticality score from manual and automatic assessment is not larger than 0.75 are displayed in green. Deviations above 1.0 are displayed in red. Test gaps in the area between red and green receive a yellow marker.

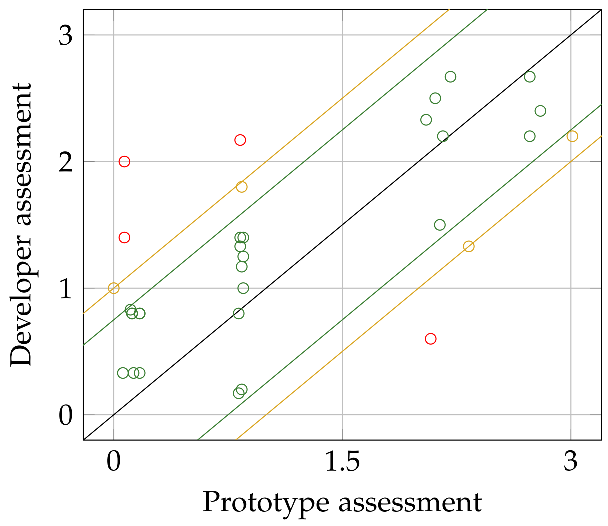


Figure 5.7.: Plot of the assessments of the prototype against the average manual ones.

Figure 5.8 shows the number and percentage of outliers for different deviation thresholds. For 0.75 the number of outliers is 8 (about 24 %), for 1.0 the number is 4 (about 12 %).

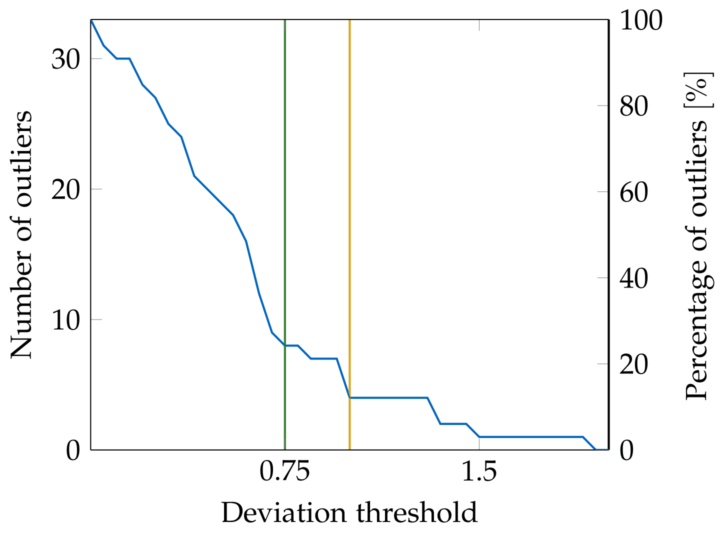


Figure 5.8.: Number and percentage of outliers with different deviation thresholds.

The exact number of test gaps in each criticality and deviation category can be found in Table 5.10. For the *pseudo* strategy the average deviation in the study is 0.65. The value from the prototype is with 0.62 slightly below this. Additionally, for the *pseudo* approach the number of critical outliers (red) is higher (six instead of four). The number of fitting test gaps is accordingly lower by two, the number of yellow outliers is the same.

Criticality	Green	Yellow	Red	Total
Low	8	1	2	11
Medium	9	1	1	11
High	8	2	1	11
Total	25 (76 %)	4 (12 %)	4 (12 %)	33 (100 %)

Table 5.10.: Number of outliers for the different criticality and deviation categories.

For this RQ, it might also be interesting how the results are connected to the experience value. For that purpose, the average deviation of the prototype assessments from the values of the more experienced participants, i.e., experience value over 0.6, was computed. This is with 0.67 slightly higher than the deviation for all participants.

In the second part of the study, the prototype is tested on further software systems and the monthly assessments are used as oracle. The evaluation on the study objects for the second part was performed with a reduced set of metrics as less information was available. In particular, there was only limited access to the issue tracker. Especially the metrics PRD and HIP did not contribute anything to the criticality score and data for REF was only available in a few cases.

The monthly assessments only point out which test gaps are considered critical. This list may not include all relevant test gaps and non-critical test gaps are not mentioned. Thus, the goal of the prioritization is that the average position of the critical test gaps is as high as possible or at least higher than the expected value. Depending on the total number of test gaps t and the number of critical test gaps c the best and worst possible average ranking are different in every case. The average rankings are normalized based on the best and worst possible value to facilitate the comparison between results.

The best possible value is computed as follows (simplified using the sum formula $\sum_{i=1}^{n} i = (n * (n+1))/2$):

$$\frac{\sum\limits_{i=1}^{c} i}{ct} = \frac{c(c+1)}{2ct}$$

This describes the case where all critical test gaps are on the first ranks. For example for $c = 3$ and $t = 7$: $(1/7 + 2/7 + 3/7)/3 = 0.29$

The worst possible value is computed with this formula:

$$\frac{\sum\limits_{i=t-c+1}^{t} i}{ct} = -\frac{c - 2t - 1}{2t}$$

This means that all critical test gaps are on the last ranks. For example for $c = 3$ and $t = 7$: $(5/7 + 6/7 + 7/7)/3 = 0.86$.

With these formulas the average ranking can be normalized to the interval of possible values. After that every result will be in the range [0,1] with 0 being the best possible result and 1 being the worst possible result. The expected value is 0.5. The average result for the automated prioritization is 0.26. Thus, it beats the expected value.

The results of the automated prioritization are mostly comparable to the manual assessments. There are some outliers, but the performance is better than the random prioritization in all cases.

Discussion

An important foundation of the manual assessments in the survey was the choice of the criticality scale. The 4-point Likert scale was chosen as compromise between gradation and usability. Some participants noted that this scale was not intuitive for them. Mental models of criticality might differ between individuals. What one person interprets as criticality 1, could mean the same as criticality 2 for another person. Thus, they would have expected a clear explanation of the steps on the scale. A suggestion of one participant was the following: *0* means needs not be tested at all, *1* means a one-time manual test would suffice, *2* means should be covered by an automated test, if possible, a manual one otherwise, and *3* means should be covered by an automated test by all means. However, the author decided not to introduce such a clear explanation during the study to not distort the remaining results compared to the previous ones. Instead this rather general explanation was added: "The criticality scale reflects the priority for closing the test gap before the release from lowest (0) to highest (3)". This should provide some clarity while not forcing a concrete mental model onto the participants. Another point regarding the criticality scale is that it might be beneficial if the scale would contain as many categories as there are categories chosen from the prioritization. For this study this would have meant a 3-point scale to reflect the three categories low, medium, and high criticality. These points should be considered for future studies. However, they are not expected to have a drastic influence in this study so that the results are still valid.

The accuracy of the results of the prototype compared to the manual assessments seems rather good: 76 % of test gaps are considered as good fit. Test gaps which are rated too high by the prototype can be considered as false-positives, ones which are rated too low are false-negatives. In Figure 5.7 on the top-left are the false-negative and on the bottom-right the false-positives. In total the automatic prioritization produced five false-positives and three false-negatives.

Table 5.11 shows an overview of all outliers including the manual criticality assessment c_m, the criticality assessment of the prototype c_p, and the difference of both $d = c_p - c_m$. That means a negative value for d indicates a false-negative, a positive value a false-positive. On the one hand, the last two columns show the metrics which are mainly responsible for the rating of the prototype and, on the other hand, the reasons which are mentioned by the participants in the survey. The outliers are sorted by severity from highest to lowest. The idea is that the outlier is the more severe, the higher the difference is and false-negatives are more severe than false-positives. The foundation of this assumption is that false-negatives have a higher potential to cause problems in the future. False-positives are less likely to produce defects than false-negatives. Thus, false-positives only require a one-time manual analysis, false-negatives

		c_m	c_p	d	Metrics (Prototype)	Reasons (Manual)
False-negatives	1	2.0	0.07	-1.93	One-liner, REF	Centrality
	2	2.17	0.83	-1.34	medium CLI/COC	Centrality, new method
	3	1.4	0.07	-1.33	REF, low CEN	Centrality, estimated defect-proneness
	4	1.0	0	-1.0	RCF	Centrality, error handling
	5	1.8	0.84	-0.96	Only small change	Centrality
False-positives	6	0.6	2.08	1.48	high LEN, HIP, PRD	-
	7	1.33	2.33	1	HIP, highest LEN, medium CLI, PRD	Testability
	8	2.2	3.0	0.8	high CLI/COC, ANF	-

Table 5.11.: Overview of outliers and respective reasons sorted by severity.

might contain defects which are detected only a while later. The effort for fixing the defect are much higher than the effort of the manual analysis of a false-positive.

The discussion of the first RQ already addressed the difference between computed and perceived metric value. This gap seems to be especially large for the *centrality* (or feature importance). All false-negatives are test gaps which have been assessed as rather central or important code by at least some of the participants. This centrality is not properly reflected in the CEN metric value. Refining this metric should be a primary goal of the further development of the approach. Other reasons which were mentioned for the false-negatives are *estimated defect-proneness*, *error handling*, and *new method*. The first one can be addressed by incorporating more defect prediction metrics. In this particular instance, the estimated defect-proneness refers to the use of the *null*-value which might cause exceptions if not handled correctly. It should be investigated whether it makes sense to let this influence the criticality score. *Error handling* has an unclear criticality direction based on the results from the study. A review of related research or future studies are necessary to clarify this before it can be incorporated in the approach. *New method* should rather be addressed by an adjustment of the metric weights. The first two false-positives in the table both have high values for the LEN metric. However, the length of the reference method did not play a significant role in the manual assessments. Even though the change metrics already have a higher rate than the metrics which belong to the reference method, the difference should likely be increased. This is in line with the occurrence count which was discussed in

RQ1. The validity of the HIP metric should be rethought. If it cannot be justified by empirical evidence, it should be removed due to the fact that it is completely missing from the reasons in the manual assessments. The same is true for the metrics which are connected to findings. On the one hand, these apparently did not influence the answers in the survey. On the other hand, defect prediction capabilities have been proven for code smells, i.e., characteristics of source code which indicate a deeper problem [Pal+16]. Only a subset of findings refers to code smells. The other ones should probably not influence the criticality score. For instance, findings which only concern the documentation can possibly be ignored. Other metrics like the size of the change set (REF) and the number of previous defects (PRD) are likely not known by the participants and should be kept as part of the algorithm.

As first benchmark the *random* and *pseudo* strategy have been selected. A question might be whether it is appropriate to assume a uniform distribution for the former. Any distribution which has a higher probability towards the center of the value range, i.e., closer to the *pseudo* strategy, would result in a lower average deviation. Thus, for that purpose the *pseudo* strategy is used as extreme edge case of the random strategy, i.e., 100 % probability for 1.5. Additionally, the lower bound of 0.75 is selected as benchmark as the most challenging value to beat for the prototype. The results show that the proposed approach yields a much lower average difference compared to the *random* strategy. In 76 % of cases it also beats the lower bound.

The *pseudo* strategy is a rather theoretical construct which has no actual use in practice. The average deviation is good in many cases as the value has the lowest expected deviation. It is optimal for the cases where all test gaps have medium criticality. For a set of only very critical and/or very uncritical test gaps it would have a very high average deviation of 1.5. However, in this case no prioritization is necessary anyway. As soon as it contains one critical test gap it is basically worthless. The proposed approach, however, can detect critical and uncritical test gap with decent accuracy according to the results of the study. On the one side, *randomly* picking an order in which the test gaps are analyzed manually may be appropriate for small sets when the effort of evaluating all test gaps is limited. For larger sets, on the other side, the sophisticated approach is recommended. It decreases the risk of missing critical test gaps when the effort to analyze the complete set is disproportionate and only the first *n* entries on the prioritized list are assessed manually.

Besides that, the automated approach can also help to overcome the limitation of missing knowledge in manual assessments. Multiple participants mentioned that they found it hard to assess the criticality as they might be unaware of the context or other important information like if the test gap is *tested elsewhere*. This again underlines that the manual assessments are not the perfect oracle. They are a valid target for the automated approach upto a certain point. But deviations are accepted and may even be desirable if the prototype uses additional information, which are not known to the participants, in a way which is based on solid empirical evidence, e.g., metrics which are known to be reliable defect predictors. These additional information should later be displayed in the UI to make the prioritization transparent to the user. Furthermore, filtering and sorting functionality can be provided to increase the flexibility and customizability of the prioritization.

The missing knowledge may be partly related to the person's experience but even seasoned developers cannot know every detail of a software product, its change history, and testing process. The results show that the average difference between assessments of the prototype and the participants was slightly higher when only the answers of the experienced participants were counted. As discussed in RQ1, they mentioned the *centrality* much more often than less experienced ones. Thus, the higher difference can be explained by the bad match of perceived and computed value for *centrality*.

For the second part of the study, a reduced metrics set was used due to technical reasons. The automated approach was still able to achieve good results and beat the *random* strategy. This may be one of the advantages of the proposed algorithm. It works also when not all metrics can be computed—additional data might still improve the results. It should be subject to future work to eliminate metrics which do not actually contribute positively to the results and replace them with better ones.

5.5.5. RQ4: Comparison to Trivial Strategies

Results

For the previous RQ the benchmark was the *random* strategy which can be applied in practice when no information about the test gaps is available. If the developer has access to some information, other strategies are possible. Obvious ones are to prioritize the set by single or a few combined metrics. The sophisticated approach which combines all metrics is only justified if it outperforms these trivial strategies. Table 5.12 shows the results of multiple strategies on the study object Teamscale. The strategy *all* refers to the proposed approach including all metrics. The other evaluated strategies are single-metric strategies of all metrics which are used in *all*. COC is the only metric in the current set which can contribute a positive or a negative value to the criticality score. The actual metrics value are in the range [-0.5,1]. Consequently, the criticality score is normalized from this range to [0,3] which is used in the survey.

Usually the metrics for RNF and RCF would need to be inverted. The assumption in this work is that removing findings makes the change less critical. However, for the given study objects the inverted values for these metrics produced even worse results. This can be explained by the fact that removed findings only occur in two out

	Score	Unique	Average	Variance	Best	Worst
All	0.03	23	0.62	0.19	0.02	1.93
CLI+LEN	0.04	21	0.74	0.31	0.01	1.98
CLI	0.04	17	0.73	0.22	0.05	1.96
LEN	0.08	14	1.16	0.71	0.00	2.67
COC	0.07	7	0.50	0.16	0.00	1.50
Findings Only	0.14	5	0.68	0.20	0.00	2.08
COM	0.15	7	1.10	0.67	0.03	2.67
REF (inv.)	0.2	6	1.17	0.49	0.17	2.74
CEN	0.22	6	1.31	0.58	0.14	2.64
PRD	0.31	4	1.23	0.56	0.17	2.67
ANF	0.43	3	1.30	0.48	0.20	2.67
HIP	0.57	2	1.13	0.47	0.17	2.67
RNF	0.66	2	1.32	0.61	0.17	2.67
Pseudo	0.67	1	0.67	0.15	0.00	1.33
RCF	0.70	2	1.39	0.60	0.17	2.67
ACF	1.36	1	1.36	0.59	0.17	2.67
CFC	1.36	1	1.36	0.59	0.17	2.67
NFC	1.36	1	1.36	0.59	0.17	2.67
Random	1.08	-	-	-	-	-

Table 5.12.: Results of different strategies for Teamscale.

of the 33 test gaps. That means, in the inverted metric value, every test gap without removed findings has the highest criticality value. The other metric value which is interpreted to contribute inversely to the criticality is REF. In this case, the inverted value is in fact significantly lower than the non-inverted value. There are two combined strategies *CLI+LEN* and *findings only*. Both have the weights which are used in the full metrics set. Finally, also the results of the *random* and *pseudo* strategy are provided for reference. The table shows for all strategies the average deviation, the variance of the deviation, and the best and worst deviation result. During the evaluation it appeared that the average deviation may not be a suitable measure to solely rate the performance of the different strategies. In this work the goal is prioritization and not classification. Therefore, strategies which do not provide an actual ranking are less interesting. All metrics which only check for a binary property essentially provide a classification as only two criticality scores exist. The *pseudo* strategy does not even provide a classification as all test gaps receive the same value. In the study, the same is true for the strategies ACF, CFC, and NFC as no critical findings have been added and no findings (neither critical nor non-critical) exist in changed code for any test gap. For that reason, the table also shows the number of unique criticality values. The average deviation and the number of unique values are combined into a performance score as *average/unique*.

With this performance score the strategy *all* shows the best result. However, it is closely followed by the strategies *CLI+LEN* and CLI. The strategy *all* also shows a very good result in the average deviation compared to the other strategies.

The result of all strategies for the remaining study objects is presented in Table 5.13. The average ranking value is again normalized to the range [0,1] based on the best and worst possible result as described for RQ3. As already mentioned, the study had to be performed with a reduced set of metrics for the remaining study objects. Thus, the strategies PRD, HIP, REF (due to missing issue tracker connection), and all strategies connected to findings (too little findings data available) are not evaluated in this part. The benchmark is again the random prioritization with an expected average deviation of 0.5. The strategy *all* also shows the best results in this case, again closely followed by CLI. All remaining strategies are not significantly better than the expected value.

	All	CLI	COC	LEN	CEN	COM
Average	0.26	0.28	0.41	0.45	0.47	0.50
Variance	0.05	0.04	0.10	0.10	0.07	0.09
Best	0.00	0.01	0.00	0.00	0.15	0.01
Worst	0.67	0.56	0.91	1.00	1.00	1.00

Table 5.13.: Results of different strategies in the second part of the study.

The proposed approach outperforms the other strategies in both parts of the study. CLI is the best single-metric strategy, combining it with LEN only yields minimal improvement.

Discussion

One interesting fact is that the proposed approach only produces slightly better results than the CLI strategy in both parts of the study. However, the results of the study yield many ideas which can be implemented to further improve the strategy in the future. After that the advantage of the sophisticated approach should be more significant.

The strategies COM and LEN perform rather poorly in both parts of the study. The reason is that the metrics value is 0 for all new methods as their reference method is empty. Thus, they are both bad as single-metric strategies. The same is true for all metrics which only contribute to the criticality score in rare cases. In that sense, the results of the strategy *findings only* are similar to the ones of the *pseudo* strategy. In combined metrics sets they can still make valid contributions in some cases while the heavy lifting is done by good single-metrics like CLI. It would have been possible that adding LEN and CLI might be promising as representation of the size of the method after the change. But the results show that adding LEN to CLI does not really yield better results than using CLI alone.

At first glance it seems strange that the COC strategy performs unexpectedly well in the first part of the study. One reason might be that it is the only metric which can contribute both positive and negative values to the criticality score. The range of the metrics value is [-0.5,1]. That means the highest complexity reduction is half the size of the biggest complexity growth. No change of complexity happened in 13 of the 33 test gaps. Due to the normalization from [-0.5, 1] to [0,3] these have the criticality value 1.0. Since this is rather close to the criticality values of the medium criticality group of test gaps (~0.84), there are only two outliers among these 13. This obviously reduces the average deviation significantly. COC only produces seven distinct values in this case which is rather low for a real prioritization. This is due to the fact that McCabe's complexity measure only contains ten steps in total. Thus, there are only so many unique values possible. In the second part of the study it is the third best strategy but with a significant distance from place two. The performance of COC as single-metric strategy could be evaluated on further study objects to get a better understanding. But the current results indicate that it is a valuable contributor to the combined strategy.

Another interesting results is that REF performs better than some other single-metric strategies. This is likely connected to the fact that larger change sets usually have a lower average of length and complexity per change. Smaller change sets often contain more complex changes.

CEN has a rather poor performance as single predictor. The problem is that there are very few test gaps which have a very high centrality score. Normalization leads to the situation that most test gaps have a CEN value close to 0. This should, on the one hand, be addressed by experimenting with different centrality measures. On the other hand, it might be possible to replace the linear normalization by one which distributes the test gaps more evenly on the range [0,1].

In one monthly assessment a whole class was recognized as untested. This could be a case where grouping would make sense, i.e., all test gaps in the same class are grouped. As these are likely related, it makes sense to present them to the developer together. A similar example are multiple test gaps inside of one function group in ABAP.

5.6. Threats to Validity

This section reflects on the possible threats to the internal and external validity of the results from this study and explains how they have been addressed.

5.6.1. Internal Validity

The internal validity describes whether the methodology and structure of the study allow confidence in the presented conclusions.

Prototype Implementation

The validity of the results depends on a correct implementation of the approach in the prototype which is the basis for the evaluation. Just as any software, this may contain defects which distort the results. The prototype uses the TGA implementation and other features of Teamscale. Many of these have been used by customers on a variety of software projects across different domains and programming languages for multiple years. Hence, the risk of relevant defects is comparatively low as they would likely have been detected and repaired already. During the development of the prioritization approach, the implementation was regularly tested on the study object Teamscale with a set of approximately 600 test gaps. This should be large enough to cover the most important edge cases. The author constantly checked the results for validity so that detected errors could be immediately corrected in the implementation. For these reasons, the risk of major defects is considered to be minimal.

There are a few known issues which either were accepted due to their rare occurrence or mitigated through specific workarounds. The first one was that for a small number of test gaps the method history contained predecessors which have no apparent connection to the method. Consequently, the computation of metrics which use the reference method was likely incorrect in these cases. In the first part of the study, any encountered occurrences have been ignored for the survey, in the second part only 1 out of over 300 test gaps was observed with this behavior. Consequently, the influence on the results should have been limited.

A workaround in the implementation was necessary to retrieve the birth issue for some cases. The Teamscale repository is in general configured to enforce commit messages to contain an issue reference in a given format, i.e., "CR#<issue number>". This check is not applied for merge commits which, therefore, can contain improperly formatted references, e.g., "Cr#123", "Cr/123" and "CR #123". This required the adjustment of the recognition pattern to allow these variations. However, in rare cases this led to the situation that in one commit message multiple issue references were detected, such as "Merge branch 'cr/111_test' into 'master'; CR#123 Migrate service". In this example, the branch name suggests a different issue id (111) than the actual commit message (123). As the number of improperly formatted issue references was far greater then the number of multiple issue references in one commit message, they were accepted for this study.

Another special treatment was necessary to retrieve the birth issue for merge commits which do not contain any issue reference. This is the case, for example, for the merge of release branches like "Merge branch release/v1.0.x into master". As the method history does not follow the method across branches, it was not easily possible to retrieve the birth issue for a test gap from a different branch. The implemented workaround was to extract the branch name from the commit message, i.e., "release/v1.0.x" for the example above, and then search the method history on this branch for commits which have an issue assigned. If another merge commit is encountered, the process is repeated. The method history depends on the character offset of the method inside the respective file. This offset is used to find the method on the other branch at the time of the merge target commit. Hence, this procedure does not work if the file has been changed on the other branch since the origin of the merge commit.

Due to its rare occurrence fixing these cases had low priority for the study and should not have had any significant influence on the results. However, in the future a cleaner implementation should be added.

Test Gap Data and Manual Assessments

The following paragraphs discuss the validity of the data of test gaps and manual assessments which is used in the different parts of the study.

The data needs to contain an appropriate balance between critical and uncritical test gaps. This is especially important as the approach estimates the relative criticality in the respective set. As discussed, there is no universal definition of criticality. Therefore, there is no objective way to assess the validity of the set in this regard. However, a manual analysis showed that the first study object contains a wide variety of test gaps, including several which can be considered as complex, as well as trivial ones. For the second part, only monthly assessments have been selected which mention both critical and uncritical test gaps explicitly or implicitly. The manual selection of test gaps is not expected to have introduced a bias. In particular, only those which were considered as duplicates due to their close relation to other test gaps were ignored for the survey. The distribution to the three versions of the survey was performed randomly with a manual verification that all of them received a balanced set of test gaps.

The evaluation which was performed by the author, e.g., the extraction and categorization of the essence of assessment reasons for RQ1, may, of course, be subject to human error. It was, however, performed to the best of his knowledge with multiple passes to double check the validity of the final result so that the amount of errors should be minimal.

Another threat to the validity of the results may be the different interpretations of the criticality scale. As there was no concrete explanation of the values on the scale, the participants might have built different mental models for themselves. Nevertheless, it is unlikely that these variations had major influences on the overall results. Additionally, only two out of the 17 participants actually voiced these concerns. A related issue are explanations which have been added to the survey in response to early feedback after the study had already started. The author took great care to not change the meaning of the survey's contents, but to only provide more details to prevent uncertainties. Therefore, these should not have threatened the validity in any way.

Similar to the fact that there is no objective definition of criticality, there is no optimal prioritization which can be used as oracle for the automated approach. In the first part of the study, the target for the criticality values is the average of the manual assessments from the survey. On the one hand, the participants might not know all the information which could influence the criticality, such as the number of previous defects. This has, however, been addressed in the discussion of results and deviations which are expected to be caused by that were accepted. On the other hand, a larger group of participants might have been slightly more effective in balancing the subjective deviations, but the group size is considered appropriate to produce valid results.

The subjectivity might have had a larger influence in the second part of the study where the accuracy of the automated prioritization was derived from the monthly assessments which are usually conducted by individuals. In this case, there is no collective wisdom which could correct subjective deviations. However, most of them do these monthly assessment regularly and, thus, might overcome this limitation through their experience with the project and the assessment of test gaps in general.

Another issue with the results of the monthly assessments was that they do not provide a complete prioritization. This was addressed through the evaluation strategy to compute the average ranking of the critical test gaps. Furthemore, it was not always possible to accurately reproduce the results. This may be caused by changes to the configuration or the coverage data in the project since the monthly assessment was conducted. Hence, only those have been selected where the deviation to the reproduced result was minimal.

The results of the study are only valid if the participants are considered as capable to appropriately assess the criticality of the test gaps. The survey has shown that most of them have many years of experience as software developers. Additionally, this was addressed by investigating the influence of the experience for the different RQs where appropriate. For the most part, this has shown to only have minor influence. Thus, this should not threaten the validity.

Finally, the data did not allow the proper evaluation of some of the metrics due to a lack of data, e.g., finding metrics. Therefore, the conclusions for these were very conservative, respecting the fact that no valid, definitive statements can be made with the current results.

5.6.2. External Validity

The external validity is a measure for the expected generalizability of the conclusions.

All study objects which are used for the evaluation in this work are industrial, closed-source projects. Theoretically, the results could be different for software from an open-source context. With Java and C#, the evaluation includes two important representatives of object-oriented programming languages. JavaScript is a widely used scripting language for web development. These three are all in the list of the ten most popular programming languages in the ranking by IEEE Spectrum in 2018 [CB18] and 2019 [19]. ABAP is with rank 40 in 2018 and 34 in 2019 at least in the Top-50. This indicates that there is a legitimate interest in these languages.

Generalizability is a known issue in many defect prediction studies [DLR12]. Even though the results of this work are promising, more work is needed to allow more well-founded insights.

6. Conclusion and Outlook

This chapter provides a short summary of this thesis and presents a selection of ideas which should be subject to subsequent research to further develop the automatic prioritization.

6.1. Summary

TGA combines static and dynamic analysis to provide insights about untested changes in software projects. These are called test gaps and are known to contain more defects than other parts of the code. Their manual assessment can be tedious and error-prone, especially for large sets. The approach which is proposed in this work aims to solve this problem through an automatic prioritization of test gaps by estimated criticality. The results of the evaluation are promising: the automatic assessment is in many cases comparable to the results of a manual assessment and it outperforms trivial strategies. It can enhance the benefits of TGA by making it more efficient and easier applicable in practice. The developers will need to spend less time on assessing test gaps and the risk of missing critical ones is reduced.

In contrast to related work in the field of defect prediction, the proposed approach is lightweight and there is no need for expensive configuration and setup as it does not depend on machine learning. The basis is a set of metrics which are known or expected to be connected to criticality. If further research proves some metrics to be wrong or other metrics to be superior, the set can easily be adjusted. The approach provides a framework which can be used with a reduced set of metrics if the available data is limited. The study has shown that this can still produce decent results.

6.2. Future Work

The approach of this thesis is based on related work in the field of defect prediction, but it chooses a different path. This uncovered several areas which should be investigated further to improve and extend the automatic prioritization in the future.

Methodology

There was no concrete model for the evaluation of the prioritization approach available. It may make sense to critically review the study methodology which was used in this work to derive a template for future studies in this field. This includes which variables are used as performance measure and which strategies serve as benchmark.

One lesson from the survey is that the criticality scale should provide a clear definition of the different criticality levels to make sure that all participants have the same mental model. The scale should also reflect the number of groups which are selected from the prototype. In the study of this thesis, four levels from the criticality scale in the survey had to be mapped to three criticality groups from the prototype. This mapping would not be necessary if the scale is chosen accordingly.

Adjustments to existing metrics

The evaluation results indicate potential for improvement by adjusting the implementation of the existing metrics. In any case, the results from the survey show that test gaps which are introduced in *Bug* issues should be considered as more critical to reduce the risk of regression. This could be realized in a separate metric or included in PRD so that the birth issue is counted as previous defect if it is a *Bug* issue (currently the birth issue is ignored).

Even though it already is one of the most important metrics according to the study results, the implementation of CLI should be adjusted. An automatic detection of changed lines in contrast to added and deleted ones will make the metric more accurate. Additionally, it should be investigated whether the assumption is true that adding lines is more critical than removing them. Otherwise, the weight of added lines should not be higher than that of deleted ones. Another issue in the current implementation is that the normalized metric values in general do not reach 1 or -1. This can be easily fixed by computing the maximum metric value instead of computing the maximum of added and deleted lines separately.

A special focus should be to improve the correlation between perceived and computed metric values. For the current metrics set, this refers to the complexity metrics (COM and COC) and CEN in particular. The addition of the total number of tokens, i.e., the smallest entities and building blocks of source code, or the number of unique token types might replace or complement the existing complexity metric to get closer to the perceived value. This could also reduce the correlation of the complexity metrics with LOC so that they become more orthogonal and, thus, complementary dimensions. Nevertheless, it has to be kept in mind that manual assessments are not the perfect oracle. Therefore, a complete fit of manual and automatic assessment is not desired.

The perceived feature importance is currently reflected in the CEN value very poorly. The approach as described by Steidl et al. [SHJ12] performs well in finding the most central classes, but does not provide a good reflection of centrality across the whole system. Modifications which could improve this result for the prioritization might be to increase the probability of the random jump and the weight of the back recommendations in the PageRank algorithm. The centrality is currently calculated on basis of files. A possibility for future research is to compute centrality on method basis and find out whether this yields an improvement. An alternative or addition to the centrality calculation would be a list of critical components which is maintained by the development team as described by Zimmermann and Nagappan [ZN08]. Test gaps in these parts of the system should receive a higher criticality score.

If the proposed adjustments for the centrality calculation do not bring the desired improvements, the normalization of metric values could be adjusted so that the values are distributed across the range more evenly. This would mitigate the effects of extreme outliers. Further studies are necessary to check whether this improves the results.

More research is required to investigate the effect of findings. A possibility is to ignore those which only affect the documentation. Additionally, the study results show no evidence that removing findings makes the code less critical. Similarly, the priority of the birth issue did not play any role for the manual assessments. It must be investigated if these have any effect on the criticality. If not, they should be removed.

The refined version of the refactoring detection, i.e., changed methods from a large change set, showed high accuracy in the study. It should be investigated whether this is a coincidence for this study object or whether this can be reproduced with other ones. The proposed approach to determine the refactoring threshold has only been tested on one study object and should be refined with further ones. If the number of refactorings for a method is known, the metric by Moser et al. [MPS08] could be added, i.e., a method which has been refactored more often is less likely to contain defects.

New metrics

The survey has shown a tendency that *shortly before release* might be a valuable addition to the metrics set. Therefore, it should likely be evaluated. Similarly, recent changes could be considered as more critical as they have been tested for a shorter time than older changes. This could also mean that recent defects receive a higher weight in PRD.

The intuition might suggest that merge commits should merely combine existing code but should not introduce any new test gaps. However, in the study, this could be observed a couple of times. Thus, it might be interesting to investigate whether test gaps from merge commits should have higher criticality. This will require a more solid understanding of the technical causes which may be inconsistencies in the coverage data rather than actual code changes. This should be clarified in subsequent work.

There are several reasons for criticality assessments which were mentioned by the participants of the study which could be interesting additions to the automated approach. Examples are whether the code is or should be *tested elsewhere* or its *testability*. It should be investigated if a formalization of these reasons is possible. If this proves to be infeasible, an alternative could be a semi-automatic approach, similar to the definition of critical components. For instance, the development team would maintain a list of parts of the code which are known to be hard to test. Test gaps in these parts receive a lower ranking in the prioritization. There have been different opinions in the survey whether test gaps from commits which are *not considered worth a review* should be ranked higher or lower. Once it is investigated whether these introduce more or less defects, a well-founded decision about the applicability of this metric is possible.

Chapter 3 about related work contains many metrics which would be valuable additions to the prioritization approach. One example are organizational and developer-centered metrics (e.g. Di Nucci et al. [Di +18]). It could also be worth to investigate whether the defect prediction approach by Nagappan et al. [NMB08] based on consecutive changes can be transferred to method level so that it can be included.

Extension of the approach

Even though the approach is currently not meant for this purpose, there might be possibilities to extend it so that it also has defect prediction capabilities. The idea is that the metric values are no longer normalized based on the minimum and maximum in the current set. Instead, the highest and lowest values are saved for each TGA session. These can be used to compute the average maximum and minimum over the entire history of the software. This allows to determine whether the current set of test gaps is rather critical or uncritical compared to the previous ones. As the prioritization is no longer relative to the current set, this could also allow to define a threshold which is used to classify the test gaps into defect-prone and defect-free. This would require a history of the software as foundation and would, thus, not work for new projects. For these, default values can be proposed from experience with other projects to allow the use of this feature from the start. In principal, this approach should also be transferable to methods in general and is not restricted to test gaps.

Besides prioritization, further support for the manual assessments could be the grouping of related test gaps. The process may be more costly than it needs to be if multiple instances of one test gap are inspected separately. If test gaps which have a logical connection can be presented together, the effort for the manual analysis might be further reduced. Ideas are grouping test gaps from the same issue or from similar regions in the code.

A. Survey

Test Gap Criticality

In this survey you are asked to assess the criticality of ten test gaps - plus some short general questions in the end. It should take no longer than 30 minutes of your time.
Thank you very much for your time and participation!
Test Gap Analysis provides test gaps on method-level. That means that all code you will see in the screenshots has not been executed by any test after the latest change.
The criticality scale reflects the priority for closing the test gap before the release from lowest (0) to highest (3).
Background and Motivation:
Quality engineers often have to face more test gaps than they can manually analyze in reasonable time - especially when test gap analysis is used in retrospective before a release. In my master's thesis we aim to develop an algorithm which proposes a ranking of test gaps according to estimated criticality to support the manual assessment in practice.
The current version of the algorithm is mainly based on metrics inspired by related research. In this survey we need your help to give us your manual assessment of some test gaps. This should, on the one hand, provide data to evaluate the performance of the algorithm. On the other hand, the gained insight will be used to refine the approach to improve performance in the future.
In the following sections, you will find a selection of test gaps from the Teamscale repository on branch 'master' in the time from June 21 to Aug 02.

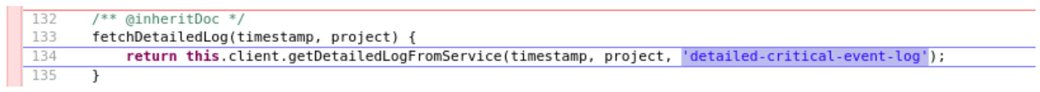
* Erforderlich

Meta-Info

Method name: fetchDetailedLog
Uniform path: engine/com.teamscale.ui/src-js/com/teamscale/ui/system/EventLogView.js
Birth issue: TS-19424 - Bug - "Migrate log services to new service framework" (https://jira.cqse.eu/browse/TS-19424)

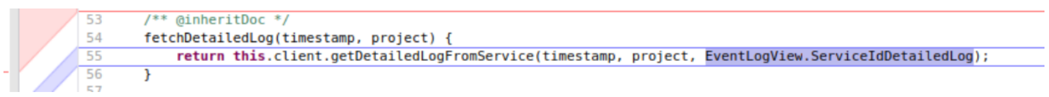
Test Gap to be evaluated

Code before change



```
/** @inheritDoc */
fetchDetailedLog(timestamp, project) {
    return this.client.getDetailedLogFromService(timestamp, project, 'detailed-critical-event-log');
}
```

Code after change (entire method untested)



```
/** @inheritDoc */
fetchDetailedLog(timestamp, project) {
    return this.client.getDetailedLogFromService(timestamp, project, EventLogView.ServiceIdDetailedLog);
}
```

Links

Delta: https://build.cqse.eu/teamscale/compare.html#cqse-all-

qc/engine%2Fcom.teamscale.ui%2Fsrc-js%2Fcom%2Fteamscale%2Fui%2Fsystem%2FEventLogView.js%23%40%23master:1559837061000&cqse-all-qc/engine%2Fcom.teamscale.ui%2Fsrc-js%2Fcom%2Fteamscale%2Fui%2Fsystem%2FEventLogView.js%23%40%23master:1563812908000
Issue in Teamscale: https://build.cqse.eu/teamscale/issues.html#/cqse-all/TS-19424

1. **How critical is this test gap?** *
   Markieren Sie nur ein Oval pro Zeile.

	Not critical (0)	1	2	Very critical (3)	Don't know
Criticality	◯	◯	◯	◯	◯

2. **Why?**

   ____

...

The survey contained 10 more test gaps in the same form as the previous one. For the sake of brevity they are skipped in this appendix.

Personal Experience

This section contains three short questions about your personal experience as a software developer, with Teamscale, and with the manual assessment of test gaps. This information might be useful to support the validity of the results from this survey.

3. **For how many years you have been working as a software developer?** *

   ____

4. **How many years have you worked with Teamscale?** *

   ____

5. **Do you have experience in the manual assessment of test gaps in one of the following contexts?**
   Wählen Sie alle zutreffenden Antworten aus.

   ☐ Retrospective
   ☐ Monthly Assesment
   ☐ Sonstiges: ________________________

General Questions

Almost done! Just a few short questions left.

6. **Shortly Before Release** *
Should test gaps which have been added shortly before a release be considered more critical? (e.g. for Teamscale everything between RC3 and release)
Markieren Sie nur ein Oval.

- Yes
- No

7. **Why?**

8. **CR#0**
In your opinion: how does the fact that a test gap was introduced in a CR#0 commit influence its criticality? (Explanation: A "CR#0 commit" refers to a commit which has no associated issue and does not undergo the review process. It is usually used for minor changes or fixes which should be applied quickly.)
Markieren Sie nur ein Oval.

	1	2	3	4	5	
Much less critical						Much more critical

9. **Why?**

10. **If you have any comments on this survey or you want to report problems with any of the questions, let me know here. For urgent feedback, which requires changes to this survey, please also write me an email.**

Acronyms

ACF Added critical findings. 17, 21, 27, 35, 51

ANF Added non-critical findings. 17, 21, 27, 35, 60

CEN Centrality. 17, 19, 21, 27, 35, 37, 48, 52, 60, 61, 63, 64

CFC Critical findings in changed code. 17, 21, 27, 35, 51

CLI Changed lines. 17, 18, 20, 21, 27, 34, 36, 50, 51, 60, 61, 63

COC Complexity change. 17, 18, 20, 21, 27, 34, 36, 37, 50–52, 60, 61, 63

COM Complexity of reference method. 16, 18, 21, 27, 35–37, 51, 61, 63

HIP High priority. 17, 19–21, 27, 35, 45, 48, 51, 60

IDE Integrated Development Environment. 25

LEN Length of reference method. 16, 18, 21, 27, 35, 36, 48, 50, 51, 60, 61

LOC Lines of Code. 6, 9–11, 18, 26, 63

NFC Non-critical findings in changed code. 17, 21, 27, 35, 51

PRD Previous defects. 17, 19, 21, 27, 35, 45, 48, 51, 60, 63, 64

RC Release Candidate. 30, 34, 38

RCF Removed critical findings. 17, 21, 27, 35, 50, 60

REF Refactoring. 17, 18, 20, 21, 27, 35, 36, 38, 45, 48, 50–52, 60, 75

RNF Removed non-critical findings. 17, 21, 27, 35, 50

RQ Research Question. 23, 24, 29, 30, 32, 34, 37, 39, 41, 43, 45, 48–51, 54, 55

TGA Test Gap analysis. 1–8, 10, 11, 16–18, 30, 35, 38, 42, 44, 53, 62, 65, 74, 75, *Glossary:* TGA

UI User Interface. 29, 49

Glossary

baseline first version of the system which is used for the comparison in Test Gap analysis or the respective timestamp. 3, 4, 16–18, 32

birth issue issue which introduced the test gap. 17–20, 28, 29, 32, 33, 35, 53, 63, 64

branch duplication of the source code in the repository to allow parallel modification, e.g. used for the development of features or the management of releases. 53, 54

change set the set of all the changes which are associated with one commit or issue. 13, 19, 20, 28, 33, 38, 48, 52, 64

changed-tested a method which has been added or changed after the baseline and executed by at least one test case after its latest modification. 4

changed-untested a method which has been added or changed after the baseline and not been tested after its latest modification. 4, 7, 8

commit a set of changes which are added to a source code repository together. 13, 17, 18, 28, 30, 34, 38, 39, 53, 54, 64, 65

defect the term "defect" refers to a static defect in the code (also often called "fault" or "bug"); it does not mean a "failure" (i.e., the possible result of a fault occurrence). iv, 1, 2, 6, 8–20, 53, 56, 62–65

field defects defects which occur in a released version of the software. 6

finding potential problem which can be assigned to a specific code region and which can be detected by static analysis. 17, 27, 35, 50, 51, 55, 64

hotfix release which repairs functionality which did not work correctly in a previous release. 5, 38

issue tracker software for managing and maintaining list of issues, usually used in collaborative settings. 12, 18, 19, 29, 30, 45

merge in version control systems which allow development on parallel branches adding changes from another branch to the current branch. 28, 53, 54, 64, 65

monthly assessment assessment of a software project by an independent third party to uncover deficits and improve quality; the TGA monthly assessment uncovers untested changes for issues that have been closed in the previous month. 5, 24–26, 31, 45, 52, 54, 55

refactoring restructuring of source code to improve readability or maintainability without changing the external behavior. 4, 13, 17–19, 28, 32–34, 38, 57, 58, 64

reference method for a test gap refers to the version of the method before it became a test gap in the analyzed time frame; is used for the computation of metrics. 16–18, 27, 48, 51, 53

repository storage location of a software project in a version control system that allows to retrieve versions of the repository at specific points in time and provides details to the change sets of all commits. 3, 18, 28

retrospective assessment of a software project by the development team itself to uncover deficits and improve quality of product and processes. 5, 31

test gap new or changed method which has not been tested in a specific timeframe. 1–8, 12, 16–55, 58, 59, 62–66, 75

tested a method which has been executed by at least one test case. 4

TGA Test Gap analysis (TGA) describes an approach to detect untested changes in a software system for a chosen timeframe. 1

working copy version of the system at the end of the analyzed timeframe in TGA. 3, 17, 18

List of Figures

List of Tables

Bibliography

[19] *Interactive: The Top Programming Languages*. 2019. URL: `https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2019` (visited on 12/12/2019).

[Aki71] F. Akiyama. “An Example of Software System Debugging.” In: *Information Processing* (1971), pp. 353–359.

[Bir+09] C. Bird, N. Nagappan, P. Devanbu, H. Gall, and B. Murphy. “Does distributed development affect software quality? An empirical case study of Windows Vista.” In: *IEEE 31st International Conference on Software Engineering, 2009*. Piscataway, NJ: IEEE, 2009, pp. 518–528. ISBN: 978-1-4244-3453-4. DOI: `10.1109/ICSE.2009.5070550`.

[BP82] V. R. Basili and B. T. Perricone. “Software errors and complexity: An empirical investigation.” 1982.

[CB18] S. Cass and P. Bulusu. *Interactive: The Top Programming Languages: Find the programming languages that are most important to you*. 2018. URL: `https://spectrum.ieee.org/static/interactive-the-top-programming-languages-2018` (visited on 07/02/2019).

[CK94] S. R. Chidamber and C. F. Kemerer. “A metrics suite for object oriented design.” In: *IEEE Transactions on Software Engineering* 20.6 (1994), pp. 476–493. ISSN: 0098-5589. DOI: `10.1109/32.295895`.

[Coh60] J. Cohen. “A Coefficient of Agreement for Nominal Scales.” In: *Educational and Psychological Measurement* 20.1 (1960), pp. 37–46. ISSN: 0013-1644. DOI: `10.1177/001316446002000104`.

[Con67] M. E. Conway. “How do committees invent?” In: (1967). URL: `https://pdfs.semanticscholar.org/cbce/35eedcde3ef152bde75950fbc7ef4c6717b2.pdf`.

[Di +18] D. Di Nucci, F. Palomba, G. de Rosa, G. Bavota, R. Oliveto, and A. de Lucia. “A Developer Centered Bug Prediction Model.” In: *IEEE Transactions on Software Engineering* 44.1 (2018), pp. 5–24. ISSN: 0098-5589. DOI: `10.1109/TSE.2017.2659747`.

[DLR12] M. D'Ambros, M. Lanza, and R. Robbes. "Evaluating defect prediction approaches: a benchmark and an extensive comparison." In: *Empirical Software Engineering* 17.4-5 (2012), pp. 531–577. ISSN: 1382-3256. DOI: 10.1007/s10664-011-9173-9.

[Ede+13] S. Eder, B. Hauptmann, M. Junker, E. Juergens, R. Vaas, and K.-H. Prommer. "Did we test our changes? Assessing alignment between tests and development in practice." In: *2013 8th International Workshop on Automation of Software Test (AST)*. Ed. by H. Zhu. Piscataway, NJ: IEEE, 2013, pp. 107–110. ISBN: 978-1-4673-6161-3. DOI: 10.1109/IWAST.2013.6595800.

[Fer74] A. E. Ferdinand. "A Theory of System Complexity." In: *International Journal of General Systems* 1.1 (1974), pp. 19–33. ISSN: 0308-1079. DOI: 10.1080/03081077408960745.

[FLP13] J. L. Fleiss, B. Levin, and M. C. Paik. *Statistical Methods for Rates and Proportions*. 3rd ed. Vol. v. 260. Wiley Series in Probability and Statistics. Somerset: Wiley, 2013. ISBN: 1118625617.

[FN99] N. E. Fenton and M. Neil. "A critique of software defect prediction models." In: *IEEE Transactions on Software Engineering* 25.5 (1999), pp. 675–689. ISSN: 0098-5589. DOI: 10.1109/32.815326.

[GFS05] T. Gyimothy, R. Ferenc, and I. Siket. "Empirical validation of object-oriented metrics on open source software for fault prediction." In: *IEEE Transactions on Software Engineering* 31.10 (2005), pp. 897–910. ISSN: 0098-5589. DOI: 10.1109/TSE.2005.112.

[Gig+12] E. Giger, M. D'Ambros, M. Pinzger, and H. C. Gall. "Method-level bug prediction." In: *2012 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. Ed. by P. Runeson. Piscataway, NJ: IEEE, 2012. ISBN: 9781450310567. DOI: 10.1145/2372251.2372285.

[Gra+00] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy. "Predicting fault incidence using software change history." In: *IEEE Transactions on Software Engineering* 26.7 (2000), pp. 653–661. ISSN: 0098-5589. DOI: 10.1109/32.859533.

[Hal+12] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. "A Systematic Literature Review on Fault Prediction Performance in Software Engineering." In: *IEEE Transactions on Software Engineering* 38.6 (2012), pp. 1276–1304. ISSN: 0098-5589. DOI: 10.1109/TSE.2011.103.

[Hal77] M. H. Halstead. *Elements of Software Science*. Amsterdam: Elsevier North-Holland, Inc, 1977. ISBN: 0-444-00205-7.

[Has09] A. E. Hassan. “Predicting faults using the complexity of code changes.” In: *IEEE 31st International Conference on Software Engineering, 2009*. Piscataway, NJ: IEEE, 2009, pp. 78–88. ISBN: 978-1-4244-3453-4. DOI: `10.1109/ICSE.2009.5070510`.

[HMK12] H. Hata, O. Mizuno, and T. Kikuno. “Bug prediction based on fine-grained module histories.” In: *34th International Conference on Software Engineering (ICSE), 2012*. Ed. by M. Glinz. Piscataway, NJ: IEEE, 2012, pp. 200–210. ISBN: 978-1-4673-1066-6. DOI: `10.1109/ICSE.2012.6227193`.

[JP16] E. Juergens and D. Pagano. *Did We Test the Right Thing? Experience with Test Gap Analysis in Practice*. Ed. by CQSE. 2016.

[Kho+96] T. M. Khoshgoftaar, E. B. Allen, K. S. Kalaichelvan, and N. Goel. “Early quality prediction: a case study in telecommunications.” In: *IEEE Software* 13.1 (1996), pp. 65–71. ISSN: 07407459. DOI: `10.1109/52.476287`.

[KM90] T. M. Khoshgoftaar and J. C. Munson. “Predicting software development errors using software complexity metrics.” In: *IEEE Journal on Selected Areas in Communications* 8.2 (1990), pp. 253–261. ISSN: 07338716. DOI: `10.1109/49.46879`.

[KWZ08] S. Kim, E. J. Whitehead, and Y. Zhang. “Classifying Software Changes: Clean or Buggy?” In: *IEEE Transactions on Software Engineering* 34.2 (2008), pp. 181–196. ISSN: 0098-5589. DOI: `10.1109/TSE.2007.70773`.

[Lip82] M. Lipow. “Number of Faults per Line of Code.” In: *IEEE Transactions on Software Engineering* SE-8.4 (1982), pp. 437–439. ISSN: 0098-5589. DOI: `10.1109/TSE.1982.235579`.

[LK77] J. R. Landis and G. G. Koch. “The Measurement of Observer Agreement for Categorical Data.” In: *Biometrics* 33.1 (1977), p. 159. ISSN: 0006341X. DOI: `10.2307/2529310`.

[MB77] R. W. Motley and W. D. Brooks. *Statistical Prediction of Programming Errors*. 1977. URL: `https://apps.dtic.mil/dtic/tr/fulltext/u2/a041106.pdf`.

[McC76] T. J. McCabe. “A Complexity Measure.” In: *IEEE Transactions on Software Engineering* SE-2.4 (1976), pp. 308–320. ISSN: 0098-5589. DOI: `10.1109/TSE.1976.233837`.

[MK92] J. C. Munson and T. M. Khoshgoftaar. “The detection of fault-prone programs.” In: *IEEE Transactions on Software Engineering* 18.5 (1992), pp. 423–433. ISSN: 0098-5589. DOI: `10.1109/32.135775`.

[MPS08] R. Moser, W. Pedrycz, and G. Succi. “A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction.” In: *Companion of the 30th international conference on Software engineering*. Ed. by W. Schäfer, M. B. Dwyer, and V. Gruhn. New York, NY: ACM, 2008. ISBN: 9781605580791. DOI: `10.1145/1368088.1368114`.

[Nag+10] N. Nagappan, A. Zeller, T. Zimmermann, K. Herzig, and B. Murphy. “Change Bursts as Defect Predictors.” In: *Proceedings, 2010 IEEE 21st International Symposium on Software Reliability Engineering*. Los Alamitos, Calif.: IEEE Computer Society, 2010, pp. 309–318. ISBN: 978-1-4244-9056-1. DOI: `10.1109/ISSRE.2010.25`.

[NMB08] N. Nagappan, B. Murphy, and V. Basili. “The influence of organizational structure on software quality.” In: *Companion of the 30th international conference on Software engineering*. Ed. by W. Schäfer, M. B. Dwyer, and V. Gruhn. New York, NY: ACM, 2008, p. 521. ISBN: 9781605580791. DOI: `10.1145/1368088.1368160`.

[NRW19] R. Niedermayr, T. Röhm, and S. Wagner. “Too trivial to test? An inverse view on defect prediction to identify methods with low fault risk.” In: *PeerJ Computer Science* 5.2 (2019), e187. DOI: `10.7717/peerj-cs.187`.

[OWB05] T. J. Ostrand, E. J. Weyuker, and R. M. Bell. “Predicting the location and number of faults in large software systems.” In: *IEEE Transactions on Software Engineering* 31.4 (2005), pp. 340–355. ISSN: 0098-5589. DOI: `10.1109/TSE.2005.49`.

[Pag+99] L. Page, S. Brin, R. Motwani, and T. Winograd. *The PageRank Citation Ranking: Bringing Order to the Web.* 1999.

[Pal+16] F. Palomba, M. Zanoni, F. A. Fontana, A. de Lucia, and R. Oliveto. “Smells Like Teen Spirit: Improving Bug Prediction Performance Using the Intensity of Code Smells.” In: *Proceedings, 2016 IEEE International Conference on Software Maintenance and Evolution*. Los Alamitos, California: IEEE Computer Society, Conference Publishing Services, 2016, pp. 244–255. ISBN: 978-1-5090-3806-0. DOI: `10.1109/ICSME.2016.27`.

[PNM08] M. Pinzger, N. Nagappan, and B. Murphy. “Can developer-module networks predict failures?” In: *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*. Ed. by M. J. Harrold. New York, NY: ACM, 2008, p. 2. ISBN: 9781595939951. DOI: `10.1145/1453101.1453105`.

[PPB18] L. Pascarella, F. Palomba, and A. Bacchelli. "Re-evaluating method-level bug prediction." In: *25th IEEE International Conference on Software Analysis, Evolution, and Reengineering : SANER 2018 : Campobasso, Italy : proceedings*. Piscataway, NJ: IEEE, 2018, pp. 592–601. ISBN: 978-1-5386-4969-5. DOI: `10.1109/SANER.2018.8330264`.

[She+85] V. Y. Shen, T.-j. Yu, S. M. Thebaut, and L. R. Paulsen. "Identifying Error-Prone Software—An Empirical Study." In: *IEEE Transactions on Software Engineering* SE-11.4 (1985), pp. 317–324. ISSN: 0098-5589. DOI: `10.1109/TSE.1985.232222`.

[SHJ12] D. Steidl, B. Hummel, and E. Juergens. "Using Network Analysis for Recommendation of Central Software Classes." In: *2012 19th Working Conference on Reverse Engineering (WCRE)*. Ed. by R. Oliveto. Piscataway, NJ: IEEE, 2012, pp. 93–102. ISBN: 978-0-7695-4891-3. DOI: `10.1109/WCRE.2012.19`.

[Sor15] I. Sora. "A PageRank based recommender system for identifying key classes in software systems." In: *2015 IEEE 10th Jubilee International Symposium on Applied Computational Intelligence and Informatics (SACI)*. Ed. by A. Szakál. Piscataway, NJ: IEEE, 2015, pp. 495–500. ISBN: 978-1-4799-9911-8. DOI: `10.1109/SACI.2015.7208254`.

[WOB08] E. J. Weyuker, T. J. Ostrand, and R. M. Bell. "Do too many cooks spoil the broth? Using the number of developers to enhance defect prediction models." In: *Empirical Software Engineering* 13.5 (2008), pp. 539–559. ISSN: 1382-3256. DOI: `10.1007/s10664-008-9082-8`.

[ZN08] T. Zimmermann and N. Nagappan. "Predicting defects using network analysis on dependency graphs." In: *Companion of the 30th international conference on Software engineering*. Ed. by W. Schäfer, M. B. Dwyer, and V. Gruhn. New York, NY: ACM, 2008. ISBN: 9781605580791. DOI: `10.1145/1368088.1368161`.