Eindhoven University of Technology

MASTER

Towards an empirical validation of the TIOBE quality indicator

Steneker, M.P.J.

Award date:
2016

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

Eindhoven University of Technology
Department of Mathematics and Computer Science

Towards an empirical validation of the TIOBE Quality Indicator

Maikel Steneker

in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science and Engineering

Supervisor: dr. Alexander Serebrenik
External supervisor: drs. Paul Jansen

Examination committee:
dr. Alexander Serebrenik
prof. dr. Jurgen Vinju
dr. George Fletcher

March 30, 2016

Abstract

We present an evaluation of the TIOBE Quality Indicator (TQI), a methodology by the company TIOBE that aims to measure software quality. The TQI is based on software metrics, a popular means to assessing various properties of software. These metrics are code-based metrics (e.g., cyclomatic complexity, code coverage, and fan-out) that can be computed automatically from a single version of the software under assessment. The role of the TQI is to compose these metrics to form a straightforward indicator of software quality.

We combine scientific literature and documentation from TIOBE to arrive at a detailed understanding of how the TQI score for an entire project is computed. We do this by presenting the theory behind each metric and describing how the metrics are aggregated and composed to form a single value. These insights result in an analysis of the weaknesses of the TQI and proposals on how to improve these aspects.

We evaluate the effectiveness of the TQI as a metric of software quality by examining how the various metrics respond to bug fixes in the software. In this study, bugs are a proxy for software quality, specifically reliability. We obtain historical information on bugs by mining software repositories. In particular, we combine information from the bug tracking system and version control system to obtain both the characteristics of each bug and the changes that were made to fix the software. We present the strengths and weaknesses of this method. Additionally, we reflect on the potential to use this setup on a larger scale to obtain a larger body of empirical evidence.

We used a survey to find projects of TIOBE customers for which we judged the data to be most reliable. The software repositories of these projects were analyzed to find the changes for each bug in the bug tracking system. We found that even when using high-quality data, extracting relevant information required considerable manual effort. From the limited analysis we were able to perform, we concluded that while metrics seem to respond to bug fixes at the project level, we found no evidence that the TQI, as a composed metric, does the same.

Acknowledgments

I would like to thank my supervisors, Alexander Serebrenik and Paul Jansen, for their help in executing this project. Alexander provided many criticisms and ideas that gave me with a lot of opportunities to improve my work. Combined with his patience, this made working with him very pleasant. Similarly, Paul help me refine my work, but he also gave me access to the people, information, and resources that made it possible to execute this project in an industrial setting. I appreciated how he was very direct in his criticism, but gave me a lot of room to flesh out my ideas.

I would also like to thank the people I worked with at TIOBE: Bram Stappers, Dennie Reniers, Jan van Nunen, Rob Goud, Sijmen Peereboom, Roos Flapper, Priscilla van der Pluijm, Thijs Engelen, Bas Kooiker, and Lucienne Baartmans. They provided me with a very stimulating and fun working environment. Not only did they help me with some of the practical challenges I faced, but they also showed a lot of interest in my work and shared their insights. I especially owe thanks to Bram and Dennie, since they proofread the text in this thesis in order to find typographical errors and other minor mistakes.

Finally, I thank the members of my graduation committee, Jurgen Vinju and George Fletcher, for taking time to discuss my work and read and grade my thesis.

Contents

Chapter 1

Introduction

Software quality is a popular topic of research [1–8]. It is one of the major research areas in the field of software engineering [3, p. 1]. The topic is also relevant from a business perspective; a sufficient level of quality is a requirement for any industrial software project to succeed [4].

Software quality is not directly observable. The quality of a software product will typically manifest itself through defects that are revealed after shipping it to the end user. Arguably, the most significant challenge in software engineering is to ensure software works properly [9, p. 261], especially as society increasingly depends on this software [10, p. 33]. Defects in the field are especially important compared to defects that are found during the development process, since they impact end users and the cost of fixing them at this stage is relatively high [11]. Since end users are impacted, the brand image of the company and their future sales performance may be negatively impacted as well. This suggests that monitoring quality in earlier stages of development is desirable. This would give some indication of the maturity and quality of a project before it is released as an end-product.

1.1 Measuring software quality

In order to measure quality, a clear definition is necessary [4]. Unfortunately, there is no consensus on how quality should be defined [8]. Garvin [12] describes five approaches to defining product quality[1]:

- The *transcendent view* describes quality as an absolute but immeasurable trait. While quality is recognizable, it cannot be defined or measured precisely. Instead, quality is a property that one can learn to recognize only through experience[2].
- The *product-based view* instead views quality as a precise and measurable attribute of a product. This means that differences in quality need to

[1] While Garvin did not speak about software products specifically, his work has been applied to software quality by Kitchenham and Pfleeger [4].

[2] In this sense, the transcendent view on quality can be compared to Plato's view on beauty: we can learn to appreciate beauty by having experiences we consider beautiful, but we never observe the abstract notion of 'beauty' directly. As such, Plato considers beauty to be objective, but undefinable.

somehow be reflected by quantitative changes of some property of the product.

- The *user-based view* views quality as a subjective quality. High quality then results from fitness to purpose of the customer.
- The *manufacturing-based view* is similar to the user-based view in that it views quality as subjective, but it focuses on the producer rather than the consumer. High quality results from the adherence to the requirements for the product.
- Finally, the *value-based view* is primarily economic. It is comparable to the manufacturing-based view, but when the price of the product increases, the quality necessarily decreases.

Depending on the approach to defining quality, the measurement method may be drastically different. Additionally, each of the views presented above share a degree of vagueness and imprecision [12]. This prompts critics of any measurement method to dismiss software quality measurement for its lack of precision. A rebuttal to this comment is offered in the form of Gilb's law [13, pp. 112–113], which has been formulated as "anything you need to quantify can be measured in some way that is superior to not measuring it at all" [14, p. 59].

1.2 TIOBE

TIOBE is a company that takes Gilb's advice to heart. Their goal is to estimate software quality using software metrics. Their framework automatically performs measurements on over 300 million Lines Of Code (LOC) every day [15]. This information is then aggregated to form a singular indicator of software quality: the TIOBE Quality Indicator (TQI) [16].

TIOBE adopts the manufacturing-based view to quality. They concede that quality is not something inherently objective [16]; any company or user can propose their own definition of quality, and there is no way of deciding which is better. While objective measurement as proposed by the product-based view is too ambitious, the TQI does strive to make the measurement of quality more objective. By being held to an industry standard, companies can meaningfully evaluate the quality of their product.

Measurements are extracted from a variety of tools [17], including (but not limited to) Coverity, Klocwork, Visual Studio, GCC, PMD, CheckStyle, and Pylint. These tools together support a variety of programming languages, including (but not limited to) C, C++, C#, Java, JavaScript, and Python.

The customers of TIOBE include ASML, Philips, TomTom, and many other companies [18]. These companies mostly, but not exclusively, focus on embedded software. The size of these companies and their projects is also diverse; the LOC for projects varies from less than 1000 lines of code to more than a million lines of code [19].

While TIOBE has achieved success in industry, it is not clear to what extent their measurement and aggregation methods are scientifically justifiable. For instance, it is not clear to what extent each metric correlates to software defects [16]. In general, metrics should not be blindly trusted, but be evaluated on a true defect history [8].

1.3 Empirical methods in software engineering research

There are multiple methods that can be used to perform empirical software engineering research [20]. Each of these methods has its own strengths and weaknesses.

The first method is the *controlled experiment*. In a controlled experiment, we control all the relevant variables. This means that we can show the existence of causal relationships. For example, we could set up an experiment in which we show that the use of an abstract interpretation tool results in a lower number of bugs in the end product. A controlled study can be costly to perform, which causes many researchers to save costs by using cheap (but non-representative) workers, such as students [21]. Another downside is that the context in which software is usually developed is not taken into account.

In a *case study*, we study data resulting from real software development in their original context [22]. We do not control these processes in any way. In particular, we do not know to what extent the context is important. When we want to apply our results in a different context, it is hard to determine whether they are valid. Case studies are also more prone to *interpreter bias*, meaning the expected results may influence the interpretation of the final results we obtain.

Case studies do not rely on random sampling, but instead choose particular instances that are interesting. This means they can aim unique or extreme cases, or instead examine a typical case. Using multiple cases can result in more valid results.

A third method is the *survey*. In a survey, respondents fill in a list of questions. This can help to gain insight from the respondents, but the results may be misleading as a result of sampling biases (i.e., the group of respondents not being the same as the group of people we intend to study) and social desirability biases (i.e., respondents answering questions in a way they think is desirable rather than truthfully).

We opt to use a mixed-methods approach. The first method we use is a survey. While a survey typically consists of an anonymous questionnaire, we record the identity of respondents in order to be able to contact them for a follow-up study. The survey is intended to reveal some identifying properties for each project, as well as assess the reliability of the data that is collected in the software repositories.

Based on the results of the survey, we can select projects which are most suitable to analyze. We analyze these in a case study. The reason is for this is that the TQI is intended as a pragmatic way to measure software quality. While exploring it in a controlled setting would be interesting, it is preferable to leverage the large volumes of measurements that have been collected to see how the TQI behaves in practice. This case study can be seen as exploratory, since we want to explore the limitations we face in practice when conducting research that involves TIOBE customers.

1.4 The ISO 25010 quality model

The manufacturing-based view of quality concedes that quality is subjective. This means we can view quality as a collection of different attributes: *quality*

factors. There is a variety of models that specify different quality factors [23]. An example of such a model is the Systems and software Quality Requirements and Evaluation (SQuaRE) model for software quality (ISO 25010 [24]). The model considers software quality to be the "degree to which a software product satisfies stated and implied needs when used under specified conditions" [24, p. 17]. It specifies the following factors of product quality [24, pp. 3–4]:

- *Functional suitability*: "degree to which a product or system provides functions that meet stated and implied needs when used under specified conditions" [24, p. 10];
- *Compatibility*: "degree to which a product, system or component can exchange information with other products, systems or components, and/or perform its required functions, while sharing the same hardware or software environment" [24, p. 11];
- *Performance efficiency*: "performance relative to the amount of resources used under stated conditions" [24, p. 11];
- *Usability*: "degree to which a product or system can be used by specified users to achieve specified goals with effectiveness, efficiency and satisfaction in a specified context of use" [24, p. 12];
- *Reliability*: "degree to which a system, product or component performs specified functions under specified conditions for a specified period of time" [24, p. 13];
- *Security*: "degree to which a product or system protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization" [24, p. 13];
- *Maintainability*: "degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers" [24, p. 14];
- *Portability*: "degree of effectiveness and efficiency with which a system, product or component can be transferred from one hardware, software or other operational or usage environment to another" [24, p. 15].

A limitation of this model, is that it leaves the way to measure these factors open to interpretation. One way to solve this is to map the various factors to software quality metrics. In the case of TIOBE, these metrics are the following:

- *Code coverage*: the percentage of code that is tested through (automatic) unit tests;
- *Abstract interpretation*: a semantic analysis that can help developers to identify potential defects;
- *Cyclomatic complexity*: a measure for the complexity of a method based on the number of independent paths through a program;
- *Compiler warnings*: warnings by the compiler that indicate something is potentially wrong with the program;
- *Coding standards*: the degree to which code adheres to coding standards;

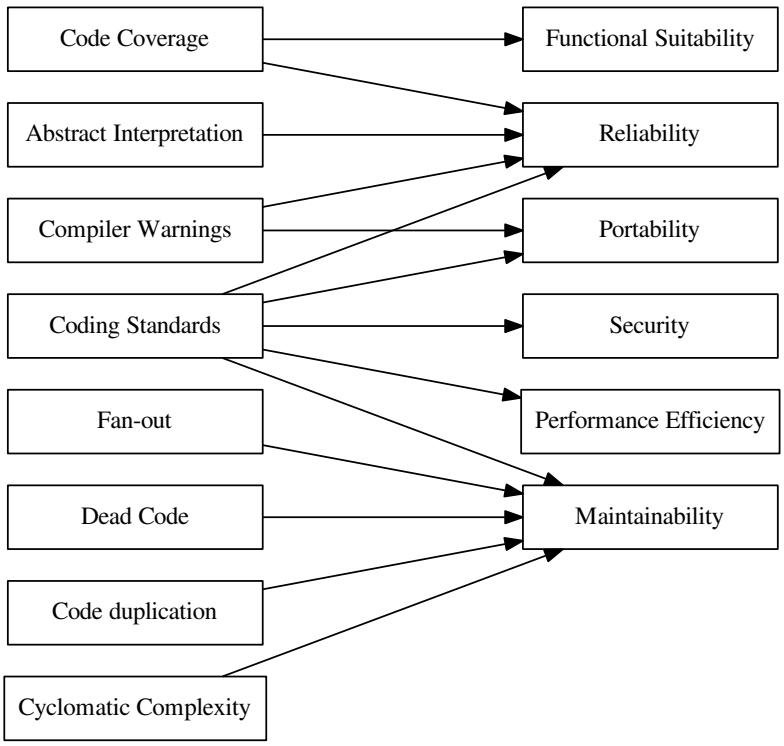


Figure 1.1: A mapping of quality metrics to quality attributes as proposed by TIOBE [16].

- *Code duplication*: the percentage of code that appears to be identical to another piece of code (100 identical consecutive tokens);
- *Fan-out*: the number of imports per module;
- *Dead code*: the percentage of code in LOC that is not reachable via any program execution.

TIOBE has constructed a mapping from SQuaRE quality factors to quality metrics [16]. This mapping is shown in Figure 1.1. It demonstrates how the quality metrics (on the left) can help assess software quality, as characterized by the factors on the right. It should be noted that this is not necessarily the only possible mapping; it is not entirely clear to what extent it is valid. Two of the factors defined in the ISO 25010 standard are missing: usability and compatibility. These are not covered by the TIOBE metrics.

TIOBE intends to validate this mapping. One way to validate metrics is by establishing a relation between the factors and metrics using statistical methods [7]. However, this requires values for each of the quality attributes. This

leads us back to the original problem of not being able to measure quality factors[3].

Instead of relating to quality factors, we can choose to relate to another metric for software quality. An interesting example of such a metric is the number of bugs. The number of bugs in a given software product is generally unknown, simply because not all bugs are known. For historical versions of the software, however, we can get information on bugs by inspecting information in a *bug tracking system*, which stores information on bugs that have been found in the software. We can therefore see these bugs as a stepping stone between the metrics TIOBE measures and the quality factors from the ISO 25010 quality model.

For this thesis, we use bugs as a proxy for software quality. Bugs are primarily a metric for the reliability of a program.We use the explicit assumption that software faults relate to software quality. The goal of the analyses are to show a relation between bugs and the various software quality metrics.

1.5 Research questions

This thesis serves both as a road map to empirical validation of the effectiveness of TQI at measuring software quality and a critical evaluation of the TQI and the metrics it is based on. This means that relevant topics include the software metrics the TQI is based on and the definition of the TQI, but also the context TIOBE operates in, specifically the software that is under analysis, the tools that are used during development, and the way the software is developed and maintained.

The main question we intend to answer is: *Is there a relation between the metrics TIOBE collects and bugs?*

We address the main question by investigating the following research questions:

1. How suitable is the information TIOBE and their customers collect for an evaluation of the TQI?
2. What steps are necessary to obtain information about bug fixes in software?
3. Do the measurements on software that the TQI aggregates respond to bug fixes in this software?
4. Can TIOBE or their customers take steps to improve the quality of this analysis?

1.6 Outline

The remainder of this thesis is structured as follows.

Chapter 2 gives an overview of software metrics, including the metrics that are used in the TQI definition. We define each of the metrics and describe what they measure and how they have been used in industry and scientific literature. We need this context to be able to understand how the TQI works.

[3] For validation purposes, experts could determine values for the quality attributes.

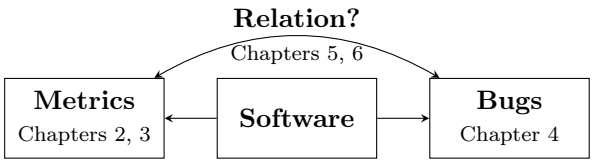


Figure 1.2: A graphical representation of the overall structure of this thesis.

Chapter 3 contains the definition of the TQI, as well as the software tools and formulas that are used to compute the TQI from "raw" measurements. We also discuss the design decisions behind the TQI and what alternative decisions could have been taken.

We discuss how we can establish a relation between metrics and bugs in Chapter 4. We do this by considering methodology from previous research and compare their strengths and weaknesses.

In Chapter 5, we describe a survey we have conducted targeting the customers of TIOBE. These customers are important, since they show what development processes are used at these companies. This information is also relevant since it allows us to get an indication of the quality of the data in the version control systems and bug tracking systems at these companies. This data can then be used to perform validation of the TQI.

Finally, we discuss a case study on some customers of TIOBE in Chapter 6 in which we examine the empirical evidence for a relation between the TQI and bugs based on historic defect data.

Figure 1.2 gives a graphical overview of the structure of this thesis. The center represents the software under analysis. We consider two different quality indicators based on this software: metrics (including the TQI and the metrics it composes) and the number of bugs. The metrics and the way these metrics are applied to the software are covered in Chapters 2 and 3. The way we find information about the bugs in the software is covered in Chapter 4. Finally, we explore the relation in the top of the figure in Chapters 5 and 6.

Chapter 2

Software metrics

The core business of TIOBE revolves around measuring software quality. Measurement is "the empirical, objective assignment of numbers, according to a rule derived from a model or theory, to attributes of objects or events with the intent of describing them" [25]. Metrics are mappings from the (components of) software to a value that represents a certain property, such as quality or complexity [25]. A wide variety of metrics have been proposed [26–31].

This chapter gives an overview of the different types of metrics and discuss some metrics in-depth. These include the metrics that are included in the TQI (see Section 1.4), as well as other metrics from literature. A classification of software metrics is presented in Section 2.1, followed by a discussion of various metrics in Sections 2.2 to 2.6. Finally, we discuss the aggregation of metrics in Section 2.7

Note Some of the metrics are regarded by some to fall outside of the definition of a metric. In particular, violations that are produced by abstract interpretation tools, coding standard checking tools and compilers (in the form of warnings) can be argued to be outputs of a tool rather than metrics.

However, according to the definition presented above, these would qualify as metrics. After all, these violations can be counted and are objectively assigned based on some theory explaining their rationale. We therefore include these metrics in this chapter, but do criticize their fitness for the purpose of measuring software quality.

Similarly, metrics may be excluded from the definition if they are compositions of other metrics. Non-composed metrics are sometimes referred to as "direct" metrics [32]. However, the usefulness of this distinction has been questioned [25], particularly since there may exist direct metrics that measure the same property as a composed metric.

2.1 Metric classifications

Software metrics can be classified into categories. There is no universally accepted classification; different classifications have been used by Kan [3], Sadowski *et al.* [33], Hata, Mizuno, and Kikuno [34], Shihab [35], Rahman and

Devanbu [36], and Nagappan, Ball, and Zeller [37]. Confusingly, different authors use the same words to refer to different concepts or use different words for the same concepts.

We consider the following categories:

- *Product metrics* [3, 32, 35]: metrics that relate to characteristics of the product.

  Product metrics are further subdivided into:

  - *Code-based metrics*[1] [33, 36, 37]: metrics that can be evaluated based on a single version of the software; and
  - *Historical metrics*[2] [34]: metrics that are evaluated over multiple versions of the software.

- *Process metrics* [3, 32]: metrics that are based on the development process.

- *Project metrics*[3] [3]: metrics that are based on properties of the project.

As indicated by Kan [3], some metrics belong to multiple categories. For example, the productivity of programmers can be considered to be a process metric or a characteristic of the programmers and therefore a project metric.

The metrics TIOBE measures are all code-based metrics [16]. This means that all metrics can be evaluated on an isolated version of the software without the need for measurements on or access to earlier versions. This has some practical advantages. It means that the metrics are all independent of the evolution of the software. Only the end product, regardless of how it was constructed, can influence the metrics and the TQI. At the same time, it means the TQI cannot use the added insight of historical, process and project metrics. These metrics have proven to be effective in the field of bug prediction [36] and could conceivably be used in the TQI to more accurately measure software quality. TIOBE measures some of these metrics and presents them to the customer without integrating them in the TQI.

2.2 Product metrics

Product metrics are metrics that relate to characteristics of the product; i.e., the software itself. This includes both metrics that can be evaluated on a single version of the code (e.g., LOC, cyclomatic complexity), as well as metrics that apply to multiple versions of the code (various metrics for the amount of change over a project's evolution). It excludes metrics that relate to the process of developing software and the project itself. Product metrics can be evaluated on the product in isolation, without considering the social or business processes surrounding it.

Code-based metrics are discussed in more detail in Section 2.3. Historical metrics are discussed in more detail in Section 2.4.

[1] Referred to as "product metrics" by Shihab [35].

[2] Referred to as "repository-based" by Sadowski *et al.* [33] and "process metrics" by Shihab [35] and Rahman and Devanbu [36].

[3] Referred to as "organizational metrics" by Hata, Mizuno, and Kikuno [34] and "process metrics" by Rahman and Devanbu [36].

2.3 Code-based metrics

The code-based metrics we discuss in this section could be classified in the following way:

- Size metrics:
  - Lines of code (Section 2.3.1)
- Complexity:
  - McCabe complexity (Section 2.3.2)
  - Halstead complexity measures (Section 2.3.3)
- Quality of structural design:
  - Code duplication (Section 2.3.4)
  - Dead/unreachable code (Section 2.3.5)
  - Cohesion and coupling (Section 2.3.6)
  - Chidamber and Kemerer metrics (Section 2.3.7)
- Test suite quality:
  - Code coverage (Section 2.3.8)
  - Mutation coverage (Section 2.3.9)
- Violations:
  - Abstract interpretation violations (Section 2.3.10)
  - Compiler warnings (Section 2.3.11)
  - Adherence to coding standard (Section 2.3.12)

2.3.1 Lines of code

The number of lines of code (LOC, also known as source lines of code or SLOC) is a widely used software metric [33, 34, 38–46]. LOC is a language-independent metric for the number of lines in the files that make up the source code. It can be used on any granularity level; we can compute LOC over a function, file, package, or project.

While LOC is intended to be a metric for size, it has also been used for effort estimation [47, p. 660] and bug prediction [38, 40, 42, 43, 48]. This is reasonably effective simply due to the trivial fact that producing code takes time and has the potential to introduce faults. This means that LOC is not only useful as a metric for software size, but also as a baseline for any model using metrics. Metrics that do not add anything to a model when LOC is already included or highly correlate with LOC are unlikely to represent anything other than size. This approach has been used to argue the redundancy of other metrics [39]. It should be noted that aggregation can play a major role in the correlation between LOC and other metrics [43]. A metric may be redundant when using a specific aggregation technique (such as the sum), but it may be useful when using other aggregation techniques or using measurements at a different granularity level.

Listing 2.1: Code to illustrate LOC metric.

```
1 for (int i = 0; i < N; i++)
2 {
3     // print the current number
4     System.out.println(i);
5 }
```

Listing 2.2: Shorter version of the code in Listing 2.1.

```
1 for (int i = 0; i < N; i++) {System.out.println(i);}
```

One weakness of LOC as a metric for size is that semantically equivalent pieces of code may have a different number of lines of code. Compare, for instance, the two pieces of code shown in Listing 2.1 and Listing 2.2.

These examples have the exact same meaning, but Listing 2.2 is written on a single line while Listing 2.1 is spread over five lines. We can alleviate this by removing comments and counting the number of statements instead of physical lines. Similarly, empty lines can be removed. Note that these solutions are language-specific, meaning we need knowledge about the language of code to determine its size.

2.3.2 McCabe complexity

Cyclomatic complexity [26] (also known as McCabe complexity) is a complexity metric for imperative methods. It is computed by representing the possible paths of execution of the program in a graph and computing its cyclomatic number (i.e., the number of regions in a planar embedding of the graph). The idea is that methods with a lot of conditional statements or loops are more complex. This means maintainability of the program is hurt, since it is hard to understand for new developers working on it. It also means testing the code properly is harder, since it is harder to reach a sufficient level of coverage. TIOBE uses cyclomatic complexity as one of the components of the TQI [16].

We consider the simple program below (Listing 2.3) as an example. Figure 2.1 shows a graph representation of this program. Each statement in the code is represented by a vertex[4] in the graph. The edges represent paths that can be executed. The graph shows both loops (the cycle between 4 and 5) and branch statements (the two outgoing edges from vertex 8, depending on the boolean value of the condition).

From the control flow graph, we can compute the cyclomatic complexity of the program by counting the number of regions the graph divides the plane in. This includes the outer region, as well as the areas enclosed by branches and loops. In this case, there are three regions, meaning the cyclomatic complexity of the program is 3.

Cyclomatic complexity is a metric that is defined for a single method; there

[4] Strictly speaking, the vertices in a control flow graph represent "basic blocks" as opposed to statements, but we abstract from this detail for simplicity.

Listing 2.3: Example of relatively simple program.

```
1  int foo(int n, int m) {
2      int sum = 0;
3
4      for (int i = 0; i < n; i++) {
5          sum += i;
6      }
7
8      if (sum > m) {
9          sum = m;
10     }
11
12     return sum;
13 }
```

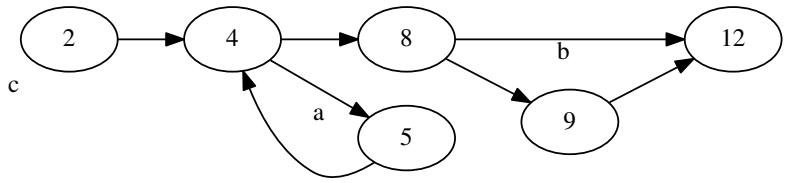


Figure 2.1: Graph representation of program in Listing 2.3. The vertices represent statements in the program. The edges represent control flow. The letters represent regions in the plane. In this case, there are three regions: one enclosed by vertices 4 and 5 (a); another enclosed by vertices 8, 9, and 12 (b); and finally the space around the graph (c).

is no notion of multiple methods or objects. Therefore, computing cyclomatic complexity for a program consisting of multiple methods and/or objects requires aggregation. The simplest aggregation technique is taking the sum of all measurements. This results in cyclomatic complexity becoming a proxy for the size of the program [39]. However, this effect is caused by the aggregation, not the metric itself [43]. Aggregation techniques for metrics are covered in Section 2.7.

2.3.3 Halstead complexity measures

Halstead complexity measures are a series of metrics aimed at measuring software size and complexity and effort estimation [49]. These metrics are all defined on a certain "problem", i.e., a set of functional requirements for a single method. The Halstead metrics are not included in the TQI, but are an interesting alternative, especially since they provide an estimate for the number of bugs in the implementation.

The following numbers are at the base to the Halstead metrics:

- η_1: number of distinct operators;
- η_2: number of distinct operands;
- N_1: number of occurrences of operators; and
- N_2: number of occurrences of operands.

Based on these numbers, Halstead defines a number of metrics [49], including:

- The *length* of a program:

$$N = N_1 + N_2 \tag{2.1}$$

- The *vocabulary* of a program:

$$\eta = \eta_1 + \eta_2 \tag{2.2}$$

- The *volume* of a program:

$$V = N \times \log_2 \eta \tag{2.3}$$

- The *difficulty* of a program:

$$D = \frac{\eta_1}{2} \times \frac{N_2}{\eta_2} \tag{2.4}$$

- *Programming effort*:

$$E = D \times V \tag{2.5}$$

- The *time required to implement* the program:

$$T = \frac{E}{18} \tag{2.6}$$

- The estimated number of *delivered bugs* in the implementation:

$$B = \frac{E^{\frac{2}{3}}}{3000} \tag{2.7}$$

Equations 2.1 to 2.3 are different size metrics. Equation 2.4 is a complexity metric. Equation 2.5 and Equation 2.6 are effort estimation metrics. Equation 2.7 is a software reliability estimate.

An issue with these metrics is that the constants seem arbitrarily chosen and have no intuitive meaning. Another problem is that the terms *operators* and *operands* are not clearly defined, leaving their meaning open to interpretation. The values for these metrics are also language-dependent, meaning the results may vary simply as a result of the choice of programming language. Another limitation is that the metrics are not defined with structured or object-oriented language in mind. Especially for larger projects, it becomes impractical to use these metrics. Finally, the Halstead metrics highly correlate with LOC [39, 50], which suggests they measure nothing more than software size.

2.3.4 Code duplication

Code duplication is the process of copying code to re-use it in a new context [51]. The amount of code duplicated code can be used as a metric for the proportion of redundant code in a project. Redundant code is syntactically or semantically equivalent to some other part of the code. Code duplication is one of the components of the TQI [16].

As a software project grows, programmers may make copies of code to adjust its functionality. While this is an easy and simple way to replicate functionality, it results in a higher volume of code that may contain duplicates of bugs. Large software system contain a considerable amount (5–10%) of duplicate code [51]. Large amounts of code duplication result in a system that is harder to maintain [52].

We can distinguish between four types of code duplication [53]:

1. Complete copies that only differ in terms of whitespace, layout, and comments;
2. Clones that may differ in terms of whitespace, layout, comments, and replacement of identifiers, literals, and types;
3. Clones in which some statements are added, removed, or replaced;
4. Different implementations of the same functionality.

Note that types 1 to 3 are related to syntactic equivalence, while type 4 describes semantic equivalence. Type-1 clones are the easiest to detect, with each consecutive type being harder to detect.

Generally, clones are only counted if they are sufficiently large. This is because when a clone is small, for instance a single statement or token, it does not pose a threat to maintainability. After all, all code is built from the same atomic tokens, so the existence of very small code clones is meaningless.

Obviously, the type of clones that are counted and the minimum length for a clone influence the measurements that are obtained as a result.

Listing 2.4: Program containing dead code.

```
1 int a;
2 a = 0;
3 a = 1;
```

Listing 2.5: Program containing unreachable code.

```
1 int n = 42;
2 n = n % 10;
3 if (n > 10) {
4     n = 0;
5 }
```

2.3.5 Dead/unreachable code

Dead code is code for which the result will never be used [54, p. 350]. A result could be a change in the state of the program in the broadest sense, including potential exception resulting from an otherwise semantically meaningless operation [54, p. 360].

Dead code is closely related to the notion of *unreachable code*, which is code that can never be executed [55]. While these concepts are different, the terms "dead" and "unreachable" code are sometimes used interchangeably. An example of the confusion of these terms is the document defining the TQI [16]. TIOBE only measures unreachable code, not dead code[5].

An example of dead code is shown in Listing 2.4. Clearly, since the statements are executed sequentially at all times, the assignment of 0 to `a` will have no effect on the remainder of the program. Line 2 can therefore be considered dead code, but not unreachable code.

The automatic detection of dead code is a challenging task, since the outcome of a statement cannot easily be predicted from a static analysis, i.e., without executing the code. As an alternative, tools can aim to identify unreachable code.

An example of unreachable code is shown in Listing 2.5. Since the modulo operator is used on `n`, it can never exceed 10. Therefore, the body of the if-statement is never executed.

While the notion of unreachable code is fairly intuitive, its detection is all but trivial—in fact, undecidable [54, p. 212]. One reason for this is the undecidability of the Halting problem. Since we cannot decide whether a given method terminates, we cannot determine with full certainty that the code towards the end of the method is ever reached. We can, however, find an over-approximation of the total body of unreachable code, for instance by checking whether a variable or method is textually referenced outside of its declaration. Tools such as ReSharper[6] and DCD[7] use this approach. Abstract interpretation can also be

[5] In fact, dead code may result in some abstract interpretation violations, which are taken into account in the TQI. However, these do not affect the "dead code" component.

[6] `https://www.jetbrains.com/resharper/`

[7] `https://java.net/projects/dcd/`

used to find dead code. While dead code detection can potentially be useful in the development process, the output of these tools may contain some false positives. Some methods may only be accessed through methods like reflection. Removing those methods would alter the functionality of the program.

2.3.6 Cohesion and coupling

An important part of programming is the division of tasks into smaller sub-tasks to deal with complexity. This type of design achieves the highest quality when it achieves *low coupling* and *high cohesion* [56]. High cohesion refers to a large degree of interdependency of components within a module. Low coupling refers to a low degree of interdependency of components outside of a module.

Henry and Kafura [29] refer to this interdependency as *fan-in* and *fan-out*. Fan-in is the number of external modules using functions in a module. Fan-out is the number of modules from which functions are imported. These numbers can be used as metrics, either directly or in a formula. Henry and Kafura use the following formula:

$$hk = sloc \cdot (fan\_in + fan\_out)^2 \tag{2.8}$$

Fan-out may be more suitable as a metric for structural complexity than fan-in [57]. The reason for this is that high fan-out is indicative of a module that provides functionality that is unrelated to the rest of the contents of the module. This is a "code smell", since this functionality should typically be divided over different modules. Fan-in, on the other hand, is an indication that the functionality is applied in a variety of contexts. A typical example of a module that achieves a high fan-in is a module that provides commonly used mathematical functions.

Another downside of the fan-in metric compared to fan-out is that it is not actionable. A module that imports too much external functionality can be "improved" by somehow reducing the number of imports (typically by splitting up a module into multiple modules). Reducing fan-in requires investigating the different contexts in which the functionality is used and assessing whether this is correct.

Martin [58] used the terms *afferent coupling* to refer to the fan-in and *efferent coupling* to refer to the fan-out of a package [58, p. 262]. He uses this to compute the *positional instability* of a package using Equation 2.9.

$$I = \frac{C_e}{C_a + C_e} \tag{2.9}$$

C_e is the number of classes outside of the package that are used and C_a is the number of classes that use classes within the package.

These metrics are all metrics for quality of structural design on a module level. TIOBE uses fan-out in their TQI definition [16].

2.3.7 Chidamber and Kemerer metrics

The Chidamber and Kemerer metrics suite for object-oriented design consists of metrics that aim to measure the quality of the design of object-oriented programs [27]. These metrics are not included in the TQI. Nevertheless, they are interesting, since they focus specifically on object-oriented programs.

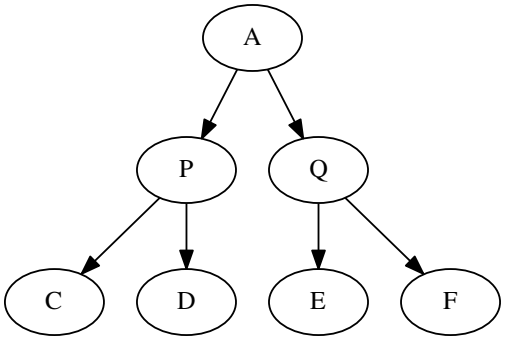


Figure 2.2: An illustration of the Depth of Inheritance Tree metric (reproduced from Chidamber and Kemerer [27]). In an inheritance tree, vertices represent classes; an edge from a class C to C' indicates that C' extends the functionality of C, i.e., C' is a subclass of C. The DIT for class A is 0; the DIT for classes P and Q is 1; the DIT for classes C, D, E and F is 2.

These metrics include the following:

- *Weighted Methods per Class* (WMC), a size metric for a class:

$$WMC = \sum_{m \in C} complexity(m) \tag{2.10}$$

  where m is a method in class C and *complexity* is a metric for the complexity of a method, such as cyclomatic complexity. We can also define $complexity(m) = 1$ for all m, in which case WMC is simply a metric for the number of methods in a class.

- *Depth of Inheritance Tree* (DIT), a metric for the number of ancestor classes that can influence a class. It measures the maximum path to the root of the inheritance tree. The behavior of deeper classes is harder to predict, since the behavior may originate from more different places. The DIT metric is illustrated in Figure 2.2.

- *Number of Children* (NOC), a metric for the scope of properties. This gives an indication of the amount of reuse, but can also be a sign of potential misuse of subclasses. It measures the number of direct children of a class in the inheritance tree. In the inheritance tree shown in Figure 2.2, the number of children of classes A, P and Q is 2; the number of children for classes C, D, E and F is 0.

- *Coupling between object classes* (CBO), a metric for coupling (see Section 2.3.6).

- *Response For a Class* (RFC), a metric for the potential communication between a class and other classes. This can be seen a a metric for com-

plexity. The definition of RFC is based on the response set RS.

$$RFC = |RS| \tag{2.11}$$

$$RS = \left(\bigcup_{m \in C} R_m \right) \cup C \tag{2.12}$$

In Equation 2.12, C is the set of methods in the class, m is a single method in the class and R_m is the set of methods called by method m.

- *Lack of Cohesion in Methods* (LCOM), a metric for cohesion (see Section 2.3.6). It measures the number of pairs of methods that share no instance variables compared to the number of methods that share some instance variable. The instance variables used by a method m are denoted as I_m.

$$P = \{(I_m, I_{m'}) \mid m, m' \in C, I_m \cap I_{m'} = \emptyset\} \tag{2.13}$$

$$Q = \{(I_m, I_{m'}) \mid m, m' \in C, I_m \cap I_{m'} \neq \emptyset\} \tag{2.14}$$

$$LCOM = \begin{cases} |P| - |Q| & \text{if } |P| > |Q| \\ 0 & \text{otherwise} \end{cases} \tag{2.15}$$

  A higher LCOM value means there is a higher number of method pairs that do not share instance variables than the number of pairs that do share instance variables. High values indicates that the class could be split up in more cohesive parts.

  The LCOM metric has been criticized [59] for its inability to differentiate between constructed examples that seem different in terms of cohesion. Another weakness is that any negative values are adjusted to zero, which can result in a lot of zero values. Henderson-Sellers [60, pp. 142-147] proposed an alternative definition of LCOM that normalizes for the number of methods in a class.

The metrics' focus on object-oriented software is one of the aspects that distinguishes them from metrics like McCabe complexity and Halstead complexity measures. The metrics are defined based on a formal theoretical basis and have been validated empirically [27].

2.3.8 Code coverage

Code coverage is a metric of test suite quality. Intuitively, it is the percentage of code that is covered by (automated) unit tests. These tests can automatically invoke parts of the software and evaluate whether their results are correct. Since automatic tests are much more reliable and cheaper to perform than manual tests, this enables software developers to test their code often, which could help to maintain high-quality software. Code coverage is a metric that can be evaluated on a low level (a single file or method), but it is usually computed over an entire project. The TQI includes code coverage as one of its components.

There are different types of coverage [61, pp. 43–49], which vary in terms of granularity and precision. The simplest form of code coverage is *statement coverage* (also known as line coverage). This is a metric for the percentage of statements that are executed by a unit test.

Listing 2.6: Example of relatively simple program.

```
1 int foo(int a, int b) {
2     int c = b;
3
4     if (a > 3 && b > 0) {
5         c = a;
6     }
7
8     return c * a;
9 }
```

Listing 2.6 shows an example of a method that can be tested using unit tests. A unit tests consists of an input (such as `foo(0,0)` or `foo(4,2)`) and an expected result for each test case (0 and 16 respectively for the given examples). This program has a total of 4 statements (two assignments, an `if` statement and a `return` statement). The test case `foo(0,0)` covers 3 of these statements (75% coverage), while the test case `foo(4,2)` covers all statements (100% coverage).

While statement coverage can give an indication of the quality of a test suite, it may be too easy to achieve. For example, if we only use the second test, we achieve a perfect coverage of 100%. However, it could be that for cases where the condition of the `if` statement evaluates to false, the result is incorrect. The notion of *decision coverage* (also known as branch coverage) alleviates this issue. It requires every branch of every conditional or loop (including `if`, `case`, `while`, and `for` statements) to be tested at least once. This means that to achieve 100% decision coverage, we need a test for which $a > 3 \wedge b > 0$ and one for which $a \leq 3 \vee b \leq 0$. The previously presented examples are an example of a test suite with 100% decision coverage for this method.

As decisions get more complex, decision coverage may be insufficient as a metric as well. After all, we only have one test case for each branch. Since a conditional may be composed of an arbitrary number of boolean variables (conditions) through conjunctions and disjunctions, there are many paths through the code that can end up untested even when 100% decision coverage is achieved. A third type of coverage, *condition coverage*, requires a test case for each of the composed conditions. That means that in our example, we need at least a case for $a > 3$ and $\neg(a > 3)$ as well as cases for $b > 0$ and $\neg(b > 0)$. Again, the provided test cases satisfy this.

Finally, an even stricter form of coverage is *path coverage*. This requires every possible path to be executed [62]. This means that every permutation of boolean values that can be assigned to the conditions in each decision should be tested. In this case, this would require a test cases for $a > 3 \wedge b > 0$, $a > 3 \wedge b \leq 0$, $a \leq 3 \wedge b > 0$ and $a \leq 3 \wedge b \leq 0$. In general, achieving 100% path coverage is impractical, since the number of paths increases exponentially as the number of boolean variables increases linearly.

Other types of code coverage include function coverage [63] (the proportion of methods that is covered by unit tests), condition/decision coverage [64, p. 9] (a combination of condition and decision coverage) and an even stronger type of coverage known as Modified Condition/Decision Coverage (MC/DC) [64, p. 9],

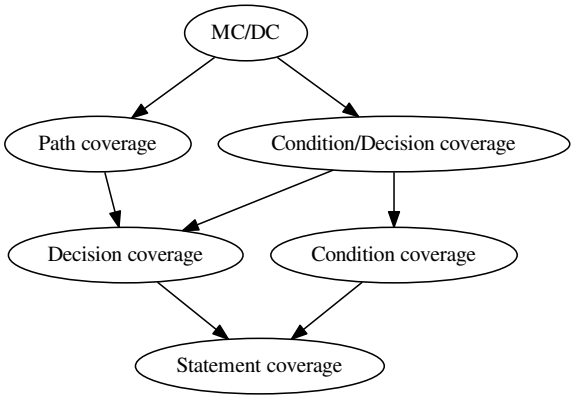


Figure 2.3: Overview of the relations between various types of code coverage. Complete path coverage implies complete decision and statement coverage. Both complete decision coverage and complete condition coverage imply complete statement coverage. However, complete condition coverage does not imply complete decision coverage or vice versa.

which we will not discuss in detail. MC/DC is applied in avionics and other critical systems domains, but it is highly sensitive to simple syntactical modifications that have no semantic meaning [65].

Note that complete condition coverage does not imply complete decision coverage or vice versa. For example, the test suite {`foo(7,-1)`, `foo(2,2)`} covers all conditions, but only one decision. Similarly, {`foo(7,-1)`, `foo(7,1)`} covers all decisions, but not all conditions. An overview of the relations between the various types of coverage is presented in Figure 2.3.

2.3.9 Mutation coverage

Mutation coverage is, like code coverage, a metric of test suite quality. It is designed to overcome a weakness of code coverage that is inherent to its definition and therefore independent of the number of paths that are covered. Mutation testing is one of the best predictors of test suite quality [62]. Despite this, it is barely used in industry, possibly due to its complexity and computational cost [62].

Recall that code coverage measures the proportion of the code (either in terms of statements, decisions, conditions, or another criterion) that is tested through unit tests. This essentially validates whether the developer thought to write tests for all functionality. However, code coverage is independent of the *quality* of these tests. For this reason, code coverage is not strongly correlated to the effectiveness of a test suite at detecting faults [66].

For instance, we could construct test inputs for a method simply by providing an input for each of its branches and achieve a reasonably high code coverage. Still, particular edge cases (such as boundary cases) may still be untested. This

Listing 2.7: Example of program for which mutants may still be correct.

```
1 int min(int a, int b) {
2     if (a < b) {
3         return a;
4     } else {
5         return b;
6     }
7 }
```

means that bugs could go undetected even with a relatively high coverage. The idea behind mutation coverage is that the quality of a test case is its ability to detect errors in implementation. That is, when a small mistake is present, one of its test cases should fail.

The process of mutation testing [67] involves creating *mutants*, modified versions of the software that is under analysis that potentially contain bugs. Since test cases are expected to fail when bugs cause the implementation to be incorrect, better test suites generally fail for a larger proportion of mutants. Mutation coverage is the percentage of mutants for which a test suite fails.

One way of generating mutants is by substituting operators by different operators. In many cases, this results in a program with different functionality that should therefore fail the test case. Note that this is not necessarily always the case. For example, consider the method in Listing 2.7. The intention of this program is to return the minimum of the supplied arguments. We can generate a mutant by replacing the $<$ operator by the $\leq$ operator. This is an example of a mutant that does not influence the correctness of the code; after all, it only impacts the trivial case where parameters `a` and `b` are equal for which either branch results in the correct result. Therefore, the goal of achieving 100% mutation coverage is meaningless. That said, mutants are generally good substitutes for real faults in software [68].

2.3.10 Abstract interpretation violations

Abstract interpretation is a technique in which the computations of a program are evaluated on abstract objects, with the goal of deriving results without executing the program [69, 70]. This is typically implemented by translating source code to a model instead of an executable program. This model is then examined for patterns that reveal potential errors. Abstract interpretation abstracts from the explicit value of a variable, and instead considers all the possible values it can have mapped to some abstract domain [71]. TIOBE uses the output of abstract interpretation tools for the TQI [16].

Consider Listing 2.8 as an example. One problem abstract interpretation may detect in this example is that the `obj` variable may have the value `null` even after line 2. This would cause a null dereference in the final line, which results in an error and potentially the termination of the program. A human programmer could very well overlook this fact; they will likely assume the `getObject()` method returns an actual object rather than a `null` value. An abstract interpretation tool would detect that `null` is one of the potential values of `obj` and

Listing 2.8: Example of a potentially faulty program for which abstract interpretation analysis may help uncover a bug.

```
1 if (obj == null) {
2     obj = getObject();
3 }
4 obj.method();
```

Listing 2.9: Example of a potentially faulty program which will trigger a compiler warning.

```
1 if (var = 0) {
2     var++;
3 }
```

flag a warning.

Abstract interpretation works by considering the range of possible values a variable can take and restricting this range based on statements in the program. This means an abstract interpretation tool may be able to deduce that `obj` can never take a `null` value. If one of the values it could take is `null`, something is wrong with the program.

While this example illustrates the usefulness of abstract interpretation, the analysis can result in false positives [72]. In this example, this could be the case if the `getObject()` method indeed guarantees that no `null` object will be returned in some way that cannot be modeled through abstract interpretation, such as reflection.

One metric we can derive from abstract interpretation is the number of violations produced by the tools. Alternatively, we can assign a weight to each type of violation to distinguish between important and less important violations. These are metrics for the number of potential bugs in the software.

2.3.11 Compiler warnings

Compiler warnings are potential errors in the source code that are discovered during compilation. These include syntactic errors for which the semantics are easily misinterpreted, portability issues and type errors. They can therefore be seen as a metric for the quantity and impact of potential bugs. TIOBE uses compiler warnings in the TQI [16].

An example of code that yields a compiler warning is shown in Listing 2.9.

Most languages use the `=` operator for assignment and expect a boolean condition, e.g., `var == 0`. Note that in this case, `var = 0` is not a boolean expression for the equality of `var` and 0, but an assignment of 0 to `var`. This assignment is executed, followed by the execution of the code in the body. A compiler will recognize that this is likely unintended and warns the programmer.

Other warnings could relate to arithmetic operations that are inconsistent across systems with different bitness[8]. This hurts the portability of the code.

[8] bitness refers to the distinction between 32- and 64-bit processor architectures

While the process of removing compiler warnings has not been shown to improve maintainability, it results in an improvement in portability [73].

2.3.12 Adherence to coding standard

Coding standards provide rules for software developers to follow while writing code. The most important reason for coding standards is maintainability; a clear set of rules makes it easier for programmers to understand the meaning of code [74]. A subset of coding standards can be used as predictor for finding fault-related lines of code [75]. The TQI also includes a metric for coding standard adherence [16].

Examples of coding standards include naming conventions (e.g., using lower case for variable names and capitals for class names), rules for indentation (e.g., use of spaces instead of tabs, indenting the body following an if-statement) and the exclusion of language features (e.g., disallowing use of `goto` statements).

To measure adherence to coding standard automatically, rules can be programmed into a tool that automatically identifies violations of the standard. This requires rules to be sufficiently precise. A meaningful metric for adherence to coding standard depends on the quality of the coding standard and a suitable classification of the weight of each of its rules.

2.4 Historical metrics

This section covers the following historical metrics:

- Number of changes (Section 2.4.1)
- Code churn (Section 2.4.2)

None of these metrics are part of the TQI. Nevertheless, we discuss them, since their effectiveness is promising [36].

2.4.1 Number of changes

The *number of changes* is a metric for the amount of change of an artifact. It is simply a count of the number of times a piece of code was changed. This can indicate which parts of software are changed most often. Code that changes often is less stable in the sense that its functionality or performance changes often. Additionally, since it is changed often, there are also more opportunities for the introduction of bugs. This means the number of changes can be used as a predictor for the number of faults.

A downside of the metric is that it likely will not reveal information that is surprising to developers; after all, they knowingly make changes to files. The metric can be improved for bug prediction by only counting changes in which a bug was fixed. Additionally, not all files that are often changed need to necessarily contain bugs. For example, a file containing references to unit tests or a list of error messages is likely to be changed with a bug fix (because a unit test or error message is added), but naturally is not the cause of the bug.

The number of changes can be used, like LOC, to test whether metrics add anything beyond a naive and simple count metric. This has been done in the past to evaluate the performance of FixCache [76].

2.4.2 Code churn

Code churn is another metric for the amount of change. Rather than simply counting the number of changes, it refers to the number of lines that were added, removed or, changed[9]. This means code churn can be computed for a certain changeset[10].

Nagappan and Ball [28] differentiate between absolute and relative code churn. Absolute code churn is simply a count of added, removed, and modified lines; relative code churn divides this number by the total size of the files that are changed. In a case study on Windows Server 2003, relative churn was a better predictor for defects than absolute churn [28].

Code churn has been shown to correlate with program quality [8, 77]. In fact, churn metrics correlate much stronger than code-based metrics [40, 44]. While this shows the potential of historical metrics in addition to code-based metrics [40], these metrics may be less useful since they are not actionable [78, 79]. Like with LOC, even if we can show an increase in code churn causes bugs, we cannot decrease code churn since this would imply making less changes and therefore not maintaining the software. In that sense, code churn is a trivial and relatively uninteresting metric.

2.5 Process metrics

The metrics in Section 2.2 all relate to the characteristics of the product. That is, the can be evaluated only on the (historical archive of) the end product: software. Process metrics evaluate software quality from another perspective, namely the development process of the software. Product management is a critical factor of software development [14], so it seems fitting to evaluate quality using these metrics.

Some examples of process metrics:

- *Effectiveness of defect removal*: a metric for the effectiveness (e.g., in terms of speed) in which defects are removed. This is particularly relevant for clients using the software. If a fault is encountered in software, it has less impact if it can quickly be resolved.

- *Quality of management*: management can play a decisive role in the productivity of software engineers. Good managers are able to supervise a software project more effectively, which results in a more effective development process.

- *Productivity*: the productivity of a team of developers can be considered to be a part of software quality as well.

TIOBE aims to measure code quality [16]. For this reason, they do not consider process metrics.

[9] The notion of a changed line of code is tricky, since any changed line can be considered to be a removal of the old line (without the change) and an introduction of the new line (containing the change).

[10] A set of changes to files that is *committed* to a *version control system*.

2.6 Project metrics

Not only the development process and the characteristics of the product, but also the properties of the project can provide insight into the quality of software. Software engineering projects are team-based efforts that depend on the strengths of the individual software developers as well as the extent to which these strengths are utilized. These factors influence productivity and software quality [80].

Example of project metrics include:

- *Number of developers*: a metric for development team size. The size of a development team can give information about the structure of the development process. Larger teams require more coordination, which impacts their central development tasks.

- *Developer experience*: a metric for the extent to which developers are familiar with the technology they are using. Prior experience provides a great advantage in terms of productivity [81].

- *Team diversity*: the diversity of the team with respect to gender, cultural background and others factors. More diverse teams tend to be more productive [82].

- *Team familiarity*: the degree to which the team members know each other. Developers generally prefer to join teams consisting of people they know. They are also more productive within these familiar teams [81].

- *Geographical distribution*: the relative locations of the members of the team. Physical distance between team members, difference in languages, cultural and organizational differences, and time zones all potentially impact the team's ability to communicate and therefore to achieve the goal of high-quality software [83, 84]. There is some evidence that when teams take the effort to take these limitations into account, software quality does not necessarily decrease [84].

- *Ownership*: the degree to which parts of the software have an "owner", i.e., someone who invented or most frequently maintains the code. Software development methodologies like Extreme Programming advocate collective code ownership [85, p. 66]. There is some evidence that a lack of any ownership, individual or otherwise, leads to lower-quality software [86].

TIOBE does not take these factors into account, since they focus on code quality [16].

2.7 Metric aggregation

Another relevant topic is metric aggregation. This refers to the process of combining multiple measurements into a single value. A good aggregation allows one to draw conclusions without having to examine the many separate measurements it aggregates separately.

The term "aggregation" is often used to refer to two different concepts [5]: *composition* and *aggregation*. These procedures are the same in the mathematical sense; they are simply functions from a set of measurements to a single value. Instead, the distinction between these concepts is in their goal.

Aggregation combines values from multiple low-level measurements (for instance, measurements on methods, classes, or files) into a single value that is intended to represent the same metric at a higher level (for instance, a package or project). For instance, cyclomatic complexity is defined on the method level. An aggregation for cyclomatic complexity would represent the same abstract notion of program complexity on a file or project level.

Composition combines values from different metrics into a single value that is intended to represent a combination of the semantics of these metrics. For instance, we can combine cyclomatic complexity and coverage to arrive at a composed metric that represents the extent to which more complex methods are tested more.

Composition and aggregation can be done in either order, but it is more meaningful to compose before aggregating in order to preserve the semantics of the composition [5].

Aggregation techniques

There are many aggregation techniques, each with their own strengths and weaknesses. We will discuss a few of them here. We use M to denote the set of measurements for each of the formulas. We assume the measurements are numbers, although this does not strictly need to be the case for all measurements.

The *sum* (Equation 2.16) is a very simple aggregation technique that simply adds up all metric values.

$$sum(M) = \sum_{m \in M} m \tag{2.16}$$

If the metrics have different weights (for example, the sizes of the various components), we can generalize the definition to the *weighted sum* (Equation 2.17).

$$wsum(M) = \sum_{m \in M} weight(m) \cdot m \tag{2.17}$$

The biggest strength of these techniques is that they are very easy to understand and implement. One big drawback is that in practice, many metrics will correlate very strongly with the size, i.e., the total of all weights. For instance, the total cyclomatic complexity aggregated over a whole project correlates very highly with its size in LOC [39].

To alleviate this issue, we can instead use the *mean* (Equation 2.18) or *weighted mean* (Equation 2.19).

$$mean(M) = \frac{\sum_{m \in M} m}{|M|} \tag{2.18}$$

$$wmean(M) = \frac{\sum_{m \in M} weight(m) \cdot m}{|M|} \tag{2.19}$$

However, the mean also has some problems. First, it "smoothes out" results [87, 88], meaning any interesting details (like specific parts of the project

or specific metrics in a composition having deviating values) are lost. In fact, the mean is suitable only for symmetrical distributions [89]. The distributions for metrics in the field of software engineering are usually skewed [90], meaning the mean is not appropriate.

To demonstrate how the mean can be problematic, consider cyclomatic complexity measurements on two software projects. Suppose that the first project consists of 1000 files of cyclomatic complexity 3. The mean cyclomatic complexity is then also 3. Suppose the second project also consists of 1000 files. The majority of these methods are similarly complex (cyclomatic complexity of 3), but 10 methods have a cyclomatic complexity of 10. This finding may be relevant; it seems like a few methods are extremely complex, which could threaten the reliability of the entire system. However, the mean cyclomatic complexity will be close ($\frac{10 \cdot 10 + 990 \cdot 3}{1000} = 3.07$) to the mean cyclomatic complexity of the first system.

For this reason, alternative aggregation techniques that originate from the field of econometrics have been applied in a software engineering context [90]. Examples of these include the Theil index, Gini index and Atkinson index. These aggregation techniques are used to compute economic inequality based on very similar distributions as the ones that are common for software metrics.

2.8 Conclusions

In this chapter, we have seen the definitions of the software metrics TIOBE uses.

TIOBE metrics are exclusively code-based product metrics. This means that only a single version is evaluated at a time. While this is simple and convenient in terms of implementation and interpreting results, the inclusion of historical metrics could be useful for defect prediction. While integrating these metrics in the TQI may not be desirable simply because historical data is not always available, support for these metrics in TICS could be useful for customers.

From the code-based metrics we discussed, three are not integrated into the TQI: the Halstead complexity measures and Mutation coverage. Halstead complexity measures can be a replacement for cyclomatic (McCabe) complexity. Adding them in addition to cyclomatic complexity is not wise, since both are complexity measures and they highly correlate [39, 50]. Mutation testing would be a great addition to the TQI, since it is a good predictor of test suite quality [62]. Unfortunately, mutation testing is very computationally expensive, meaning it is infeasible for large code bases. Even normal code coverage tools take so long to run that their execution is limited to weekly or monthly runs for large projects. Mutation testing requires, by nature, multiple runs of the test suite. Mutation testing could be a nice addition to TICS for smaller projects. Finally, the Chidamber and Kemerer metrics are interesting since they are specifically aimed at object-oriented software. Almost all software that the TQI is used on uses the object-oriented paradigm, but the metrics in the TQI are predominantly defined at the method (cyclomatic complexity, abstract interpretation) or line (compiler warnings, coding standards) level.

The TQI is discussed in more detail in Chapter 3. It includes a discussion of how the metrics from this chapter are combined to result in a metric for overall project quality.

Chapter 3

The TIOBE Quality Indicator

In order to assess the validity of the TQI, we need to have a clear understanding of how it works. We opt for a top-down approach. The description of the TQI in this chapter summarizes a few documents written by TIOBE describing the TQI in detail [16, 91].

We start by introducing the TQI in terms of its goal and presentation in Section 3.1. We discuss the compliance factor, an aggregation method specifically aimed at metrics for violations (such as coding standard violations or compiler warnings) in Section 3.2. Section 3.3 describes how the metrics are integrated and composed to form the TQI. Next, we describe the TQI in detail and consider the design decisions and their ramifications in Section 3.4. Finally, Section 3.5 describes the evolution of the TQI definition over time.

3.1 Introduction to TQI

TIOBE customers are interested in an assessment of the quality of their software. The tool TICS can perform such an assessment automatically. A TICS assessment of a software product results in a grade between A and F. This grade is based on the TQI score, which is a number on a scale of 0 to 100. This scale is divided into categories, forming a mapping from a TQI score to the corresponding grade. The categories with their corresponding names and TQI score ranges are displayed in Figure 3.1.

The reduction from scores to grades is primarily aimed at managers. A single grade is a very simple representation of quality. This is attractive from a marketing perspective; it enables managers to acquire some indication of software quality without needing to understand intricate details.

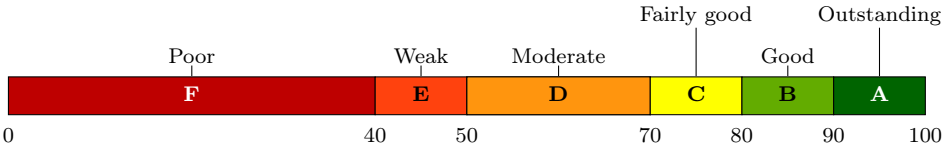


Figure 3.1: Distribution of categories on TQI score.

Note that the distribution of categories over TQI scores (Figure 3.1) is not uniform. Instead, some categories have been enlarged to create what TIOBE describes as "a Gaussian-like distribution" [16]. Also note that the reduction from a number to category causes the results to be less actionable, which is undesirable [78]. Another undesirable aspect is that the reduction of a number to categories causes a "staircase effect". This means that meaningful fluctuations in the TQI score may not result in a different category, while tiny fluctuations that are less meaningful may result in a sudden change of category. This means that this category should be seen as an addition, not a replacement, of the TQI score.

3.1.1 Metric inclusion criteria

A subset of the metrics that were presented in Chapter 2 are included in the TQI. In deciding which metrics are included into the TQI, TIOBE used three criteria:

1. **Automatically calculable**: for a metric to be included in the TQI, it should be possible to compute its value using software tools without human intervention for individual measurements;

2. **Based on a single version**: the calculation of a metric should not require older versions of the software;

3. **Standardizable**: it should be possible to decide which values for metric are considered "good" and which are considered "bad".

The first requirement means that any manual measurement (such as rating a system manually or separately activating tools) is not acceptable. This results in the exclusion of process and project metrics. This requirement is necessary because, the goal of the TIOBE is to automatically measure software quality [15].

The second requirement results in the exclusion of historical metrics. This is because all historical metricss are an indicator of changes in the software. As such, they are based on an comparison of at least two versions of the software. This requirement is included because it enables the assessment of software quality based on a single version, instead of producing a result only after multiple version have been analyzed.

For the final requirement, we should be able to establish a *direction* and a *threshold* for a metric. In this context, the term "direction" refers to the direction of a correlation coefficient. For example, cyclomatic complexity increases as the complexity of a project increases. Since complexity is considered bad, the direction of this metric is negative. A higher code coverage, on the other hand, implies a larger portion of the code is tested, which is considered good. Note that these directions are established by TIOBE based on face validity; empirical results may invalidate the directions for some metrics.

Thresholds are values that establish a demarcation between "good" and "bad" values for a metric. For example, McCabe [26] suggests an upper bound of 10 for the cyclomatic complexity of a method. TIOBE considers average cyclomatic complexity values under 3 to be good and values above 5 to be very bad [16]. Like most thresholds, these thresholds are based on experience or a viewpoint on which metric values are acceptable [92].

Table 3.1: Weights for the metrics the TQI composes.

Metric	Weight
Code Coverage	20%
Abstract Interpretation	20%
Cyclomatic Complexity	15%
Compiler Warnings	15%
Coding Standards	10%
Code Duplication	10%
Fan-out	5%
Dead Code	5%

An alternative to this is to take a more statistical approach [93], in which thresholds are derived that can differentiate between commonly found values and the remaining, uncommon values. While this method can give additional insight, it cannot serve as a replacement for evaluation by an expert. One reason for this is that common practices are not necessarily good practices. In addition to this, the results of the analysis depend on the sample that was used.

3.1.2 Composition

The TQI score is composed of aggregates that represent the following 8 attributes:

1. Code coverage
2. Abstract interpretation
3. Cyclomatic complexity
4. Compiler warnings
5. Coding standards
6. Code duplication
7. Fan-out
8. Dead code

Each of these attributes is assigned a number on the same scale as the TQI, i.e., 0 to 100. Each attribute also corresponds to a category (A to F) in the same way as the TQI. This means that raw data is first aggregated to form a variety of scores that are then composed into a single quality indicator. This differs from the approach by Mordal-Manet *et al.* [5]; they advice to first compose (i.e., combine code coverage, abstract interpretation and other attributes) and then aggregate afterwards (i.e., combine different attributes to form a single indicator of quality). The goal of this is for attributes at lower level to "retain the intended semantic of the composition" [5].

Attributes are composed using a weighted mean. The weights for the metrics that the TQI is composed of are presented in Table 3.1. In some cases, some components may be missing. One reason for this is the absence of a license for

tools. Abstract interpretation is done through tools that are relatively expensive and therefore are not purchased by all companies. Similarly, some languages, such as JavaScript, are typically interpreted instead of compiled. This means there are no compiler warnings available in a typical use case. In the case of JavaScript, TIOBE uses the Closure Compiler[1] to obtain compiler warnings. The TQI is defined to be conservative in this case. That is, absent metrics are assigned a value of zero. The rationale behind this is that the TQI should never give a false sense of security. This means that a lack of measurements in one of the categories can never result in a higher TQI score.

The mapping from a raw metric to its equivalent value in the $[0, 100]$ range is different for each metric. Some metrics, such as code coverage, undergo a linear transformation for their inclusion in the TQI. For other metrics, such as abstract interpretation, the so-called compliance factor is computed based on the number and severity of the various violations, which is then adjusted for inclusion in the TQI.

The TQI and the metrics it is based on are presented to the customer in a label resembling an energy label. An energy label is a familiar concept, which makes it easy to understand. It allows management to base their decisions on a broad impression, while engineers can use the more detailed information. Figure 3.2 contains an example of such a label.

3.2 TIOBE compliance factor

The compliance factor [91] is an invention by TIOBE that is used to compute a single number based on a set of violations of various types. It was originally defined for use with coding standard violations under the name "static confidence factor" [91], but has subsequently been renamed to "compliance factor" and is now used for abstract interpretation, coding standard violations and compiler warnings [16].

3.2.1 Static defect count

The compliance factor is based on the notion of a coding standard, weights for each of the coding standard rules and the number of violations for every rule. A *coding standard* C is defined as a set of rules. Every rule $\mathcal{R} \in C$ is assigned a *defect probability* $DP(\mathcal{R})$, which is the probability that the violation of the rule corresponds to a real software defect.

We denote the number of violations of a rule $\mathcal{R}$ within a program π as $V_{\mathcal{R}}(\pi)$. Using a weighted sum, TIOBE computes the *static defect count* for a program using Equation 3.1.

$$SDC_C(\pi) = \sum_{\mathcal{R} \in C} (DP(\mathcal{R}) \cdot V_{\mathcal{R}}(\pi)) \tag{3.1}$$

The static defect count is the estimated number of bugs in the program based on the violations of the coding standard.

[1] `https://developers.google.com/closure/compiler/`

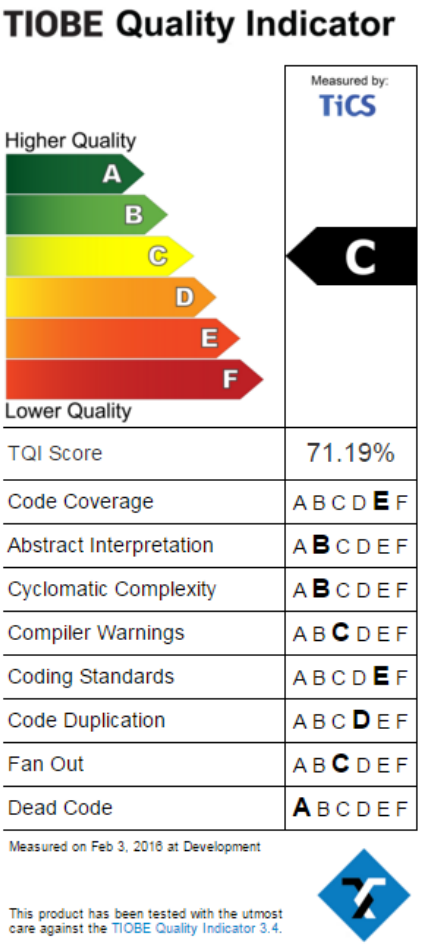


Figure 3.2: An example of a label produced by TICS. It includes a TQI score and an A–F grade for each of the metrics.

3.2.2 Estimating defect probability from severity levels

The static defect count is based on the notion of defect probability. While the mathematical definition is unambiguous, the defect probability of a rule is hard to establish. In practice, experts assign a *severity level* to each rule. This is a strictly positive natural number that indicates how serious a violation of this rule is. Lower values indicate more serious violations. Given a severity level SL for a rule $\mathcal{R}$, TIOBE estimates the defect probability using Equation 3.2.

$$DP(\mathcal{R}) = 4^{-SL(\mathcal{R})} \tag{3.2}$$

Note that the choice for a base of 4—and even the choice for an exponential growth—is somewhat arbitrary.

3.2.3 Controlling for size and coverage

The following aspects of static defect count are problematic:

- Dependence on the number of rules;
- Dependence on the size of the code;
- The absence of data due to failing tools.

The compliance factor is an attempt to mitigate the impact of these weaknesses.

The dependence on the number of rules is referred to as the "coding standard extension paradox" [91]. The paradox lies in the fact that adding more rules to the coding standard (which makes it more useful) results in an apparent decrease in code quality. This is because the static defect count can only increase as a result of the addition of more rules. This is accounted for by dividing by the number of rules of the same severity level. This results in adjusted definitions for defect probability (Equation 3.3) and static defect count (Equation 3.4).

$$DP^*(\mathcal{R}) = \frac{4^{-SL(\mathcal{R})}}{|\{\mathcal{R}' | SL(\mathcal{R}) = SL(\mathcal{R}')\}|} \tag{3.3}$$

$$SDC^*_C(\pi) = \sum_{\mathcal{R} \in C} (DP^*(\mathcal{R}) \cdot V_{\mathcal{R}}(\pi)) \tag{3.4}$$

Similarly, larger codebases will typically result in a larger amount of defects. This is simply a result of the aggregation technique and results in the metric to become a proxy for size[2]. For this reason, the static defect count is also divided by the size of the code in LOC.

Finally, if a tool crashes, this results in a lower static defect count. To prevent the compliance factor from giving a false sense of security, the *violation coverage* $VC_C(\pi)$ is taken into account. The violation coverage is the proportion of the program in LOC that could be checked for violations, i.e., for which the tool did not crash. This way, if only 80% of the program is checked, the maximum score becomes 80% and the score is scaled to the range $[0, 80]$ instead of $[0, 100]$.

[2] This is discussed in more detail in Section 2.7.

3.2.4 Mapping to $[0, 100]$ scale

Finally, the *static confidence factor* (or compliance factor) is a mapping of the static defect count to a scale of 0 to 100, normalized by the number of lines of code (Equation 3.5).

$$SC_C(\pi) = 100 \cdot VC_C(\pi) \cdot \frac{KLOC(\pi)}{SDC^*_C(\pi)} \tag{3.5}$$

This formula matches the intuition that software with a higher volume (KLOC, 1000 lines of code) and a lower estimated defect count (SDC_C) gets a higher score. If only a limited amount of the code can be checked, the score is decreased through the violation coverage VC_C.

3.3 Metric integration into TQI

Each of the metrics is integrated into the TQI using a formula that maps it to a value in the interval $[0, 100]$. We introduce the following notations in order to write down these formulas in a concise way:

- We distinguish between metrics by using subscript abbreviations in capital letters, namely TC (test coverage), AI (abstract interpretation), CC (cyclomatic complexity), CW (compiler warnings), CS (coding standard violations), CD (code duplication), FO (fan-out) and, DC (dead code);
- The raw metric value (e.g., the average cyclomatic complexity or abstract interpretation violations): m_{X};
- The compliance factor for a metric: $SC(m_{\text{X}})$;
- The TQI value (i.e., the value that has been mapped to the interval $[0, 100]$) for a metric X: TQI_{X}.

The formulas for all TQI metrics are presented in Equations 3.6 to 3.13.

$$TQI_{\text{TC}} = \min(0.75 \cdot m_{\text{TC}} + 32.5, 100) \tag{3.6}$$
$$TQI_{\text{AI}} = \max(SC(m_{\text{AI}}) \cdot 2 - 100, 0) \tag{3.7}$$
$$TQI_{\text{CC}} = \min(\max(140 - 20 \cdot m_{\text{CC}}, 0), 100) \tag{3.8}$$
$$TQI_{\text{CW}} = \max(100 - 50 \cdot \log_{10}(101 - SC(m_{\text{CW}})), 0) \tag{3.9}$$
$$TQI_{\text{CS}} = SC(m_{\text{CS}}) \tag{3.10}$$
$$TQI_{\text{CD}} = \min(-30 \cdot \log_{10}(m_{\text{CD}}) + 60, 100) \tag{3.11}$$
$$TQI_{\text{FO}} = \min(\max(120 - 5 \cdot m_{\text{FO}}, 0), 100) \tag{3.12}$$
$$TQI_{\text{DC}} = \max((100 - 2 \cdot m_{\text{DC}}), 0) \tag{3.13}$$

Note that TQI_{CS} is equal to the compliance (confidence) factor based on coding standards. Since this value is already in the range $[0, 100]$, no modifications to this value are necessary. Although both TQI_{AI} and TQI_{CW} are already in the range $[0, 100]$ as well, these are transformed to achieve a more even spread across categories.

Finally, the TQI of the composed metrics is defined as follows (Equation 3.14):

$$\begin{aligned} TQI &= \sum_{\text{X} \in \{\text{TC,AI,CC,CW,CS,CD,FO,DC}\}} weight(\text{X}) \cdot TQI_{\text{X}} \\ &= 20\% \cdot TQI_{\text{TC}} \\ &+ 20\% \cdot TQI_{\text{AI}} \\ &+ 15\% \cdot TQI_{\text{CC}} \\ &+ 15\% \cdot TQI_{\text{CW}} \\ &+ 10\% \cdot TQI_{\text{CS}} \\ &+ 10\% \cdot TQI_{\text{CD}} \\ &+ \ 5\% \cdot TQI_{\text{FO}} \\ &+ \ 5\% \cdot TQI_{\text{DC}} \end{aligned} \tag{3.14}$$

The weights in Equation 3.14 are established by TIOBE based on their perception of the importance of each of these metrics. At this time, there is no reasoning provided for these weights and no evidence showing their appropriateness for quality assessment.

3.4 Implementation

Even with the definitions of the metrics the TQI is composed of (from Chapter 2) and the way raw metrics are integrated in the TQI, we have abstracted from some of the details in the implementation of the TQI. We will now discuss some of these details and critically evaluate some of the choices that have been made for the aggregation of metrics into the TQI.

We use clear labels to indicate where we discuss a shortcoming of the TQI. The remaining text is simply a description of how the metrics are computed.

3.4.1 Constants

The constants in the TQI formulas (Equations 3.6 to 3.13) and the weights for the weighted average (Equation 3.14) are somewhat arbitrary. That is, constants like 0.75, 32.5, 140, and other numbers that appear in these equations have not been derived from theory, but are instead chosen to create a desirable distribution for the metrics [16]. These distributions are desirable in the sense that projects are distributed over the categories, as opposed to most projects ending up in the same category. In addition, it ensures a grade of A is hard to achieve, since only a very small subset of projects end up in this category. TIOBE aims to base the weights for the metrics on statistical data in the future [16], but for now the weights are established from industry experience.

Shortcoming 1 The constants in the TQI formulas lack a strong theoretical or statistical basis.

Table 3.2: Overview of code coverage tools and the types of coverage they support. These are all tools that are supported by the TICS framework [17]. The supported types of coverage were taken from the official web pages [94–104] for each tool.

Tool	Statement	Function	Decision (branch)	Condition	Condition/Decision	MC/DC	Path
BullseyeCoverage (Bullseye) [94]		✓	✓	✓	✓		
Squish Coco (FrogLogic) [95]	✓		✓	✓			
CTC (Testwell) [96]	✓	✓	✓	✓	✓	✓	
gcov/lcov (SourceForge) [97]	✓	✓	✓				
PureCoverage (IBM) [98]	✓	✓					
C++Test (Parasoft) [99]	✓		✓	✓		✓	✓
VectorCAST (Vector Software) [100]	✓		✓			✓	
NCover (NCover) [101]	✓		✓	✓			
OpenCover (open source) [102]	✓	✓	✓				
Jacoco (open source) [103]	✓	✓	✓				
Cobertura (open source) [104]	✓		✓				

3.4.2 Type of code coverage

As we discussed in Section 2.3.8, there are various types of code coverage, with the most popular being statement coverage, decision coverage and condition coverage. Measuring code coverage involves running the tests while the implementation code is instrumented[3]. As such, it is highly specific to the language and framework. This means coverage for different software projects is typically collected by different tools.

An overview of tool support for various types of code coverage is shown in Table 3.2. All of these tools are supported by TICS [17]. Almost all tools support statement coverage. Decision coverage is also almost universally supported. However, there is no type of coverage that is supported by all tools. This makes it difficult to decide which type of coverage to include in the TQI. For this reason, the average of statement, decision and condition coverage is taken (if available). For example, a project with 100% statement coverage, 90% decision coverage and an unknown condition coverage results in a TQI coverage of $\frac{100\%+90\%}{2} = 95\%$.

The reason why the straightforward statement coverage is not supported by all tools is that Bullseye, the developer of BullseyeCoverage, chooses not to provide this metric [105]. This is because they think statement coverage should only be used if no other coverage metric is available [105]. The underlying reason is that it is particularly sensitive to computational statements instead of decisions [63], which are arguably more important to test.

This design decision is problematic, since it means that the coverage score is

[3] Instrumentation is the process of inserting instructions into source or binary code with the goal of collecting data about the execution.

highly dependent on the tool that is used to measure the coverage. Decision and condition coverage are typically lower than statement coverage[4]. This means that any given project scores better using a tool that only supports statement coverage, since it will reach a higher coverage relatively easily. A high score is much more difficult to obtain when using a tool that only supports decision and condition coverage. This is particularly unfortunate, since the different types of coverage correlate so highly that we can conclude they measure the same thing [66].

Shortcoming 2 The TQI code coverage score is highly dependent on the coverage types supported by the tool.

This suggests an alternative approach of choosing a particular type of coverage and approximating it using a statistical method like linear regression when it is not available. While imperfect, this method does not have the drawbacks discussed earlier. A downside of this method is that the semantics of approximated metrics are not as clear. For example, developers can understand that a low statement coverage is caused by tests not executing all lines. To improve coverage, they can look at the parts of code that are not covered and write tests for them. However, if we estimate statement coverage from any of the other coverage types, the estimated coverage may not increase when more statements are covered.

3.4.3 Abstract interpretation tools

Abstract interpretation is inherently language-specific, so it is accomplished with a different tool [17] depending on the project.

Shortcoming 3 The differences between tools, particularly the severity levels for violations and the overall volume of violations, could strongly impact the resulting compliance factor and TQI score.

Abstract interpretation tools are typically not free. This has resulted in one[5] TIOBE customer opting to disable abstract interpretation in order to save costs on license fees. This choice results in a potential 20% decrease in the overall TQI due to its high weight (see Table 3.1).

Shortcoming 4 A lack of an expensive license can negatively affect a company's TQI score.

3.4.4 Interpretation of cyclomatic complexity values

TIOBE considers average cyclomatic complexity values under 3 to be good and over 5 to be very bad. Equation 3.8 maps the interval $[3, 5]$ of average cyclomatic

[4] For an intuition of why this is, refer to Section 2.3.8. The average coverage measurements observed in software TIOBE monitors are 31.10%, 27.53% and 21.93% for statement, decision and condition coverage respectively.

[5] at the time of writing

complexity linearly to TQI scores $[40, 80]$. This corresponds to categories B to E. The remaining two categories (A and F) are reserved for the cyclomatic complexities under 3 or above 5.

Since TICS stores measurements per file, there is no functionality in place to store the cyclomatic complexity per method. To still be able to compute a sensible average cyclomatic complexity, TIOBE stores both the total cyclomatic complexity of a file (the sum of cyclomatic complexity measurements) and the number of methods in a file. To compute the average cyclomatic complexity on any granularity level, the total cyclomatic complexity is simply divided by the total number of methods.

Note that an average cyclomatic complexity of 7 results in a score of 0; higher complexity is ignored. Similarly, an average complexity of 2 is sufficient to reach the maximum score of 100.

Shortcoming 5 The TQI cyclomatic complexity score represents a limited range only.

3.4.5 Compiler verbosity settings

Compilers may suppress a subset of the warnings they generate. The reasoning behind this is that too many unimportant warnings may overwhelm the user, obscuring important warnings in the process. When assessing the quality of the product, hiding warnings is undesirable. For this reason, TIOBE uses the highest possible warning level for all compilers.

Note that this does not solve the problem of different compilers being stricter or less strict than others. While the compliance factor has been designed to deal with these differences, it is not clear to what extent it is effective.

Shortcoming 6 The compliance factor lacks empirical validation.

Another issue with compiler warnings is that they are not necessarily independent. Therefore, a single problem may result in many compiler warnings, meaning the volume of compiler warnings may be relatively meaningless.

Shortcoming 7 The number of warnings may be meaningless if warnings are dependent.

3.4.6 Coding standard adherence

Coding standards are evaluated through the TIOBE compliance factor [91] (see Section 3.2). This allows one to compare projects in terms of coding standard adherence.

Companies choose their own coding standard; TIOBE simply measures their adherence to this standard and includes this in the TQI. This means that a company could easily "trick the system" by choosing a lenient coding standard, or a coding standard with a lot of rules that have little chance of occurring in

practice. In that sense, the current policy is closer to a self-assessment than an objective measure for quality. This means adherence to coding standards measures a company's success at achieving its goals, not the quality of the software it produces. At the very least, adherence to coding standards can only be compared meaningfully if the coding standards are equivalent. It should be noted that most companies do use the default coding standard TIOBE provides, with only some larger companies making adjustments by adding their own rules.

Shortcoming 8 The ability of companies to choose their own coding standard enables companies to artificially inflate their TQI score.

More importantly, there may be legacy code in the system that does not adhere to coding standards. A programmer seeking to improve quality is therefore motivated to change this code to adhere to the standard. Considering that every change has a (possibly small, but non-zero) chance of introducing a fault [106], enforcing a strict adherence to a coding standard and encouraging developers to make changes to adhere to it may decrease the reliability of the software [75].

Shortcoming 9 The negative effect of legacy code on the TQI coding standard score may prompt developers to introduce bugs during the process of resolving violations.

3.4.7 Duplicated code

TIOBE uses an implementation of the Rabin-Karp algorithm [107] to detect code clones. This algorithm is restricted to detecting syntactic equivalence. In particular, it can detect type-1 and possibly type-2 clones. The ability to detect type-2 clones depends on the tokenizer, which extracts tokens from source code. TIOBE uses a tokenizer to allow the detection of type-2 clones for C, C++, C#, Objective-C, PL/SQL, Python, Scala, and VB.NET, but the clone detection for other languages (such as Java and JavaScript) is restricted to type-1 clones. This distinction is implicit; there is no indication within TICS which type of clones are detected.

Shortcoming 10 The type of code clones that are used for duplicated code are unknown.

Since small code clones are less interesting, TIOBE only considers clones consisting of at least 100 consecutive tokens. The degree of code duplication is then measured by counting the number of lines of code that contain a clone and dividing it by the total number of lines of code in the system.

If two (or more) pieces of code are considered a clone, both (or all) of the code contributes to the percentage of duplicated code. The rationale behind this is that when a single piece of code is cloned multiple times, this should increase the percentage of duplicated code.

Listing 3.1: Illustration of different import mechanisms in Java.

```
1 import java.util.List;
2 import java.util.ArrayList;
3 import java.util.LinkedList;
4
5 import java.util.*;
```

3.4.8 Measurement of fan-out

TIOBE defines fan-out as "the average number of imports per module" [16]. Identifying imports is highly language-dependent, since languages use different concepts and keywords to facilitate importing modules. For C and C++, the number of `include` statements is used; for Java the number of `import` statements is used. For C#, instead of counting `using` statements, a more complicated analysis is used. This has prompted the introduction of a separate tool under the name TICSCil (which is discussed later in this section).

While the choice of imports as a proxy for fan-out seems to make sense, it should be noted that in some languages, a class can rely on other components without explicitly importing them. Java is an example of this. While it is customary to import a class before using it (i.e., `import java.util.List;` before declaring `List list;`), this is not necessary (we can instead directly declare `java.util.List list;`).

Shortcoming 11 Some languages may allow the use of external modules without requiring `import` statements.

It could be argued that these examples, while theoretically valid, are not common in practice. It is highly unusual to use classes without explicitly importing them. Using wildcards in import statements is generally discouraged as well [108].

We discuss some language-specific challenges to counting imports in the paragraphs below.

Wildcards

A problem with counting a number of statements is that the same imports can be achieved in different ways. One example of this is illustrated in the Java code in Listing 3.1. The `import` statements in lines 1 to 3 can be replaced by the single statement in line 5. In fact, this can be done for an arbitrary number of imports. This "trick" is dependent on the structure of the modules that are imported.

To deal with this issue, TIOBE uses a heuristic to estimate the number of imports for a wildcard statement. The current estimate is 5 [16]. This estimate was previously used for all languages for which fan-out is measured. With the introduction of TICSCil, it is no longer used for C#. The remaining languages, including Java, still use this heuristic.

Dynamic imports

Counting imports is even trickier in dynamic languages, like Python. The approach by TIOBE fails for the following reasons:

- A high-level import of a module gives access to the entire module, including all of the submodules it contains. For example, consider a module X which contains modules A, B, and C. We can import these modules using three statements `import X.A; import X.B; import X.C`. However, we can accomplish the exact same result by using `import X`.
- Extending on the previous example: we can use `from X import A, B, C` to achieve the same result. A statement like this would actually also be counted as three imports.
- Artificially decreasing fan-out can be accomplished by placing all imports for an entire software system in a single file, and importing that file in every other file. This achieves the minimal amount of imports without compromising functionality.

  This can be done because imports are transitive, i.e., importing X also gives access to all modules that are imported by X. For example, if X contains a line `import Y`, we can access `Y` without importing it by importing X and referring to `X.Y`.

  This is just an example of "cheating the system"; a design like this is actually bad in terms of maintainability and execution speed and can therefore not be recommended.

Shortcoming 12 Counting imports to measure fan-out is problematic in dynamic languages.

Namespaces

C# is an example of a language that uses an import system based on namespaces. This means that all imports are comparable to imports using wildcards in Java, in that the entire namespace is imported instead of the portion inside of it that is used. Namespaces can consist of hundreds of classes, but typically very few are actually used. This means the heuristic of 5 imports per `using` statement is especially imprecise, resulting in inaccurate measurements.

For this reason, TIOBE introduced a separate tool under the name TICSCil. This tool uses the .NET framework to accurately determine how many external classes are used in each file. Since this tool is language-specific, the remaining languages still use the same heuristic of 5 imports per wildcard.

The introduction of TICSCil has had a massive influence on measurements, as can be seen in Figure 6.8 (page 72).

3.4.9 Dead code

As we pointed out earlier (Section 2.3.5), the concept referred to by TIOBE as "dead code" is actually "unreachable code". The dead code component is

included in the TQI in the form of the percentage of lines of code that are unreachable. A higher percentage results in a lower score.

When measuring unreachable code, TIOBE does not consider any unused variables or similar unused code, but restricts to unused files and methods. Dead code is measured in terms of LOC. This means that an unreachable method or file is counted more heavily when it is larger. It could be argued that the number of separate occurrences of unreachable code is a more reasonable metric, since each of these occurrences signals that some method or file is no longer used.

Dead code can be detected through abstract interpretation. If this results in a violation, it is taken into account for the abstract interpretation component of the TQI.

3.4.10 Order of aggregation, composition and TQI integration

A raw metric value undergoes at least three processes before it impacts the TQI of the entire project. These are:

1. *Aggregation*: the process of aggregating the measurements on multiple files to obtain a value representing the metric over the entire project;
2. *TQI integration*: the process of applying one of the formulas in Section 3.3 to obtain a TQI component with a value in the domain $[0, 100]$;
3. *Composition*: the process of combining metrics into a single TQI value using a weighted average.

These processes are executed in the order they are listed above. This order has an influence on the TQI values.

Mordal-Manet *et al.* [5] suggest to compose the metrics before aggregating them. Their reason for this choice is that this will "retain the intended semantic of the composition" [5]. This is reasonable, since the aggregation over files has unintended side effects, such as hiding details by smoothing them out [87, 88]. On the other hand, it removes the possibility of using a different aggregation for different metrics. This is a challenge, since not all metrics are evaluated on the same granularity level. For instance, cyclomatic complexity is computed per method, while fan-out is computed per class.

Shortcoming 13 The order of aggregation and composition used for the TQI is discouraged by literature.

TIOBE does offer composed measurements on lower levels (e.g., file level and package level). For these cases, the aggregation is simply restricted to the requested level. The formulas mapping the measurements to the $[0, 100]$ domain are applied here as well.

This does have some unintended side effects. In some cases, the overall TQI value for a project is higher or lower than all of its top-level packages. This means that a project consisting of two packages P and Q could have an overall TQI value in the B range, while P and Q are both assigned values that result in an A score. This is an artifact resulting from the order of these steps that

can be confusing to end users, who may expect the value to result from some kind of weighted average.

Shortcoming 14 The integration of a metric into the TQI may lead to confusing side-effects.

3.5 Evolution of TQI definition

So far, we have discussed a single version of the TQI. However, the TQI definition is updated periodically. In this section, we consider the consequences of these changes.

An important property of any metric, including the TQI, is the consistency of its values over time [24, p. 11]. This allows one to compare the current value to historical values of the TQI. Fortunately, the consistency of these values is enforced by TICS. The TQI is computed dynamically from the raw metrics at all times.

This dynamic recomputation also has a negative side. When the TQI definition changes, historical values also change, i.e., a more recent definition of the TQI is retrospectively applied. This may be confusing to the end user. We should note that these changes only occur when the version of TICS is upgraded, meaning the TQI does not change as long as the client stays on the same version.

At the start of this graduation project (September 2015), the extent to which TIOBE used versioning for TICS and the TQI was limited. While each version of the TQI was assigned a version number, there was no changelog comparing historical versions. TICS did refer to a specific version of the document, but only the latest version of the TQI document was available on TIOBE's website. This means that when a user was using an old version of TICS and read the TQI definition, they were presented with a version that was possibly inconsistent with the implementation. Only a manual comparison of the version numbers could reveal this inconsistency.

A recommendation resulting from this project was to start maintaining a version history and present historical versions of the TQI document online. TIOBE has already decided to adhere to this recommendation; version history is now presented on the TIOBE website [109]. Additionally, previous versions of the TQI document are available. This is an important improvement, since it allows anyone to investigate how the TQI definition changed over time. Figure 3.3 contains the current changelog for the TQI.

3.6 Conclusions

In this chapter, we have seen how the TQI is computed based on raw metrics. Using this information, we can propose some recommendations to improve the TQI.

First, the constants in the various TQI integration formulas (Equations 3.6 to 3.14) seem arbitrarily chosen. They result from the metric data that has been collected in the past, but are periodically updated. A better way of establishing these constants would be to base them on empirical evidence. One way to do this

3.7 Relaxed TQI for code duplication a bit.

3.6 Added multiple compiler constraint to TQI.

3.5 Made a distinction between internal and external fan out.

3.4 Updated the average cyclomatic complexity.

3.3 Excluded header files from metric "Code Duplication".

3.2 Updated TQI document for improved Code Duplication calculation.

3.1 Updated TQI document for new C# fan out calculation.

3.0 Factored out metric coverage in TQI definition.

2.2 Added percentages to the TQI scores.

2.1 Adjusted TQI metric boundaries.

2.0 Improved the TQI definition for compiler warnings.

1.2 Added recommended TQI levels.

1.1 Improved the TQI energy label.

Figure 3.3: Changelog for the TQI (reproduced from the TIOBE website [109]).

would be to apply regression and use the resulting coefficients as an indication of the importance of each metric.

As we discussed in Section 3.5, TIOBE recently started maintaining a change-log on their website [109]. Another improvement in terms of traceability could be made by giving insight in the origin of metrics. For instance, when a user sees a change in the TQI score, TICS could show which metric(s) and file(s) caused this change. This would be even more useful if changes in versions of tools are also recorded. Due to its dependency on external tools, it is inevitable that the historic measurements recorded by TICS sometimes change purely as a result of changing (the version of) a tool, such as an abstract interpretation tool or compiler. These tools are typically updated by the TIOBE service team. If they would store information about these upgrades in the TICS database, this could help users to understand the cause of changes in measurements.

The next change is related to code coverage. Currently, the average of the available coverage metrics is taken, depending on the tool that is being used. A better alternative is to simply use a widely supported type of code coverage (statement or decision coverage) directly. This is justifiable, since aggregated code coverage measurements correlate strongly. The other coverage metrics can be offered within the TICS interface, but do not need to be part of the TQI definition.

Another change is to only support standardized coding standards for inclusion in the TQI. While some customers may find it attractive to add more rules in order to comply to some alternative coding standard, no company should be allowed to add or modify rules that could influence their TQI score.

The problems we found with the computation of fan-out can mostly be addressed by developing tools that inspect imports more closely.

Finally, TIOBE can consider to change the order of aggregation and composition after the advice of Mordal-Manet *et al.* [5]. This could prevent some of the confusing effects we discussed in Section 3.4.10.

Chapter 4

Establishing the relation between metrics and bugs

Since the topics of software quality and software metrics are popular among researchers and relevant for industry, there exists a large body of literature related to this thesis.

The most important goal of this chapter is to present the various approaches to empirical validation of software metrics. We discuss how repository mining can be used to obtain data on defects in Section 4.1. One particularly relevant example of this field, bug prediction, is covered in Section 4.2. Bug prediction is relevant because research in this area uses techniques to examine historical occurrence of faults, which can also be applied to validate software metrics. We discuss some details of the Git version control system that become relevant when applying bug prediction in Section 4.3. We discuss the distinctions we can make between various kinds of bugs in Section 4.4. The generalizability of results obtained from bug prediction is covered in Section 4.5. Finally, we consider some previous research on the TQI in Section 4.6.

4.1 Repository mining

As explained in Chapter 1, we want to use bugs as a proxy for software quality to validate the TQI metrics. Unfortunately, establishing the number of bugs in a piece of software is no trivial task. Bugs can only be counted if they have been found in the first place.

However, most companies do keep a historical record of faults that have been encountered in their software project. By automatically mining software archives, we can find big volumes of data that can be used as evidence to test a hypothesis [110]. This is attractive from a statistical point of view, since a large sample size can provide evidence for our hypotheses. It is also a relatively inexpensive way to conduct research. If we grant that faults in the history of the project are similar to faults that will appear in the future, we can evaluate metrics on historical data to assess their validity.

Examples of software archives include version control systems and bug tracking systems, but also the source code of the software itself and documentation.

4.1.1 Data reliability

While repository mining results in a large amount of data, this data is generally noisy and incomplete [110]. The quality of this data is of vital concern, since noisy data can lead to incorrect conclusions [111–113].

An example of a problem that impacts the validity of data originating from a bug tracking system is that some bugs may be reported multiple times. This is the case, for example, when multiple people independently find an issue without realizing it is induced by the same root cause [114–116]. The reverse is also possible. The fix for a problem could be distributed over multiple commits. This is particularly relevant when the task of fixing bugs and testing the software is not assigned to the same person. A developer may attempt to fix a bug and commit it. A tester then tests the fix, and finds a flaw. The bug is then reopened, and the developer commits a second fix.

It has been suggested that industrial projects may be subject to stricter regulations [112], which could result in higher-quality data compared to data originating from open-source projects. TIOBE customers in particular are often subject to Food and Drugs Administration (FDA) regulations [117].

4.1.2 Version control systems

Version control systems are a major source for repository mining. A Version Control System (VCS) is a system that keeps track of different revisions of (the source code of) software. Additionally, it contains documentation in the form of commit messages, which explain why a change was made and may refer to other documentation.

A version control system stores *revisions* (versions) of software or other files in a structured way. Users *commit* changes to a version control system to store them. Each commit results in a unique version, which is typically an integer that is incremented for every new commit.[1] Some VCSs, such as CVS, record change information per file. Most modern VCSs instead use a notion of *changeset* to refer to the contents of a commit over multiple files, and therefore use a single version number for all files that are tracked by the system.

Most VCSs provide *branching* functionality. This means that multiple slightly different versions of the software are tracked at once. These branches can later be *merged* (either manually, automatically, or through a combination of the two), such that the changes that were committed to the separate branches are combined. A popular application of branches is to use a stable and unstable branch, which provides the distinction between the version of the software that is deployed to customers and a version with newer (possibly unfinished or buggy) features. Another possibility is to use *feature branches*, each of which contains a single version that is later merged with some "main" branch. The underlying structure of a version control system is a Directed Acyclic Graph (DAG).

In general, a version control system provides functionality to easily retrieve older versions of the files it tracks and functionality to compare (`diff`) versions. Another feature that is often provided is annotation (popularly known as `blame`), which shows the origin (i.e., the last modification) of each line in a file.

We can distinguish between *centralized* and *distributed* version control systems. A centralized VCS consists of a server that keeps track of the history of

[1] Git is an exception to this rule; it uses SHA-1 hashes.

revisions. A distributed VCS is locally stored by each user. This means the entire history, including all previous changes, can be accessed without communicating with an external server.

Examples of centralized version control systems include CVS and Subversion. Examples of distributed version control systems include Mercurial and Git.

In the context of repository mining, a distributed VCS is different from a centralized one. Distributed systems are preferable since they can generally be accessed in their entirety. Local changes that could not be recovered when using a centralized system can be investigated. However, this extra data also introduces more chances to misinterpret the version history [118].

4.1.3 Bug tracking systems

Bug tracking systems are another important source for information for repository mining. A bug tracking system is an application that maintains a record of faults that have been encountered in a software project.

The following fields that are typically stored for each entry in a bug tracking system:

- **Submitter**: the person who reported the bug to the bug tracking system.
- **Assignee**: the developer who has been chosen to handle the ticket.
- **Class**: a field that is used to make the distinction between change requests and bugs. A change request is a generic request to change some aspect of the program; a bug is an implementation error that should be fixed. These are sometimes referred to as Problem Report (PR) and Change Request (CR) respectively.
- **Priority**: an indication of how quickly the ticket should be handled.
- **Severity**: the extent to which the defect can affect the software. This is not necessarily the same as priority; a bug may result in great damage (i.e., be very severe) but not given a high priority because it is extremely rare.
- **Status**: the status of the ticket. Possibly values may include "open", "closed" and "won't fix".
- **Date/time**: the moment the ticket was introduced. Usually, the moment of closing the ticket is also registered.

This information can be used to acquire real-time data on defects in the project history, the time it takes to handle a ticket and the effect of priority or severity on the way tickets are handled. All of these are potentially interesting for software engineering research.

Some bug tracking systems are integrated with one or more version control systems. This allows users to easily retrieve a commit that belongs to a bug ticket and vice versa. Alternatively, commit messages may be used to refer to a ticket identifier [119]. This concept is discussed in more detail in Section 4.2.1.

The data from a bug tracking system may be of low quality. One issue could be that not all bugs are reported in the system, which could cause a potential bias [120]. Another is that the commit through which a bug is fixed

is not always documented [121]. Finally, bugs may be assigned the wrong class label [122] (e.g., a ticket may be labeled as a bug while it is actually a change request). Some authors claim these imperfections are relatively harmless [121], while others point out they have likely led to wrong conclusions in previous research [112, 120].

4.2 Bug prediction

Bug prediction is an example of a field in which repository mining is applied. Bug prediction is the process of estimating the likelihood of software defects in software. The underlying assumption is that the occurrence of bugs (both their time of introduction and their location in the source code) resembles historical occurrences of bugs that have been documented in the bug tracking system.

Bug prediction has been applied successfully to predict the time of occurrence [123, 124] and location [37, 125] of bugs.

In the remainder of this section, we describe how the data from various software repositories can be combined in order to estimate the time and location of bug introductions and fixes.

4.2.1 Linkage

Linkage refers to the process of linking the tickets from the bug tracking system to the information in the version control system. This allows us to determine when a bug was fixed and which files were changed to accomplish this fix.

The linkage process is illustrated in Figure 4.1. We establish a link between the bug tracking system and version control system by examining the commit message for a commit in the version control system. If this message contains a reference to a bug ticket (such as "#394"), we can consider the commit to be *bug-fixing*. In order for this process to succeed, we need to rely only on the accuracy of the references from commit messages to bug tickets. Of course, a number can also refer to something other than a bug ticket. If commit messages follow a stricter structure, we can ensure a higher quality of linkage by searching for this structure instead of any integer.

When we establish linkage, this means that we have linked all documentation about the bug from the bug tracking system to the changes that resolved the bug.

While bug-fixing commits are interesting, the introduction of a bug is even more interesting. After all, it is not the fixes for bugs we want to predict, but the existence of non-reported bugs. Even for bugs that have been fixed, their moment and location of introduction is not well-documented.

However, a combination of information from the version control system and heuristics can help us to trace back through the project history to estimate the introduction of a bug. We accomplish this using the blame- or annotate-functionality of the version control system to find the commit in which the relevant line was changed last before the bug fix. We then consider this commit to be *bug-inducing*.

One well-known way of automatically identifying commits that fix a reported bug is the SZZ algorithm [126, 127]. It uses a mixture of keyword analysis and heuristics to link a bug report to a commit. A machine-learning alternative that

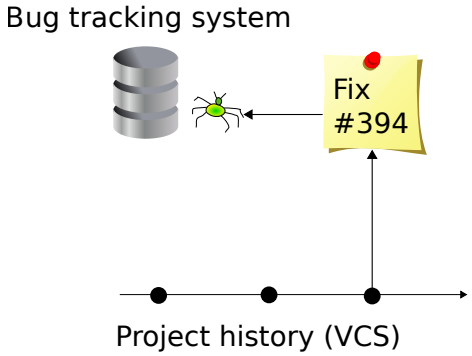


Figure 4.1: An illustration of linkage between tickets in a bug tracking system and commits in a version control system.

performs slightly better is ReLink [128]. We should emphasize that the reliability and effectiveness of all of these methods has been severely questioned [41, 78, 79, 112, 113, 129, 130]. This is partly due to the unreliability of the data. Another important factor is the challenges in validation of bug prediction models.

4.2.2 Data analysis

Once we obtain the relevant data, we can use statistical analysis to infer relations between quality measures and bug counts. There are different approaches to this.

Regression

A variety of papers [38, 131–133] have proposed regression models for bug prediction, specifically negative binomial regression. Using such a model, we try to predict the number of bugs using one or more metrics. This allows us to check whether the metrics significantly predict the number of bugs. This is a good indication that these metrics predict quality.

A regression model produces a coefficient for each independent variable. This is similar to the way the TQI is computed, suggesting that these coefficients could provide weights that more accurately predict the number of bugs. Care needs to be taken with this, since the coefficients do not necessarily add up to 100 and some forms of regression (like negative binomial regression) do not predict the number of bugs directly, but the log of the number of bugs instead.

A variation of the negative binomial regression that is applicable to bug count data is zero-inflated negative binomial regression. In most cases, the majority of observations (files, commits, software versions or some other unit to which a bug count can be assigned) will have 0 bugs. Zero-inflated models deal with this by considering the existence of a bug (zero or non-zero) and the number of bugs as separate variables [134].

When using regression, it is important to make sure the distribution of the data matches the theoretical distribution the regression expects.

The usage of regression for bug prediction has been criticized for its inability to discover relations of cause and effect [30]. Unless all variables that could

influence the dependent variable are present, regression models only show correlation. This somewhat limits the practical application of regression models.

Classification

Bug prediction can also be considered a classification problem [76, 78, 79, 125]. This problem can be phrased as: "does a component (file, class, sub-project) contain a bug or not?" In some cases, the approach is simply to list the files that are most likely to contain bugs.

These bug prediction methods are often able to identify the buggiest files in a project, but they are often simply a proxy for code churn. That is, listing the files that have been modified most often is as effective as more complicated classification models [76]. This limits their usefulness, since developers are often aware which files change often and are aware that they are defect-prone [78].

Classification in the context of bug prediction can be challenging because most commits do not cause bugs. This results in imbalanced data [79]. Techniques like resampling can improve the results [79].

Alternative technique for TQI validation

Both regression analyses and machine learning techniques are techniques to build a model that can predict the occurrence of bugs. Both techniques attempt to find the original location of bugs through heuristics. The SZZ algorithm [126] is a well-known example of this. The algorithm was later improved to remove some shortcomings that were present in the original version [127, 135], but since there is no ground truth it is hard to know whether acceptable levels of accuracy are achieved.

To avoid this threat to validity, we could simply investigate *bug-fixing* instead of *bug-inducing* commits. That is, we consider all links between a commit c and bug ticket b. We then find the parent c' of c. We can then consider the difference in TQI (or any other metric) between c' and c. Since the number of bugs is lower in c, we would expect c to have a higher quality than c'.

There are a couple of problems with this approach. First, we need to restrict the metrics we use or adapt their definitions. Some metrics, like duplicate code, have a different meaning in the context of a limited subset of files. Another issue is that the difference in quality between these revisions may be almost non-existent. In fact, some metrics may indicate quality has decreased after a bug has been fixed. For example, consider a bug that is fixed by adding a missing if-statement. This statement will cause an increase in cyclomatic complexity. Therefore, we may be tempted to conclude cyclomatic complexity is not a positive predictor of the number of bugs, because we only look at this isolated change. The absolute cyclomatic complexity of the function (before and after) may be much more worthwhile. Finally, the TQI is meant as a general indication of quality over an entire project. Even if we find that the metrics it is composed of are very effective at predicting the number of bugs, this arguably provides little evidence for the notion that the aggregated TQI is a good predictor of bug density.

4.3 Git internals

The version control system we deal with primarily for the context of this project is Git[2]. This is because this is the VCS we encountered in the case study. Git is an example of a distributed VCS. While the use of Git is similar to other VCSs for end users, its inner workings become relevant when investigating the history of a project. Note that while our general methodology works on all VCSs, we can only fully understand the context of development by taking a close look at the specific technology that is being used.

This section is intended to be a primer to the internals of Git. Many of the concepts in Git are shared by other version control systems, but some of the details are different. Since these details are relevant for the methodology of our case study (Chapter 6), we present them here.

Files At its core, Git is a simple key-value store. The values it stores are *files*; the keys are the SHA-1 hashes of the files. Note that Git tracks the *contents* of a file, not its location in the file system or file name. At this level, there is also no notion of a revision of a file; we simply provide file contents to Git and get a hash so we can retrieve the contents later using this hash.

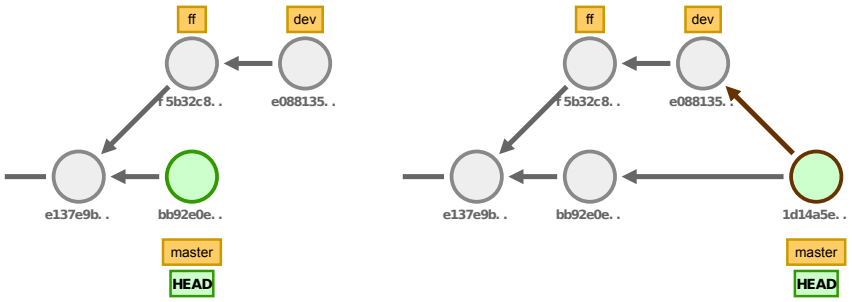
Trees A *tree* is essentially Git's version of directories or folders. This enables grouping files together in a hierarchical structure and provide names for files. Trees are implemented as files following a certain structure. This structure resembles a textual table specifying for each child of the tree (a) whether it is a file (blob) or tree; (b) its SHA-1 hash; and (c) its name. Trees allow us to store directory structures containing named files and sub-directories. Using the concepts we have seen so far, we can provide a directory structure to Git and later retrieve it using the provided hash. Recall that a tree is implemented as a file, so we can compute its SHA-1 hash just like we can with any other file.

Commits A *commit* object provides a way to track versions of trees. Like trees, commits are implemented through files. This file contains (a) the SHA-1 hash of parent[3] commit(s); (b) the name, email, and date of the author; and (c) the name, email, and date of the committer. The distinction between an author and committer is subtle. The author of a commit is intended to be the person who produced its contents. The committer is the person that added the changes to the VCS. In many cases, these are the same person. An example where this is not the case is if the author is someone who does not have access to the VCS. This author can then offer changes for review to someone who is more intimately involved with the project and can act as committer. A commit object again results in a SHA-1 hash that identifies it. All commits form a DAG. This DAG is not necessarily connected; if there are multiple root commits they can each form a separate connected component.

References Finally, Git uses *references* to provide names to specific commits in the DAG structure. A reference is essentially a name that points to a certain

[2] `http://git-scm.com`

[3] A root commit has no parents; a "normal" commit has one parent; a merge commit has two (or theoretically more) parents.



(a) before merge (b) after merge

Figure 4.2: Illustration of an explicit merge. The circles denote commits; the yellow rectangles denote branch references; the arrows denote parent-child relations. When we merge the `dev` branch into the `master` branch, a new merge commit is constructed and the `master` branch is updated to refer to this commit. Images generated using *Explain Git with D3* [136] and used with permission.

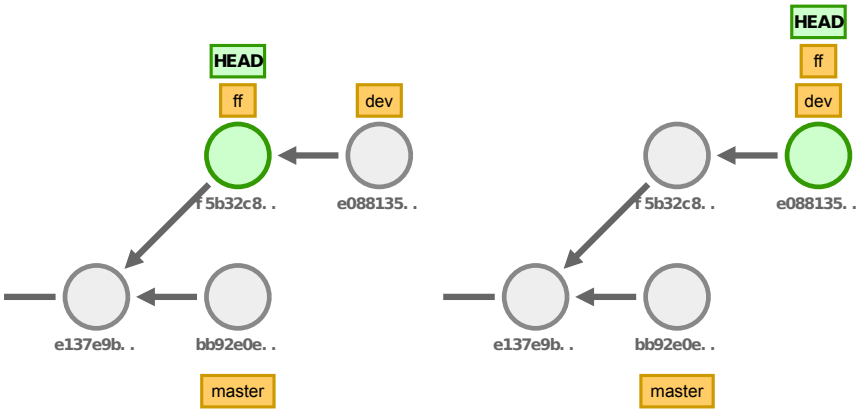
commit (through its hash). References include various kinds of *tags* as well as *branches*. A tag is used to provide a name for a specific revision. This allows a user to easily reference a certain version of the software. For instance, a stable version 1.0 may be given the tag `1.0` to allow developers to easily access the revision without having to remember a hash. A branch is similar, in that it simply points to the most recent commit in a branch. Whenever changes are committed to this branch, a new commit object is created and the branch is updated to refer to the new commit.

Using these concepts, we can consider what happens internally when executing some operations on the Git VCS.

Explicit merge An explicit merge is the standard way of merging two branches. Consider a branch `master` that refers to commit a and a branch `dev` that refers to commit b. Merging the changes of branch `dev` into branch `master` involves creating a merge commit c and updating `master` so that it refers to c. This process is illustrated in Figure 4.2.

Fast-forward merge Consider a branch `dev` that refers to commit a and a branch `ff` that refers to commit b. When both branch `dev` and `ff` change before the two are merged together, a merge commit is necessary to integrate the two changesets. However, there are situations where this merge commit is unnecessary. For example, `dev` may be a feature branch that is merged back before any changes are made in `ff`. In this case, we can omit the merge commit and simply update `ff` to make it refer to a. This is called a *fast-forward merge*. Fast-forward merges remove unnecessary clutter from version history, which can help developers. However, it also obscures the details of the history in terms of branches and merges. This process is illustrated in Figure 4.3.

Rebase A *rebase* is another way of integrating changes from one branch into another. Again, consider a branch `master` that refers to commit a and a branch `dev` that refers to commit b. Suppose branch `master` contains changes to some



(a) before merge (b) after merge

Figure 4.3: Illustration of a fast-forward merge. The circles denote commits; the yellow rectangles denote branch references; the arrows denote parent-child relations. When we merge the `dev` branch into the `ff` branch, we simply move the pointer of the `ff` branch. We can do this because `ff` contains no commits that `dev` does not contain. Images generated using *Explain Git with D3* [136] and used with permission.

files, with branch `dev` containing changes to other files. Since the intersection of these files is empty, the merge commit again contains no changes. However, we cannot simply update either branch to refer to the latest commit of the other branch, since this would result in lost changes.

Instead of merging, a developer can opt to rebase changes. This means that the changes from branch `dev` are applied one by one starting from commit a. This means that the commits in branch `dev` are replaced by new commits, which have a different parent, a modified tree and (because of this) different hashes.

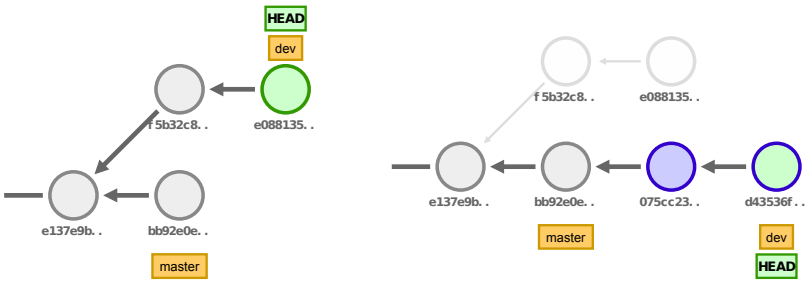
This process is sometimes referred to as *rewriting history*, since the original version history of branch `dev` is modified to make it seem like the changes were applied directly to branch `master`. While the term has a negative connotation (since a VCS is supposed to store history), this approach has the advantage of removing clutter from version history.

Rebasing can also be used to rewrite different aspects of history. For example, commits can be *squashed* together to appear like a single commit, or commit messages, authors and committers can be edited. All of these changes result in a new hash for the commit that is being edited. All subsequent commits are then also updated, since their reference to the parent commit must be updated. This process is illustrated in Figure 4.4.

Cherry-picking Cherry-picking is the process of moving one or more commits from one location in the DAG to another. It is a special case of rebasing. Cherry-picking may be useful in a situation where change that is made in one branch is useful in another, but merging the two branches together is impractical.

Advantages and disadvantages

The Git structure poses the following challenges for repository mining [118]:



(a) before rebase (b) after rebase

Figure 4.4: Illustration of a rebase. The circles denote commits; the yellow rectangles denote branch references; the arrows denote parent-child relations. When we rebase the `dev` branch into the `master` branch, we construct a new commit for each commit in the `dev` branch that contains the same changes, and apply it on the `master` branch. Note that the hashes of the rebased commits (blue circles) have changed! Images generated using *Explain Git with D3* [136] and used with permission.

- There is no explicit mainline. Most centralized VCSs have a notion of one branch that is considered to be default (e.g., `trunk` in Subversion). Most Git repositories contain a `master` with the same meaning, but there is no need for such a branch (or even any branch at all) to exist.
- There is no explicit indication of the branch a commit was committed on. Instead, we know which commit is currently at the top of a branch and we can find its parents. However, we cannot always ascertain the name of the branch a commit was originally committed to. (Note that a commit strictly speaking does not even need to be committed "to a branch"; it can simply be added anywhere in the DAG without updating any branch pointers.)
- Implicit merges, particularly fast-forward merges, make it difficult to find out to which branch a change was applied after merges have occurred.
- Before changes are pushed, they can be rewritten locally. This can be done to "simplify" the structure of the history. The downside of this is that the real, unmodified history is lost.

It does provide the following benefits:

- Distributed version control systems store the entire history locally. This means that we can obtain a complete history simply by cloning (i.e., checking out) the repository. This local copy can be accessed without any network latency.
- Since the history is stored locally, we can also collect data without intervening with the work of developers using the VCS. When working with a centralized VCS, executing expensive operations may overload the server and impact the availability of the service for other users.
- When attempting to trace back the origin of a change in a version control system, it may be hard to find an atomic commit. This is particularly

true for big projects that go multiple iterations of integrating merges into a mainline [137].

4.4 Differences between bugs

Until now, we have made no distinction between different bugs. This is an over-simplification; in reality, bugs are not all equivalent in terms of impact, cost to fix and overall importance.

A major distinction is the difference between pre- and post-release bugs. Pre-release bugs occur during the development of the software before it is used by end users. Post-release bugs are those that are discovered after the initial release of the software. Typically, these are discovered "in the field" by customers. Post-release bugs are arguably more important, since fixing them is much more expensive [11]. Camilo, Meneely, and Nagappan [45] found a weak correlation between pre- and post-release bugs, but these classes of bugs are not equivalent [30]. While some studies explicitly take both pre- and post-release bugs into account [138], many more primarily target post-release bugs [1, 28, 37, 40, 45, 48, 84, 139].

TIOBE claims software quality is determined by the number of defects found after release, the severity of these effects, and the effort required to solve the defects [16]. This means that we can represent quality better if we also take severity and fix time into account. Severity is a field in most bug tracking systems, meaning we can extract it. The time to fix a bug could be determined from the time difference between the bug report and the bug-fixing commit, but this only tells us how much time passed between reporting and fixing the bug, not how much time was spent on fixing it.

In some bug tracking systems, some reported bugs are actually feature requests, meaning they are not related to defects. This bias may cause a bug prediction model to predict changes instead of bugs [112].

4.5 Generalizability concerns

Since this approach of counting bugs is technology-specific, we need to decide which companies and/or projects deserve our focus. Ideally, we would like a large and diverse sample. This is because literature typically focuses on one [38, 140] or few (under five) [132] companies for practical reasons. This affects the generalizability of these studies; their conclusions may not be valid for other companies [141]. This is especially true since measurements may be impacted by company-specific policies [87]. The wealth of data TIOBE has access to can limit the problem of generalizability; even if we can only study a limited sample, careful selection can greatly improve generalizability [142].

Generalizability is a big challenge in the field of bug prediction. A large majority of bug prediction models does not work when applied on a different project [129]. We should therefore be very careful not to overestimate the relevance of results from a case study.

4.6 Previous TQI research

In order to validate the TQI, TomTom has examined the correlation between the number of bugs per file and the TQI of the file [133]. The metrics this study compares are the number of bugs (extracted from the version control system and bug tracking system), the TQI, LOC and all of the metrics the TQI is based on. Each of the metrics correlates significantly with the number of bugs, except for code coverage and fan-out. Note that this analysis does not take other metrics into account! This means that a metric like LOC may be a *confound*: an extraneous variable that correlates with some metrics (independent variables) and the number of bugs (dependent variable). When we adapt our statistical model to include it, the correlations may no longer be significant. Additionally, the strength of the correlations is weak (between 0.3 and 0.4), with the TQI itself even reaching a correlation under 0.2.

The research also consists of a negative binomial regression model with the number of bugs as dependent variable and the total and average cyclomatic complexity as independent variables. Both of these metrics turned out to be significant. It should be noted that the total cyclomatic complexity likely acted as a proxy for the size of the files. The average cyclomatic complexity does seem to be a predictor of the number of bugs.

Since the measurements are performed per file, effects of the aggregation did not influence this study.

As we discussed in Section 4.5, we should be careful not to generalize these results to other projects. To validate these findings, we can repeat the set-up on other projects.

4.7 Conclusions

In this chapter, we explored scientific literature that is relevant to our goal of validating the TQI. We have seen that while repository mining can be used to acquire large amounts of data, this data may be noisy and contain complicated structures. The internals of Git in particular are relevant. While we can cheaply access this data, the lack of explicit indications of the branch a changeset was committed to and the option to rewrite history are problematic.

When we combine the data from a version control system with bug tickets in a bug tracking system, we can find bug-fixing commits. This gives us access to the time and location of a bug fix. We can track down bug-inducing commits as well, but this is highly dependent on heuristics that may be invalid.

To analyze the resulting information, we can apply regression or classification. In addition, we propose an alternative technique in which we simply compare measurements before and after a bug fix. While this technique has its own challenges, it does not rely on heuristics to find bug-inducing commits.

We also need to be aware of biases in the data and generalizability concerns. Naturally, the set of bugs we consider in an analysis influences the results we obtain. We also need to be aware that drawing general conclusions from these results is problematic.

Finally, we discussed some previous research that specifically focused on the TQI. While this study yielded some preliminary results, more research is needed to assess the validity of the TQI.

Chapter 5

Survey

TIOBE collects data for a large set of companies. This sample is very attractive from a research perspective, since it allows us to apply bug prediction in a larger setting than usually is the case. At the same time, applying bug prediction on all projects is not feasible for this project. Instead, we should select a set of projects that we can extract the most valuable results from.

In order to do this, we propose a survey that reveals similarities and differences between projects and companies. In addition, this survey also intends to reveal more practical matters. For example, we will be using the bug tracking system for each project to obtain information about bugs in the system. For this reason, it is useful to know which bug tracking system(s) are being used. From a practical point of view, we may select projects that use the same bug tracking software. This allows us to extract all information through a common interface. We need to be careful to not introduce a bias; some bug tracking systems may be harder to use than others, resulting in bug under-reporting.

We start this chapter by considering which information we should ask for in Section 5.1. This leads to a set of questions, which we present in Section 5.2. Finally, we discuss the results of the survey in Section 5.3.

5.1 Relevant questions

For practical reasons, we should ask about the following:

- which bug tracking systems are being used;
- whether bugs are reported only within the development team, only from outside sources (i.e., customers complaining about problems in the field) or both;
- the reliability and completeness of information in the bug tracking system (e.g., are some bugs not reported, what happens to incorrect tickets, and how likely is wrong information in the bug tracking system);
- the extent to which the bug tracking system is linked to the version control system (e.g., a prescribed format for commit messages that includes a bug ticket reference, a requirement to include the reference at least somewhere, or hooks to prevent commits that do not adhere to this standard);

- policies related to bug reporting.

We do not ask about version control systems. Although these could be relevant, TIOBE already has access to this information.

Hall *et al.* [41] discuss context criteria that determine the practical applicability of research to a specific setting. These criteria are as follows:

- source of data;
- maturity of the system;
- size in KLOC;
- application domain;
- programming language.

TIOBE has this data for all projects, so we do not need to ask these questions in a survey.

Petersen and Wohlin [143] present a more exhaustive list of criteria. They do not just consider context of the *product*, but also the following other context facets:

- processes (activities, documentation);
- practices, tools, techniques (usage of CASE tools, usage of development techniques such as agile);
- people (roles, experience, number of developers);
- organization (hierarchical or other model, certification, geological distribution);
- market (number of customers, market segments, strategy, constraints).

Information on these criteria is harder to obtain through survey questions and also more sensitive. Recall that we save identifiable information for each survey participant. For this reason, process facets are disregarded for the initial survey.

5.2 Survey contents

The introductory text is located in Figure 5.1. The email that will be sent to participants is located in Figure 5.2.

The questions in the survey are as follows:

1. Which bug tracking system(s) do you use? (JIRA, IBM Rational, MantisBT, Serena TeamTrack, Bugzilla, Trac, other)
2. If you use multiple bug tracking systems, are these used by specific users (e.g., developers vs. users), sub-projects or phases of the project (e.g., internal development vs. deployed software)?
3. If you use multiple bug tracking systems, are these separate systems connected in any way? (open question)

Welcome to our survey on software projects.
In this study, we aim at getting an overview of projects that are using TIOBE TICS. We are looking for a limited set of projects that we can use in a case study on bug prediction. In this case study, we assess the ability of the TIOBE TQI to predict the number of bugs in a system.
Your participation is voluntary and confidential. We do not guarantee anonymity, since we would like to identify the projects that are most useful to our study and contact you. Results will be anonymized if they are published. If you are hesitant to share information or would like more details about how this information will be used, feel free to contact us. All questions can be skipped by leaving the field empty or choosing "no answer".
This survey is being carried out by Maikel Steneker <m.p.j.steneker@student.tue.nl> (Eindhoven University of Technology) in collaboration with TIOBE.
We thank you in advance for your participation in this study. If you have any questions or further remarks, feel free to contact us.

Figure 5.1: Introductory text to survey.

Dear FIRSTNAME,
My name is Maikel Steneker. I am a computer science student and I am currently working on my graduation project at TIOBE. The goal of this project is to apply bug prediction in an industrial setting.
In order to select projects that are most suitable for this for a case study, we are conducting a survey about the use of bug tracking systems for your software project. I would be very grateful if you could fill in this information. This will take approximately ten minutes of your time.
The survey can be found at <SURVEYURL>. Thank you in advance for your help! Feel free to contact me for any questions or remarks.
Kind regards,
Maikel Steneker <m.p.j.steneker@student.tue.nl>

Figure 5.2: Email that was sent to survey participants.

4. What percentage of bugs that are found would you estimate end up in the bug tracking system(s)? Note that we are looking for an estimate; we do not expect you to give a very precise indication of this. (open question)

5. Which of the following information is typically part of commit messages in the version control system of your project?

   - A short description of the change;
   - A longer, more detailed description of the change;
   - A reference to a ticket in the bug tracking system;
   - An explanation of why the change was made (e.g., to fix a bug or add a feature).

6. Is there a prescribed format for commit messages, which dictates where information is located within a commit message? For example: each commit

starts with a reference to a ticket in the bug tracking system, followed by a short textual description of at most 80 characters and a longer description on the next line.

7. If there is a prescribed format for commit messages: if developers violate this prescribed format, what kind of system (automatic or manual) is used (if any) to handle this violation? Please elaborate. (open question)
8. If there is a system in place to handle commits that violate this prescribed format, does this system check for any of the following mistakes?
   - referring to a ticket that is not yet assigned;
   - referring to a ticket that is assigned to another developer.
9. Developers may sometimes postpone committing their changes and then commit a set of unrelated changes at once. To what extent would you estimate this practice takes place within your project?
10. Developers may also commit their work early even if it is not finished, (e.g., before a lunch break or at the end of a working day). To what extent would you estimate this practice takes place within your project?

5.3 Results

We invited a total of 30 TIOBE customers to fill in the survey. Over a course of two weeks, we received 16 complete responses (a 53% response rate). The participants were the contact persons that each customer provides to TIOBE. The job description of these people varies between team lead, software architect and software developer. Generally, these are people with a thorough understanding of the project they work for. They are aware of the systems that are used during the development process. It should be noted that since these people may have a higher position than most other developers in their team, they may have an inaccurate view of the extent to which undesirable practices take place within their project. After all, developers may hide some of their actions from superiors if they expect a negative reaction to them.

The customers of TIOBE are a variety of companies that are mostly active within the embedded space. The size of these companies varies; some use TICS for a single project, while others have more than 300 different projects. Often, these projects are strongly related. For example, a single, large platform could consist of dozens of sub-projects. These sub-projects may be developed by a different group of people, but typically share a lot of characteristics: the same bug tracking and version control systems as well as the same development practices. To distinguish between projects that are closely related and independent projects, we used the expertise of the managing director. He also provided the contact information for each of the active projects.

Projects were considered active if there was at least one status report since January 1, 2015. A status report is an email that is automatically sent by TICS to report metric values. This is typically done daily or weekly. The choice of our cut-off date of January 1 is somewhat arbitrary. The period of time since this date is short enough to find most inactive projects, but long enough to allow projects that are only occasionally checked. This is primarily the case for very

large projects for which extracting metrics such as code coverage takes a lot of time.

We should stress that these results are based on a very small sample ($N = 16$) with a bias towards a few companies in which the response rate was higher. Therefore, these results are not generalizable and serve little other purpose than the selection of projects for our case study and potential follow-up studies.

5.3.1 Bug tracking systems

In our limited sample, IBM Rational was the most popular bug tracking system, with 6 out of 16 (38%) of projects using it. Another three projects used JIRA. The remaining projects used Team Foundation Server, ClearQuest (a precursor to IBM Rational), Mantis, Trac, or an in-house developed bug tracking system.

Two projects combine multiple bug tracking systems. In one case, one bug tracking system was used by developers, while another was intended for all other users. Bug tickets in the system used by developers were required to refer to tickets in the other to enable traceability. In the other case, developers used either one of the systems exclusively, or a combination of the two. From the answer given in the survey, there seem to be no rules that indicate when to use which system. This seems problematic not only for the context of this research, but also in general; a combination of two systems that have no dedicated task or intended user base may impede finding relevant information.

5.3.2 Proportion of tracked bugs

A bug tracking system only provides information about bugs that have been added to its database. The estimated proportion of the bugs that end up in the database gives an indication of the completeness of this information. The survey respondents reported a wide variety of estimations. Most projects seem to track almost all bugs, with answers in the range of 90% to 100%. However, there were also some projects that score noticeably lower, around 50% or 60%. In some instances, only 10% or 20% of bugs is estimated to end up in the bug tracking system.

It should be noted that not all respondents may have interpreted this question in the same way. An important distinction between bugs is whether they appear pre- or post-release. Pre-release bugs are arguably less important, since they are fixed during development and therefore do not hurt the quality of the final product. Post-release bugs are more expensive to fix and negatively impact end users. At the same time, disregarding pre-release bugs entirely is also problematic. Fixing bugs during development also takes time, meaning high amounts of pre-release bugs could be costly and even prevent the product from releasing at all.

The respondents that indicated 100% of their bugs were in the bug tracking system all noted the caveat that they only counted post-release bugs.

5.3.3 Contents of commit messages

We asked participants which components were typically part of a commit message. All participants indicated a commit message should include a description of the change that was made in the commit. Most participants indicated this

description should be short. A fifth (20%) of participants expect an explanation of why a change was made. Finally, the part of a commit message that we were most interested in is a reference to a bug tracking system. Most participants (80%) indicated that this reference is part of the message.

The primary goal of this question was to assess to what extent linkage between a version control system and bug tracking system is possible. This linkage is easiest to achieve if there is a prescribed format for commit messages. Half of the projects use a prescribed format for commit messages. All but one include a field for a reference to a bug ticket, but the use of this format is not enforced in most companies. That said, some companies do fill in this default template for each commit message or validate that all information is present in code review.

5.3.4 Piggybacking

Finally, we asked whether piggybacking was an issue within the software projects of the participants. Piggybacking refers to the process of adding unrelated changes to a commit instead of committing them separately. There are several motivations to do this. One motivation is the cost of a commit. A large amount of commits may clutter the log. Another aspect of this cost is that continuous integration platforms may run all tests for each commit. Doing this for small changes may be considered a waste. Finally, programmers may want to make changes that are not part of any ticket. For example, a bug tracking system may be focused on bugs and functional feature requests, while a developer wants to refactor some code for clarity. If a system is in place to prevent these commits from being integrated, developers may resort to tactics like piggybacking.

Another related phenomenon is committing changes early. Developers may commit their (unfinished) work at certain times (e.g., before lunch or at the end of the day) even though their work is not yet done. This can be seen as the reverse of piggybacking; instead of adding unrelated changes to a commit, relevant changes are not a part of the final commit.

Participants indicated that neither of these practices were common in their projects and happened only in very specific circumstances. Participants universally agreed that it was desirable to include all relevant changes in a commit.

5.4 Conclusions

In this chapter, we presented a survey aimed at TIOBE clients. The most important conclusion is that the amount of traceability offered by the information from bug tracking systems and version control systems varies widely. Some companies claim to track almost all bugs, while others expect to have information only on a small subset. Most participants do expect a reference to a bug ticket in commit messages.

Since the sample size is very small, these findings cannot be generalized to a larger group of companies. However, we can use the information we collected to assess the suitability of projects for a case study based on data from software repositories.

Chapter 6

Case study

To evaluate the TQI, we perform a case study on select TIOBE customers. These were selected based on the results of the survey and data from the TICS Enterprise Dashboard (TED) database [19].

We consider two different TIOBE customers for this case study, which we refer to as "Company A" and "Company B". These customers were selected for a case study based on their reports of high data quality in the survey. These companies then made data from their version control system, bug tracking system, and TICS installation available for this study. We give a general overview of this data in Section 6.1. In Section 6.2, we consider the shape of the distributions of the TQI and other metrics. We discuss the findings based on the data from Company A in Section 6.3. In Section 6.4, we do the same for Company B. We discuss the applicability of SZZ based on the data in Section 6.5. Section 6.6 contains a qualitative analysis for the data we extracted from software repositories, which reveals the practical difficulties we encountered when finding the files that were changed for each bug fix. Finally, we present an analysis of the extent to which the TQI responds to bug fixes in Section 6.7.

6.1 Data overview

We have access to the following data:

TED database This consists of reports that result from each TICS run. Reports provide a global overview of a project. This overview consists of metrics which have been aggregated over all files in the project. This means that the TED database cannot be used to inspect projects at a more granular level (e.g., at file level). This database is maintained centrally by TIOBE.

Bug tickets This is a table that consists of a list of tickets, including the class (i.e., bug or change request), severity, status (i.e., resolved, open, or some other status) and date/time (at least of initial introduction, possibly of closing) for each ticket. The history of a ticket (i.e., changes to it over time, such as how the ticket status changed over time) is not included.

TICS database This is a database that is located on a server at the customer. TIOBE does not necessarily have access to this database, but for the purpose of this project, we were provided a copy. The TICS database contains a history of all measurements that were performed by TIOBE on the file level. From this data, we can derive a value for any metric at any granularity level.

SZZ output This is the output that our implementation of SZZ produces. This consists of a list of commits, including their revision number/hash, date and time, and message. We store an optional reference to a ticket in the bug tracking system, which has been extracted from the commit message. We do not consider multiple references to tickets. We did not expect multiple references to be common based on the results of the survey, and indeed did not find any messages in which multiple tickets were referenced. Finally, there is a list of candidates that has been constructed using the traceback functionality of SZZ. These are commits that potentially caused the bug that the ticket describes. This entry contains a reference to all these candidates, as well as the file from which this was traced back.

6.2 Metric distributions

The data from the TED database provides an interesting insight into the various metrics TIOBE collects. In this section, we discuss the shape of the distributions for the metrics it contains.

In general, software metrics have skewed distributions [90, 92, 144]. One of the goals for the TQI is to approach a Gaussian (normal) distribution [16]. An example of a normal distribution is shown in Figure 6.1. Note that this distribution is symmetrical, with most observations appearing around the mean and values that are further away from the mean being rarer. A Gaussian distribution has some desirable properties when used for the TQI. Extreme scores are relatively rare, while average scores are more common. This is very much a subjective design decision. Additionally, the distribution should spread over the entire range of 0 to 100 in order to make it easier to distinguish low- and high-quality projects.

Figure 6.2 contains a plot for the TQI distribution over the more than a thousand projects that are using TICS. While it does not resemble a normal distribution, the resulting distribution is not especially skewed (skewness: 0.096; kurtosis: 2.137). One obvious observation is that the plot contains two peaks: one around 35, and another (albeit smaller) around 75. We hypothesize this is because some of the values the TQI is composed of are "cut off" when they are outside of the $[0, 100]$ range. When a sufficient number of measurements is negative or exceeds 100, this will result in a peak at 0 and 100 in the probability density plot. Since the TQI is simply a weighted mean of multiple of these values, we can imagine these peaks ending up in the probability density plot for the TQI as well.

An example of one of the metrics that is integrated into the TQI is duplicated code (Figure 6.3). This is an example of a metric that shows a radically different distribution compared to its raw counterpart. While Figure 6.3a shows a very skewed distribution (skewness: 3.620; kurtosis: 22.845), Figure 6.3b shows a

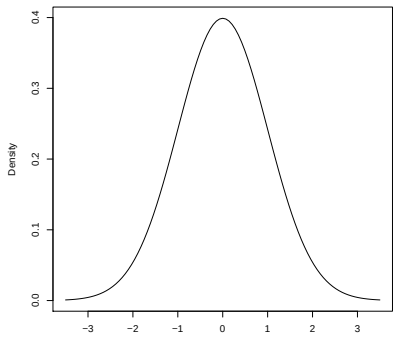


Figure 6.1: An example of a normal distribution ($\mu = 0, \sigma^2 = 1$).

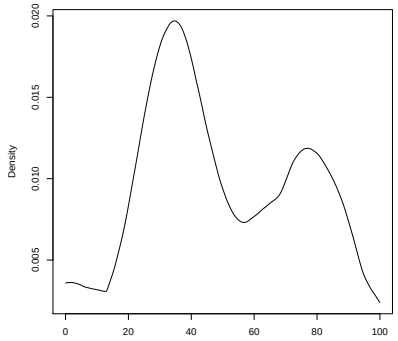
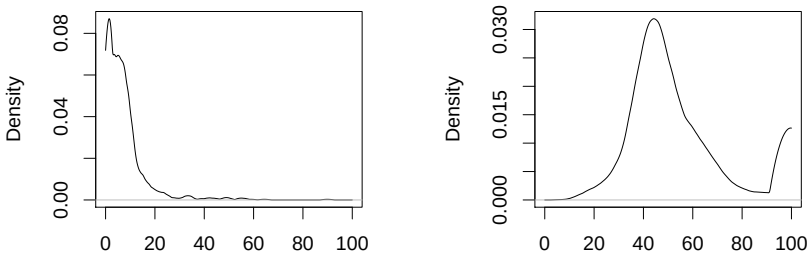
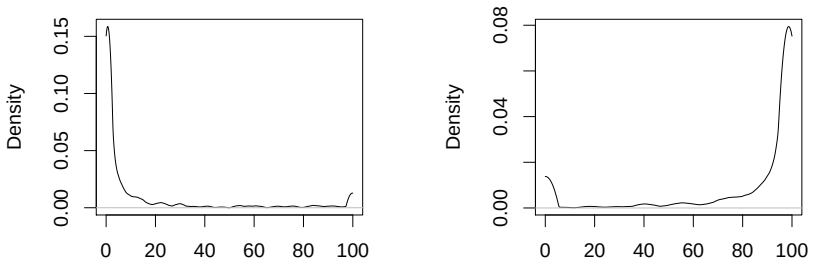


Figure 6.2: The distribution of the TQI values of all projects using TICS.



(a) raw (b) transformed for TQI integration

Figure 6.3: Probability density plots for duplicate code. The plot on the left represents the "raw" metric, which is a percentage. The plot on the right represents the TQI component that is included in the TQI.



(a) raw (b) transformed for TQI integration

Figure 6.4: Probability density plots for dead code. The plot on the left represents the "raw" metric, which is a percentage. The plot on the right represents the TQI component that is included in the TQI.

largely symmetrical distribution that is much more evenly spread out over the full range of admitted values (skewness: 0.954; kurtosis: 2.882).

The dead code metric (Figure 6.4) is very different from the duplicated code metric. Again, the distribution is very skewed (skewness: 2.497; kurtosis: 7.922). However, the results in another (reversed) skewed distribution (skewness: -1.949; kurtosis: 5.284), instead of a more evenly spread one.

From this analysis, we can conclude that the TQI follows a distribution which, unlike many other software metrics, is not skewed. However, it also does not closely resemble a Gaussian distribution. A possible reason for this is the variation in the distributions of the metrics it is based on.

Taking the formulas into account, this difference is hardly surprising. The duplicate code formula uses a logarithm, which results in its range of values being spread out over a much larger range. Conversely, the dead code formula simply scales the values linearly and cuts off values that exceed the maximum value of 100.

Plots for all metrics are included in Appendix B.

6.3 Company A

This section contains a discussion of the data specific to Company A. We discuss the measurements from the TICS database in Section 6.3.1. We discuss the data obtained from the version control system and bug tracking system in Section 6.3.2.

6.3.1 High-level description of TICS database

This section contains a high-level description of the results. The focus is on some interesting patterns that can be observed in the data.

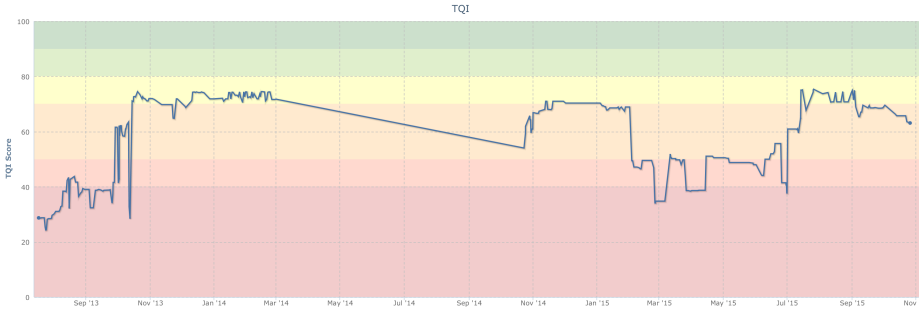


Figure 6.5: TQI level over time.

The project was measured in the period from July 2013 to February 2014. At this point, the license for TICS was temporarily discontinued, meaning data was no longer collected. After renewal of the license, measurements were performed from November 2014 to October 2015.

Figure 6.5 shows the TQI over this period of time. A few things stand out from this plot. First of all, there is a straight line close to the center of the figure caused by the absent measurements described earlier. Adjacent measurements are connected by a straight line. Since the TQI level in October was lower than the TQI level in March, a decreasing line is shown for the months in which no measurements occurred. Second, there is a lot of variation in the TQI. While some variation in quality is to be expected for software during its development, the peaks are not. For example, on October 11, the TQI is 63.56% in the morning, but only 32.82% in the evening. Days later, on October 14, the TQI has increased to 71.03%. This poses the question whether the data is reliable enough in its current state.

To examine what causes these effects, we look at the individual metrics the TQI is composed of. Since the TQI is a (weighted) average, a big variation in a single metric could very well be the cause of drops in TQI such as the ones we observe. One such metric is the abstract interpretation component of the TQI. Recall that this is a value that depends on the amount and types of violations found by an abstract interpretation tool and maps to a domain between 0 and 100. Figure 6.6 shows that this value indeed has an influence on the TQI and may partly cause the peaks we observe. It is very common for the abstract interpretation component to jump from a value close to 100 to a value of 0. Closer inspection of the underlying database reveals that in some cases, the abstract interpretation tool does not run for a variety of reasons (e.g., a missing license, a parsing error). Such a failed run typically results in a value of 0. Unfortunately, we have no guarantee on the value for failed runs of tools. This is because a tool could fail halfway through its run and result in a value that is based on a limited number of files within the project. Dealing with this incomplete data is a challenge both for this project and TICS in general.

6.3.2 SZZ data

The number of commits in the Subversion repository is 7876. We successfully linked 2333 (29.62%) of these to tickets. Extracting references to tickets was done using the prescribed format indicated in the survey. This format included

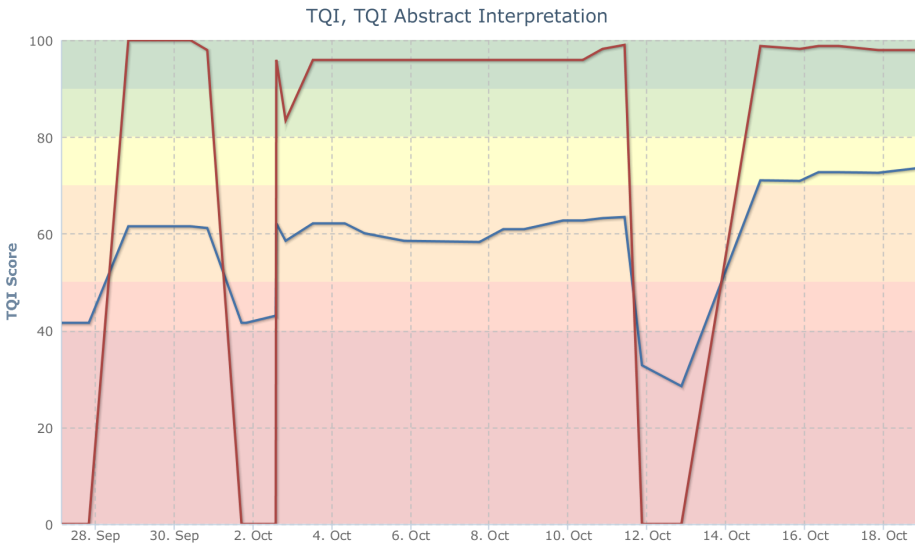


Figure 6.6: TQI level (blue) compared to abstract interpretation (red).

a specific position in which a reference is located. These were extracted using a regular expression. In practice, commit messages slightly deviated from this format in a significant number of cases. Had we opted to enforce this format less strictly, we should be able to link 3674 (46.65%) commits to their bug tickets. In this case, we would simply use the first number that occurs in the commit message. This means the chance of misinterpreting some numbers as ticket references increases, but we do obtain a better linkage. Note that in both cases, we do not need to rely on heuristics, as was done in the original paper [126].

An interesting observation is that many tickets are referred to by multiple commits. These are likely *partial fixes*, i.e., a work-in-progress solution for the ticket it refers to. One difficulty when dealing with partial fixes is the distinction between a partial and a full fix. If we find a single revision that refers to a ticket, it may be a full fix. However, it may also be a partial fix that has not been followed up by subsequent other partial fixes yet. Therefore, it is important to check whether a fix refers to a ticket that has been closed. Note that even if a ticket has been closed, it may be re-opened later. We assume this is something that does not happen often. Instead, we would expect a ticket to be set to an intermediate state (e.g., implemented) before being verified and being closed or re-opened. The number of commits per ticket varies widely, ranging from 1 to 79 with a median of 2 ($M = 5.52, SD = 9.46$). Some of the tickets with a large number of commits are large refactorings that are divided into more atomic changesets. However, for some of the other outliers, we could not derive from the data why this particular ticket took so much effort to implement.

Using SZZ, we can only trace back changes that included a deleted or modified line. Therefore, changes that do not modify any files or only add new lines to files will be excluded from the analysis. This is important, since we are introducing a potential bias by excluding these changes. We also exclude

commits for which the size of the diff[1] exceeds a certain threshold, which was in this case set to 10000 characters. The reason for the existence for such a threshold is two-fold. First, very large patches tend to modify a lot of lines. Each of these line potentially originated from a different revision. Since we aim to find a single origin, any result we obtain is meaningless. There is also a more practical reason to exclude large commits. Since the running time of the annotate command for the version control system is dependent on number and size of the files, processing large commits typically takes a lot of time. It is therefore infeasible to process the largest commits.

We encountered a total of 964 commits that were too large to process, as well as 446 commits that did not include any modifications that can be used to trace back. This leaves only 923 commits (39.56% of the linked commits) that can be processed.

6.4 Company B

This section contains a discussion of the data specific to Company B. We discuss the measurements from the TICS database in Section 6.4.1. We discuss the data obtained from the version control system and bug tracking system in Section 6.4.2.

6.4.1 High-level description of TICS database

In this section, we describe interesting patterns we can observe in the data.

This company has divided their code into 10 separate projects. These projects are all evaluated separately. However, we can also view an aggregation of the measurements over all projects. This aggregation is shown in Figure 6.7. It should be noted that one of the TQI components, namely abstract interpretation, has been disabled for all of these projects. The reason for this is that the tool for abstract interpretation requires a specific license which has not been purchased by Company B. This not only means that we cannot evaluate abstract interpretation, but also that the aggregated TQI score for this project is lower simply because 20% of the TQI gets the minimal score of 0 by default. Excluding the abstract interpretation component could potentially result in a TQI score above 80, resulting in a B label (good) instead of a D label (moderate).

Again, we see the TQI is typically stable, but at times peaks or suddenly jumps to another height. Figure 6.8 shows how one such peaks is caused by the fan-out component. Over the course of a single day, the average fan-out decreased from 31.88 to 12.26. This decrease in fan-out improves the TQI by 3.21%.

Looking at this issue in more detail reveals a new version of TICS was installed two weeks prior. This new version introduced TICSCil, a tool that was specifically developed to improve the computation of fan-out for C# projects. Originally, fan-out was approximated by counting the number of `using` statements and assuming (as a heuristic) that each of these namespaces provides

[1] We use the noun "diff" as a term meaning *the output of a* `diff` *command*. Variants of this command are embedded in most version control systems, such as Git and Subversion.

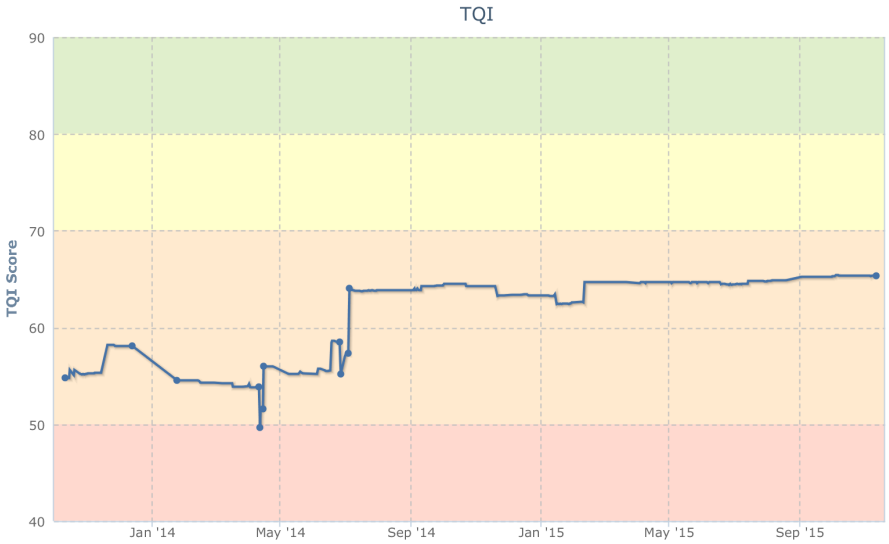


Figure 6.7: TQI level over time (aggregation over all company B projects).

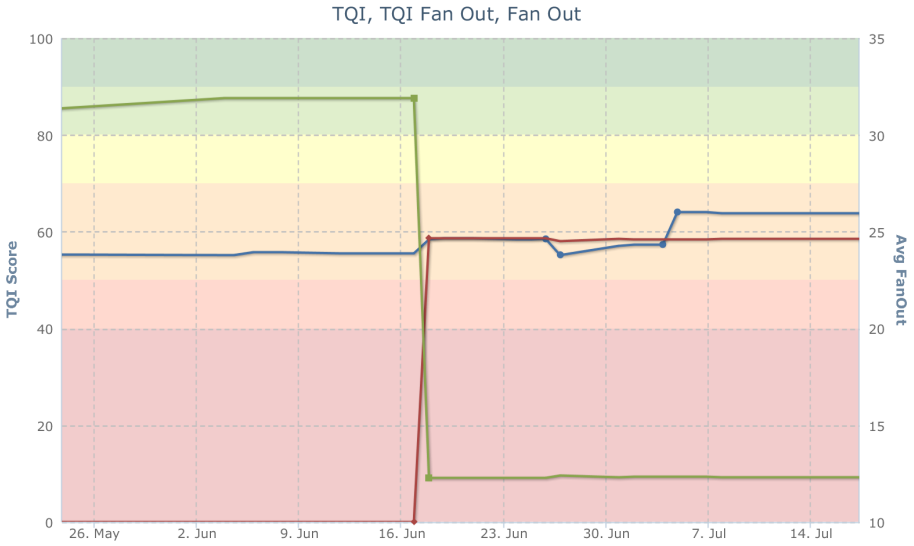


Figure 6.8: TQI level (blue) compared to the TQI fan-out component (red) and average fan-out (green).

5 classes[2]. The new version instead counts the number of classes that are used from the imported namespaces.

Even more surprising than sudden changes in values are the peaks, in which the TQI value first drops, and then increases to a different value from the one before the peak. This is most clearly seen around the beginning of May 2014 and July 2014. We found these peaks corresponded to peaks in the TQI component for compiler warnings. This value is normalized for the size of the project (i.e., LOC). We found that these peaks as well as other sudden jumps correspond to changes of 5000 to 20000 lines of code. This suggests that, even if it is not entirely clear yet what caused these changes exactly, they were a result of major changes in the codebase as opposed to imperfections of TICS or the underlying tools. Further investigation is needed to confirm whether these changes represent a meaningful change in quality.

Changes like these are particularly hard to deal with, since it is not easy to detect which version of TICS is responsible for a metric value. A straightforward look at the version of TICS at the time of the run is not sufficient, since unmodified files may still report values measured by an earlier version.

6.4.2 SZZ data

The data to facilitate analysis through the SZZ algorithm was obtained in a slightly different way than the data for Company A. There are multiple reasons for this. There are multiple subtle but relevant changes that were made for a variety of reasons.

The first change is the way tickets were linked. For Company A, we chose to only consider a commit linked if it adhered exactly to the specified format for commit messages. That is, if the format was of the shape "Ticket #999: fixed bug", a commit message omitting the number sign (#) will be considered to be unlinked. This choice was made to ensure a high precision at the cost of recall. This helps ensure a high quality of the data and saves valuable computation time that should only be spent on commits that are, in fact, intended to be linked to a bug report. In hindsight, we found that violations of the standard format were more common than anticipated, while numbers that did not refer to tickets are sufficiently rare. Therefore, we opted for a relaxed approach in which we consider any number to be a reference to a ticket.

Other changes are related to the speed at which we could process the data of Company B. Since we were able to work with a copy of a repository on a USB drive as opposed to a remote server, the data collection script could process each of the commits much faster. This allowed us to extract SZZ data for all commits (as opposed to just linked commits) and increase the maximum value of a diff to 100000; a ten-fold increase. Still, the data extraction was significantly faster (around 40 minutes instead of around 15 hours for a comparable number of commits).

Due to these changes, we ended up with data that is much more complete. We collected data from two separate repositories, both containing a different set of sub-projects. The first of these repositories consists of 7017 commits, from which 2192 were excluded for a missing or wrong linkage. Note that most of

[2] Note that this approach is still used for Java, where the use of wildcard imports is less common. The superior approach of TICSCil may be adopted for Java in the future.

these are relatively old commits from a time where referring to a ticket was not the norm. If we consider the commits for the past year, we find 1192 out of 1216 commits (98.02%) are linked. The second repository consists of 1647 commits, with 1621 (98.42%) of them being linked. This repository is more recent, which explains the consistently high quality of linkage.

As we saw in the dataset for Company A, most bugs are not fixed in a single commit. A clear difference is that changes are typically made in so-called *feature branches*. A feature branch is created to isolated the change for a single feature that is added or changed in the system. Note that a feature is not necessarily new functionality in this context. Instead, it can be any change that is made to the product in response to a ticket. This suggests that we can consider the commits in such a branch to form a transaction together which changes the code in the desired way.

As mentioned earlier, some commits were excluded because their changes were too large to process in a reasonable amount of time. For Company B, this was the case for 108 of the commits in the first dataset and 32 commits in the second database. Note that this low number is partly caused by the higher threshold we set during data collection. We attempted to qualitatively analyze some of the commits that exceeded even this higher threshold by manually reading their commit messages and the information in the associated ticket. It turns out that almost all of these commits can be identified as large refactors. For example, many of these commits simply moved files within the projects. While this change is not very meaningful, especially in the context of a bug fix, it causes the diff output to grow very large. This is because each file that is part of the directory structure is listed separately in this file, and there may be many references in other files that need to be changed to import from the correct location. Other refactorings added copyright notices to all source files in the project or changed the line endings of all files to use the same standard. Some other big changes included the addition of a WDSL (Web Services Description Language) file and the addition of a suite of unit tests, both of which can be large in size and are therefore not comparable to most changes.

6.5 Applicability of SZZ

SZZ, the algorithm that can be used to identify bug-inducing changes by tracing back through version history from bug-fixing changes, has some known limitations. Before analyzing the data in-depth, we can consider how important these changes are. We will use the larger dataset from Company B for this, since it is sufficiently large and complete.

Recall that this dataset contains 7017 commits in total. A small subset of 382 commits is linked to a bug ticket (i.e., a ticket that has been classified in the bug tracking system as "bug" as opposed to "change" or other category). These relate to 81 bug tickets. We found that 337 out of 1004 of the changes to a single file within these commits were either additions to a file or addition or removal of a complete file. This means that SZZ disregards more than a third of changes in the bug-fixing commits in this dataset. This affects 96 (25.13%) of the bug-fixing commits, with 85 (22.25%) consisting entirely of changes that SZZ cannot analyze. While this does not discount the results of SZZ entirely, we introduce a potential bias on top of the imperfect heuristics that SZZ uses.

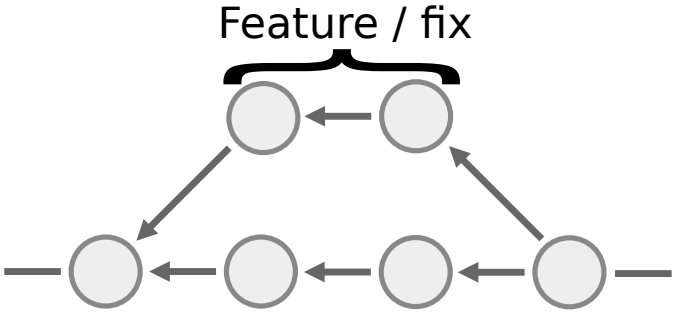


Figure 6.9: An example of a feature branch. The two commits at the top are contained in a separate branch, which contains all relevant changes for a feature or bug fix.

6.6 Using linkage to find changes for bug fixes

As we discussed in Chapter 4, we use data from software repositories to find out which changes were made for each bug fix. We do this by finding the identifiers for bug tickets in the bug tracking system, linking them to commits in the version control system, and inspecting which files were changed by these commits. In this section, we discuss some observations we made during the execution of this process on the dataset from Company B.

6.6.1 Theoretical ideal case

In the most simple case, each bug ticket corresponds to a single commit, which contains all relevant changes for the fix. In practice, there are often multiple commits. In the particular project we are studying, the developers have adopted a *feature branch* model. This means that a new branch is created for each bug ticket. In a brief conversation, a developer working on the project indicated that he was not aware of any cases in which developers would deviate from this workflow. This means that we can simply isolate the changes for a ticket by finding where the branch was created and where it was merged back. All commits in between these two must then be part of the bug fix. This is illustrated in Figure 6.9.

6.6.2 Practical

Unfortunately, we discovered this workflow is often not perfectly matched. We discuss some of the reasons for this here.

Fast-forward merges As we discussed in Chapter 4, Git automatically performs fast-forward merges to simplify project history. We discovered this happened in multiple instances. Fortunately, we could rely on the strong linkage for this project, meaning individual commits each contained a reference to a ticket. Without this, it would be extremely challenging to decide which changes belong to each ticket.

The project would be easier to analyze automatically if all changes were explicitly merged. Developers can achieve this by disabling fast-forward changes.

At the same time, this makes the history harder to interpret for developers themselves.

Rebases Despite the suggestions by the development team, we did find some evidence of rebases occurring in the project. These rebases were rare and did not affect the data we used in our analysis.

We found two different reasons for rebases. The first was cherry-picking. In the rare instances where this occurred, we found a clear indication of this in the commit message. This makes it possible to manually include or exclude these changes. It does impact the scalability of an analysis like this; when investigating a larger project, it will likely not be possible to deal with a large number of rebases. The second was that some commit messages were edited before they were merged back to the main branch. The reason for this was that they did not contain a reference to any ticket. By rebasing the commits, the developers were able to keep the history of the project as fully documented as possible.

Commits to main branch In some rare cases, changes were committed directly to the main branch. These changes usually did not correspond to any bug ticket, and were tagged to relate to TICS. From the commit messages, we can deduce developers made these changes as a response to results within TICS. These changes included fixes of warnings, changes to adhere closer to the coding standard, and refactorings to reduce cyclomatic complexity or fan-out.

We can view some of these changes as treating metrics as goals, which is undesirable [145]. However, in general, these changes seem reasonable improvements of quality, meaning the TICS system is used exactly as intended. The only improvement to this practice would be to create tickets for these changes to improve traceability.

6.7 Responsiveness of metrics

Due to the limitations of SZZ, particularly its reliance on heuristics, we instead examine whether the TQI metrics react to a bug fix.

6.7.1 Expectations

The goal of the TQI is to give an indication of software quality. One attribute of software quality is reliability, which relates to the number of bugs in the software. Therefore, when a bug is fixed, the quality of the software should increase. This means the TQI should increase.

Each of the TQI components has a positive or negative influence on the TQI. Therefore, for the TQI to increase, at least some of these components should change in the direction that indicates higher quality. These expected directions are included in Table 6.1.

Note that these hypotheses are purely based on the way the TQI is defined. We would not expect these to universally apply to all commits. For example, it is very reasonable to expect that some bugs are caused by a missing check, which can be fixed by an `if`-statement. This would *increase* cyclomatic complexity against our expectations.

Table 6.1: Expected direction for TQI components.

Metric	Expected direction	Rationale
TQI	+	The TQI is meant to be an indicator of software quality. The reliability of software increases when bugs are fixed.
Code Coverage	+	A higher code coverage means a larger portion of the code is tested.
Abstract Interpretation*	+	A higher AI compliance means there are less potential issues.
Cyclomatic Complexity	−	Lower complexity means the software is easier to understand and maintain.
Compiler Warnings*	+	A higher compiler warnings compliance means there are less compiler warnings.
Coding Standards*	+	A higher coding standards compliance means the code adheres better to coding standards.
Code Duplication	−	A lower amount of code duplication means the software is smaller and changes need to be made only once.
Fan-out	−	A lower fan-out means there are less external dependencies.
Dead Code	−	A lower percentage of dead code means there is less unnecessary code in the project.

* These metrics all use the TIOBE compliance factor, meaning they are mapped to the interval $[0, 100]$. For all of these metrics, a higher compliance factor indicates a higher quality, which corresponds to a lower number of violations.

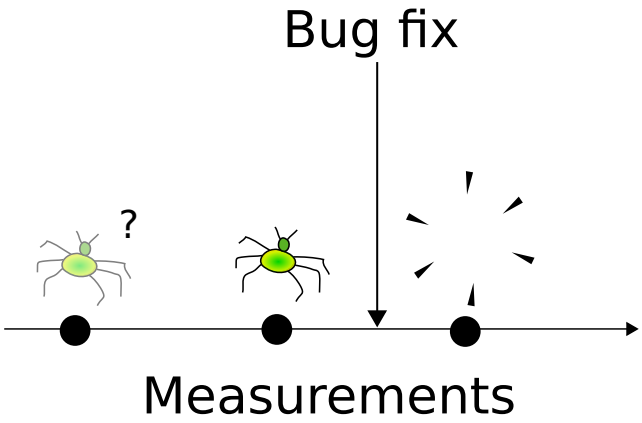


Figure 6.10: Illustration of time line of measurements with some bugs. The measurement before the bug fix is performed on a version of the software that definitely contains the bug. Assuming the fix was successful, the measurement after the bug fix was performed on a version of the software that definitely does not contain the bug. Any earlier measurements may or may not contain the bug, so we need heuristics if we want to consider them.

Another caveat is that bug fixes may, in reality, slightly decrease the quality of the software due to technical debt [6, 146]. A developer may decide to fix a bug in a somewhat crude manner to get the product to a correctly working state as quickly as possible. This temporary decrease in quality could later be resolved, but this would not be visible in this analysis.

6.7.2 Methodology

For this analysis, we consider two measurement timestamps for each bug: one right before the bug fix, and one right afterwards (see Figure 6.10). At these times, we obtain the value for all TQI metrics. Our goal is to find out whether the metrics significantly changed from the time before the bug was changed to afterwards. These metrics are aggregated over the whole project; we do not consider the values for a single file.

Since these measurements are paired (i.e., each measurement before the bug fix forms a pair with the measurement after the bug fix), the most obvious test is a paired t-test. This test assumes that the distributions we compare should both be normally distributed. Unfortunately, software metrics are often skewed [5, 144], which means this assumption is violated. A Shapiro-Wilk test for normality[3] [147] confirms this is the case for all of our metrics both before and after the bug fix.

For this reason, we use a non-parametric alternative for the t-test, which does not have the normality assumption. We use a Wilcoxon signed-rank test[4] [148]

[3] The null hypothesis for a Shapiro-Wilk test is that the data is normally distributed. We use a standard significance threshold of $\alpha = 0.05$. Since the p-value is below this threshold, we reject the null hypothesis and conclude the distribution is non-normal.

[4] The null hypothesis for the Wilcoxon signed-rank test is that both variables come from the same distribution. In our case, this means that there is no significant difference between the measurements before and after the bug fix.

on each of the metrics separately.

Since we know an expected direction for each test, we could use a one-tailed test instead of a two-tailed test. A one-tailed test is more powerful, meaning it is more likely to find an effect if it exists. This comes at the cost of only testing for one direction of the effect. Any effect in the opposite direction of what we expect would go undetected. Since our expectations are based on imperfect simplifications, we instead opt for the more conservative two-tailed test.

6.7.3 Results

The results of these tests are shown in Table 6.2.

First, these results do not include all metrics as they were specified earlier. Abstract interpretation is missing since this specific company does not have a license for the required tool. We performed two more tests for fan-out. The reason for this is that the way fan-out is estimated was changed at the introduction of TICSCil. For this reason, we also consider the fan-out measurements that occurred before and after this introduction in separate tests. It turns out that neither of these tests have a significant result.

The first column of interest is the significance (p-value) of each of the tests. This denotes the probability we can observe data like this under the assumption that the null hypothesis (measurements before and after being the same) is correct. If this probability is sufficiently low, we can reject the null hypothesis and conclude that there is a significant difference between the measurements before and after the bug fix. A significance level of $\alpha = 0.05$ is commonly used [149; 150, pp. 889–891].

It should be noted that the significance levels cannot be meaningfully ordered. That is, we cannot say the metric for coding standard violations per KLOC with $p = 0.137$ is "more significant" than the metric for duplicated code with $p = 0.138$. Instead, we compare both to α and conclude that neither is significant at $\alpha = 0.05$. We can, however, use multiple α levels (for instance: $0.05, 0.01, 0.001$) to distinguish highly significant and less significant metrics.

Another note is that due to the number of tests we perform (16; one for each metric and two extra for the fan-out metric), the Type I error rises above the specified α level. If the probability of not making a Type I error in a single test is $1 - \alpha$, the error increases to $(1 - \alpha)^n$ when performing n tests. We can adjust for this in various ways. One is the so-called Bonferroni correction, which simply reduces the α level to $\frac{\alpha}{n}$. A more sophisticated method has been presented by Benjamini and Hochberg [151]. We did not apply either correction, but consider this potential problem when interpreting the results.

Since significance only reveals the existence of an effect and not its magnitude, we also look at the effect size using Cliff's delta. This effect size is negligible for all metrics. This means that the differences between measurements before and after a bug fix are so small that the difference has limited practical applicability.

6.7.4 Discussion

We can now compare the findings from Table 6.2 to the expectations as formulated in Table 6.1.

Table 6.2: Results of the responsiveness tests.

Metric	p	**Direction**	**Sample size**
TQI	0.671	−	63
Average Cyclomatic Complexity	0.006	−	63
Coding Standard Compliance	0.001	+	63
Coding Standard Violations	$< .001$	+	63
Coding Standard Violations per KLOC	0.137	+	63
Compiler Warnings per KLOC	$< .001$	−	63
Duplicated Code	0.138	−	41
Fan-out	0.406	−	49
Lines of Code	$< .001$	+	63
Unit Test Coverage: Branches	0.017	−	42
Unit Test Coverage: Functions	0.126	−	42
Fan-out (before TICSCil)	0.222	+	30
Fan-out (after TICSCil)	0.195	−	18
Compiler Warnings Compliance	$< .001$	+	25
Dead Code	$< .001$	−	62
Unit Test Coverage: Statements	0.66	−	37

As we would expect, the average cyclomatic complexity and compiler warnings per KLOC decrease. Similarly, coding standard compliance increases. However, we also find unexpected results in the opposite direction: branch coverage decreases and coding standard violations increase. The significance of the decrease in branch coverage may be a result of multiple hypothesis testing; this test is no longer significant when using a stricter α value of 0.01. This is not the case for coding standard violations. The tests suggest that coding standard compliance and coding standard violations, which should move in opposite directions, move in the same direction. This remarkable result requires more attention to find out what causes this effect.

We find that the number of lines of code increases. The p-value for this test reveals it to be highly significant. In fact, it is possible that some of the other metrics are influenced by this increase as well. We cannot detect this difference using the current experiment set-up; using techniques such as regression we could find relationships among the metrics.

Finally, the difference in TQI is highly insignificant ($p = 0.671$). This means that the TQI does not react to a bug fix. This is most likely caused by the aggregation over multiple metrics using the mean. Any small effects are easily compensated by other changes in other metric values. This results in an effect that is almost certainly insignificant, unless all metrics represent the same abstract concept. Software quality clearly consists of multiple attributes, which explains the results we found.

6.7.5 Threats to validity

The validity conclusions we draw may be jeopardized by a few limitations. Perry, Porter, and Votta [152] make the distinction between construct validity, internal validity, and external validity.

Construct validity

Construct validity is the extent to which the variables we use for the test accurately represent the abstract hypothesis we want to test. Construct validity is impacted in a variety of ways.

First and foremost, validity is impacted by the data we use. Not only may there be implementation errors in TICS, which is responsible for collecting and storing the measurements; the scripts that were used to extract data via the API may be imperfect due to misunderstandings about how it should be used. We believe this risk is relatively minor, since the API is relatively simple in terms of use, we had direct contact with its developers and we found no inconsistencies between the data as presented in TICS and the data that was extracted from the API.

Our statistical methods form some other threats to validity. First of all, the meaning of the tests is not entirely intuitive. Each test tests for a significant difference between metric values before and after each bug fix respectively. The presence of such a difference does not mean that the metric has a predictive value for the number or presence of bugs.

Another limitation is that the test examines the absolute difference caused by a bug fix, but it does not compare this to the difference caused by any other change of the software. When repeating the tests for non-bug commits, we obtain a result that is very similar: the direction of the changes remains the same, with the p-value of most tests decreasing. The decrease in p-value is likely caused by the increased population size. This suggests that the changes in the metrics may be a result not of a bug fix, but of any change to the software in general. This is a problem that some bug prediction models have also suffered from [112].

Finally, we violate some of the assumptions of the Wilcoxon test. One assumption is statistical independence of observations. This assumption universally applies to almost all tests, but is often violated when dealing with real data [153]. The Wilcoxon test in particular is conservative when limited dependency between samples is introduced [154], meaning the power of the test is weakened and we are less likely to find a result if it is there. Another assumption is the absence of ties in the pairs of observations. This is not an assumption per se, but in real-world data, ties are very uncommon. This is because as long as the measurements are sufficiently accurate, we would not expect any two measurements to be exactly the same if only because of noise in the measurements. In computer science, and our measurements in particular, ties are no impossibility, since the tools are deterministic and may result in the exact same value even after a change.

Internal validity

Internal validity is the extent to which changes in dependent variables can be attributed to independent variables. This notion is particularly relevant when using a more complicated model, such as regression. In that case, the absence of control variables is an important threat to internal validity. In this case, the tests do not take control variables into account at all. This means that we cannot tell whether, for instance, the increase in compiler warnings can be directly attributed to an increase of the size of the project.

External validity

External validity is the extent to which the study's results generalize to other settings outside of this particular case study. We emphasize that this case study is just a single example of a software project within a single company. This means that the results cannot be applied to other projects. In order to do this, we would need to repeat the study to other projects.

6.8 Conclusions

In this chapter, we have described the majority of the empirical research that was conducted for this project.

First, we examined data from the TED database TIOBE maintains to examine the shapes of the distributions of the various metrics. We found that these distributions vary widely. Since the TQI composes these metrics using a weighted mean, this diversity makes it hard to interpret the distribution of the TQI itself. This suggests that this an area where the TQI can be improved.

We collected data from two companies, which we refer to as Company A and Company B. We worked with these companies to obtain a historical record of measurements on their projects that are stored in the TICS database. Additionally, we applied the technique of linkage, as proposed by the SZZ algorithm [126], to obtain information on the commits that contain the changes to fix bugs in older versions of the software.

Extracting relevant information from the software repositories proved to be very challenging. Even with relatively small projects that we selected for their suitability, it was difficult to correctly attribute changes to a bug fix. This means that validating the TQI through the investigation of real industry projects is a bigger undertaking than one might expect; one that is problematic even in the most advantageous of circumstances.

The historical measurements for the projects we examined revealed some interesting patterns. We saw that in a real database, the TQI can drastically change in a short amount of time. This raises some concerns about the reliability of the data that is collected, but it also reveals an area in which TICS, the software that is used to present the data to the end user, can be improved: the addition of a type of *root cause analysis* that can be used to investigate which files, metrics, and/or projects are the cause of a change in a higher granularity level. Another pattern that was clearly observable was the introduction of TICSCil, which resulted in a big drop in fan-out in the project of Company B.

Finally, we investigated how each of the metrics respond to a bug fix. We found that while some metrics showed a statistically significant increase or decrease after a bug fix had occurred, the changes were negligible in effect size. The most likely reason for this is that the aggregation of measurements causes differences in a small subset of files to result in very minor TQI fluctuations. When the metrics are composed to form the TQI, we do not find a significant difference in the change in TQI score. Similarly, this is likely caused by composition. Unfortunately, we cannot draw any other conclusions about the effectiveness of the TQI and the metrics it is based on. We believe this is a topic for future research.

Chapter 7

Conclusions

In this chapter, we reflect on the research questions that we presented at the start of this thesis and use the findings we presented in previous chapters to answer them. We also propose some possibilities for future work.

The main question we posed is: *Is there a relation between the metrics TIOBE collects and bugs?*

In addition, we presented the following research questions:

1. How suitable is the information TIOBE and their customers collect for an evaluation of the TQI?
2. What steps are necessary to obtain information about bug fixes in software?
3. Do the measurements on software that the TQI aggregates respond to bug fixes in this software?
4. Can TIOBE or their customers take steps to improve the quality of this analysis?

We discuss research questions 1 through 4 in Sections 7.1 to 7.4. We address the main question in Section 7.5. Finally, we discuss future work in Section 7.6.

7.1 Suitability of data for TQI validation

The first aspect we wanted to investigate was the extent to which the data TIOBE and their customers collect can be used to empirically validate the TQI. We used a survey (discussed in Chapter 5) to obtain an impression of the customers of TIOBE. Partly as a result of the suggestion in literature that the more regulated nature of industrial projects could result in a higher data quality in software repositories, we expected to find very usable information. While some projects were reported to maintain a very well-documented history of encountered defects, there was a lot of variation amongst the different projects. From this, we conclude that while the quality of the data stored in the software repositories of some projects is certainly high enough to acquire linkage, this is not the case for all projects.

We found that even when we pre-selected the projects for which we expected the analysis to be most successful, there were significant practical difficulties in

interpreting the data correctly for the case study (see Chapter 6). For version control systems, it was difficult to decide exactly which changes contributed to a bug fix. For the historical metric data, it was difficult to find out what caused a change in a measurement.

7.2 Steps to take for data acquisition

The second aspect we investigated is how we can empirically validate the TQI. In Chapter 4 we discussed how we can mine software repositories, including bug tracking systems and version control systems, to obtain the data which we can use to analyze the effectiveness of the TQI.

We found that to obtain information about the bugs in the history of the project, we needed a thorough understanding of the internals of a version control system. These internals are important for the interpretation of the structure and contents of the commits. We found that distributed version control systems pose their own set of advantages and disadvantages. These are important to take into account when mining software repositories.

Even with the large body of research in the field of bug prediction, we found the current state of the art techniques still relied on a heuristics-based algorithm (SZZ), the validity of which is questioned. While we can use linkage to find the commit and changeset which were used to fix a bug, finding the commit which introduced the bug is a far more challenging operation. Due to the problems with this methodology, we did not end up using SZZ for our analysis.

7.3 Responsiveness of metrics to bug fixes

To investigate the effectiveness of the metrics for which we have measurements, we compared the values before and after a bug fix. We covered this in Chapter 6. Based on the definition of the TQI, we proposed an expected direction for each of the metrics. We then examined whether each of the metrics increased or decreased. We found our intuitions were correct for metrics like average cyclomatic complexity and the number of compiler warnings, but the compliance factor based on this metric moved in the opposite direction. Finally, the TQI itself did not significantly change after a bug fix, most likely because it is a composition of other metrics. For each of the metrics, we found that the effect size was negligible. This is most likely due to aggregation, since relatively small changes become almost negligible when aggregating to a higher level and including a lot of unchanged files.

These results are too preliminary to draw a strong conclusion. Further research is needed to determine what the relation between these metrics and bug fixes is.

7.4 Improvement of the analysis process

From our limited experience with the investigation of software repositories, we can make some recommendations to improve the quality of these information sources.

We found that a major challenge in the interpretation of the data was a result of the use of certain operations in the version control system, particularly rebasing and fast-forward merging. These issues impact the scalability of empirical methods that use software repositories, since their resolution requires a fair amount of manual effort. A straightforward recommendation to improve on this situation is to replace the use of these operations by normal (explicit) merges, which have a similar result.

However, we should stress that this recommendation should not be applied without vigilance. While explicit merges simplify the automatic analysis of software repositories, they may become harder to understand for humans. Since a version control system is used daily by humans, this trade-off may not be worth the investment.

7.5 Relation between TQI and bugs

In conclusion, we have not been able to establish the validity of the TQI as an indicator of software quality.

While this study has revealed some interesting insights that can be used to improve the TQI, such as the distributions of the measurements and the theoretical basis behind the various metrics and aggregation techniques, we encountered some difficulties in the collection and interpretation of our data that proved to be difficult to overcome within the time-frame set for the project. We hope that the insights of this work can be used for a more thorough assessment of the validity of the TQI.

7.6 Future work

We propose a few directions of future work.

The first subject that can be researched in more detail is the way the TQI is composed. We gave an overview of how the different metrics are combined into the TQI and offered some criticism and potential improvements, but there is more work to be done here. In particular, it would be useful to collect a set of mathematical properties the TQI should satisfy, and adjust its definition based on this.

In order to empirically assess to what extent these adjustments form an improvement compared to the current definition, a statistical bug prediction model could be built. Such a model can directly be compared and its strengths can be incorporated into the TQI definition.

In addition to this, the current approach we used for validation (based on differences before and after a bug fix) could be improved upon by investigating the results at various granularity levels. We would expect lower levels of aggregation to be more responsive to changes, which should result in a larger effect size.

Finally, generalizing the results of any case study is difficult. In order to understand the effectiveness of the TQI better, we need a much larger and more diverse set of projects. By performing a set of case studies, we can attempt to collect sufficient data to deduce findings that apply to a larger subset of TIOBE customers.

Appendices

Appendix A

TED Data collection

An important data source for the project is the TED database [19]. TED is an abbreviation of "TICS Enterprise Dashboard". This dashboard provides an overview of all projects the user has access to, including all metric values as well as historical values.

The TICS Enterprise Dashboard connects to a database in which all metric data is stored. Historically, TICS has sent reports by email to customers containing the latest metric data. For each run, TICS sends an email. TED has been built on top of this system. When the need arose to collect all data in a central database, the choice was made to send these emails to a central email address. The TED database simply collects each of these emails and stores the gzipped email in the database as a Binary Large OBject (BLOB).

TICS reports have had various formats over the years. Naturally, an email can contain any text. Initially, reports were textually formatted tables. Later, the choice was made to switch to reports in JSON format. A strength of the JSON format is that it is easy to parse automatically while still being human-readable. It can also be extended with new fields easily.

The data collection consists of two broad steps:

1. querying the database;
2. parsing the email messages to extract metric data.

A.1 Database query

We combine the `Owner`, `Project` and `Report` tables to find the name of the company, name of the project and all reports. For each project, we select the latest report that was successful. The query to do this is included below.

```
1 WITH successful AS (
2     SELECT o."Name" "Owner", p."Name" "Project",
          "Site", r."Text" "Email", r."Time" "Time"
3     FROM "Owner" o
4     INNER JOIN "Project" p
5     ON o."Id" = p."OwnerId"
6     INNER JOIN "Report" r
7     ON r."ProjectId" = p."Id"
```

```
    WHERE r."Success" = 1 AND r."Text" IS NOT NULL
), latest AS (
    SELECT "Project", MAX("Time") "Time"
    FROM successful
    GROUP BY "Project"
)
SELECT successful."Owner", successful."Project",
    successful."Site", MIN("Email") "Email",
    successful."Time"
FROM successful
INNER JOIN latest ON latest."Project" =
    successful."Project"
WHERE latest."Time" = successful."Time"
GROUP BY successful."Owner", successful."Project",
    successful."Site", successful."Time"
```

A.2 Parsing email

The data that results from the database query is processed by a Python script. This script takes the BLOB that contains the email for each report, decompresses it, extracts the body of the message, attempts to find a JSON structure within this body and finally extracts all metrics from this structure.

The entire script is included below.

```
#!/usr/bin/env python
import csv
import fdb
import email
import json
import zlib
from operator import itemgetter
from pprint import pprint

def strip(s):
    lines = s.split('\n')
    try:
        start = s.split('\n').index('[') # start of
            JSON object (first occurrence of line
            containing only '[')
        result = ''.join(lines[start:])
    except ValueError:
        result = s
    start = result.index('[')
    end = result.rindex(']') + 1 # end of JSON object
        (last occurrence of ']')
    result = result[start:end]
    result = result.replace("'", '"') # replace any
        single quotes by double quotes to support
        invalid JSON
```

```
    return result

def make_pretty(dictionary):
    '''Transforms a dictionary to a prettier format.

    >>> pprint(make_pretty([{'key1': 'val1'},
       {'key2': 'val2'}]))
    {'key1': 'val1', 'key2': 'val2'}
    >>> pprint(make_pretty([{'dict': [{'key1':
       'val1'}, {'key2': 'val2'}]}]))
    {'dict': {'key1': 'val1', 'key2': 'val2'}}
    '''
    if isinstance(dictionary, list) and
       len(dictionary) > 0 and
       isinstance(dictionary[0], dict):
        # merge all dicts in the list
        result = {}
        for d in dictionary:
            if isinstance(d, dict):
                for key in d:
                    val = make_pretty(d[key])
                    result.update({key: val})
        return result
    elif isinstance(dictionary, list) and
       len(dictionary) > 0 and
       isinstance(dictionary[0], list):
        result = []
        l = dictionary
        for elt in l:
            result.append(make_pretty(elt))
        return result
    else:
        return dictionary

def parse_email(mail):
    mail_dec = zlib.decompress(mail,
       16+zlib.MAX_WBITS)
    message = email.message_from_string(mail_dec)
    json_obj = message.get_payload()
    try:
        result = json.loads(strip(json_obj))
        return result
    except ValueError:
        # ignore silently XXX
        return None

def get_main_language(loc):
    '''
    Derive main language from loc metric.
    '''
```

```
64      languages = {elt['Language']: elt['Value'] for
            elt in loc.get('Values', {}) if
            elt['Language'] != 'All'}
65      try:
66          return max(languages.iteritems(),
                key=itemgetter(1))[0]
67      except ValueError:
68          return None
69
70  def find(structure, key, val):
71      '''
72      Finds a key, value pair in `structure` with
            structure[key] == val
73      '''
74      if not structure:
75          return {}
76      for d in structure:
77          if d[key] == val:
78              return d
79      return {}
80
81  if __name__ == "__main__":
82      import doctest
83      doctest.testmod()
84
85      import pickle
86      with open('./recent_dump.dmp') as f: cur =
            pickle.load(f)
87
88      all_metrics = set()
89      for (owner, project, site, mail, time) in cur:
90          structure = make_pretty(parse_email(mail))
91          if structure:
92              enabled_metrics =
                    structure.get('Header').get('Enabled
                    metrics')
93              if enabled_metrics:
94                  all_metrics.update(enabled_metrics)
95      all_metrics = sorted(all_metrics)
96
97      ms = ('DUPLICATEDCODE',
98            'ABSTRACTINTERPRETATION/VIOLATIONSPERLEVEL',
99            'ABSTRACTINTERPRETATION/SUPPRESSIONSPERLEVEL',
100           'UNITSTATEMENTCOVERAGE',
101           'UNITBRANCHCOVERAGE',
102           'UNITFUNCTIONCOVERAGE',
103           'UNITDECISIONCOVERAGE',
104           'DEADCODE',)
105     cols = ['Owner', 'Project', 'Site', 'Time', 'Main
            language', 'Number of files', 'LOC'] +
```

```
        all_metrics
    with open('recent_metrics.csv', 'w') as csv_file:
        csv_writer = csv.writer(csv_file,
            delimiter=',', quotechar='"',
            quoting=csv.QUOTE_MINIMAL)
        csv_writer.writerow(cols)
        for (owner, project, site, mail, time) in cur:
            mail = make_pretty(parse_email(mail))
            if not mail: mail = {}
            metrics = mail.get('Metrics')
            loc = find(metrics, 'Metric name', 'LOC')
            main_language = get_main_language(loc)
            loc = find(loc.get('Values'), 'Language',
                'All').get('Value')
            num_files = max(find(metrics, 'Metric
                name', m).get('Analyzed',
                {}).get('Total') for m in ms)
            metrics = [find(find(metrics, 'Metric
                name', m).get('Values'), 'Language',
                'All').get('Value') for m in
                all_metrics]
            row = [owner, project, site, time,
                main_language, num_files, loc] +
                metrics
            csv_writer.writerow(row)
```

Appendix B

Probability density plots

This chapter contains some plots of the distributions of the various metrics that TIOBE uses. These are generated using the R script below.

```
1  #!/usr/bin/Rscript
2  require(moments) # install.packages('moments')
3  ted <- read.csv("../ted/ted.csv")
4  dir.create('output', showWarnings = FALSE)
5
6  pdf('output/tqi.pdf')
7  plot(density(ted$TQI.Score, na.rm = TRUE, kernel =
       "epanechnikov", from = 0, to = 100), main="",
       xlab="")
8  dev.off()
9  skewness(ted$TQI.Score)
10 kurtosis(ted$TQI.Score)
11
12 TQI.AbstrInt <- ted$AbstrInt*2-100
13 TQI.AbstrInt[TQI.AbstrInt < 0] <- 0
14 pdf('output/ai_raw.pdf')
15 par(cex=2)
16 plot(density(ted$AbstrInt, na.rm = TRUE, kernel =
       "epanechnikov", from = 0, to = 100), main="",
       xlab="")
17 dev.off()
18 pdf('output/ai_tqi.pdf')
19 par(cex=2)
20 plot(density(TQI.AbstrInt, na.rm = TRUE, kernel =
       "epanechnikov", from = 0, to = 100), main="",
       xlab="")
21 dev.off()
22
23 TQI.Cyclox <- 140 - 20 * ted$Cyclox
24 TQI.Cyclox[TQI.Cyclox < 0] <- 0
25 TQI.Cyclox[TQI.Cyclox > 100] <- 100
26 pdf('output/cc_raw.pdf')
```

```
27 par(cex=2)
28 plot(density(ted$Cyclox, na.rm = TRUE, kernel =
       "epanechnikov", from = 0), main="", xlab="")
29 dev.off()
30 pdf('output/cc_tqi.pdf')
31 par(cex=2)
32 plot(density(TQI.Cyclox, na.rm = TRUE, kernel =
       "epanechnikov", from = 0, to = 100), main="",
       xlab="")
33 dev.off()
34
35 TQI.CompWarn <- 100 - 50 * log10(101-ted$CompWarn)
36 TQI.CompWarn[TQI.CompWarn < 0] <- 0
37 pdf('output/cw_raw.pdf')
38 par(cex=2)
39 plot(density(ted$CompWarn, na.rm = TRUE, kernel =
       "epanechnikov", from = 0, to = 100), main="",
       xlab="")
40 dev.off()
41 pdf('output/cw_tqi.pdf')
42 par(cex=2)
43 plot(density(TQI.CompWarn, na.rm = TRUE, kernel =
       "epanechnikov", from = 0, to = 100), main="",
       xlab="")
44 dev.off()
45
46 TQI.DupCode <- -30 * log10(ted$DupCode) + 70
47 TQI.DupCode[TQI.DupCode > 100] <- 100
48 pdf('output/dup_raw.pdf')
49 par(cex=2)
50 plot(density(ted$DupCode, na.rm = TRUE, kernel =
       "epanechnikov", from = 0, to = 100), main="",
       xlab="")
51 dev.off()
52 pdf('output/dup_tqi.pdf')
53 par(cex=2)
54 plot(density(TQI.DupCode, na.rm = TRUE, kernel =
       "epanechnikov", from = 0, to = 100), main="",
       xlab="")
55 dev.off()
56 skewness(ted$DupCode, na.rm = TRUE)
57 kurtosis(ted$DupCode, na.rm = TRUE)
58 skewness(TQI.DupCode, na.rm = TRUE)
59 kurtosis(TQI.DupCode, na.rm = TRUE)
60
61 TQI.FanOut <- 120 - (5 * ted$FanOut)
62 TQI.FanOut[TQI.FanOut < 0] <- 0
63 TQI.FanOut[TQI.FanOut > 100] <- 100
64 pdf('output/fanout_raw.pdf')
65 par(cex=2)
```

```
66  plot(density(ted$FanOut, na.rm = TRUE, kernel =
        "epanechnikov", from = 0), main="", xlab="")
67  dev.off()
68  pdf('output/fanout_tqi.pdf')
69  par(cex=2)
70  plot(density(TQI.FanOut, na.rm = TRUE, kernel =
        "epanechnikov", from = 0, to = 100), main="",
        xlab="")
71  dev.off()
72
73  TQI.DeadCode <- 100 - (2 * ted$DeadCode)
74  TQI.DeadCode[TQI.DeadCode < 0] <- 0
75  pdf('output/dead_raw.pdf')
76  par(cex=2)
77  plot(density(ted$DeadCode, na.rm = TRUE, kernel =
        "epanechnikov", from = 0, to = 100), main="",
        xlab="")
78  dev.off()
79  pdf('output/dead_tqi.pdf')
80  par(cex=2)
81  plot(density(TQI.DeadCode, na.rm = TRUE, kernel =
        "epanechnikov", from = 0, to = 100), main="",
        xlab="")
82  dev.off()
83  skewness(ted$DeadCode, na.rm = TRUE)
84  kurtosis(ted$DeadCode, na.rm = TRUE)
85  skewness(TQI.DeadCode, na.rm = TRUE)
86  kurtosis(TQI.DeadCode, na.rm = TRUE)
87
88  TQI.unitSCov <- (0.75 * ted$unitSCov) + 32.5
89  TQI.unitSCov[TQI.unitSCov > 100] <- 100
90  pdf('output/coverage_raw.pdf')
91  par(cex=2)
92  plot(density(ted$unitSCov, na.rm = TRUE, kernel =
        "epanechnikov", from = 0, to = 100), main="",
        xlab="")
93  dev.off()
94  pdf('output/coverage_tqi.pdf')
95  par(cex=2)
96  plot(density(TQI.unitSCov, na.rm = TRUE, kernel =
        "epanechnikov", from = 0, to = 100), main="",
        xlab="")
97  dev.off()
98
99  TQI.CodingStd <- ted$CodingStd
100 pdf('output/cs_raw.pdf')
101 plot(density(ted$unitSCov, na.rm = TRUE, kernel =
        "epanechnikov", from = 0, to = 100), main="",
        xlab="")
102 dev.off()
```

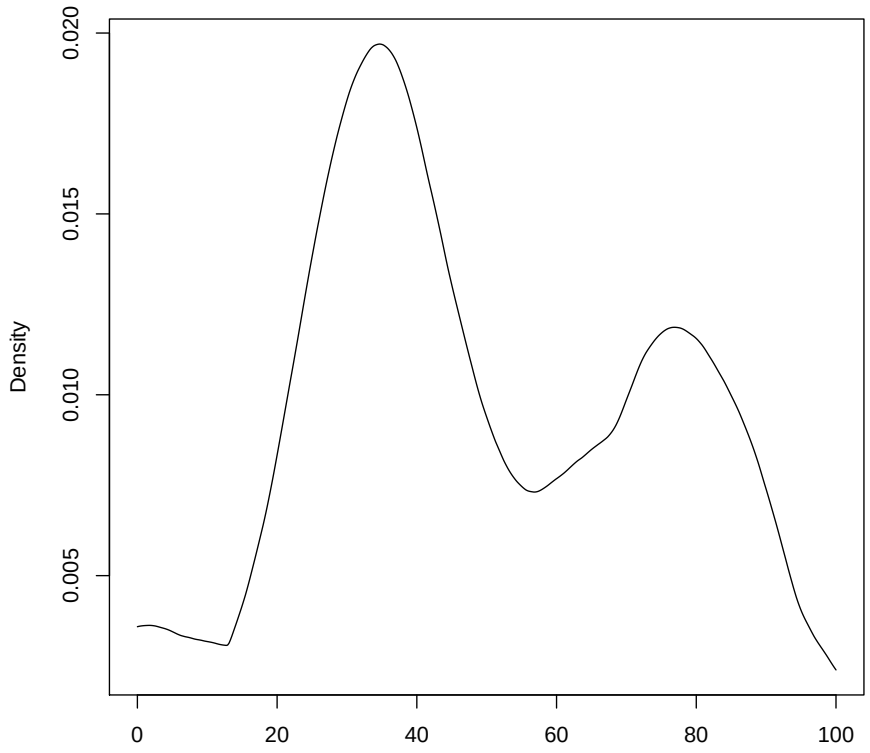
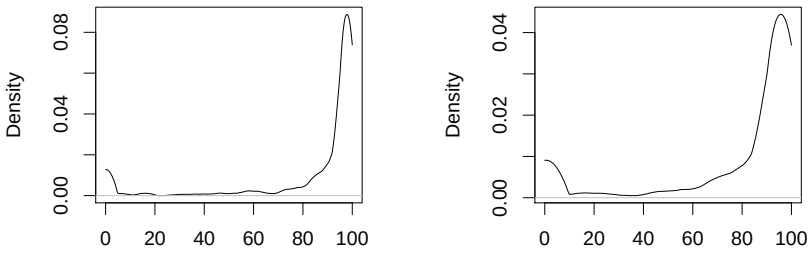


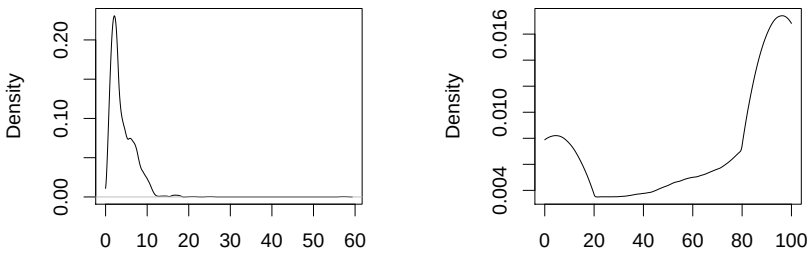
Figure B.1: Density plot for the TQI.

```
103
104 pdf('output/normal.pdf')
105 curve(dnorm, type="l", xlab="", ylab="Density", xlim
        = c(-3.5,3.5))
106 dev.off()
```



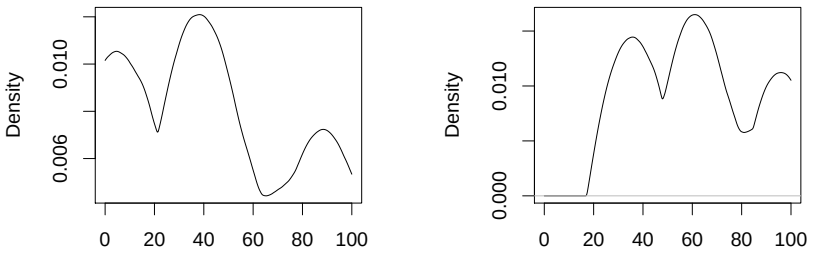
(a) raw (b) transformed for TQI integration

Figure B.2: Density plots for abstract interpretation.



(a) raw (b) transformed for TQI integration

Figure B.3: Density plots for cyclomatic complexity.



(a) raw (b) transformed for TQI integration

Figure B.4: Density plots for code coverage.

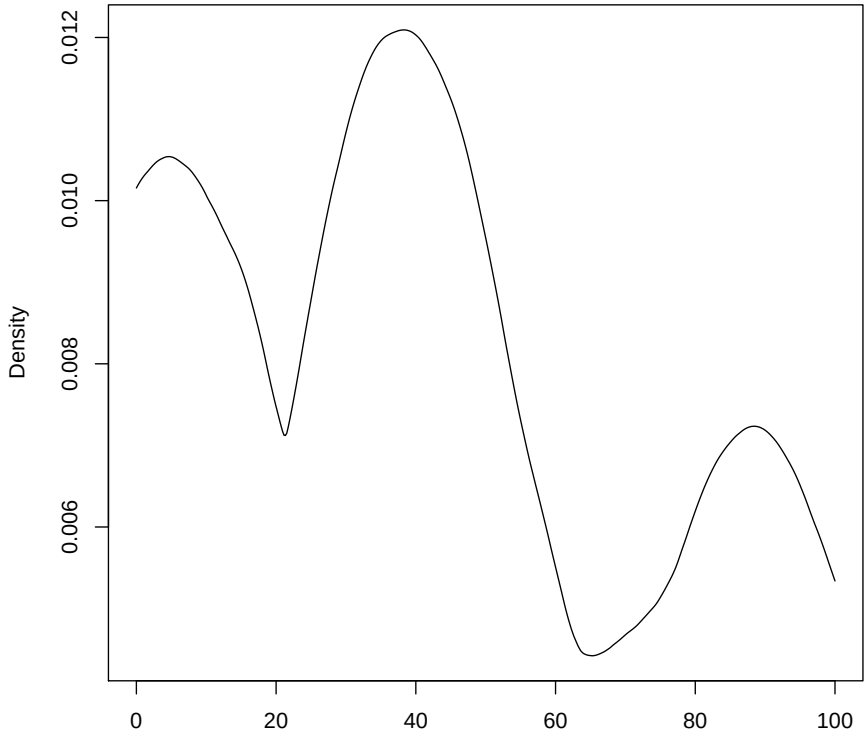
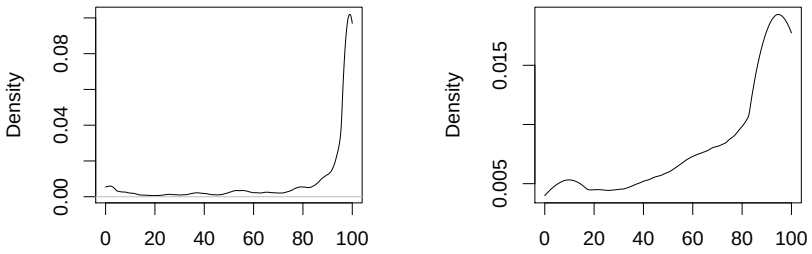
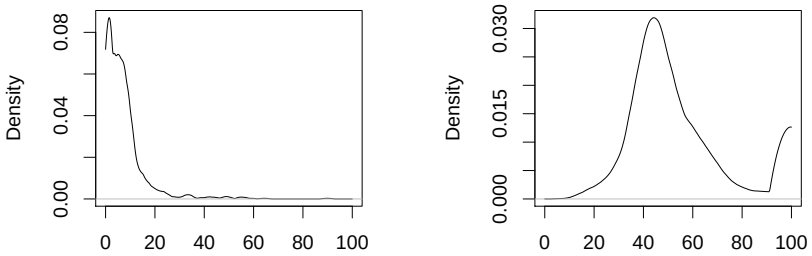


Figure B.5: Density plots for coding standards.



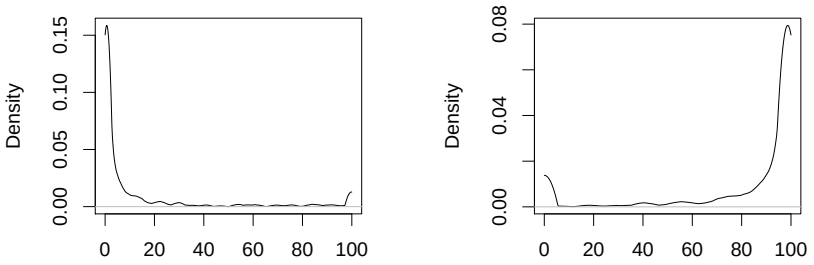
(a) raw (b) transformed for TQI integration

Figure B.6: Density plots for compiler warnings.



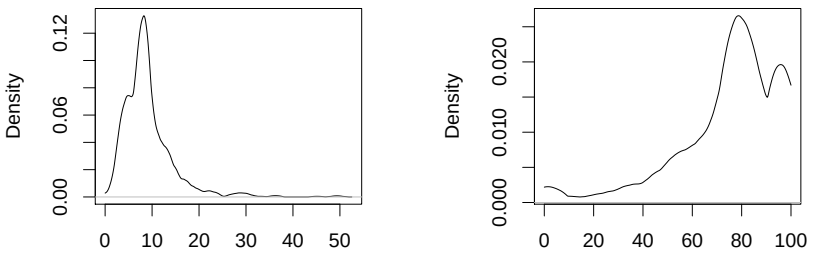
(a) raw (b) transformed for TQI integration

Figure B.7: Density plots for duplicate code.



(a) raw (b) transformed for TQI integration

Figure B.8: Density plots for dead code.



(a) raw (b) transformed for TQI integration

Figure B.9: Density plots for fan-out.

Glossary

abstract interpretation "A method for designing approximate semantics of programs which can be used to gather information about programs in order to provide sound answers to questions about their run-time behaviours" [70]. See Section 2.3.10. *pp.* 4, 22, 23, 28, 31, 32, 35, 38, 43, 46, 69–71, 79

aggregation The process of combining the measurements on multiple low-level components (e.g., methods, files) to obtain a value representing the same metric on a higher level (e.g., the entire project) [5]. The term "aggregation" is sometimes used to refer to *composition*. *pp.* 2, 6, 9, 14, 26–29, 31, 34, 36, 43, 46, 52, 58, 65, 71, 72, 78, 80, 82–85

branch coverage *pp.* 20, 80, *see* decision coverage

bug A *fault*. *pp.* 6, 7, 14, 15, 22–25, 32, 40, 47–52, 57, 59, 61, 63, 64, 74, 76–78, 81, 83, 103

bug prediction The process of estimating the likelihood of software *defects* in software [48]. *pp.* 10, 11, 24, 47, 50–52, 57, 59, 61

bug report *pp.* 50, 60, 73, *see* bug ticket

bug ticket A report about a *bug* that has been found in a software project. Bug tickets are submitted to a *bug tracking system*. *pp.* 49, 50, 52, 58, 59, 63, 64, 70, 74

bug tracking system "A software application that helps in tracking and documenting the reported software bugs" [155]. *pp.* 6, 7, 47–51, 57–66, 68, 71, 74, 75, 84, 103, 106

bug-fixing commit Commit that fixes a bug in the software. *pp.* 50, 52, 57, 74

bug-inducing commit Commit that introduces a bug into the software [156]. *pp.* 50, 52, 58, 74, 107

changeset A set of changes to files that is *committed* to a *version control system*. *pp.* 25, 48, 58, 104

Chidamber and Kemerer metrics A suite of metrics that aim to measure the quality of the design of object-oriented programs [27]. See Section 2.3.7. *pp.* 17, 28

code churn Metric for the amount of change in source code, expressed in the number of lines that were added, removed, or changed [28]. See Section 2.4.2. *pp.* 25, 52

code coverage A metric of test suite quality [66]. See Section 2.3.8. *pp.* 4, 12, 19–21, 28, 31, 32, 37, 38, 46, 58, 63, 77, 104–107

code duplication The process of copying code to re-use it in a new context [51]. The amount of code duplicated code can be used as a metric for the proportion of redundant code in a project. See Section 2.3.4. *pp.* 5, 15, 31, 35, 40, 77

coding standard A set of rules for software developers to follow while writing code [108]. See Section 2.3.12. *pp.* 4, 9, 24, 28, 29, 31, 32, 34, 35, 39, 40, 46, 76, 77, 79, 80, 107

cohesion The degree of interdependency of components within a module [56]. See Section 2.3.6. *pp.* 17, 19

commit The act of providing changes to a *version control system* for storage. The word "commit" is often used as a noun to refer to the changes themselves (changeset). *pp.* 25, 48, 50–52, 59, 62–64, 66, 69–71, 73, 74, 76, 81, 103

commit message A textual message that may be used to describe the intention of a commit. Commit messages are intended to be human-readable, but may also be used to include machine-readable information, such as a reference to documentation. *pp.* 48–50, 55, 59, 61–64, 66, 70, 73, 74

commit parent The "predecessor" of a commit. That is, the commit for which the contents were modified to arrive at the current version. Parents are particularly relevant in a distributed *VCS*, where the parent of a commit is stored explicitly. *pp.* 52, 53, 55, 56

compiler warnings Potential errors in the source code that are discovered during compilation. See Section 2.3.11. *pp.* 4, 23, 24, 28, 29, 31, 32, 35, 39, 73, 77, 80, 81, 84

compliance factor A method to compute a single number based on a set of violations of various types [91] (see Section 3.2). *pp.* 29, 32, 34, 35, 38, 39, 77, 107

composition The process of combining different metrics into a single value that is intended to represent a combination of the semantics of these metrics [5]. *pp.* 9, 27–29, 31, 36, 43, 46, 52, 69, 82, 103, 107

condition coverage *Code coverage* metric based on the proportion of conditions in each decision being covered [61, pp. 45–46]. A condition is a single boolean value that a decision can be based on. See Section 2.3.8. *pp.* 20, 21, 37, 38

condition/decision coverage A combination of condition and *decision coverage* [61, pp. 46–47]. *pp.* 20, 37

coupling The degree of interdependency of components outside of a module [56]. See Section 2.3.6. *pp.* 17, 18

cyclomatic complexity *pp.* 4, 10, 12, 14, 18, 27, 28, 30, 31, 35, 38, 39, 43, 52, 58, 76, 80, 84, *see* McCabe complexity

dead code Code for which the result will never be used [54, p. 350]. See Section 2.3.5. *pp.* 5, 16, 31, 35, 43, 77

decision coverage *Code coverage* metric based on the proportion of decision branches covered by unit tests [61, pp. 44–45]. See Section 2.3.8. *pp.* 20, 21, 37, 38, 46, 104

defect A problem which, if not corrected, could cause an application to either fail or to produce incorrect results [157]. Often used as a synonym for "fault". *pp.* 1, 2, 4, 7, 25, 32, 34, 47, 50, 57, 103

defect prediction *pp.* 28, *see* bug prediction

diff A tool to compare files line by line [158]. The term "diff" can be used both to refer to the command itself (typeset as `diff` in this document) and the output that the tool produces, i.e., a list of added, removed, or modified lines along with information about their location in the file. The `diff` command is not only available as a stand-alone application, but is also integrated in version control systems such as Subversion and Git. *pp.* 48, 71, 74

external validity The extent to which a study's results generalize to other settings outside of the study [152]. *pp.* 80, 82

fan-out The number of modules from which functions are imported. See Section 2.3.6. *pp.* 5, 17, 31, 35, 41–43, 46, 58, 71, 72, 76, 77, 79, 82, 107

fault A manifestation of an error in software [157]. *pp.* 6, 11, 21, 22, 24, 25, 40, 47, 49, 103, 105

function coverage Code coverage metric based on the proportion of functions (methods) in a class, module, or program that is invoked by a unit test. *pp.* 20, 37

functional suitability "Degree to which a product or system provides functions that meet stated and implied needs when used under specified conditions" [24, p. 10]. *pp.* 4, 106

Halstead complexity measures A series of metrics aimed at measuring software size and complexity and effort estimation [49]. See Section 2.3.3. *pp.* 14, 28

import A *statement* that declares the usage of some other class or package. It is known as `import` (Java, Python), `using` (C#) and `require` (PHP, Node.js). *pp.* 5, 17, 41, 42, 46

lines of code Metric for software size. See Section 2.3.1. *pp.* 2, 11, 12, 35, 40, 43, 73, 80

linkage The process of linking the tickets from a *bug tracking system* to the information in a version control system [120]. *pp.* 50, 51, 64, 70, 73, 74

maintainability "Degree of effectiveness and efficiency with which a product or system can be modified by the intended maintainers" [24, p. 14];. *pp.* 4, 24, 106

measurement "The empirical, objective assignment of numbers, according to a rule derived from a model or theory, to attributes of objects or events with the intent of describing them" [25]. *pp.* 2, 3, 7, 9, 10, 14, 15, 26–28, 32, 39, 42, 43, 46, 58, 66, 68, 69, 71, 78, 79, 81, 82, 84, 85, 103

metric *pp.* 10, 11, 15, 29, 46, 47, 58, 66, 82, 84, 95, 103–107, *see* software metric

Modified Condition/Decision Coverage Type of *code coverage* that is applied in avionics and other critical system domains [65]. *pp.* 20, 109

mutation coverage A metric for test suite quality that is designed to overcome a weakness of *code coverage* [67]. See Section 2.3.9. *pp.* 21, 22

normal distribution A continuous probability distribution that forms the basis of many statistical tests. *pp.* 66, 78

path coverage *Code coverage* metric based on the proportion of execution paths through a method that are covered [62]. See Section 2.3.8. *pp.* 20, 21, 37

quality factor A characteristic of *software quality* [23]. Examples include: *reliability*, *functional suitability*, and *maintainability*. *pp.* 3–6

quality metric *Metric* for software quality. *pp.* 4–6, 107

reliability "Degree to which a system, product or component performs specified functions under specified conditions for a specified period of time" [24, p. 13]. *pp.* 4, 106

repository mining The process of extracting data from *software archives* [110]. *pp.* 47–50, 58

software archive Data which is generated during the software development process [110]. Examples of software archives include *version control systems* and *bug tracking systems*, but also the source code of the software itself and documentation. *pp.* 47, 106

software metric Mapping from the (components of) software to a value that represents a certain property, such as quality or complexity [25]. Metrics are covered in Chapter 2. *pp.* 2, 6, 9, 28, 47, 66, 78, 106, 107

software quality The quality of software. For a discussion of the various definitions of this concept, refer to Chapter 1. *pp.* 1, 2, 4–6, 9, 10, 25, 26, 29, 30, 47, 57, 76, 77, 80, 85, 106, 107

statement The smallest stand-alone unit of an imperative programming language that describes some action that the program should execute. While a statement is not necessary equivalent to a line (multiple statements could appear on a single line and vice versa), this distinction is not always made. *pp.* 12, 13, 15, 16, 19–21, 37, 41, 42, 52, 71, 76, 105, 107

statement coverage *Code coverage* metric based on the proportion of statements covered by unit tests [61, p. 44]. See Section 2.3.8. *pp.* 19–21, 37, 38, 46

static defect count The estimated number of bugs in a program based on violations of the coding standard [91]. This concept is used to define the compliance factor. Static defect count is presented in Section 3.2.1. *pp.* 32, 34, 35

statistical test A test that uses statistical methods to infer whether some null hypothesis can be rejected. The null hypothesis is typically the default position that there is no interesting effect to be observed. The test can be used to determine whether this null hypothesis can be rejected, providing evidence for the alternative test that there is an interesting effect [150, p. 1020]. *pp.* 78–81, 106

SZZ Algorithm to find bug-inducing commits using heuristics that are applied on information from the version control system. Named after the authors: Śliwerski, Zimmermann, and Zeller [126]. *pp.* 50, 52, 65, 66, 70, 73, 74, 76, 82, 84

test coverage *pp.* 35, 80, *see* code coverage

TICS The primary product by TIOBE. TICS periodically runs a variety of tools to obtain measurements and stores these in a database. *pp.* 28, 29, 33, 37, 39, 40, 44, 46, 62, 65–69, 71, 73, 76, 81, 82, 89

TICS Enterprise Dashboard An application developed by TIOBE that provides an overview of all projects the user has access to, including all current metric values as well as historical values. *pp.* 65, 89, 109

TICSCil A tool to compute *fan-out* for C# projects, developed internally by TIOBE. *pp.* 41, 42, 71, 73, 79, 80, 82

TIOBE The company where this research was carried out. TIOBE measures *software quality* using *software metrics* and has created the TQI. *pp.* 2–7, 9, 10, 12, 16, 17, 22, 23, 25, 26, 28–30, 32, 34, 36, 38–46, 48, 57, 59, 60, 62, 64–66, 82, 83, 85, 95, 107

TIOBE Quality Indicator A *composition* of *quality metrics* that forms a metric for *software quality* by TIOBE [16]. *pp.* 2, 109

TQI integration The process of applying one of the formulas in Section 3.3 to obtain a TQI component with a value in the domain $[0, 100]$. *pp.* 28, 29, 35, 36, 43, 44, 66–68, 99–102

unreachable code Code that can never be executed [55]. See Section 2.3.5. *pp.* 16, 42, 43

version control system System that keeps track of different revisions of (the source code of) software [159]. *pp.* 7, 25, 47–51, 53, 56, 58, 60–62, 64, 65, 68, 71, 75, 84, 85, 103–106, 109

Acronyms

BLOB Binary Large OBject *pp.* 89, 90

DAG Directed Acyclic Graph *pp.* 48, 53, 55, 56

KLOC Kilo (1000) Lines Of Code *pp.* 35, 60, 80

LOC Lines Of Code *pp.* 2, 5, 10–12, 15, 24, 25, 27, 34, 43, 58, 73

MC/DC Modified Condition/Decision Coverage *pp.* 20, 21, 37

SHA-1 Secure Hashing Algorithm 1 *pp.* 48, 53

SLOC Source Lines Of Code *pp.* 11, 17

SQuaRE Systems and software Quality Requirements and Evaluation *pp.* 4, 5

TED TICS Enterprise Dashboard *pp.* 65, 66, 82, 89

TQI TIOBE Quality Indicator *pp.* 2, 3, 6, 7, 9, 10, 12, 14–17, 19, 22–24, 28–33, 35–40, 43–47, 51, 52, 58, 65–73, 76–78, 80, 82–85, 98, 107, 108

VCS Version Control System *pp.* 48, 49, 53–56, 104

Bibliography

[1] G. Bavota and B. Russo, "Four eyes are better than two: On the impact of code reviews on software quality," in *ICSME*, IEEE Computer Society, 2015, pp. 81–90.

[2] G. Concas, M. Marchesi, C. Monni, M. Orrú, and R. Tonelli, "Predicting software quality through network analysis," in *SATToSE*, 2015.

[3] S. H. Kan, *Metrics and Models in Software Quality Engineering*, Second. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[4] B. Kitchenham and S. L. Pfleeger, "Software quality: The elusive target," *IEEE Software*, vol. 13, no. 1, pp. 12–21, 1996.

[5] K. Mordal-Manet, N. Anquetil, J. Laval, A. Serebrenik, B. Vasilescu, and S. Ducasse, "Software quality metrics aggregation in industry," *Journal of Software: Evolution and Process*, vol. 25, no. 10, pp. 1117–1135, 2013.

[6] A. Potdar and E. Shihab, "An exploratory study on self-admitted technical debt," in *ICSME*, IEEE Computer Society, 2014, pp. 91–100.

[7] N. F. Schneidewind, "Methodology for validating software metrics," *IEEE Trans. Software Eng.*, vol. 18, no. 5, pp. 410–422, 1992.

[8] T. Zimmermann, N. Nagappan, and A. Zeller, "Predicting bugs from history," in *Software Evolution*, T. Mens and S. Demeyer, Eds., Springer, 2008, pp. 69–88.

[9] I. Sommerville, *Software Engineering*. Pearson, 2011.

[10] H. Weber, *The software factory challenge*. IOS Press, 1997.

[11] B. W. Boehm and V. R. Basili, "Software defect reduction top 10 list," *IEEE Computer*, vol. 34, no. 1, pp. 135–137, 2001.

[12] D. A. Garvin, "What does product quality really mean," *Sloan management review*, vol. 26, no. 1, 1984.

[13] T. Gilb, *Competitive Engineering: A Handbook For Systems Engineering, Requirements Engineering, and Software Engineering Using Planguage*. Newton, MA, USA: Elsevier Butterworth-Heinemann, 2005.

[14] T. DeMarco and T. Lister, *Peopleware: Productive Projects and Teams*, Second. New York, NY, USA: Dorset House Publishing Co., Inc., 1999.

[15] TIOBE Software. (2016). TIOBE software: The coding standards company, [Online]. Available: `http://www.tiobe.com/index.php/content/company/Home.html` (visited on Jan. 27, 2016).

[16] P. Jansen. (2012). The TIOBE Quality Indicator: A pragmatic way of measuring code quality, [Online]. Available: `http://www.tiobe.com/content/paperinfo/TIOBEQualityIndicator.pdf` (visited on Oct. 19, 2015).

[17] TIOBE Software. (2016). TIOBE software: Fact sheet, [Online]. Available: `http://www.tiobe.com/index.php/content/TICS/FactSheet.html` (visited on Jan. 27, 2016).

[18] ——, (2016). TIOBE software: Customers, [Online]. Available: `http://www.tiobe.com/index.php/content/TICS/Customers.html` (visited on Jan. 27, 2016).

[19] ——, "TICS enterprise dashboard (TED) database," Accessed: Jan. 27, 2016.

[20] S. Easterbrook, J. Singer, M. Storey, and D. Damian, "Selecting empirical methods for software engineering research," in *Guide to Advanced Empirical Software Engineering*, F. Shull, J. Singer, and D. I. K. Sjøberg, Eds., London: Springer London, 2008, pp. 285–311.

[21] G. J. Myers, "A controlled experiment in program testing and code walkthroughs/inspections," *Commun. ACM*, vol. 21, no. 9, pp. 760–768, 1978.

[22] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Software Engineering*, vol. 14, no. 2, pp. 131–164, 2009.

[23] A. B. Al-Badareen, M. H. Selamat, M. A. Jabar, J. Din, and S. Turaev, "Software quality models: A comparative study," in *ICSECS*, J. M. Zain, W. M. B. W. Mohd, and E. El-Qawasmeh, Eds., ser. Communications in Computer and Information Science, vol. 179, Springer, 2011, pp. 46–55.

[24] ISO/IEC, "ISO/IEC 25010 — systems and software engineering — systems and software quality requirements and evaluation (SQuaRE) — system and software quality models," International Organization for Standardization, Tech. Rep., 2010.

[25] C. Kaner and W. P. Bond, "Software engineering metrics: What do they measure and how do we know?" In *METRICS*, IEEE Computer Society, 2004, pp. 1–12.

[26] T. J. McCabe Sr., "A complexity measure," *IEEE Trans. Software Eng.*, vol. 2, no. 4, pp. 308–320, 1976.

[27] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Software Eng.*, vol. 20, no. 6, pp. 476–493, 1994.

[28] N. Nagappan and T. Ball, "Use of relative code churn measures to predict system defect density," in *ICSE*, G. Roman, W. G. Griswold, and B. Nuseibeh, Eds., ACM, 2005, pp. 284–292.

[29] S. M. Henry and D. G. Kafura, "Software structure metrics based on information flow," *IEEE Trans. Software Eng.*, vol. 7, no. 5, pp. 510–518, 1981.

[30] N. E. Fenton and M. Neil, "Software metrics: Roadmap," in *ICSE*, A. Finkelstein, Ed., ACM, 2000, pp. 357–370.

[31] G. Baxter, M. R. Frean, J. Noble, M. Rickerby, H. Smith, M. Visser, H. Melton, and E. D. Tempero, "Understanding the shape of Java software," in *OOPSLA*, P. L. Tarr and W. R. Cook, Eds., ACM, 2006, pp. 397–412.

[32] "IEEE standard for a software quality metrics methodology," *IEEE Std 1061-1998*, Dec. 1998.

[33] C. Sadowski, C. Lewis, Z. Lin, X. Zhu, and E. J. Whitehead Jr., "An empirical analysis of the FixCache algorithm," in *MSR*, 2011, pp. 219–222.

[34] H. Hata, O. Mizuno, and T. Kikuno, "Bug prediction based on fine-grained module histories," in *ICSE*, M. Glinz, G. C. Murphy, and M. Pezzè, Eds., IEEE, 2012, pp. 200–210.

[35] E. Shihab, "An exploration of challenges limiting pragmatic software defect prediction," PhD thesis, Queen's University, 2012.

[36] F. Rahman and P. T. Devanbu, "How, and why, process metrics are better," in *ICSE*, D. Notkin, B. H. C. Cheng, and K. Pohl, Eds., IEEE, 2013, pp. 432–441.

[37] N. Nagappan, T. Ball, and A. Zeller, "Mining metrics to predict component failures," in *ICSE*, L. J. Osterweil, H. D. Rombach, and M. L. Soffa, Eds., ACM, 2006, pp. 452–461.

[38] T. J. Ostrand and E. J. Weyuker, "Predicting bugs in large industrial software systems," in *ISSSE*, A. De Lucia and F. Ferrucci, Eds., ser. Lecture Notes in Computer Science, vol. 7171, Springer, 2011, pp. 71–93.

[39] I. Herraiz and A. E. Hassan, "Beyond lines of code: Do we need more complexity metrics?" In *Making Software: What Really Works, and Why We Believe It*, A. Oram and G. Wilson, Eds., Sebastopol, Calif., USA: O'Reilly Media, 2010, pp. 125–141.

[40] N. Nagappan and T. Ball, "Evidence-based failure prediction," in *Making Software: What Really Works, and Why We Believe It*, A. Oram and G. Wilson, Eds., Sebastopol, Calif., USA: O'Reilly Media, 2010, pp. 415–434.

[41] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, "A systematic literature review on fault prediction performance in software engineering," *IEEE Trans. Software Eng.*, vol. 38, no. 6, pp. 1276–1304, 2012.

[42] R. M. Bell, T. J. Ostrand, and E. J. Weyuker, "Looking for bugs in all the right places," in *ISSTA*, L. L. Pollock and M. Pezzè, Eds., ACM, 2006, pp. 61–72.

[43] D. Landman, A. Serebrenik, E. Bouwers, and J. Vinju, "Empirical analysis of the relationship between CC and SLOC in a large corpus of Java methods and C functions," *Journal of Software: Evolution and Process*, 2015.

[44] E. Giger, M. D'Ambros, M. Pinzger, and H. C. Gall, "Method-level bug prediction," in *ESEM*, P. Runeson, M. Höst, E. Mendes, A. A. Andrews, and R. Harrison, Eds., ACM, 2012, pp. 171–180.

[45] F. Camilo, A. Meneely, and M. Nagappan, "Do bugs foreshadow vulnerabilities? A study of the chromium project," in *MSR*, IEEE, 2015, pp. 269–279.

[46] N. E. Fenton and N. Ohlsson, "Quantitative analysis of faults and failures in a complex software system," *IEEE Trans. Software Eng.*, vol. 26, no. 8, pp. 797–814, 2000.

[47] B. W. Boehm, "Software engineering economics," *IEEE Trans. Softw. Eng.*, vol. 10, no. 1, pp. 4–21, 1984.

[48] M. D'Ambros, M. Lanza, and R. Robbes, "An extensive comparison of bug prediction approaches," in *MSR*, 2010, pp. 31–41.

[49] M. H. Halstead, *Elements of software science*, ser. Operating and programming systems. Elsevier, 1977.

[50] J. K. Kearney, R. L. Sedlmeyer, W. B. Thompson, M. A. Gray, and M. A. Adler, "Software complexity measurement," *Commun. ACM*, vol. 29, no. 11, pp. 1044–1050, 1986.

[51] I. D. Baxter, A. Yahin, L. M. de Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *ICSM*, IEEE Computer Society, 1998, pp. 368–377.

[52] J. H. Johnson, "Substring matching for clone detection and change tracking," in *ICSM*, H. A. Müller and M. Georges, Eds., IEEE Computer Society, 1994, pp. 120–126.

[53] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Sci. Comput. Program.*, vol. 74, no. 7, pp. 470–495, 2009.

[54] A. W. Appel and J. Palsberg, *Modern Compiler Implementation in Java*, Second. Cambridge University Press, 2002.

[55] A. Srivastava, "Unreachable procedures in object-oriented programming," *LOPLAS*, vol. 1, no. 4, pp. 355–364, 1992.

[56] M. Page-Jones, "Comparing techniques by means of encapsulation and connascence," *Commun. ACM*, vol. 35, no. 9, pp. 147–151, 1992.

[57] D. N. Card and W. W. Agresti, "Measuring software design complexity," *Journal of Systems and Software*, vol. 8, no. 3, pp. 185–197, 1988.

[58] R. C. Martin, *Agile Software Development: Principles, Patterns, and Practices*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2003.

[59] M. Hitz and B. Montazeri, "Chidamber and kemerer's metrics suite: A measurement theory perspective," *IEEE Trans. Software Eng.*, vol. 22, no. 4, pp. 267–271, 1996.

[60] B. Henderson-Sellers, *Object-oriented Metrics: Measures of Complexity*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996.

[61] G. J. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing*, Third. Wiley Publishing, 2011.

[62] R. Gopinath, C. Jensen, and A. Groce, "Code coverage for suite evaluation by developers," in *ICSE*, New York, NY, USA: ACM, 2014, pp. 72–82.

[63] S. Cornett. (2014). Code coverage analysis, [Online]. Available: `http://www.bullseye.com/coverage.html` (visited on Dec. 7, 2015).

[64] K. J. Hayhurst, D. S. Veerhusen, J. J. Chilenski, and L. K. Rierson, "A practical tutorial on modified condition/decision coverage," NASA Langley Research Center, Tech. Rep., 2001.

[65] A. Rajan, M. W. Whalen, and M. P. E. Heimdahl, "The effect of program and model structure on MC/DC test adequacy coverage," in *ICSE*, W. Schäfer, M. B. Dwyer, and V. Gruhn, Eds., ACM, 2008, pp. 161–170.

[66] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *ICSE*, 2014.

[67] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *IEEE Computer*, vol. 11, no. 4, pp. 34–41, 1978.

[68] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, "Are mutants a valid substitute for real faults in software testing?" In *FSE*, 2014.

[69] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *POPL*, R. M. Graham, M. A. Harrison, and R. Sethi, Eds., ACM, 1977, pp. 238–252.

[70] ——, "Abstract interpretation frameworks," *J. Log. Comput.*, vol. 2, no. 4, pp. 511–547, 1992.

[71] N. D. Jones and F. Nielson, "Abstract interpretation: A semantics-based tool for program analysis," in *Handbook of Logic in Computer Science*, S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, Eds., vol. 4, Oxford, UK: Oxford University Press, 1995, pp. 527–636.

[72] D. Octeau, S. Jha, M. Dering, P. D. McDaniel, A. Bartel, L. Li, J. Klein, and Y. Le Traon, "Combining static analysis with probabilistic models to enable market-scale Android inter-component analysis," in *POPL*, R. Bodik and R. Majumdar, Eds., ACM, 2016, pp. 469–484.

[73] T. T. Pearse and P. W. Oman, "Maintainability measurements on industrial source code maintenance activities," in *ICSM*, IEEE Computer Society, 1995, pp. 295–303.

[74] Sun Microsystems, Inc. (1999). Code conventions for the Java programming language, [Online]. Available: `http://www.oracle.com/technetwork/java/javase/documentation/codeconventions-139411.html` (visited on Sep. 18, 2015).

[75] C. Boogerd and L. Moonen, "Assessing the value of coding standards: An empirical study," in *ICSM*, IEEE Computer Society, 2008, pp. 277–286.

[76] F. Rahman, D. Posnett, A. Hindle, E. T. Barr, and P. T. Devanbu, "Bugcache for inspections: Hit or miss?" In *SIGSOFT*, 2011, pp. 322–331.

[77] S. G. Elbaum and J. C. Munson, "Code churn: A measure for estimating the impact of code change," in *ICSM*, IEEE Computer Society, 1998, p. 24.

[78] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. J. Whitehead Jr., "Does bug prediction support human developers? findings from a google case study," in *ICSE*, 2013, pp. 372–381.

[79] M. Tan, L. Tan, S. Dara, and C. Mayeux, "Online defect prediction for imbalanced data," in *ICSE*, IEEE, 2015, pp. 99–108.

[80] E. J. Whitehead Jr., "Collaboration in software engineering: A roadmap," in *ICSE*, L. C. Briand and A. L. Wolf, Eds., IEEE Computer Society, 2007, pp. 214–225.

[81] C. Casalnuovo, B. Vasilescu, P. T. Devanbu, and V. Filkov, "Developer onboarding in GitHub: The role of prior social links and language experience," in *ESEC/FSE*, E. Di Nitto, M. Harman, and P. Heymans, Eds., ACM, 2015, pp. 817–828.

[82] B. Vasilescu, D. Posnett, B. Ray, M. G. J. van den Brand, A. Serebrenik, P. T. Devanbu, and V. Filkov, "Gender and tenure diversity in GitHub teams," in *CHI*, B. Begole, J. Kim, K. Inkpen, and W. Woo, Eds., ACM, 2015, pp. 3789–3798.

[83] D. Šmite, "Global software development projects in one of the biggest companies in Latvia: Is geographical distribution a problem?" *Software Process: Improvement and Practice*, vol. 11, no. 1, pp. 61–76, 2006.

[84] C. Bird, N. Nagappan, P. T. Devanbu, H. C. Gall, and B. Murphy, "Does distributed development affect software quality? an empirical case study of Windows Vista," *Commun. ACM*, vol. 52, no. 8, pp. 85–93, 2009.

[85] K. Beck and C. Andres, *Extreme Programming Explained: Embrace Change*, Second. Addison-Wesley Professional, 2004.

[86] M. Greiler, K. S. Herzig, and J. Czerwonka, "Code ownership and software quality: A replication study," in *MSR*, M. Di Penta, M. Pinzger, and R. Robbes, Eds., IEEE, 2015, pp. 2–12.

[87] K. Mordal-Manet, J. Laval, S. Ducasse, N. Anquetil, F. Balmas, F. Bellingard, L. Bouhier, P. Vaillergues, and T. J. McCabe Sr., "An empirical model for continuous and weighted metric aggregation," in *CSMR*, Mar. 2011, pp. 141–150.

[88] B. Vasilescu, A. Serebrenik, and M. G. J. van den Brand, "By no means: A study on aggregating software metrics," in *WETSoM*, G. Concas, E. D. Tempero, H. Zhang, and M. Di Penta, Eds., ACM, 2011, pp. 23–26.

[89] B. Vasilescu, "Analysis of advanced aggregation techniques for software metrics," Master's thesis, Eindhoven University of Technology, 2011.

[90] B. Vasilescu, A. Serebrenik, and M. G. J. van den Brand, "You can't control the unfamiliar: A study on the relations between aggregation techniques for software metrics," in *ICSM*, IEEE Computer Society, 2011, pp. 313–322.

[91] P. Jansen, R. L. Krikhaar, and F. Dijkstra. (2007). Towards a single software quality metric, [Online]. Available: `http://www.tiobe.com/content/paperinfo/DefinitionOfConfidenceFactor.html` (visited on Sep. 10, 2015).

[92] P. Oliveira, F. P. Lima, M. T. Valente, and A. Serebrenik, "RTTool: A tool for extracting relative thresholds for source code metrics," in *ICSME*, IEEE Computer Society, 2014, pp. 629–632.

[93] K. A. M. Ferreira, M. A. S. Bigonha, R. S. Bigonha, L. F. O. Mendes, and H. C. Almeida, "Identifying thresholds for object-oriented software metrics," *Journal of Systems and Software*, vol. 85, no. 2, pp. 244–257, 2012.

[94] Bullseye Testing Technology. (2012). BullseyeCoverage measurement technique, [Online]. Available: `http://www.bullseye.com/measurementTechnique.html` (visited on Feb. 18, 2016).

[95] froglogic. (2016). Squish coco, [Online]. Available: `http://www.froglogic.com/squish/coco/` (visited on Feb. 18, 2016).

[96] Testwell. (2015). Testwell CTC++ description, [Online]. Available: `http://www.testwell.fi/ctcdesc.html` (visited on Feb. 18, 2016).

[97] Linux Test Project. (2015). Coverage: Lcov, [Online]. Available: `http://ltp.sourceforge.net/coverage/lcov.php` (visited on Feb. 18, 2016).

[98] IBM. (2002). Collecting coverage data rational PureCoverage, [Online]. Available: `ftp://ftp.software.ibm.com/software/rational/docs/v2002/dev_tools/purecov/html/ht_ov_collecting.htm` (visited on Feb. 18, 2016).

[99] Parasoft. (2016). Code coverage analysis - parasoft, [Online]. Available: `https://www.parasoft.com/capability/code-coverage-analysis/` (visited on Feb. 18, 2016).

[100] Vector Software. (2016). Breaking down three different types of code coverage, [Online]. Available: `http://www.vectorcast.com/blog/2015/01/breaking-down-three-different-types-code-coverage` (visited on Feb. 18, 2016).

[101] NCover. (2016). Code central product features, [Online]. Available: `http://www.ncover.com/products/code-central/features/coverage` (visited on Feb. 18, 2016).

[102] S. Wilde. (2015). Code coverage: Opencover, [Online]. Available: `https://github.com/OpenCover/opencover/wiki/Code-Coverage` (visited on Feb. 18, 2016).

[103] Mountainminds GmbH & Co. KG and Contributors. (2009). Jacoco: Coverage counter, [Online]. Available: `http://eclemma.org/jacoco/trunk/doc/counters.html` (visited on Feb. 18, 2016).

[104] shakhal. (2014). Cobertura FAQ, [Online]. Available: `https://github.com/cobertura/cobertura/wiki/FAQ` (visited on Feb. 18, 2016).

[105] S. Cornett. (2011). What is wrong with statement coverage, [Online]. Available: `http://www.bullseye.com/statementCoverage.html` (visited on Dec. 7, 2015).

[106] E. N. Adams, "Optimizing preventive service of software products," *IBM Journal of Research and Development*, vol. 28, no. 1, pp. 2–14, 1984.

[107] R. M. Karp and M. O. Rabin, "Efficient randomized pattern-matching algorithms," *IBM Journal of Research and Development*, vol. 31, no. 2, pp. 249–260, 1987.

[108] Google. (2014). Google Java style, [Online]. Available: `https://google.github.io/styleguide/javaguide.html` (visited on Mar. 7, 2016).

[109] TIOBE Software. (2016). TIOBE Quality Indicator change history, [Online]. Available: `http://www.tiobe.com/tqi/changes` (visited on Feb. 23, 2016).

[110] K. S. Herzig and A. Zeller, "Mining your own evidence," in *Making Software: What Really Works, and Why We Believe It*, A. Oram and G. Wilson, Eds., Sebastopol, Calif., USA: O'Reilly Media, 2010, pp. 517–529.

[111] L. Prechelt and A. Pepper, "Why software repositories are not used for defect-insertion circumstance analysis more often: A case study," *Information & Software Technology*, vol. 56, no. 10, pp. 1377–1389, 2014.

[112] K. S. Herzig, S. Just, and A. Zeller, "It's not a bug, it's a feature: How misclassification impacts bug prediction," in *ICSE*, 2013, pp. 392–401.

[113] C. Bird, A. Bachmann, E. Aune, J. Duffy, A. Bernstein, V. Filkov, and P. T. Devanbu, "Fair and balanced?: Bias in bug-fix datasets," in *ACM SIGSOFT*, H. van Vliet and V. Issarny, Eds., ACM, 2009, pp. 121–130.

[114] R. Premraj and T. Zimmermann, "The art of collecting bug reports," in *Making Software: What Really Works, and Why We Believe It*, A. Oram and G. Wilson, Eds., Sebastopol, Calif., USA: O'Reilly Media, 2010, pp. 435–451.

[115] A. Alipour, A. Hindle, and E. Stroulia, "A contextual approach towards more accurate duplicate bug report detection," in *MSR*, T. Zimmermann, M. Di Penta, and S. Kim, Eds., IEEE Computer Society, 2013, pp. 183–192.

[116] A. Lazar, S. Ritchey, and B. Sharif, "Improving the accuracy of duplicate bug report detection using textual similarity measures," in *MSR*, P. T. Devanbu, S. Kim, and M. Pinzger, Eds., ACM, 2014, pp. 308–311.

[117] Food and Drug Administration, "General principles of software validation; final guidance for industry and FDA staff," FDA, Tech. Rep., 2012.

[118] C. Bird, P. C. Rigby, E. T. Barr, D. J. Hamilton, D. M. Germán, and P. T. Devanbu, "The promises and perils of mining git," in *MSR*, M. W. Godfrey and J. Whitehead, Eds., IEEE Computer Society, 2009, pp. 1–10.

[119] M. Brandtner, P. Leitner, and H. C. Gall, "Intent, tests, and release dependencies: Pragmatic recipes for source code integration," in *SCAM*, IEEE Computer Society, Sep. 2015, pp. 11–20.

[120] A. Bachmann, C. Bird, F. Rahman, P. T. Devanbu, and A. Bernstein, "The missing links: Bugs and bug-fix commits," in *FSE*, G. Roman and K. J. Sullivan, Eds., ACM, 2010, pp. 97–106.

[121] G. Schermann, M. Brandtner, S. Panichella, P. Leitner, and H. C. Gall, "Discovering loners and phantoms in commit and issue data," in *ICPC*, A. De Lucia, C. Bird, and R. Oliveto, Eds., ACM, 2015, pp. 4–14.

[122] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, A. Ihara, and K. Matsumoto, "The impact of mislabelling on the performance and interpretation of defect prediction models," in *ICSE*, IEEE, 2015, pp. 812–823.

[123] M. Wedel, U. Jensen, and P. Göhner, "Mining software code repositories and bug databases using survival analysis models," in *ESEM*, H. D. Rombach, S. G. Elbaum, and J. Münch, Eds., ACM, 2008, pp. 282–284.

[124] H. Zhang, L. Gong, and S. Versteeg, "Predicting bug-fixing time: An empirical study of commercial software projects," in *ICSE*, D. Notkin, B. H. C. Cheng, and K. Pohl, Eds., IEEE, 2013, pp. 1042–1051.

[125] D. Wang, H. Zhang, R. Liu, M. Lin, W. Wu, and H. Hu, "Predicting bugs' components via mining bug reports," *CoRR*, vol. abs/1010.4092, 2010.

[126] J. Śliwerski, T. Zimmermann, and A. Zeller, "When do changes induce fixes?" In *MSR*, 2005.

[127] S. Kim, T. Zimmermann, K. Pan, and E. J. Whitehead Jr., "Automatic identification of bug-introducing changes," in *ASE*, IEEE Computer Society, 2006, pp. 81–90.

[128] R. Wu, H. Zhang, S. Kim, and S. Cheung, "Relink: Recovering links between bugs and changes," in *ESEC/FSE*, T. Gyimóthy and A. Zeller, Eds., ACM, 2011, pp. 15–25.

[129] T. Zimmermann, N. Nagappan, H. C. Gall, E. Giger, and B. Murphy, "Cross-project defect prediction: A large scale experiment on data vs. domain vs. process," in *ESEC/FSE*, Amsterdam, The Netherlands: ACM, 2009, pp. 91–100.

[130] S. Kim, H. Zhang, R. Wu, and L. Gong, "Dealing with noise in defect prediction," in *ICSE*, R. N. Taylor, H. C. Gall, and N. Medvidovic, Eds., ACM, 2011, pp. 481–490.

[131] Ö. Sari and O. Kalipsiz, "Bug prediction for an ATM monitoring software - use of logistic regression analysis for bug prediction," in *ICEIS*, 2015, pp. 382–387.

[132] 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," *Empirical Software Engineering*, vol. 13, no. 5, pp. 539–559, 2008.

[133] K. Kevrekidis and J. Udo, "A regression analysis on the complexity of software components and the number of software faults of a navigation software application," TomTom International B.V., Tech. Rep., 2014.

[134] M. Ridout, C. G. B. Demétrio, and J. Hinde, "Models for count data with many zeros," in *IBC*, vol. 19, 1998, pp. 179–192.

[135] A. Pepper, "Analyse von Defekten in einem großen Softwaresystem durch die Auswertung von Versionsverwaltungssystemen," Master's thesis, Freie Universität Berlin, 2011.

[136] W. Wang. (2015). Explain Git with D3, [Online]. Available: `http://onlywei.github.io/explain-git-with-d3/` (visited on Mar. 9, 2016).

[137] D. M. Germán, B. Adams, and A. E. Hassan, "Continuously mining distributed version control systems: An empirical study of how linux uses git," *Empirical Software Engineering*, pp. 260–299, 2015.

[138] R. Moser, W. Pedrycz, and G. Succi, "A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction," in *ICSE*, W. Schäfer, M. B. Dwyer, and V. Gruhn, Eds., ACM, 2008, pp. 181–190.

[139] G. Denaro and M. Pezzè, "An empirical evaluation of fault-proneness models," in *ICSE*, W. Tracz, M. Young, and J. Magee, Eds., ACM, 2002, pp. 241–251.

[140] C. Gerpheide, *Bug prediction at box*, Honors project at Eindhoven University of Technology, 2015.

[141] T. Menzies, A. Butcher, A. Marcus, T. Zimmermann, and D. R. Cok, "Local vs. global models for effort estimation and defect prediction," in *ASE*, P. Alexander, C. S. Pasareanu, and J. G. Hosking, Eds., IEEE Computer Society, 2011, pp. 343–351.

[142] M. Nagappan, T. Zimmermann, and C. Bird, "Diversity in software engineering research," in *ACM SIGSOFT*, B. Meyer, L. Baresi, and M. Mezini, Eds., Saint Petersburg, Russia: ACM, Aug. 2013, pp. 466–476.

[143] K. Petersen and C. Wohlin, "Context in industrial software engineering research," in *ESEM*, ACM / IEEE Computer Society, 2009, pp. 401–404.

[144] A. Myrvold, "Data analysis for software metrics," *Journal of Systems and Software*, vol. 12, no. 3, pp. 271–275, 1990.

[145] E. Bouwers, J. Visser, and A. van Deursen, "Getting what you measure," *Commun. ACM*, vol. 55, no. 7, pp. 54–59, 2012.

[146] M. Tufano, F. Palomba, G. Bavota, R. Oliveto, M. Di Penta, A. De Lucia, and D. Poshyvanyk, "When and why your code starts to smell bad," in *ICSE*, IEEE, 2015, pp. 403–414.

[147] S. S. Shapiro and M. B. Wilk, "An analysis of variance test for normality (complete samples)," *Biometrika*, vol. 52, no. 3/4, pp. 591–611, 1965.

[148] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics Bulletin*, vol. 1, no. 6, pp. 80–83, 1945.

[149] T. Dybå, V. B. Kampenes, and D. I. K. Sjøberg, "A systematic review of statistical power in software engineering experiments," *Information & Software Technology*, vol. 48, no. 8, pp. 745–755, 2006.

[150] N. J. Salkind, *Encyclopedia of Measurement and Statistics*. SAGE Publications, 2006.

[151] Y. Benjamini and Y. Hochberg, "Controlling the false discovery rate: A practical and powerful approach to multiple testing," *Journal of the Royal Statistical Society*, vol. 29, no. 4, pp. 289–300, 1995.

[152] D. E. Perry, A. A. Porter, and L. G. Votta, "Empirical studies of software engineering: A roadmap," in *ICSE*, A. Finkelstein, Ed., ACM, 2000, pp. 345–355.

[153] W. Kruskal, "Miracles and statistics: The casual assumption of independence," *Journal of the American Statistical Association*, vol. 83, no. 404, pp. 929–940, 1988.

[154] R. J. Serfling, "The Wilcoxon two-sample statistic on strongly mixing processes," *The Annals of Mathematical Statistics*, vol. 39, no. 4, pp. 1202–1209, 1968.

[155] K. Kluza, M. Baran, S. Bobek, and G. J. Nalepa, "Overview of recommendation techniques in business process modeling," in *KESE*, G. J. Nalepa and J. Baumeister, Eds., ser. CEUR Workshop Proceedings, vol. 1070, CEUR-WS.org, 2013.

[156] O. Kononenko, O. Baysal, L. Guerrouj, Y. Cao, and M. W. Godfrey, "Investigating code review quality: Do people and participation matter?" In *ICSME*, IEEE Computer Society, 2015, pp. 111–120.

[157] ISO/IEC, "ISO/IEC/IEEE 24765 — systems and software engineering — vocabulary," International Organization for Standardization, Tech. Rep., 2010.

[158] J. W. Hunt and M. D. McIlroy, "An algorithm for differential file comparison," Bell Laboratories, Tech. Rep., 1976.

[159] J. S. Haikin, *Version control system for software code*, US Patent 6,757,893, Jun. 2004.