This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2016.2622264, IEEE Transactions on Software Engineering

Preventing Defects: The Impact of Requirements Traceability Completeness on Software Quality

Patrick Rempel and Parick Mäder
Technische Universität Ilmenau
Software Engineering for Safety-Critical Systems Group
patrick.rempel@tu-ilmenau.de and patrick.maeder@tu-ilmenau.de

Abstract—Requirements traceability has long been recognized as an important quality of a well-engineered system. Among stakeholders, traceability is often unpopular due to the unclear benefits. In fact, little evidence exists regarding the expected traceability benefits. There is a need for empirical work that studies the effect of traceability. In this paper, we focus on the four main requirements implementation supporting activities that utilize traceability. For each activity, we propose generalized traceability completeness measures. In a defined process, we selected 24 medium to large-scale open-source projects. For each software project, we quantified the degree to which a studied development activity was enabled by existing traceability with the proposed measures. We analyzed that data in a multi-level Poisson regression analysis. We found that the degree of traceability completeness for three of the studied activities significantly affects software quality, which we quantified as defect rate. Our results provide for the first time empirical evidence that more complete traceability decreases the expected defect rate in the developed software. The strong impact of traceability completeness on the defect rate suggests that traceability is of great practical value for any kind of software development project, even if traceability is not mandated by a standard or regulation.

Index Terms—requirements traceability; traceability completeness; traceability metrics; change impact analysis; requirements satisfaction analysis; source code justification analysis; software quality; error proneness; defects; bugs; empirical validation; regression analysis

1 Introduction

Requirements traceability is broadly recognized as a critical element of any rigorous software development process [12], but especially for building high-assurance and safety-critical software systems [65]. It is defined as the "ability to follow the life of a requirement in both a backward and a forward direction" [38]. Finkelstein argues that "[requirements traceability] provides the information management support for [..] complex multi-threaded customer relationships and the technical substrate for rapid system evolution" [30]. The resulting network of interrelated software artifacts can be used to "answer questions of both the software product and its development process" [11].

However, the implementation of requirements traceability leads to increased development effort and documentation workload, which can only be compensated by higher quality [16, 3, 14, 77] and reduced overall costs if traceability is applied purposefully [99]. Cleland-Huang et al. assume for a traceability cost model that the manual creation of a trace link among two artifacts takes on average 15 minutes [13]. Depending on the artifact types, the average creation times can vary between 10 and 45 minutes [47]. This often leads to a traceability benefit problem [52]. Among stakeholders, requirements traceability is often unpopular due to the extra effort it creates and the unclear benefits and beneficiaries [4, 78, 79]. In fact, there is little hard evidence regarding the benefits to expect from creating software traceability.

Therefore, we previously conducted a controlled experiment in which we studied the effect of traceability on subjects performing software development tasks [75, 76]. We found that subjects that had traceability available for a task, performed on average 24% faster and created 50% more accurate solutions than the subjects that performed tasks without traceability. These results are very encouraging and show expectable benefits, especially regarding effort reduction and the output accuracy, on a discrete task basis. However, the experiment poses three limitations with respect to the generalizability of results. First, subjects were using complete and correct traces, which is rarely the case in industry. Second, subjects were no project stakeholders and therefore not familiar with the project. Third, subjects submitted their solution to a task, but had no chance to incrementally improve it through debugging.

These limitations make it hard to generalize results beyond a single task perspective and to determine an aggregated effect for a whole development project. Furthermore, the observed higher correctness achieved by subjects working with traceability raises an interesting research question which the preceding experiments cannot answer. Does the capturing and usage of requirements traceability within a development project lead to a software product of higher quality? To answer this question, we conducted a large case study assessing the effect of requirements traceability in supporting implementation tasks for 24 medium- to large-scale open-source development projects. In preparation for this study, we derived a description model from the existing body of knowledge that characterizes use cases of traceability in the context of software development. Based on this description model, we further derive a set of traceability completeness measures, where each measure quantifies

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2016.2622264, IEEE Transactions on Software Engineering

missing trace links with respect to the necessities of a traceability use case. We found a strong correlation between the degree to which traceability was available in support of different implementation scenarios and software quality, which we quantified as defect rate. Our study presents for the first time strong evidence that purposed traceability in fact raises the quality of implemented source code.

The remainder of the paper is organized as follows. We formulate the research questions for our study in Section 2. In Section 3, we develop a preliminary description model for our study by characterizing the requirements implementation process and deriving traceability use cases, which are asserted to be beneficial for the implementation process. Section 4 introduces the design of our study and the applied data collection procedure. In Section 5, we describe how we statistically analyzed the collected data. The statistical analysis results are presented in Section 6. We discuss these results in Section 7 with regard to our formulated research questions. In Section 8, we discuss potential threats to the validity of the discovered results and how we mitigated them. In Section 9, we discuss work related to our study and finally, in Section 10 we draw conclusions and outline future research directions.

2 Research Questions

Software quality has been the subject of numerous empirical research in the past (e.g., [31, 88, 87, 42, 68]), leading to a large body of empirical knowledge on that topic. Thereby, a variety of influential factors on software quality has been studied, such as requirements attributes [31, 104], organizational attributes of software development projects [88, 10], or software complexity metrics [42, 87, 119, 9].

The work presented in this paper focuses on the completeness of requirements traceability in software projects and how it impacts the implementation quality of requirements. Requirements traceability is broadly recognized as a critical factor for software quality [12]. Researchers argue that requirements traceability supports software engineers in understanding complex systems as well as to account for the consequences of ongoing changes [30]. Practitioners argue that traceability can help to determine completion, is essential to analyze the impact of changes, and provides confidence that the requirements are satisfied [110]. Additionally, software safety standards, such as DO-178C [111], ISO-26262-6 [54], and IEC-61508-3 [51] and software maturity standards, such as CMMI [17] and SPICE [55] explicitly mandate requirements traceability.

Although requirements traceability is a commonly accepted quality of rigorous software development processes [12], empirical evidence on the impact of traceability completeness on the software product quality is lacking. In an attempt to close this gap, we studied the effect of requirements traceability availability on the completing time and correctness of specific development tasks in a controlled experiment [75, 76]. As discussed in the previous section, the experiment investigated two extreme cases only, where traceability is either missing completely or a complete set of trace links is provided. The results of recent empirical traceability studies [80, 106, 109, 78, 70, 69] revealed that these two extreme cases do not reflect current software engineering practice. All 30 analyzed projects used requirements traceability to support various software engineering activities, but failed to implement complete sets of trace links with respect to the traceability usage goals [106, 109, 103]. The degree of completeness greatly varied among the analyzed projects. All these empirical results led to the hypothesis that the degree of requirements traceability completeness is a software implementation quality driver, because important software engineering activities for the assurance of implementation quality depend on requirements traceability. Based on this hypothesis, we derived the following three research questions:

RQ-1: Does the degree to which existing artifacts are traceable (traceability completeness) with respect to the implementation supporting activities high-level impact analysis, low-level impact analysis, requirements satisfaction analysis, and source code justification analysis affect software quality?

RQ-2: Does the effect investigated in RQ-1, if any exists, vary for the four different implementation supporting activities?

RQ-3: What are the orientation (negative vs. positive) and the intensity of the effect investigated in RQ-1, if any exists?

In projects with large development teams, software engineers often need to understand relationships between artifacts that were created or maintained by other software engineers. Hence, cooperative development requires communication between the involved contributors [48]. Requirements traceability can be used to communicate artifact relationships among contributors. We hypothesize that traceability is more beneficial for development projects with a higher amount of distinct requirements engineers or developers. The need to perform quality assurance activities re-occurs with every software change. Since people's memories typically fade over time [24], we hypothesize that captured traceability information is particularly beneficial if time spans between requirement changes or code changes are longer. Based on these hypotheses, we derived the following two research questions:

RQ-4: Is the effect investigated in RQ-1, if any exists, additionally impacted by the development team size?

RQ-5: Is the effect investigated in RQ-1, if any exists, additionally impacted by the time span between software changes?

3 Description Model

In this section, we derive a description model from the existing body of knowledge on requirements traceability that characterizes major traceability use cases in the context of software development. Researchers found through case studies that the benefits of available requirements traceability depend on the intended usage [27]. Different intended traceability usage scenarios may require different artifacts to be related [106]. For example, the demonstration of regulatory compliance requires traceability between regulatory codes and requirements, while in contrast, the demonstration of implementation completeness requires traceability

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2016.2622264, IEEE Transactions on Software Engineering

between requirements and source code. In the following three subsections, we first derive two general implementation scenarios during software development (Section 3.1), we then identify four common traceability use cases that we discuss in the context of the derived implementation scenarios (Section 3.2), and finally we discuss the concept of requirements traceability completeness with respect to the four common traceability use cases (Section 3.3).

3.1 Software Implementation

In this paper, the term implementation refers to the process of developing source code to realize new or changed software requirements. This realization process typically involves intermediate artifacts. Gunter et al. [46] proposed a "reference model for requirements and specifications" and distinguish three general artifact types: *requirements specification*, *design specification*, and *source code*. We denote R as a set of requirements specifications that explicitly describe what customers expect from a software system, D as a set of design specifications that contain explicit instructions on how to build a software system in order to satisfy R, and S as a set of source codes that implement D in order to build the software system. As described by Rajlich and Bennett [98], the software development life-cycle spans three different stages: *initial development*, *evolution*, and *servicing*. While in the *initial development* stage, only new requirements are discovered and implemented, in the two later stages (*evolution* and *servicing*) new requirements may still be discovered and implemented but also existing requirements are changed. Changes can be triggered, for example, by the customer, by changes in applied libraries or technologies or by discovered issues in the exiting implementation. We denote C as a set of requirement change specifications that describe how a software system is supposed to be changed to meet newly emerged, shifted, or misunderstood customers' expectations. For the further discussion of implementation scenarios and traceability use cases, we refer to C as a separate artifact that is related via trace link to the initial requirement. Different tools implement this change concept in different ways (e.g., as new version of the requirement or as a separate change request artifact). However, the implication remain the same, independently of how the concept is implemented. In Figure 1, all described artifact sets are shown as Venn diagram, each set contains illustrating example artifacts and is mapped to corresponding implementation phases and software life-cycle stages. In general, two implementation scenarios can be distinguished:

SC-1: implementing a new requirement

SC-2: implementing a requirements change

We will refer back to these two generic implementation scenarios when we derive and discuss the four major traceability use cases in their context within the next subsection.

3.2 Traceability Use Cases in the Context of Software Implementation

A necessary precondition to achieve requirements traceability is the explicit creation of trace links [39]. As depicted in Figure 1, vertical and horizontal traceability relationships are distinguished in the literature [97, 71, 94]. While vertical traceability refers to dependencies among artifacts that are part of a single work product within the software development process (e.g., a dependency between two requirements artifacts), horizontal traceability refers to relationships between artifacts that are part of different work products (e.g., a source code artifact implementing a design artifact).

The usage of traceability refers to the activity of following trace links from a source artifact to a target artifact [39] to achieve an explicit goal in a development project [95]. Rierson [110] argues that using traceability provides two main benefits for stakeholders. First, it is essential to change impact analysis, and second, it helps to determine completion. Thereby, impact analysis refers to identifying the consequences of implementing a new or a changed requirement [7]. Completion analysis refers to resolving either the traceability from requirements to their implementation or vice versa and allows determining whether or not all the specification and the implementation are complete [99]. In a requirements traceability case study, Ramesh et al. [99] found that traceability usage practices vary among different project roles. While project managers and requirements engineers are concerned with the change impact on and the completeness of requirements artifacts, system engineers and developers are concerned with the change impact on and the completeness of source code artifacts. Empirical studies on traceability usage substantiated these findings [100, 4, 8, 74, 26, 91, 92, 28, 89, 22].

In summary, analyzing change impact and determining completeness are the most common traceability use cases in practice. Traceability users are either concerned with requirements or source code artifacts. Hence, we derived the following traceability use cases that are relevant for the implementation of new requirements (SC-1) and for the implementation of changed requirements (SC-2):

UC-1: Analyzing the impact of a new or changed requirement on dependent requirements (*High-level impact analysis*).

UC-2: Analyzing the impact of a new or changed requirement on dependent code artifacts (*Low-level impact analysis*).

UC-3: Analyzing if a new or changed requirement is satisfied by code artifacts (*Requirements satisfaction analysis*).

UC-4: Analyzing if a new or changed code artifact is justified by a requirement (*Source code justification analysis*).

In the following subsections, we discuss these four traceability use cases and refer especially to three characteristics: the *source artifacts* on which the analysis is applied, the *trace link paths* that are followed to conduct the analysis, and the *destination artifacts* to which trace link paths are resolved. Additionally, we provide illustrating examples for each discussed use case.

3.2.1 UC-1: High-level impact analysis

The first traceability use case (UC-1) is concerned with the analysis of effects that a new or a changed requirement has on its dependent requirements artifacts. Vertical traceability relations from a new or changed requirement to dependent requirements or requirement changes are to be resolved by the stakeholder in order to identify what other requirements

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2016.2622264, IEEE Transactions on Software Engineering

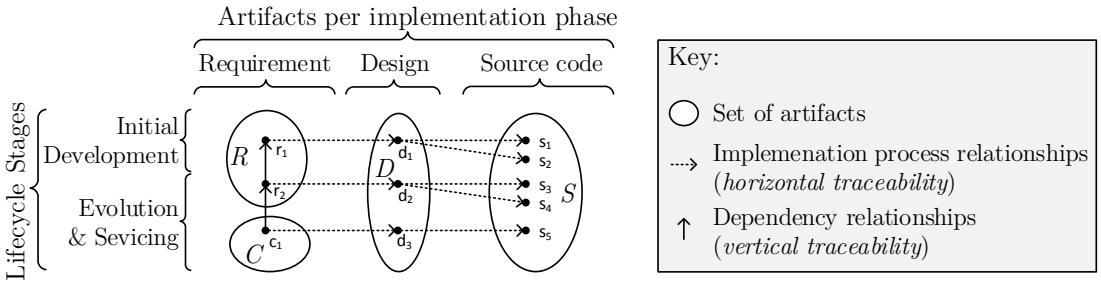
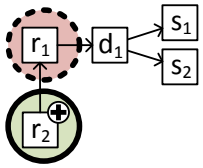
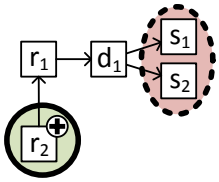
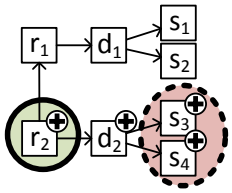
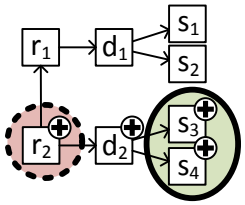
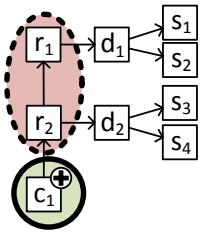
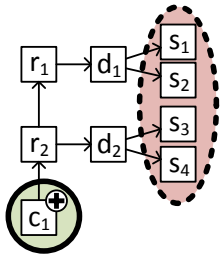
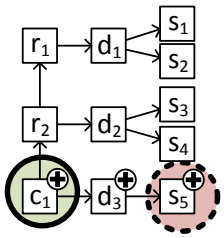
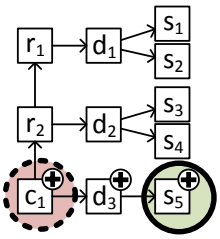


Fig. 1. Artifacts and their traceability across the software life-cycle stages: initial development, evolution, and servicing (adapted from [46, 98])

TABLE 1
Overview of the four traceability use cases (UC-1, UC-2, UC-3, and UC-4) for each implementation scenario (SC-1 and SC-2)

	Traceability Use Cases			
	Conducted throughout the entire lifecycle		Conducted after a requirement was implemented	
	UC-1: High-level impact analysis	UC-2: Low-level impact analysis	UC-3: Requirements satisfaction analysis	UC-4: Source code justification analysis
SC-1: Implementing a new requirement				
SC-2: Implementing a requirements change				

are potentially impacted. Thus, we denote UC-1 as "High-level impact analysis". The *source artifacts* for UC-1 depend on the implementation scenario. For SC-1 (implementing a new requirement), the source artifact is an element of R. Accordingly, for SC-2 (implementing a changed requirement) the source artifact is an element of C. The *destination artifacts* are dependent requirement artifacts, which are potentially impacted by the new or changed requirement. For both implementation scenarios, the destination artifacts are elements of $R \cup C$. The *trace link paths* between source and destination artifacts consists of vertical trace links across requirement artifacts only. Thus, trace paths that support UC-1 can consist of any of the following trace links in any sequence: $r \rightarrow r$, $r \rightarrow c$, $c \rightarrow c$, and $c \rightarrow r$, where $r \in R$ and $c \in C$. Figure 2 defines all possible trace link paths for UC-1 in a syntax graph like notation.

Examples: The first column in Table 1 shows examples of UC-1 applied in the scenarios SC-1 and SC-2. In the example that illustrates SC-1, the set of source artifacts consists of the newly created requirement r_2 and the set of target artifacts consists of r_1, which is the only dependent requirement of r_2. The artifacts r_2 and r_1 are connected through the trace link path $r_2 \rightarrow r_1$. In the example that illustrates SC-2, the set of source artifacts consists of a changed requirement c_1 and the set of target artifacts consists of r_1 and r_2 which are the dependent requirements of c_1. While c_1 and r_2 are connected through the trace link path $c_1 \rightarrow r_2$, c_1 and r_1 are connected through the trace link path $c_1 \rightarrow r_2 \rightarrow r_1$.

3.2.2 UC-2: Low-level impact analysis

The second traceability use case (UC-2) is concerned with the analysis of effects that a new or a changed requirement has on its dependent source code artifacts. Therefore, not only vertical but also horizontal traceability relations from the new or changed requirement through dependent

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2016.2622264, IEEE Transactions on Software Engineering

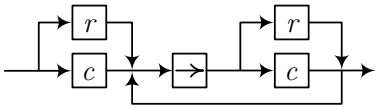


Fig. 2. Syntax graph of possible trace link paths supporting UC-1 (the box with an arrow inside represents a trace link)

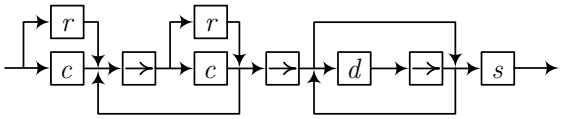


Fig. 3. Syntax graph of possible trace link paths supporting UC-2 (the box with an arrow inside represents a trace link)

requirement artifacts to source code artifacts that implement the dependent requirement artifacts are to be resolved by the stakeholder in order to identify what source code artifacts are potentially impacted. Thus, we denote UC-2 as "Low-level impact analysis". The *source artifacts* for UC-2 depend on the implementation scenario. For SC-1, the source artifact is an element of R, while for SC-2 the source artifact is an element of C. The *destination artifacts* are dependent artifacts at source code level, which are potentially impacted by the new or changed requirement. For both implementation scenarios, the destination artifacts are elements of S. The *trace link paths* between source and destination artifacts consist of two parts. Analogous to UC-1, the first part consists of any of the following trace links in any sequence: $r \rightarrow r$, $r \rightarrow c$, $c \rightarrow c$, and $c \rightarrow r$, where $r \in R$ and $c \in C$. The second part consists of a trace link path between a requirement and a source code artifact with zero to many intermediate design artifacts: $r[\rightarrow d]^* \rightarrow s$ or $c[\rightarrow d]^* \rightarrow s$, where $d \in D$, $s \in S$, $r \in R$, and $c \in C$. Figure 3 provides a generalized definition of all possible trace link paths that support UC-2 in a syntax graph like notation.

Examples: The second column in Table 1 shows examples of UC-2 applied in the scenarios SC-1 and SC-2. In the example that illustrates SC-1, the set of source artifacts consists of the newly created requirement r_2 and the set of target artifacts consists of s_1 and s_2, which are the source codes that implement the requirement r_1, which is a dependent requirement of r_2. The artifacts r_2 and s_1 are connected through the trace link path $r_2 \rightarrow r_1 \rightarrow d_1 \rightarrow s_1$ and the artifacts r_2 and s_2 are connected through the trace link path $r_2 \rightarrow r_1 \rightarrow d_1 \rightarrow s_2$. In the example that illustrates *SC-2*, the set of source artifacts consists of the changed requirement c_1 and the set of target artifacts consists of s_1, s_2, s_3, and s_4, which are the source codes that implement the dependent requirements r_1 and r_2. The following trace paths exists: c_1 and s_1 are connected through $c_1 \rightarrow r_2 \rightarrow r_1 \rightarrow d_1 \rightarrow s_1$, c_1 and s_2 are connected through $c_1 \rightarrow r_2 \rightarrow r_1 \rightarrow d_1 \rightarrow s_2$, c_1 and s_3 are connected through $c_1 \rightarrow r_2 \rightarrow d_2 \rightarrow s_3$, and c_1 and s_4 are connected through $c_1 \rightarrow r_2 \rightarrow d_2 \rightarrow s_4$.

3.2.3 UC-3: Requirements satisfaction analysis

The third traceability use case (UC-3) is concerned with completeness determination for new or changed requirements. Therefore, horizontal traceability is resolved by the stakeholder to follow the implementation process subsequently from its originating requirement to its final result

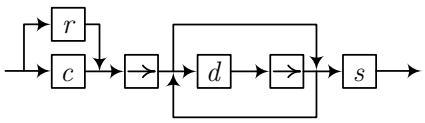


Fig. 4. Syntax graph of possible trace link paths supporting UC-3 (the box with an arrow inside represents a trace link)

(source code) in order to identify whether or not all stated requirements are satisfied by source code. We denote UC-3 as "Requirements satisfaction analysis". The *source artifacts* depend on the implementation scenario. For SC-1, the source artifact is an element of R, while for SC-2 the source artifact is an element of C. The *destination artifacts* are the artifacts at source code level that implement the new or changed requirement. Thus, for both implementation scenarios, the destination artifacts are elements of S. The *trace link paths* between source and destination artifacts consist of horizontal trace links only. Each trace link path of UC-3 connects a requirement with a source code artifact through zero to many intermediate design artifacts: $r[\rightarrow d]^* \rightarrow s$ or $c[\rightarrow d]^* \rightarrow s$, where $d \in D$, $s \in S$, $r \in R$, and $c \in C$. Figure 4 provides a generalized definition of all possible trace link paths supporting UC-3 in a syntax graph like notation.

Examples: The third column in Table 1 shows examples of UC-3 applied in the scenarios SC-1 and SC-2. In the example that illustrates SC-1, the set of source artifacts consists of the newly created requirement r_2 and the set of target artifacts consists of s_3 and s_4, which are the source code artifacts that implement r_2. While r_2 and s_3 are connected through the trace link path $r_2 \rightarrow d_2 \rightarrow s_3$, the artifacts r_2 and s_4 are connected through the trace link path $r_2 \rightarrow d_2 \rightarrow s_4$. In the example that illustrates SC-2, the set of source artifacts consists of the changed requirement c_1 and the set of target artifacts consists of s_5, which is the source code artifact that implements c_1. The artifacts c_1 and s_5 are connected through $c_1 \rightarrow d_3 \rightarrow s_5$.

3.2.4 UC-4: Source code justification analysis

The fourth traceability use case (UC-4) is concerned with determining whether or not each source code artifact satisfies at least one requirement. Therefore, the stakeholder resolves horizontal traceability in reverse direction, from the source code to its originating requirements. If a source code artifact is not traceable to at least one originating requirement, the implemented software may contain unintended functionality, which is not justified by any requirement. We denote UC-4 as "Source code justification analysis". The *source artifacts* are the newly created artifacts at implementation level (S). The *destination artifacts* depend on the supported implementation scenario and are either elements of R for SC-1 or elements of C for SC-2. The *trace link paths* between source and destination artifacts consist of horizontal trace links only. Each trace link path of UC-4 connects a source code artifact with a requirement artifact through zero to many intermediate design artifacts: $s[\rightarrow d]^* \rightarrow r$ or $s[\rightarrow d]^* \rightarrow c$, where $d \in D$, $s \in S$, $r \in R$, and $c \in C$. Figure 5 provides a generalized definition of all possible trace link paths supporting UC-4 in a syntax graph like notation.

Examples: The fourth column in Table 1 shows examples of UC-4 applied in the scenarios SC-1 and SC-2. In the

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2016.2622264, IEEE Transactions on Software Engineering

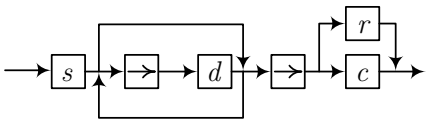


Fig. 5. Syntax graph of possible trace link paths supporting UC-4 (the box with an arrow inside represents a trace link)

example that illustrates SC-1, the set of source artifacts consists of the newly created source code artifacts s_3 and s_4 and the set of target artifacts consists of r_2, which is the justification for the source code artifacts s_3 and s_4. While the artifacts s_3 and r_2 are connected through the reverse trace link path $s_3 \rightarrow d_2 \rightarrow r_2$, the artifacts s_4 and r_2 are connected through the reverse trace link path $s_4 \rightarrow d_2 \rightarrow r_2$. In the example that illustrates scenario SC-2, the set of source artifacts consists of the newly created source code s_5 and the set of target artifacts consists of c_1, which is the justification of s_5. The artifacts s_5 and c_1 are connected through the reverse trace link path $s_5 \rightarrow d_3 \rightarrow c_1$.

3.3 Requirements Traceability Completeness

As stated in research question *RQ-1*, the main goal of this work was to study if the degree of requirements traceability completeness with respect to implementation supporting activities depending on traceability influences software quality. In Section 3.2, we characterized these implementation supporting activities as traceability use cases. Each traceability use case and the underlying analysis require different trace link paths to be executable. A traceability use case can be performed for each element within the set of *source artifacts*. Hence, each artifact that is an element of the *source artifact* set of a traceability use case must be traceable to one or many artifacts that are an element of the *destination artifacts* set of this use case through a path of trace links that conform to the use case specific *trace link path* definition (see Figures 2 – 5). The number of source artifacts that misses traceability with respect to the concerned traceability use case determine the degree of traceability completeness per entity of analysis. The more artifacts with missing traceability the lower is the traceability completeness as required by the traceability use case and the underlying analysis. In Section 4.3, we provide the exact definitions to measure the traceability completeness for each of the four identified traceability use cases. We consider the proposed measures a suitable approximation for quantifying traceability completeness with respect to a traceability use case. However, since the proposed measures solely quantify the degree of non-traced artifacts, our measurement approach implies the risk that in cases where one source artifact is related to multiple target artifacts, missing traces cannot be recognized and the source artifacts may be incorrectly considered as complete. For example, if a project member is creating traces from one requirement to multiple source code artifacts and accidentally misses one of the code artifacts. Thus, the validity of the proposed measures depends on the quality of the created trace links. We provide a critical discussion of the studied cases' traceability quality and a more thorough discussion of trace quality threats in Section 8.1.

4 Study Design

To investigate our research questions (see Section 2), an in-depth analysis of software development artifacts and their traceability is required. For this study, we focused on open-source projects due to the public availability of the required artifacts and traceability information. We collected development artifacts and traceability data from 24 open-source software projects with industrial relevance. In this section, we describe how we selected appropriate projects, the projects' characteristics, and the process we implemented to collect the data for our analysis.

4.1 Case Selection Strategy

We considered projects from five publicly available open-source repositories: Apache [120], JBoss [59], Codehaus [18], SourceForge [118], and GoogleCode [37] as potential cases for our study. We defined the following *criteria* for a case to be included in our study. The case:

1) must capture the development artifacts (requirements, requirement changes, defect descriptions, and source code) that are part of the two implementation scenarios SC-1 and SC-2 (see also Section 3.1),
2) must capture horizontal and vertical requirements traceability that is required to perform the traceability use cases UC-1 to UC-4 (see also Section 3.2),
3) must be under active development for at least three years, and must have evolved over at least five stable releases to ensure that the analyzed project is mature.

The following search strategy enforced these criteria.

First, we extracted a list of available projects from each repository ordered by project popularity. We determined the projects' popularity by the number of downloads.. We iterated over each list and manually checked whether or not the project is under active development, the project history is long enough, the project's issue tracker contained requirement artifacts, requirement changes, unique artifact identifiers, as well as trace links between requirements artifacts (see first, second, and third case inclusion criterion). If none of the first 20 projects' issue trackers fulfilled all requirements, we randomly selected 20 additional projects from the list and repeated our manual check. If none of these 40 projects' issue trackers fulfilled all requirements, we skipped the search for projects in this repository (GoogleCode and SourceForge).

Second, we checked whether or not the project's Software Configuration Management (SCM) system comprised commit logs containing references to artifact identifiers of the project's issue tracker (see first and second case inclusion criterion). At this stage, we identified 17 that fulfilled all inclusion criteria.

Third, we applied the information oriented selection strategy *Maximum Variation Cases* [32] to draw representative samples with varying project characteristics from among the remaining cases. We considered five dimensions with three categories per dimension: organization (Apache, Codehaus, JBoss), requirements (<1k, 1k–5k, >5k), lines of code (<50k, 50k–200k, >200k), horizontal trace links (<500, 500–2k, >2k), and vertical trace links (<5k, 5k–20k, >20k). To maximize the utility of information from our samples, we defined that each dimension×category tuple must be

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2016.2622264, IEEE Transactions on Software Engineering

represented by at least four cases. For tuples that were not represented by at least four cases after the second step, we manually searched for suitable cases. Eventually, we gained a list of 24 cases.

4.2 Unit of Analysis

Although we collected artifacts from 24 projects, we analyzed the data at a more granular level to increase the number of data point for the statistical analysis. We utilized the fact that each studied development project comprised multiple components and that the majority of the studied software development artifacts and defect reports were associated with one or many software component. Hence, the unit of analysis for this study is a software component.

4.3 Independent Variables

To address our research questions (Section 2), we identified eight measures that were computed for artifacts of the 24 studied software projects (see Section 4.6). In this section, we describe these measures, which serve as independent variables in our statistical analysis.

Traceability completeness with respect to UC-1 ($\overline{Q}_{trc,UC\text{-}1}$). The traceability use case UC-1 (see also Section 3.2.1) requires trace link paths as defined in Figure 2.

The number of artifacts that miss traceability to support UC-1 is quantified as follows:

$$|\{a|a \in C \wedge traceable_{C \to RC}(a) = \varnothing\}| \quad (1)$$

The function $traceable_{C \to RC} : C \to 2^{R \cup C}$ maps a requirement change ($a \in C$) onto a set of requirements and requirement changes ($R \cup C$) that have a direct or transitive traceability relationship with a. Thus, if $traceable_{C \to RC}(a)$ is an empty set, the requirement change a has no direct or transitive traceability relationship to any R or C. For each software component i, we quantified the traceability completeness with respect to UC-1 as $\overline{Q}^{\,i}_{trc,UC\text{-}1}$.

Traceability completeness with respect to UC-2 ($\overline{Q}_{trc,UC\text{-}2}$): The traceability use case UC-2 (see also Section 3.2.2) requires trace link paths as defined in Figure 3. The number of artifacts that miss traceability to support UC-2 is quantified as follows:

$$\sum_{a \in R \cup C} \frac{|traceable_{RC \to P}(traceable_{RC \to RC}(a)) = \varnothing|}{|traceable_{RC \to RC}(a)|} \quad (2)$$

The function $traceable_{RC \to P} : R \cup C \to 2^P$ maps a requirement or requirement change ($a \in R \cup C$) onto a set of source codes (S) that have a direct or transitive traceability relationship with a. Similarly, the function $traceable_{RC \to RC} : R \cup C \to 2^{R \cup C}$ maps a requirement or requirement change ($a \in R \cup C$) onto a set of requirements and requirement changes ($R \cup C$) that have a direct or transitive traceability relationship with a. The set $traceable_{RC \to P}(traceable_{RC \to RC}(a)) = \varnothing$ contains all requirements and requirement changes that depend on a but are not traceable to any source code artifact and thus misses the traceability to support UC-2. Each trace link path of UC-2 starts with a vertical trace link, which is also required by UC-1 and therefore considered by the traceability completeness measure for UC-1. To eliminate dependencies between the measures for UC-1 and UC-2, we normalize the number of requirements and requirement changes that depend on a but are not traceable to any source code artifact with the number of requirements and requirement changes that have a direct or transitive traceability relationship with a. For each software component i, we quantified the traceability completeness with respect to UC-2 as $\overline{Q}^{\,i}_{trc,UC\text{-}2}$.

Traceability completeness with respect to UC-3 ($\overline{Q}_{trc,UC\text{-}3}$): The traceability use case UC-3 (see also 3.2.3) requires trace link paths as defined in Figure 4. We quantify the number of artifacts that miss traceability with to support UC-3 as follows:

$$|\{a|a \in R \cup C \wedge traceable_{RC \to P}(a) = \varnothing\}| \quad (3)$$

Analogous to UC-2, the mapping function $traceable_{RC \to P}(a)$ is leveraged. The artifact a has no direct or transitive traceability relationship to any S if $traceable_{RC \to P}(a)$ is an empty set. This indicates that a misses traceability to support UC-3. For each software component i, we quantified the traceability completeness with respect to UC-3 as $\overline{Q}^{\,i}_{trc,UC\text{-}3}$.

Traceability completeness with respect to UC-4 ($\overline{Q}_{trc,UC\text{-}4}$): The traceability use case UC-4 (see also Section 3.2.4) requires trace link paths as defined in Figure 5. We quantify the number of artifacts that miss traceability to support UC-4 as follows:

$$|\{a|a \in P \wedge traceable_{P \to RC}(a) = \varnothing\}| \quad (4)$$

The function $traceable_{P \to RC} : P \to 2^{R \cup C}$ maps a source code artifact a onto a set of requirements or requirement changes that have a direct or transitive traceability relationship with a. If $traceable_{P \to RC}(a)$ is an empty set, the artifact a has no direct or transitive traceability relationship to any R or C and misses traceability to support UC-4. For each software component i, we quantified the traceability completeness with respect to UC-4 as $\overline{Q}^{\,i}_{trc,UC\text{-}4}$.

Number of requirements contributors (N_{RCtrb}): Requirements within a software project are created or modified by requirements contributors. We counted the number of distinct issue tracker account names who created requirements or requirement changes as N^i_{RCtrb} per software component i.

Number of source code contributors (N_{SCtrb}): Changes to source codes within a Software Configuration Management (SCM) tool are issued through commits by code contributors. We counted the number of distinct SCM tool account names who issued commits to the SCM repository as N^i_{SCtrb} per software component i.

Average time span between requirements modifications (M_{RDist}): To calculate the average time span between consecutive requirements and requirement changes, all artifacts of a software component were ordered by their creation timestamp. We computed the average time span between adjacent artifacts of the ordered set as M^i_{RDist} per software component i.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2016.2622264, IEEE Transactions on Software Engineering

Average time span between source code modifications (M_{SDist}): All source code commits were ordered by time-stamp and the average time span between adjacent changes was computed as M^i_{SDist} per software component i.

4.4 Dependent Variable

Our research questions are concerned with only one effect to study, the software quality of the analyzed software projects (see Section 2). Kitchenham and Pfleeger [67] suggest the number of defects as a measure for "software manufacturing quality" and subsequent studies, conducted by various researchers, successfully applied this measure in studies similar to ours. Therefore, we selected the *number of defects* (N_{Def}) as measure to operationalize the software quality within the 24 studied projects. All studied projects used an issue tracker system to document defects and their resolution. Project contributors file their discovered defects in this system. We used these filed defects to calculate the number of defects as N^i_{Def} per software component i. However, the existence of a defect issue does not necessarily imply the existence of a software defect. Hence, we only considered defects within the issue tracker that had the resolution types: *done*, *implemented*, and *fixed*. We excluded all defects with the resolution types: *cannotreproduce*, *communityanswered*, *duplicate*, *goneaway*, *incomplete*, *invalid*, *notaproblem*, *wontfix*, and *worksasdesigned*.

4.5 Data Collection Process

The data collection process for this study consisted of five steps (see Figure 6). We sequentially applied this process to all 24 cases to collect the necessary data for our empirical analysis.

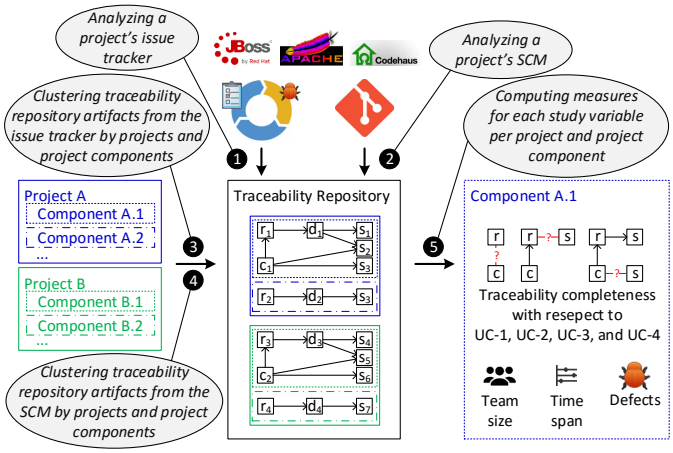


Fig. 6. Overview of the applied data collection process consisting of five steps

Step ❶: **Analyzing a project's issue tracker.** Common to all projects was that they used the issue tracker tool Jira [5] to manage requirements, requirement changes, design artifacts, and defects. The issue tracker tool supports the creation of trace links between any issue types. All projects extensively used this feature to establish traceability among the managed artifacts. Further, all projects followed a requirements change process as depicted in the lower row of Table 1. Accordingly, requirement changes are captured as newly created requirement change artifacts in the issue tracker, which are linked to the changed requirement through vertical trace links. Each requirement change artifact carries a unique identifier. We implemented a collector application that leverages the web service interface of the issue tracker and parses its standardized XML representation for artifacts and trace links. Through the developed application, we downloaded and parsed all requirements, requirement changes, design artifacts and defect descriptions, as well as the trace links between these artifacts.

Step ❷: **Analyzing a project's SCM.** We implemented another collector application that downloads all code changes and commit messages from the SCM repository. While 16 of the selected cases leveraged Git [36] as SCM system, the remaining 8 cases leveraged Subversion [127]. Every commit message was parsed for issue artifact identifiers, to discover horizontal trace links between the referred requirements or design artifact kept within the issue tracker and the source code artifacts of that commit. Since requirement changes are captured separately from requirements in the issue tracker, source code artifacts are explicitly linked to either the requirement or any related requirement change that was created throughout the change process. All collected and parsed software development artifacts and trace links (Step ❶ and ❷) were stored in a study repository for further analysis.

Step ❸: **Clustering traceability repository artifacts from the issue tracker by software components.** All studied cases decomposed the developed software into software components, where each component encapsulated a set of related functions or data. In this study, a single project component was the smallest entity of analysis (see Section 4.2). To enable a component based analysis of the study repository, for each collected artifact a unique component identifier was stored from which the artifact stemmed. The Jira issue tracker provides an issue attribute that stores the project components to which an artifact belongs. However, this attribute is not mandatory within the issue tracker tool and can potentially be empty for each artifact. Several artifacts were not mapped to any component, and thus were excluded from the measure computation (refer to Step ❺). An analysis showed that across all projects more than 80% of the issue tracker artifacts were mapped to one or many components.

Step ❹: **Clustering traceability repository artifacts from the SCM by software components.** Source code artifacts committed to the SCM were organized in folders that reflected the software components. However, for some components the naming conventions slightly differed between the issue tracker and the SCM. In these cases, the issue tracker component name comprised the main concept and additional descriptive words while the corresponding SCM component name solely comprised the main concept. We manually created a mapping table per project to match related software component names used in the issue tracker and the SCM. We validated all mapping tables through consensus decision-making. Issue tracker components, for which we could not find a component folder within the SCM were excluded from the analysis. These excluded components were created for project administration purposes such as "build-process", "website", and "documentation". Due

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2016.2622264, IEEE Transactions on Software Engineering

to this artifact to component mapping, the study repository could be clustered by software components.

Step ❺: **Computing measures for each study variable per project and project component.** In this step, based on the clustered artifacts and trace links within the central repository, the measures for the independent variables discussed in Section 4.3 ($\overline{Q}^{\,i}_{trc,UC\text{-}1}$, $\overline{Q}^{\,i}_{trc,UC\text{-}2}$, $\overline{Q}^{\,i}_{trc,UC\text{-}3}$, $\overline{Q}^{\,i}_{trc,UC\text{-}4}$, N^i_{RCtrb}, N^i_{SCtrb}, M^i_{RDist}, and M^i_{SDist}) and for the dependent variable (N^i_{Def}) discussed in Section 4.4 were computed per software component i. For this purpose, we implemented a computation application that uses the open source library QuickGraph [62], which provides generic graph data structures and algorithms. The resulting measure data were stored for further statistical analysis.

4.6 Studied Cases

Table 2 shows characteristics of the studied cases. For all projects, we gained raw data by collecting relevant project artifacts referenced at the projects' websites and created till August 16^{th}, 2014. We make the raw data on which this study is based publicly available to ensure that our study can be reproduced and replicated [105]. The second column (*Requirements*) summarizes the number of requirements and requirement changes per studied software project. Across all projects, an average of 1,277 requirements exists per project, with a minimum of 41 requirements (IronJacamar) and a maximum of 4,464 requirements (mongoDB). The third column (*Lines of Code*) shows the lines of code per project. On average, a project was implemented by 232,733 lines of code, with a minimum of 6,457 lines (TorqueBox) and a maximum of 942,409 lines (GeoTools). The fourth column (*Commits to SCM*) shows the number of committed changes to the SCM. On average, 7,001 changes were committed to the project's SCM, with a minimum of 177 (Smooks) and a maximum of 26,679 (mongoDB). The fifth column (*Vertical trace links*) shows the number of vertical trace links per project. On average, each project captured 11,222 vertical trace links, with a minimum of 24 (Gumtree) and a maximum of 101,752 (JBoss Seam2). The last column (*Horizontal trace links*) shows number of horizontal trace links per project. On average, each project captured 30,937 horizontal trace links, with a minimum of 143 (Grails) and a maximum of 226,116 (Apache Lucene).

All analyzed trace links were created by respective project members as ordinary part of their development. Vertical trace links were created through the issue trackers' traceability functionality and horizontal trace links through commit messages submitted to the SCM. To establish horizontal trace links, project members added unique issue tracker artifact ids in squared brackets to the commit log of changed source code. Commit logs containing multiple issue tracker artifact ids established multiple horizontal trace links.

TABLE 2
Characteristics of the studied projects

Project	Requirements	Lines of code	Commits to SCM	Vertical trace links	Horizontal trace links
Activiti BPMN [1]	790	19,605	4,667	344	24,974
Apache Archiva [121]	522	83,972	7,536	1,034	20,189
Apache Axis2 [122]	1,117	144,412	11,151	1,772	45,964
Apache Derby [123]	2,046	264,350	7,940	6,570	49,310
Drools [57]	115	256,088	9,484	61,194	63,328
Errai Framework [58]	321	27,863	6,873	15,496	6,039
GeoServer [33]	1,853	349,094	3,311	2,950	44,761
GeoTools [34]	1,633	942,409	3,460	1,912	26,817
Grails [41]	1,752	74,355	14,597	2,084	143
Groovy [43]	1,763	149,909	11,617	1,622	20,293
Groovy-Eclipse [44]	397	13,418	2,839	390	16,745
GumTree [45]	585	383,201	601	24	164
Apache Hadoop [124]	3,046	806,763	10,509	7,588	97,918
HornetQ [49]	334	352,306	10,901	26,518	22,187
IronJacamar [53]	41	85,908	2,161	23,098	9,780
JBoss Narayana [60]	514	297,996	1,341	44,710	12,876
Apache Lucene [125]	2,619	693,928	12,862	1,834	226,116
mongoDB [86]	4,464	653,705	26,679	9,548	48,962
Apache Pig [126]	1,151	29,870	2,810	1,798	15,934
JBoss Seam2 [113]	1,652	24,143	11,305	101,752	10,373
Smooks [114]	314	78,202	177	166	1285
SonarQube [116]	2,948	72,911	12,031	3,248	28,872
Sonar CSharp [117]	241	12,643	1,392	140	170
TorqueBox [61]	428	6.457	4,576	2,361	24,830

TABLE 3
Summary statistics across all analyzed software projects (N = 24)

Variable	Mean	SD	Min	1^{st}Q	Median	3^{rd}Q	Max
$\overline{Q}_{trc,UC\text{-}1}$	451.8	502.5	0.0	1.0	189.0	826.0	1403.0
$\overline{Q}_{trc,UC\text{-}2}$	5.6	5.0	0.3	1.8	4.8	7.2	19.7
$\overline{Q}_{trc,UC\text{-}3}$	367.1	346.9	0.0	112.3	215.0	649.0	1088.0
$\overline{Q}_{trc,UC\text{-}4}$	51,955.1	66,063.0	59.0	11,059.8	22,124.5	65,936.5	270,140.0
T [years]	6.0	2.6	2.1	3.9	5.1	8.6	10.9
N_{RCtrb}	9.3	5.8	1.0	6.3	8.0	13.0	26.0
N_{SCtrb}	37.3	27.2	3.0	15.5	26.0	57.3	97.0
M_{RDist} [years]	1.57	0.90	0.01	1.05	1.49	1.96	4.53
M_{SDist} [years]	0.21	0.19	0.01	0.08	0.13	0.30	0.69
N_{Def}	1,209.2	1,224.0	9.0	277.8	899.0	1,614.3	5,038.0

5 Data Analysis

Based on the measures captured within the data collection process (see also Section 4.5) we performed statistical analyses to study the impact of requirements traceability on software quality. All statistical analyses were performed within the R environment [130].

Table 3 shows *mean*, *standard deviation* (SD), *minimum* (Min), *first quartile*, *median* (*second quartile*), *third quartile* and *maximum* (Max) of all quantification measures across the 24 studied projects. Similarly, Table 4 shows all quantification measures for the 610 software components that the 24 projects were composed of.

To study the impact of traceability on the software quality, a generalized linear Poisson regression model was applied. Poisson regressions can be applied to situations in which the response variable is a number of events that occur in a given period of time, which is the case for the studied number of defects within a software component (N_{Def}). Figure 7 shows the number of N_{Def} across all software components, which indeed closely follows a Poisson distribution. The linear Poisson regression model is defined as

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2016.2622264, IEEE Transactions on Software Engineering

TABLE 4
Summary statistics across all analyzed components (N = 610)

Variable	Mean	SD	Min	1^{st}Q	Median	3^{rd}Q	Max
$\overline{Q}_{trc,UC\text{-}1}$	27.5	40.9	1.0	4.0	13.0	35.0	325.0
$\overline{Q}_{trc,UC\text{-}2}$	0.5	0.2	0.0	0.4	0.5	0.7	1.0
$\overline{Q}_{trc,UC\text{-}3}$	19.3	30.7	1.0	3.0	8.0	23.0	216.0
$\overline{Q}_{trc,UC\text{-}4}$	3,624.8	9,262.6	1.0	221.8	802.5	2,339.8	105,457.0
T [years]	4.6	2.8	0.01	2.8	3.5	6.5	10.9
N_{RCtrb}	3.4	3.1	1.0	1.0	2.0	4.0	26.0
N_{SCtrb}	18.9	16.0	1.0	8.0	14.0	25.0	97.0
M_{RDist} [years]	1.68	1.62	0.00	0.003	1.41	2.59	7.36
M_{SDist} [years]	0.23	0.36	0.00	0.00	0.04	0.35	2.71
N_{Def}	52.8	93.4	1.0	5.0	19.0	57.0	820.0

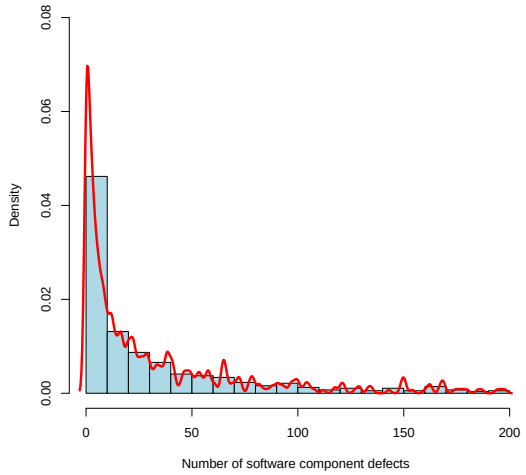


Fig. 7. The histogram of the number of defects N_{Def} across all software components of all studied projects shows a Poisson distribution. The red curve visualizes the density estimates.

follows:

$$ln(\lambda_i) = \beta_0 + \sum_{h=1}^{n} \beta_h x_{hi} + \varepsilon_i \quad (5)$$

where λ_i is the expected N_{Def} for a software component i, β_0 is the intercept of the model, x_{hi} are the independent variables describing the modeled phenomena, β_h are the slopes of the independent variables, n is the number of independent variables, and ε_i is the random error of the regression model.

The linear Poisson regression model (see Equation 5) assumes that the period of time is constant across observations. As the duration of development varies for each software component on which the number of defects occur, the development time per software components must be included in the model. As described in [63], we include the variable T to the model, which changes the linear Poission regression model as follows:

$$ln\left(\frac{\lambda_i}{T_i}\right) = \beta_0 + \sum_{h=1}^{n} \beta_h x_{hi} + \varepsilon_i \quad (6)$$

where T_i is the period over which a software component i was developed. This period was computed by calculating the timespan between the create date of the oldest artifact and the change data of the most recent changed artifact of a software component i. The linear Poisson regression model rests on the assumption of independently distributed error terms for the individual observations. This assumption implies that no relationships exists among the individuals. The 610 analyzed software components in this study stemmed from 24 distinct software projects. However, the software projects differ among others in terms of size, developers, and release cycles. Thus, there may indeed exist relationships between software components of the same software project. To avoid violating the model's assumption of independent errors, we decided to use a multi-level model design that incorporates potential intra-project relationships among software components. Thereby, software components are nested within software projects and each software component appears in one project only. In the context of multi-level regression models, there are potentially separate intercepts and slopes of the independent variables for each cluster of software components that belong to one project. The two-level Poisson regression model is described as follows:

$$ln\left(\frac{\lambda_{ij}}{T_i}\right) = \gamma_{00} + U_{0j} + \sum_{h=1}^{n} \left(\gamma_{h0} x_{hij} + U_{hj} x_{hij}\right) + \varepsilon_i \quad (7)$$

where λ_{ij} is the expected number of defects per software component i in software project j, γ_{00} is the cluster independent intercept part that remains constant across all software projects, U_{0j} is the cluster dependent intercept part that varies across the projects, γ_{h0} is the cluster independent slope of the independent variables x_{hij}, and U_{hj} are the cluster dependent slopes of x_{hij}.

An initial multi-level generalized linear Poisson regression model was fitted, consisting of all independent variables ($\overline{Q}_{trc,UC\text{-}1}$, $\overline{Q}_{trc,UC\text{-}2}$, $\overline{Q}_{trc,UC\text{-}3}$, $\overline{Q}_{trc,UC\text{-}4}$, N_{RCtrb}, M_{RDist}, N_{SCtrb}, M_{SDist}) and all possible second-order interactions between these independent variables describing the number of defects in software components clustered into the analyzed software projects $(1|ProjectRnd)$. A significant interaction between two independent variables indicates that the relationship between one independent and the dependent variable depends on the level of another independent variable. This initial model was tested for over-dispersion (Residual deviance = 4,468.7; Residual degree of freedom = 495). The ratio of 9.03 between the residual deviance and the degree of freedom was larger than 1.0 and clearly indicated over-dispersed data. To compensate for this over-dispersion, we additionally assumed a random between-transects error at component level $(1|CompRnd)$ that is additive on the scale of the linear predictor as proposed by Maindonald & Braun [83] and is considered superior to the constant multiplier used in the alternative quasi-Poisson error assumption.

According to the principle of parsimony for statistical modeling [20], statistical models should feature as few parameters as possible to be minimal adequate. To derive such a minimal adequate model from the initial statistical model, we applied a stepwise model simplification process as proposed by Crawley [20]. After fitting the initial model, we iteratively removed the least significant terms from the model and applied a Chi-squared test to find out whether the removal caused a significant increase in deviance over the initial model. If the Chi-squared test rejected the null-hypothesis that the deviance was increased significantly,

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2016.2622264, IEEE Transactions on Software Engineering

we continued the simplification process with the reduced model. The resulting simplified model consists of the following parts:

- *Variables:* $\overline{Q}_{trc,UC\text{-}1}$, $\overline{Q}_{trc,UC\text{-}2}$, $\overline{Q}_{trc,UC\text{-}3}$, N_{RCtrb}, M_{RDist}, and M_{SDist}.
- *Interactions:* $\overline{Q}_{trc,UC\text{-}1} : \overline{Q}_{trc,UC\text{-}3}$, $\overline{Q}_{trc,UC\text{-}1} : N_{RCtrb}$, and $\overline{Q}_{trc,UC\text{-}2} : \overline{Q}_{trc,UC\text{-}3}$.
- *Random terms:* $(1|ProjectRnd)$ and $(1|CompRnd)$.

A potential problem when using a regression model arises when predictor variables are highly correlated. This multicollinearity amongst the effect-causing variables can inflate the variance amongst these variables in the model. The Variance Inflation Factor (VIF) would indicate multicollinearity if it exceeds a value of 10 (see Belsley et al. [6]), which was not the case for our model: VIF($\overline{Q}_{trc,UC\text{-}1}$) = 5.09, VIF($\overline{Q}_{trc,UC\text{-}2}$) = 1.1, VIF($\overline{Q}_{trc,UC\text{-}3}$) = 3.4, VIF($\overline{Q}_{trc,UC\text{-}4}$) = 1.07, VIF($N_{RCtrb}$) = 2.62, VIF($M_{RDist}$) = 1.44, and VIF($M_{SDist}$) = 1.32.

6 Results

Table 5 shows the random effects of the fitted model. The random effect values were used to calculate the Intra Class Correlation (ICC) as described in [29] for the 24 project clusters (ICC = 0.066). The ICC indicates that only 6.6% of the variance in software component defects can be attributed to the project membership of a software component, which implies a very weak relationship among the defects for two components of the same project. This very low ICC value implies that the multilevel data structure of our study (610 components from 24 different projects) do not impact the dependent variable. Hence, the study setup can be considered to be valid to investigate the effect of the independent variables (see Section 4.3) on the dependent variable (see Section 4.4) among the 610 analyzed components.

TABLE 5
Random effects of the fitted multi-level Poisson regression

Groups	Observations	Variance	SD
$ProjectRnd$	24	0.144	0.379
$CompRnd$	610	0.661	0.813

Table 6 summarizes the fixed effects of the finally fitted model. Column Est_α shows the estimated coefficients of this model. In Poisson regression the modeled dependent variable is the natural logarithm of the conditional mean (see Section 6). To facilitate a more intuitive interpretation, the estimated variable coefficients are additionally exponentiated (see column e^{Est_α}) and so transformed into the original scale of the dependent variable. The column *z-value* shows the test statistics for the null hypothesis that the associated coefficient is equal to 0. The column $Pr(>|z|)$ shows the p value for the test that the coefficient is equal to 0. A p value less than 0.05 indicates statistical significance. The studied independent variables of the fitted model have very different scales of measurement (see Tables 3 and 4). To evaluate the relative importance of one independent variable in relation to the others, we transformed each independent variable X_i into X_i^T with a standardized scale across all variables. We established comparability of the relative causal effect across independent variables by standardizing each variable to common characteristics [66], namely mean $\bar{X}_i$ and standard deviation $SD(X_i)$ of the variable X_i in the sample: $X_i^T = (X_i - \bar{X}_i)/SD(X_i)$. This linear transformation rescaled each independent variable to have a zero mean and standard deviation one. We independently fitted this standardized model. Column Est_β shows the estimated coefficients of the fitted standardized model, where the units of each independent variable are standard deviations. This means that while the estimated coefficients of the non-standardized model (column Est_α) reflect the actual causal law, the estimated coefficients of the standardized model reflect the relative contribution of the independent variables. Column RK_β shows the rank order of a variable's relative importance with lower ranks referring to higher relative importance on the causal effect.

TABLE 6
Fixed effects of the fitted multi-level Poisson regression

Effect variable	Est_α	e^{Est_α}	z-value	Pr(>|z|)	Est_β	RK_β
$Intercept$	-3.323	0.036	-27.498	<.0001 ***		
$\overline{Q}_{trc,UC\text{-}1}$	0.023	1.023	6.087	<.0001 ***	0.734	1
$\overline{Q}_{trc,UC\text{-}2}$	0.018	1.018	5.926	<.0001 ***	0.508	2
$\overline{Q}_{trc,UC\text{-}3}$	0.011	1.011	2.915	0.004 **	0.212	4
N_{RCtrb}	0.098	1.103	5.559	<.0001 ***	0.428	3
M_{RDist}	0.002	1.002	3.165	0.002 **	0.066	8
M_{SDist}	-0.001	0.999	-3.032	0.002 **	-0.131	6
$\overline{Q}_{trc,UC\text{-}1} : \overline{Q}_{trc,UC\text{-}3}$	-.0001	.9999	-3.899	<.0001 ***	-0.083	7
$\overline{Q}_{trc,UC\text{-}1} : N_{RCtrb}$	-.0002	.9998	-2.134	0.032 *	-0.059	9
$\overline{Q}_{trc,UC\text{-}2} : \overline{Q}_{trc,UC\text{-}3}$	-.0002	.9998	-4.548	<.0001 ***	-0.194	5

significance codes for Pr(>|z|): < .0001 '***', ≤0.01 '**', ≤0.05 '*'

The first row of the table shows the estimated intercept (-3.323) and estimates the logarithm of N_{Def} across software components when each of the independent variable would be zero. As it is not possible, for example, to have zero requirements contributors in a software project, the intercept value is only of minor practical relevance in the interpretation of this statistical model.

The second row shows that the variable $\overline{Q}_{trc,UC\text{-}1}$ significantly affects the defect rate ($Pr(>|z|) < 0.0001$). Each additional requirement lacking the necessary traceability to dependent requirements for supporting use case "high-level impact analysis" (refer to Section 3.2.1) increases the expected software component defect rate by 2.3% and has the highest relative importance among the independent variables on the causal effect. The third row shows that the variable $\overline{Q}_{trc,UC\text{-}2}$ also significantly affects the defect rate ($Pr(>|z|) < 0.0001$). Each dependent requirement that does not feature the necessary traceability to support use case "low-level impact analysis" (refer to Section 3.2.2) increases the expected defect software component rate by 2% and has the second highest relative importance among the independent variables on the causal effect. The fourth row shows that the variable $\overline{Q}_{trc,UC\text{-}3}$ also significantly affects the defect rate ($Pr(>|z|) = 0.004$). Each additional requirement that does not feature the necessary traceability to the implementing source code for use case "requirements satisfaction analysis" increases the expected defect rate by 1.1% and has the fourth highest relative importance among the independent variables on the causal effect.

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2016.2622264, IEEE Transactions on Software Engineering

The variables $\overline{Q}_{trc,UC\text{-}4}$ and N_{SCtrb} were eliminated during the stepwise model simplification process due to their insignificance for the deviance of the dependent variable, and thus do not appear in Table 6. This implies that neither the degree of traceability completeness of a software component with respect to UC-4 nor the number of source code contributors of a software component have a significant impact on the software component's defect rate.

The results in Table 6 also show that N_{RCtrb} has a strongly significant effect ($Pr(>|z|) < 0.0001$) on the defect rate. One additional requirements contributor per software component increases the expected defect rate by 10.3% and has the third highest relative importance. M_{RDist} has a significant effect ($Pr(>|z|) = 0.002$) on the defect rate. A one year increase of the M_{RDist} increases the expected defect rate by 0.2% and has the eighth highest relative importance. M_{SDist} has a significant effect ($Pr(>|z|) = 0.002$) on the defect rate. A one year increase of the M_{SDist} decreases the expected defect rate by 0.1% and has the sixth highest relative importance.

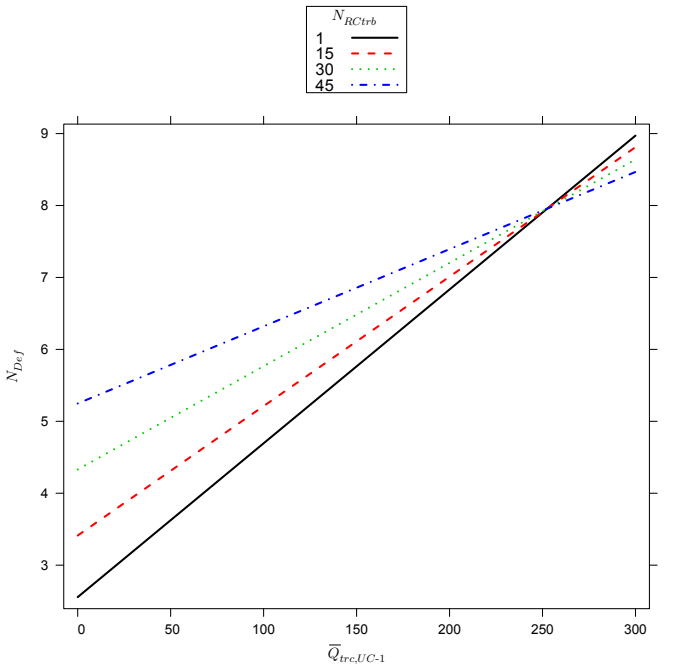


Fig. 8. Interaction effect between incomplete traceability for use case UC-1 ($\overline{Q}_{trc,UC\text{-}1}$) and the number of requirements contributors (N_{RCtrb})

In addition to the discussed independent variables, the regression analysis also showed three interactions between these variables to have a significant effect on the dependent variable. The interaction effect of $\overline{Q}_{trc,UC\text{-}1} : N_{RCtrb}$ is disordinal with a crossing point at 270 requirements (see Figure 8). For software components with less or equal to 270 requirements, more contributors intensify the effect that $\overline{Q}_{trc,UC\text{-}1}$ has on the expected defects. For components with more than 270 requirements, more contributors attenuate this effect. The interaction effect of $\overline{Q}_{trc,UC\text{-}1} : \overline{Q}_{trc,UC\text{-}3}$ means that a higher amount of $\overline{Q}_{trc,UC\text{-}3}$ attenuates the effect of $\overline{Q}_{trc,UC\text{-}1}$ on the N_{Def} (see Figure 9). Similarly, the interaction effect of $\overline{Q}_{trc,UC\text{-}1} : \overline{Q}_{trc,UC\text{-}3}$ means that a

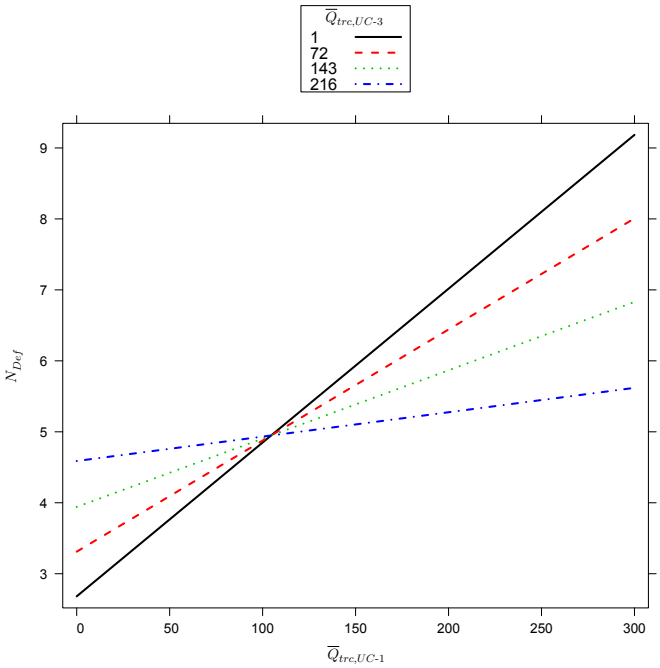


Fig. 9. Interaction effect between incomplete traceability for use case UC-1 ($\overline{Q}_{trc,UC\text{-}1}$) and incomplete traceability for use case UC-3 ($\overline{Q}_{trc,UC\text{-}3}$)

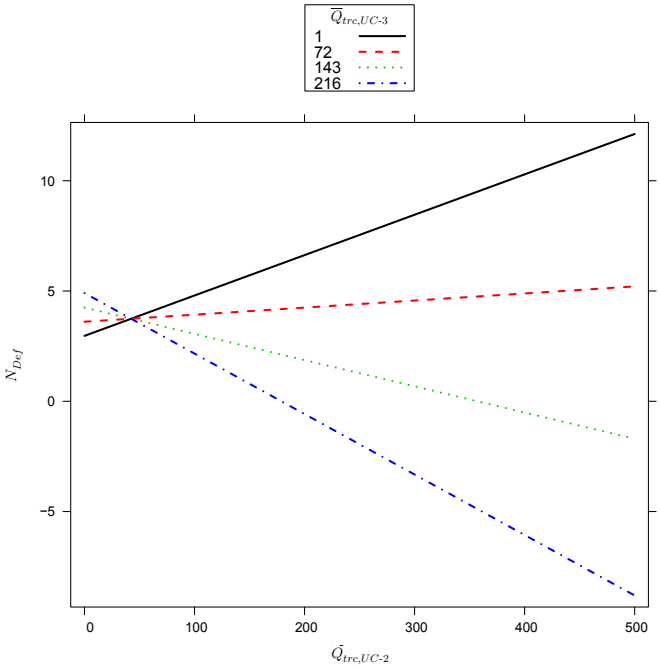


Fig. 10. Interaction effect between incomplete traceability for use case UC-2 ($\overline{Q}_{trc,UC\text{-}2}$) and incomplete traceability for use case UC-3 ($\overline{Q}_{trc,UC\text{-}3}$)

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2016.2622264, IEEE Transactions on Software Engineering

higher amount of $\overline{Q}_{trc,UC\text{-}3}$ attenuates the effect of $\overline{Q}_{trc,UC\text{-}2}$ on N_{Def} (see Figure 10).

7 Discussion

In this section, we discuss the implications of the study's statistical results (see Section 6) with regards to our research questions (refer to Section 2).

7.1 Does traceability completeness affect software quality?

To investigate *RQ-1*, we conducted a two-level Poisson regression analysis (see Section 5). The results indicate that the degree to which artifacts are traceable has a statistically significant impact on the number of defects, and thus on the software quality. Thereby, the level of significance was less or equal to 0.01 for three of the studied traceability use cases (UC-1, UC-2, and UC-3). This finding is of great practical value for software project managers who need to decide if and how much traceability should be captured. While the work of Cleland-Huang et al. [13] as well as Heindl and Biffl [47] provided clarification on the costs to be expected for implementing requirements traceability, the benefits to be expected to compensate these costs remaind unclear [4]. The results of this study provide for the first time empirical evidence, based on open source software projects that are frequently used in industrial developments, that the implementation of requirements traceability to enable the software engineering activities *high-level impact analysis*, *low-level impact analysis*, and *requirements satisfaction analysis* significantly reduces the defect rate of the implemented software. Since a reduced software defect rate entails less need for software maintenance, well established cost estimation schemes for software maintenance tasks [115, 2] can be used to quantify the expected savings. This enables practitioners to perform cost-benefits analyses with respect to the implementation of requirements traceability, providing a reliable basis for decision makers in software projects.

The results also show that the degree of traceable artifacts required for use case UC-4 has no significant impact on the dependent variable. These results confirmed our findings in [8] that traceability is less important for software implementation tasks.

> Finding 1 (RQ-1). Traceability **completeness affects software quality significantly**.

The finding of prior controlled experiments [75, 56, 76], conducted within artificial software development environments, that the availability of traceability implies a higher software implementation correctness has been partly confirmed by our study. However, contrary to the previous studies, the results of this study show that this effect does not exists for all kinds of requirements traceability. This finding is of great value for decision makers in software projects who solely aim to improve software quality. For example, projects not required to implement traceability by legal obligations or safety guidelines could potentially focus their requirements traceability implementation to those trace links that are expected to improve software quality most.

7.2 Does the effect vary for traceability use cases?

The study results our statistical analysis also provide answers to *RQ-2* and *RQ-3*. They show that the degree of completeness of traceability required to support the use cases "High-level impact analysis" (refer to Section 3.2.1), "Low-level impact analysis" (refer to Section 3.2.2), and "Requirements satisfaction analysis" (refer to Section 3.2.3), influences the number of defects and thus the software quality.

> Finding 2 (RQ-2). The **effect** of traceability **completeness** on **software quality varies** among traceability use cases: UC-1 $\succeq$ UC-2 $\succeq$ UC-3.

For each of the three use cases, improving the degree of traceability completeness leads to an improved expected software quality. The relative importance of the degree of traceability completeness for the expected software quality improvement varies among the three use cases: UC-1 ranks first, UC-2 ranks second, and UC-3 ranks fourth within the regression model. The degree of traceability completeness for UC-4 does not significantly affect software quality. Accordingly, the following prioritization order is suggested: UC-1 $\succeq$ UC-2 $\succeq$ UC-3.

> Finding 3 (RQ-3). An **increase** of traceability **completeness** also **increases software quality**.

7.3 Is the effect impacted by development team size?

As summarized in Table 6, the number of requirements contributors N_{RCtrb} has a weak significant interaction with the traceability availability of UC-1, which is relevant with respect to *RQ-4*. As shown in Figure 8, the interaction effect of $\overline{Q}_{trc,UC\text{-}1} : N_{RCtrb}$ varies depending on the number of requirements within the software component. For software components with less or equal than 270 requirements, more contributors intensify the effect that an increase in $\overline{Q}_{trc,UC\text{-}1}$ is associated with increased expected defects. For components with more than 270 requirements, more contributors attenuate this effect. The number of programming code contributors N_{SCtrb} has no significant interaction with the traceability availability of any use case.

The results of our study rejected our hypothesis that the team size impacts the effect of traceability completeness on the defect rate. Contrary to our initial assumption, the results of our study show that the effect does not significantly vary across projects with different team sizes, ranging from 1 to 26 requirements contributors and 1 to 97 source code contributors per component. This implies that the findings we discussed in the Sections 7.1 and 7.2 hold at least for team sizes varying between 1 and 97 contributors. Managers of software projects at any team size within this range can expect a decreased number of defects from improving the degree of the traceability completeness as previously discussed. A possible reason for this result could be the fact that we solely analyzed open source projects, which typically comprise project members that work from many different and often widely distributed locations. Even in

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2016.2622264, IEEE Transactions on Software Engineering

projects with small team sizes information about artifact relationships needs to be documented explicitly.

> Finding 4 (RQ-4). The **effect** of traceability **completeness** on **software quality** is **not dependent** upon the development **team size**.

7.4 Is the effect impacted by time span between software changes?

Table 6 shows that there is no significant interaction between the requirements change time span M_{RDist} or programming code change time span M_{SDist} and any traceability availability measure. This means for *RQ-5* that the results of our study reject our hypothesis that longer requirements change time spans or source code change time spans make the use of traceability more valuable, because developers may forget requirements dependencies over time.

The results of our study also rejected our hypothesis that the time span between software changes impacts the effect of traceability completeness on the defect rate. The effect does not significantly vary across projects with different time spans between software changes, ranging from less than 1 day to more than 7 years for requirements changes and less than 1 day to more than 2 years for software changes. This implies that the findings we discussed in the Sections 7.1 and 7.2 hold at least for average change time spans in those ranges. A possible reason for this result could be the complexity and maturity of the analyzed projects. We concluded that memorizing relationships between artifacts in projects with this complexity is of minor practical relevance.

> Finding 5 (RQ-5). The **effect** of traceability **completeness** on **software quality** is **not dependent** upon the **time span** between code changes.

7.5 Limitations

A potential limitation of this study is the fact that our study does not consider the actual usage of the available trace links, because none of the analyzed projects provide explicit statistics on how often a specific trace link path was traversed by whom. While there is the theoretical chance that the implemented trace links were not actually used, we regard this as not realistic for the following three reasons. First, all analyzed projects define guidelines for project contributors that explicitly demand the creation of trace links. The Apache Derby project [123], for example, encourages developers in their project guidelines [128] to "*include the following in a commit log message: the ID of the JIRA issue*" and to "*make sure you use the format DERBY-NNN so that JIRA picks it up*". Similarly, the *Guide for Hadoop Core Committers* [129] states "*commit message should include a JIRA issue id [..]*". Similar guidelines exist for all analyzed projects. Further, these guidelines also urge project stakeholders to use captured trace links to support and improve software engineering activities. The Grails project [41], for example, explains in section *Contributing to Grails* of the project's reference documentation [40] "*This may not seem particularly important, but having a JIRA issue ID in a commit message means that we can find out at a later date why a change was made [..]*". Second, the fact that so much manual effort (average: 11,222 horizontal links and 30,937 vertical trace links per project) was spent to manually create trace links is a clear indicator that projects' stakeholders expect benefits from traceability usage, and thus make actual use of them. Third, a manual analysis of representative samples drawn from the textual comments within the issue tracker attached to requirements revealed that these comments often provide arguments based on existing artifact dependencies represented by captured trace links. The following discussion between developers of the Apache Derby project exemplifies the usage of trace links. The context is that requirement change DERBY-6482 is related to the requirement change artifacts DERBY-6256 and DERBY-3684, the bug artifact DERBY-6683, and the task artifact DERBY-6459. The assignee of the related task issue DERBY-6459 asked the reporter of this issue: "*Would you like me to hold off on finishing DERBY-6459 until this work is completed? [..] I created a second patch for DERBY-6459 to incorporate the changes you made for DERBY-6256. I'm wondering if it might make sense to commit the existing patch [..] and then file another one for future VTI changes.*" Statements like these demonstrate that stakeholders are aware of the complex dependencies among artifacts and use them for their development work.

8 Threats to Validity

In this section, we discuss threats to the validity of our study and how we mitigated them. To systematize the discussion, we structure potential threats into four common categories: construct validity, external validity, internal validity, and reliability (see [112, 132]).

8.1 Construct Validity

Software quality. The effect to be measured in our study is the software quality of a developed software component. We operationalized this effect as the number of defects that were filed and assigned to one or many software components within the issue tracker throughout the project's development. This effect operationalization as number of defects could be a threat to construct validity if the filed defect issues do not sufficiently represent the occurred software defects. Assuming that the issue tracking and management was performed properly, because all 24 studied cases are widely disseminated and mature open source software tools with a broad user group, the number of filed defects that were accepted by the developers as a defect that needs to be fixed should be a good indicator of the software quality of a software component. Also, this measure has already been applied in other studies in the past [119, 87, 31]. The assumption that the issues were tracked carefully by the analyzed projects is supported by the fact that each of them explicitly defined a development process for project contributors, which among others prescribe to always create a defect issue if an software defect was detected. Since we studied open-source projects where anyone is allowed to file defect issues, a potential threat exists that issues were filed

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2016.2622264, IEEE Transactions on Software Engineering

that do not represent an actual software defect. To mitigate this threat, we excluded all issues that were not accepted by the developers as a valid defect.

Traceability completeness. We proposed the four measures $\overline{Q}_{trc,UC\text{-}1}$, $\overline{Q}_{trc,UC\text{-}2}$, $\overline{Q}_{trc,UC\text{-}3}$, and $\overline{Q}_{trc,UC\text{-}4}$ to operationalize the degree of traceability completeness with respect to the four most common traceability usage scenarios derived through the analysis of existing literature. To ensure the construct validity of these measures, we had to address the questions to which degree our proposed measures are suitable to measure the degree of traceability completeness in support of the four use cases. Hence, we developed in Section 3 a qualitative description model that provides for each of the four traceability use cases a generic definition of all possible trace link paths between source and target artifacts. Assuming that our derived description model is valid, the derived traceability completeness measures should be a good indicator of the traceability completeness of the analyzed software components.

Team size and change time span. For the construct validity of the other independent variables, we had to address the question to which degree the measures N_{RCtrb} and N_{SCtrb} measure the concept team size as well as M_{RDist} and M_{SDist} measure the concept change time span. The team of an open source development project typically consists of people who contribute to the project by following a predefined project contribution process. Accordingly, we consider the number of requirements and source code contributors as a valid measure for the team size of open source projects. Since all analyzed projects followed a predefined contribution process, all changes to the developed software were initiated by new issue tracker items followed by commits to the SCM repository. This led us to the assumption that the time span between issue tracker items and SCM repository commits that are ordered by creation time is a valid measure for the change time span.

Documentation quality. A necessary precondition for analyzing the projects on a per-component basis is that the issue tracker items and the committed SCM items are assigned to one or many components of the developed software. Hence, there exists a potential threat that missing item to component assignments skew the operationalization measures that were used to study the effect of traceability completeness on the software quality. To mitigate this threat, we analyzed whether a project contained artifacts that were not assigned to any component. Although, we found these widow artifacts in all projects, the percentage was less than 1% across all projects. Therefore, we consider our operationalization measures as not skewed by missing artifact to component assignments. Another potential threat to construct validity is that developers created incorrect artifact to component assignments by mistake. Although we cannot completely rule out that implementation mistakes were made by the developers, we consider this threat sufficiently mitigated by the fact that every implementation activity of the analyzed projects is reviewed by at least a second person.

Traceability Quality. The analyzed trace links were created manually by project members in all analyzed projects. This implies the risk that semantically incorrect trace links were created or trace links were forgotten by mistake. The following three aspects indicate a very high trace link quality in all projects. First, all projects' quality assurance process is based on the created trace links. Three projects (Apache Archiva, Apache Axis2, and Apache Hadoop) established a process where changes are automatically tested in a continuous manner and in a second step reviewed by humans. The remaining 21 projects established a manual process where changes are reviewed and tested by humans. All 24 projects have in common that the quality of the established trace links is implicitly verified through this process. Second, the explicit change approval process in all 24 projects ensures that the four-eyes-principle is applied for each manually created trace link. Third, the openness of all 24 projects (all development artifacts are publicly available) enables anyone to participate in the project and review the created trace links. Due to these facts, we consider the risk of incorrect or forgotten trace links sufficiently mitigated.

8.2 External Validity

For our study, we solely focused on open-source projects, since those were the only available projects to us that provided all the necessary information to conduct this study. A potential threat to external validity arises when we want to generalize our findings to a wider population that includes commercial developments. To identify commonalities and differences between open-source and closed-source projects, we compared the 24 analyzed open-source projects with 17 closed-source projects, which we analyzed in two previous studies [106, 108]. We found that the used software lifecycle management tools (Jira, Git) of the 24 open-source projects was also the most popular combination of tools for the closed-source projects. While open-source tools primarily focus on automated unit tests for software quality assurance, the majority of closed-source projects applied a combination of automated unit tests and manual user acceptance tests. Both project types typically applied a similar agile development methodology. However, the sizes of development teams were significantly different. The average team size of open-source projects was on average ten times bigger than the team size of the compared closed source projects. Since we found that the team size does not significantly influences the impact of the degree of traceability completeness on the software quality (see Section 7), we consider our findings to be generalizable to a large population of closed-source projects. However, replications of our study with closed-sourced projects are required to justify our assumption by further empirical evidence.

Another potential threat to the external validity could be an insufficient distribution of projects across characteristics like project size, number of releases, project longevity, number of contributors, project domain, and used software lifecycle tools. As outlined in Table 2, our study featured software projects of varying sizes ranging from small projects with few requirements (e.g., IronJacamar: 41 requirements) to very big projects with many requirements (e.g., mongoDB: 4,464 requirements). Projects also varied in terms of number of releases, project longevity, number of contributors, and project domains (see Tables 3 and 4).

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2016.2622264, IEEE Transactions on Software Engineering

Thus, we consider our results as generalizable to projects with varying characteristics. However, all analyzed projects used the same set of tools for managing the software lifecycle. Additionally, the distribution of the studied projects could probably further be improved by adding more diverse projects.

8.3 Internal Validity

We applied regression analysis, which is a common statistical tool to identify explanatory independent variables that are related to the dependent variable. We found that the degree of traceability completeness has a statistically significant impact on the software quality. Such statistical relationships do not necessarily entail a causal relationship, but rather provide empirical evidence. Only controlled experiments, where one independent variable is varied in a controlled manner and all other independent variables remain constant, could really demonstrate causality. However, conducting controlled experiments in such big and complex software development projects is unrealistic.

Another potential threat to internal validity is the fact that studied software components were developed over different time intervals. To mitigate this threat, we added the development time of a software component T as a normalization parameter to the regression model (refer to Section 6).

There is a threat that project-specific characteristics that are not covered by our metrics, additionally influence the dependent variable. Such hidden influential relationship would bias the results of the regression analysis. To mitigate this threat, we applied generalized linear mixed model regression analysis with the project as a random factor. The relatively low intra class correlation (ICC = 0.066) indicates very similar distributed variances across the 24 projects and leaves less room for hidden factors.

Since our dependent variable is measured as count data, we applied a Poisson regression model. However, the violation of assumptions that are inherent to the Poisson regression, is another potential threat to internal validity. Thus, we tested our statical model for over-dispersion and mitigated the in fact over-dispersed data by adding a random between-transects error at component level, as proposed by Maindonald and Braun [83].

8.4 Reliability

Data. The 24 studied cases, from which we drew our conclusions, were selected by the authors of this study. This selection might be biased due to certain experiences or preferences of the authors. To mitigate this threat, we specified a set of case selection criteria upfront (see Section 4.1), which we derived from our research questions, the examined traceability use cases, and the proposed traceability metrics. This selection strategy ensured that we selected cases, which are suitable for the studied problem. However, other researchers could have selected other projects.

Data collection. A potential threat exists in the collection and preparation of the project data used to measure dependent and independent variables. To avoid especially manual bias during the project data preparation and to ensure reproducible results, we fully automated the process of data collection and preparation. Due to the public availability of the project artifacts and the fully automated collection and analysis process, our study can be replicated and additional projects could be included to further broaden the data corpus. We carefully verified our tool that automates this process. Therefore, we validated intermediate results of the process manually and cross-checked the data for inconsistencies and contradictions.

Statistical analysis. Another potential threat may exist in the statistical modeling and the elimination of irrelevant factors from the model, which was manually conducted by the researchers. To mitigate this threat, we applied a clearly defined and reproducible model simplification process as defined by Crawley [20]. Additionally, we make the raw data on which this study is based publicly available to ensure that our study can be reproduced and replicated [105].

9 Related Work

We carefully searched for previous work with relation to the study reported in this paper and classify the discussion into four topics: previous empirical work on traceability in general, previously proposed traceability metrics, previous work that studied specifically the impact of traceability on software quality, and previous work that studied other factors than traceability for their influence on software quality.

9.1 Empirical Work on Software Traceability in General

Gotel and Finkelstein [38] reported the findings of a year-long empirical study on requirements traceability conducted in 1992. The study involved around one hundred software development practitioners, holding a variety of positions within a large organization and with experience of up to 30 years. The authors were concerned with understanding and exposing the scope of the problem area and found multiple perspectives on what traceability was expected to enable and on the problems experienced. However, the authors did not report about the actual benefits of traceability to their subjects.

Arkley and Riddle [4] reported on a survey of nine software projects, small to multi-national in scope, undertaken using questionnaires and interviews. The authors identified three issues related to traceability: the necessity for extra entry data when using traceability tools; a lack of understanding on how to employ traceability; and the lack of perceived direct benefits to the main development process. Our study especially addresses the third point in that we can demonstrate for the first time that there is clear benefit in terms of higher software quality from recording and maintaining traceability information within a project.

Ramesh et al. [99] emphasized on the high costs of creating and maintaining traceability. The authors concluded that these costs can potentially be compensated by higher quality and reduced overall maintenance costs if traceability is applied purposefully. While the authors' conclusion was a vision statement at the time when it was written, we can now provide empirical evidence that purposed traceability in fact leads to higher product quality, which entails reduced

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/TSE.2016.2622264, IEEE Transactions on Software Engineering

software maintenance costs. However, it remains a future exercise to set the benefits of software traceability in relation to its costs.

Dömges and Pohl [96, 27] claimed that neglecting traceability completely or capturing traces in an unstructured or incomplete manner will lead to reduced system quality, expensive iterations of defect corrections, and increased project costs. Since the authors did not support their visionary claim by empirical results, it has been subject to many controversial debates among researchers and practitioners. We can now demonstrate the practical relevance and applicability of these visionary predictions through empirical evidence.

9.2 Requirements Traceability Metrics

Various researchers proposed traceability metrics to characterize traced software artifacts. For example, Pfleeger and Bohner [94] proposed software maintenance metrics for traceability graphs. They distinguish vertical traceability metrics (i.e., cyclomatic complexity and size) and horizontal traceability metrics. While vertical metrics are meant to characterize the developed product, horizontal metrics are meant to characterize the development process. In preparation of our study, we proposed a set of traceability use case specific metrics that measure the completeness of the implemented trace links within a project with respect to a traceability use case.

To generically measure the complexity of requirements traceability, Costello and Liu [19] proposed the use of linkage statistic metrics. Dick [25] extends the idea of analyzing traceability graphs by introducing trace link semantics, which he calls rich traceability. Main advantage of his approach is that propositional reasoning can be applied to analyze traceability relationships for their consistency. Hull et al. [50] carry on with the idea of analyzing rich traceability graphs and propose further metrics: *breadth* is related to the coverage and measures the progress of a development phase, *depth* measures the number of granularity layers per development phase, *growth* is related to the potential change impact, *balance* measures the distribution of growth factors, and *latent change* measures the impact on a change. Regan et al. [101] proposed multiple traceability metrics for the categories management issues, social issues, and technical issues, which served as a foundation for a traceability assessment model [102].

While all the proposed requirements traceability metrics were meant to measure specific characteristics of the requirements traceability graph, no empirical evidence is available on whether and how these metrics support practitioners with software engineering activities to improve the software quality. We used these metrics as an initial inspiration for the definition of the proposed set of traceability completeness measures.

9.3 Impact of Requirements Traceability on the Implementation Quality

In previous work, we studied the effect of traceability on developers' performance when performing software implementation and maintenance tasks [75, 76] within a controlled experiment. We found that study participants performed on average 24% faster and created 50% more correct solutions if they had traces between relevant requirements and the source code available when solving a task.

Jaber et al. [56] replicated the experiment with similar development tasks from another software development project. They even observed a 86% improvement in task accuracy.

In [104], we investigated whether existing traceability can be used to measure the complexity of requirements in agile software development projects. Therefore, we proposed the three complexity metrics *number of related requirements*, *average distance to related requirements*, and *requirements information flow* to assess the complexity of vertical traces among the requirements. We found that these complexity metrics can be leveraged to predict the error proneness of requirements implementations, and thus to estimate the requirements' implementation risk. Eventually, this information can then be used to rank requirements implementation tasks within the project's backlog according to the implementation risk in a much more objective way than it is possible today.

9.4 Factors Influencing Implementation Quality

There exists a variety of previous work that empirically investigated other factors than traceability for their influence on software implementation quality. Fitzgerald et al. [31] studied the impact of requirements attributes such as *number of words*, *bag of words*, *requirements creator*, or *requirements implementer* on the software quality. Similarly to our study, the authors investigated two open-source projects due to the public availability of the development artifacts. The authors used naive Bayes, decision table, linear regression and M5P-tree classification algorithms for investigating the relationship and concluded that linear regression and M5P-tree produced the best results.

The studies of Nagappan et al. [88] and Cataldo and Herbsleb [10] investigated the influence of organizational attributes on software quality. Each study solely investigated closed-source development projects of one company. The authors applied logistic regression analysis, which is a common approach for binary dependent variables, because they investigated the impact on the error proneness rather than the actual number of defects per component.

Since complexity is considered as an important driver for software defects, a variety of studies investigated the impact of software complexity metrics on the software quality. Graves et al. [42] studied the impact of metrics such as *lines of code*, *McCabe*, *functions*, or *breaks* on the software quality of a closed-source legacy system that comprised 80 modules. Similar to our approach, the authors fitted a generalized linear model to investigate the effect. Nagappan et al. [87] analyzed the impact of complexity metrics on the software quality for five different closed-source projects by fitting one logistic regression model for each project. For each project they found a set of metrics that correlated with the software quality. However, there was no single set of metrics that fitted all projects. Contrary to our study, the authors did not apply a multi-level regression analysis to overcome this problem.

As the manual creation and maintenance of trace links is associated with high costs [47], researchers studied information retrieval based approaches to support automated trace

recovery scenarios [84, 15, 23, 81, 82, 90, 64, 107, 73, 72]. However, automated trace recovery implies the risk that potentially incorrect trace links are created [131, 85]. To address this traceability quality problem, Panichella et al. [93] leveraged structural artifact information to improve the correctness of the recovered traces. Further, Dasgupta et al. [21] successfully incorporated relevant documentation artifacts into the automated retrieval process. It was also shown that a combination of multiple information retrieval approaches can improve the overall recovery performance [35].

10 Conclusions and Future Work

In this paper we focused on studying the impact of requirements traceability on software implementation quality. Requirements traceability has long been recognized as an important quality of a well-engineered software. Among stakeholders, traceability is often unpopular due to the extra effort it creates and the unclear benefits and beneficiaries. So far there was little hard evidence regarding the benefits to expect from creating and maintaining traceability.

We identified and investigated the four most relevant software engineering activities that depend on the availability of requirements traceability. These refer (1) to impact analysis on the requirements level and (2) between requirements and source code, as well as (3) to completeness analysis of source code vs. requirements and (4) requirements vs. source code. We proposed four measures to quantify the degree to which a project's requirements traceability is complete with respect to these four software engineering activities. Furthermore, we conducted an empirical study with 24 open-source software development projects and applied these measures on the granularity of software components. The results of our study show empirical evidence that requirements traceability has a significant impact on the software implementation quality of the studied software components. For three of the four identified implementation supporting activities we found that the more complete the necessary traceability to execute these activities the lower the defect rate within the analyzed software component.

Our results provide for the first time empirical evidence that improving the degree of traceability completeness indeed decreases the defect rate to be expected, and thus helps to raise implementation quality of the developed software. This finding supports practitioners with project planing on whether and what traceability to be established for a software development, because the direct impact on software implementation quality can be estimated. The results also provide concrete figures on the benefits to be expected from capturing specific trace links.

The tool we developed to automatically collect and analyze project specific software development artifacts can be used by practitioners to automatically calculate their degree of traceability implementation completeness. Similar to existing initiatives (e.g., SonarQube [116]) that continuously calculate and monitor software complexity metrics of open source projects, we plan to setup a webpage that continuously monitors the traceability completeness with respect to different software engineering activities requiring traceability. This webpage will provide a ready-to-use tool for practitioners that does not require any project installation or customization.

A direct integration of our proposed measures into the dashboard of existing software lifecycle management tools could provide valuable information to project managers about the status of the the development projects. Additionally, software engineers who conduct an activity that requires traceability could use this information to find out whether or not and to which extent existing traceability can be used for this activity.

Future work will focus on analyzing the costs for establishing traceability. We plan to further investigate if the proposed metrics can be used to detect hot spots among the components for which additional traceability would have the highest impact on the overall defect rate.

Acknowledgment

We are funded by the German Ministry of Education and Research (BMBF) grants: 01IS14026A, 01IS16003B and by the Thüringer Aufbaubank (TAB) grant: 2015FE9033. We would like to thank Jana Wäldchen and Michael Rzanny for sharing their statistical expertise and providing early feedback on our empirical validation methods.

References

[1] Activiti Team. Activiti: A light-weight workflow and Business Process Management (BPM) Platform targeted at business people, developers and system admins [Online]. http://activiti.org/, 2015.

[2] Yunsik Ahn, Jungseok Suh, Seungryeol Kim, and Hyunsoo Kim. The software maintenance project effort estimation model based on function points. *Journal of Software Maintenance and Evolution: Research and Practice*, 15(2):71–85, 2003. doi: 10.1002/smr.269.

[3] Nasir Ali, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. Trustrace: Mining Software Repositories to Improve the Accuracy of Requirement Traceability Links. *IEEE Trans. Software Eng.*, 39(5):725–741, 2013. doi: 10.1109/TSE.2012.71.

[4] P. Arkley and S. Riddle. Overcoming the traceability benefit problem. In *Proceedings of the 13th IEEE International Conference on Requirements Engineering*, pages 385–389. IEEE, 2005. doi: 10.1109/RE.2005.49.

[5] Atlassian. Jira Software - Issue & Project Tracking for Software Teams [Online]. https://www.atlassian.com/software/jira, 2015.

[6] David A. Belsley, Edwin Kuh, and Roy E. Welsch. *Regression Diagnostics: Identifying Influential Data and Sources of Collinearity*, volume 571. John Wiley & Sons, 2005. ISBN 978-0-471-05856-4.

[7] S.A. Bohner. Impact analysis in the software change process: A year 2000 perspective. In *Proceedings of the International Conference on Software Maintenance*, pages 42–51. IEEE, 1996. doi: 10.1109/ICSM.1996.564987.

[8] Elke Bouillon, Patrick Mäder, and Ilka Philippow. A Survey on Usage Scenarios for Requirements Traceability in Practice. In Joerg Doerr and Andreas L. Opdahl, editors, *Requirements Engineering: Foundation*

for Software Quality, volume 7830, pages 158–173. Springer, 2013. ISBN 978-3-642-37421-0 978-3-642-37422-7.

[9] Lionel C Briand, Jürgen Wüst, John W Daly, and D Victor Porter. Exploring the relationships between design measures and software quality in object-oriented systems. *Journal of systems and software*, 51(3): 245–273, 2000. doi: 10.1016/S0164-1212(99)00102-8.

[10] Marcelo Cataldo and James D Herbsleb. Factors leading to integration failures in global feature-oriented development: An empirical analysis. In *Proc. of the 33rd International Conference on Software Engineering (ICSE)*, pages 161–170. IEEE, 2011. doi: 10.1145/1985793.1985816.

[11] Center of Excellence for Software Traceability (COEST). Software Traceability [Online]. http://coest.org/bok/index.php/Main_Page, 2015.

[12] Center of Excellence for Software Traceability (COEST). What is Traceability? [Online]. http://coest.org/index.php/what-is-traceability, 2015.

[13] J. Cleland-Huang, G. Zemont, and W. Lukasik. A heterogeneous solution for improving the return on investment of requirements traceability. In *Proceedings of the 12th IEEE International Requirements Engineering Conference*, pages 214–223. IEEE, 2004. doi: 10.1109/ICRE.2004.1335680.

[14] Jane Cleland-Huang, Carl K. Chang, and Mark Christensen. Event-based traceability for managing evolutionary change. *IEEE Transactions on Software Engineering*, 29(9):796–810, 2003. doi: 10.1109/TSE.2003.1232285.

[15] Jane Cleland-Huang, Brian Berenbach, Stephen Clark, Raffaella Settimi, and Eli Romanova. Best Practices for Automated Traceability. *Computer*, 40(6):27–35, 2007. doi: 10.1109/MC.2007.195.

[16] Jane Cleland-Huang, Orlena C. Z. Gotel, Jane Huffman Hayes, Patrick Mäder, and Andrea Zisman. Software traceability: Trends and future directions. In *Proceedings of the Future of Software Engineering*, pages 55–69. ACM Press, 2014. doi: 10.1145/2593882.2593891.

[17] CMMI Product Team. CMU/SEI-2006-TR-008: CMMI for Development, Version 1.2. Technical report, Carnegie Mellon University / Software Engineering Institute, 2006.

[18] Codehaus Team. Codehaus: A collaborative software development environment for projects with open-source (but business friendly) licences [Online]. http://www.codehaus.org, 2015.

[19] Rita J. Costello and Dar-Biau Liu. Metrics for requirements engineering. *Journal of Systems and Software*, 29 (1):39–63, 1995. doi: 10.1016/0164-1212(94)00127-9.

[20] Michael J Crawley. *The R Book*. John Wiley & Sons, 2012. ISBN 978-0-470-97392-9.

[21] Tathagata Dasgupta, Mark Grechanik, Evan Moritz, Bogdan Dit, and Denys Poshyvanyk. Enhancing Software Traceability by Automatically Expanding Corpora with Relevant Documentation. In *Proc. of the 29th IEEE International Conference on Software Maintenance*, pages 320–329. IEEE, September 2013. doi: 10.1109/ICSM.2013.43.

[22] Jose Luis de la Vara, Alejandra Ruiz, Katrina Attwood, Huáscar Espinoza, Rajwinder Kaur Panesar-Walawege, Ángel López, Idoya del Río, and Tim Kelly. Model-based specification of safety compliance needs for critical systems: A holistic generic metamodel. *Information and Software Technology*, 72: 16–30, 2016. doi: 10.1016/j.infsof.2015.11.008.

[23] Andrea De Lucia, Fausto Fasano, and Rocco Oliveto. Traceability management for impact analysis. In *Proceedings of the Frontiers of Software Maintenance Conference*, pages 21–30. IEEE, 2008. doi: 10.1109/FOSM.2008.4659245.

[24] Denise Dellarosa and Lyle E. Bourne Jr. Text-Based Decisions: Changes in the Availability of Facts Due to Instructions and the Passage of Time. Technical report, DTIC Document, 1982.

[25] Jeremy Dick. Rich traceability. In *Proceedings of the 1st International Workshop on Traceability in Emerging Forms of Software Engineering, Edinburgh, Scotland*, pages 18–23, 2002.

[26] Bogdan Dit, Annibale Panichella, Evan Moritz, Rocco Oliveto, Massimiliano Di Penta, Denys Poshyvanyk, and Andrea De Lucia. Configuring topic models for software engineering tasks in TraceLab. In *Proc. of the 7th International Workshop on Traceability in Emerging Forms of Software Engineering*, pages 105–109, 2013. doi: 10.1109/TEFSE.2013.6620164.

[27] Ralf Dömges and Klaus Pohl. Adapting traceability environments to project-specific needs. *Communications of the ACM*, 41(12):54–62, 1998. doi: 10.1145/290133.290149.

[28] Davide Falessi, Shiva Nejati, Mehrdad Sabetzadeh, Lionel Briand, and Antonio Messina. SafeSlice: A model slicing and design safety inspection tool for SysML. In *Proc. of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, page 460. ACM Press, 2011. doi: 10.1145/2025113.2025191.

[29] W Holmes Finch, Jocelyn E Bolin, and Ken Kelley. *Multilevel Modeling Using R*. CRC Press, 2014. ISBN 978-1-4665-1585-7.

[30] Antony Finkelstein. Foreword of Software and Systems Traceability. In Jane Cleland-Huang, Olly Gotel, and Andrea Zisman, editors, *Software and Systems Traceability*. Springer, 2012. ISBN 978-1-4471-2238-8.

[31] Camilo Fitzgerald, Emmanuel Letier, and Anthony Finkelstein. Early failure prediction in feature request management systems. In *Proc. of the 19th IEEE International Requirements Engineering Conference*, pages 229–238. IEEE, 2011. doi: 10.1109/RE.2011.6051658.

[32] B. Flyvbjerg. Five Misunderstandings About Case-Study Research. *Qualitative Inquiry*, 12(2):219–245, April 2006. doi: 10.1177/1077800405284363.

[33] GeoServer Team. GeoServer: An open source server for sharing geospatial data [Online]. http://geoserver.org, 2015.

[34] GeoTools Team. GeoTools: An open source Java library that provides tools for geospatial data [Online]. http://geotools.org, 2015.

[35] Malcom Gethers, Rocco Oliveto, Denys Poshyvanyk, and Andrea De Lucia. On integrating orthogonal information retrieval methods to improve traceability

recovery. In *Proc. of the 27th IEEE International Conference on Software Maintenance*, pages 133–142, 2011. doi: 10.1109/ICSM.2011.6080780.

[36] Git Team. Git: A free and open source distributed version control system designed to handle everything from small to very large projects with speed and efficiency [Online]. http://git-scm.com, 2015.

[37] Google. Google Code Project Hosting: A free collaborative development environment for open source projects [Online]. https://code.google.com/, 2015.

[38] O.C.Z. Gotel and C.W. Finkelstein. An analysis of the requirements traceability problem. In *Proceedings of 1st International Conference on Requirements Engineering*, pages 94–101. IEEE, 1994. doi: 10.1109/ICRE.1994.292398.

[39] Orlena Gotel, Jane Cleland-Huang, Jane Huffman Hayes, Andrea Zisman, Alexander Egyed, Paul Grünbacher, Alex Dekhtyar, Giuliano Antoniol, Jonathan Maletic, and Patrick Mäder. Traceability Fundamentals. In Jane Cleland-Huang, Orlena Gotel, and Andrea Zisman, editors, *Software and Systems Traceability*, pages 3–22. Springer, 2012. ISBN 978-1-4471-2238-8 978-1-4471-2239-5.

[40] Graeme Rocher, Peter Ledbrook, Marc Palmer, Jeff Brown, Luke Daley, Burt Beckwith, and Lari Hotari. Contributing to Grails - Reference Documentation [Online]. http://docs.grails.org/3.0.2/guide/contributing.html, 2016.

[41] Grails Team. The Grails Framework - a powerful Groovy-based web application framework for the JVM [Online]. http://grails.org, 2015.

[42] Todd L Graves, Alan F Karr, James S Marron, and Harvey Siy. Predicting fault incidence using software change history. *Software Engineering, IEEE Transactions on*, 26(7):653–661, 2000. doi: 10.1109/32.859533.

[43] Groovy Team. Groovy: A multi-faceted language for the Java platform [Online]. http://groovy.codehaus.org/, 2015.

[44] Groovy Team. Eclipse tooling support for the Groovy programming language [Online]. http://groovy.codehaus.org/Eclipse+Plugin, 2015.

[45] GumTree Team. GumTree: A scientific suite for performing beamline experiments [Online]. http://gumtree.codehaus.org/, 2015.

[46] Carl A Gunter, Elsa L Gunter, Michael Jackson, and Pamela Zave. A reference model for requirements and specifications. *IEEE Software*, 17(3):37–43, 2000. doi: 10.1109/ICRE.2000.855609.

[47] Matthias Heindl and Stefan Biffl. A case study on value-based requirements tracing. In *Proceedings of the 10th European Software Engineering Conference*, pages 60–69. ACM Press, 2005. doi: 10.1145/1081706.1081717.

[48] James D Herbsleb, Daniel J Paulish, and Matthew Bass. Global software development at siemens: Experience from nine projects. In *Proc. of the 27th International Conference on Software Engineering*, pages 524–533. IEEE, 2005. doi: 10.1109/ICSE.2005.1553598.

[49] HornetQ Team. HornetQ: An open source project to build a multi-protocol, embeddable, very high performance, clustered, asynchronous messaging system [Online]. http://hornetq.jboss.org/, 2015.

[50] Elizabeth Hull, Ken Jackson, and Jeremy Dick. *Requirements Engineering*. Springer, 3rd edition, 2011. ISBN 978-1-84996-404-3.

[51] IEC. IEC 61508-3: Functional safety of electrical/electronic/programmable electronic safety-related systems - Part 3: Software requirements. Technical report, IEC, 2010.

[52] Claire Ingram and Steve Riddle. Cost-Benefits of Traceability. In Jane Cleland-Huang, Orlena Gotel, and Andrea Zisman, editors, *Software and Systems Traceability*, pages 23–42. Springer, 2012. ISBN 978-1-4471-2238-8.

[53] IronJacamar Team. IronJacamar: An implementation of the Java EE Connector Architecture 1.7 specification [Online]. http://www.ironjacamar.org/, 2015.

[54] ISO. ISO:26262-6:2011 Road vehicles - Functional safety - Part 6: Product development at the software level. Guideline, ISO, 2011.

[55] ISO and IEC. ISO/IEC 15504:2004 Information technology – Process assessment. Standard, ISO/IEC, 2004.

[56] K. Jaber, B. Sharif, and Chang Liu. A Study on the Effect of Traceability Links in Software Maintenance. *IEEE Access*, 1:726–741, 2013. doi: 10.1109/ACCESS.2013.2286822.

[57] JBoss Community. Drools: A Business Rules Management System (BRMS) solution [Online]. http://www.drools.org, 2015.

[58] JBoss Community. Errai Framework: A Java/GWT web framework for building rich-client web applications [Online]. http://erraiframework.org/, 2015.

[59] JBoss Community. JBoss: A family of a lightweight, cloud-friendly, enterprise-grade products that help enterprises innovate faster, in a smarter way [Online]. http://www.jboss.org/, 2015.

[60] JBoss Community. Narayana: A transactions toolkit which provides support for applications developed using a broad range of standards-based transaction protocols [Online]. http://narayana.jboss.org/, 2015.

[61] JBoss Community. TorqueBox: A new kind of Ruby application platform that supports popular technologies such as Ruby on Rails and Sinatra, while extending the footprint of Ruby applications to include built-in support for services such as messaging, scheduling, caching, and daemons. [Online]. http://www.torquebox.org, 2015.

[62] Jonathan de Halleux. QuickGraph: Graph Data Structures And Algorithms for .NET [Online]. http://quickgraph.codeplex.com, 2015.

[63] Robert Kabacoff. *R in Action*. Manning, 2011. ISBN 978-1-935182-39-9.

[64] Ed Keenan, Adam Czauderna, Greg Leach, Jane Cleland-Huang, Yonghee Shin, Evan Moritz, Malcom Gethers, Denys Poshyvanyk, Jonathan Maletic, Jane Huffman Hayes, Alex Dekhtyar, Daria Manukian, Shervin Hossein, and Derek Hearn. TraceLab: An Experimental Workbench for Equipping Researchers to Innovate, Synthesize, and Comparatively Evaluate Traceability Solutions. In *Proc. of the 34th International Conference on Software Engineering*, pages 1375–1378,

2012. ISBN 978-1-4673-1067-3.

[65] Timothy Patrick Kelly. *Arguing Safety-a Systematic Approach to Managing Safety Cases*. University of York, 1999.

[66] Jae-On Kim and G Donald Ferree. Standardization in causal analysis. *Sociological Methods & Research*, 10(2): 187–210, 1981. doi: 10.1177/004912418101000203.

[67] Barbara Kitchenham and Shari Lawrence Pfleeger. Software quality: The elusive target. *IEEE Software*, 13(1):12–21, 1996. doi: 10.1109/52.476281.

[68] Segla Kpodjedo, Filippo Ricca, Philippe Galinier, Yann-Gaël Guéhéneuc, and Giuliano Antoniol. Design evolution metrics for defect prediction in object oriented systems. *Empirical Software Engineering*, 16(1): 141–175, 2011. doi: 10.1007/s10664-010-9151-7.

[69] Hongyu Kuang, Patrick Mäder, Hao Hu, Achraf Ghabi, LiGuo Huang, Jian Lü, and Alexander Egyed. Can method data dependencies support the assessment of traceability between requirements and source code?: Traceability Assessment based on Method Data Dependencies. *Journal of Software: Evolution and Process*, 27(11):838–866, 2015. doi: 10.1002/smr.1736.

[70] Johan Linåker, Patrick Rempel, Björn Regnell, and Patrick Mäder. How Firms Adapt and Interact in Open Source Ecosystems: Analyzing Stakeholder Influence and Collaboration Patterns. In Maya Daneva and Oscar Pastor, editors, *Requirements Engineering: Foundation for Software Quality*, volume 9619, pages 63–81. Springer, 2016. ISBN 978-3-319-30281-2.

[71] Mikael Lindvall and Kristian Sandahl. Practical implications of traceability. *Software Practice and Experience*, 26(10):1161–1180, 1996.

[72] Lukas Linsbauer, Roberto E. Lopez-Herrejon, and Alexander Egyed. Recovering traceability between features and code in product variants. In *Proc. of the 17th International Software Product Line Conference*, pages 131–140, 2013. doi: 10.1145/2491627.2491630.

[73] Lukas Linsbauer, Stefan Fischer, Roberto E. Lopez-Herrejon, and Alexander Egyed. Using Traceability for Incremental Construction and Evolution of Software Product Portfolios. In *Proc. of the 8th IEEE/ACM International Symposium on Software and Systems Traceability*, pages 57–60, 2015. doi: 10.1109/SST.2015.16.

[74] Andrea De Lucia, Massimiliano Di Penta, and Rocco Oliveto. Improving Source Code Lexicon via Traceability and Information Retrieval. *IEEE Transactions on Software Engineering*, 37(2):205–227, 2011. doi: 10.1109/TSE.2010.89.

[75] Patrick Mäder and Alexander Egyed. Assessing the effect of requirements traceability for software maintenance. In *Proc. of the 28th IEEE International Conference on Software Maintenance*, pages 171–180. IEEE, 2012. doi: 10.1109/ICSM.2012.6405269.

[76] Patrick Mäder and Alexander Egyed. Do developers benefit from requirements traceability when evolving and maintaining a software system? *Empirical Software Engineering*, pages 1–29, 2014. doi: 10.1007/s10664-014-9314-z.

[77] Patrick Mäder, Orlena Gotel, and Ilka Philippow. Rule-Based Maintenance of Post-Requirements Traceability Relations. In *Proceedings of the 16th IEEE International Requirements Engineering Conference*, pages 23–32. IEEE, 2008. doi: 10.1109/RE.2008.24.

[78] Patrick Mäder, Orlena Gotel, and Ilka Philippow. Motivation Matters in the Traceability Trenches. In *Proceedings of the 17th IEEE International Requirements Engineering Conference*, pages 143–148. IEEE, 2009. doi: 10.1109/RE.2009.23.

[79] Patrick Mäder, Orlena Gotel, and Ilka Philippow. Getting back to basics: Promoting the use of a traceability information model in practice. In *Proceedings of the ICSE Workshop on Traceability in Emerging Forms of Software Engineering*, pages 21–25. IEEE, 2009. doi: 10.1109/TEFSE.2009.5069578.

[80] Patrick Mäder, Paul L. Jones, Yi Zhang, and Jane Cleland-Huang. Strategic Traceability for Safety-Critical Projects. *IEEE Software*, 30(3):58–66, 2013. doi: 10.1109/MS.2013.60.

[81] Anas Mahmoud and Nan Niu. Using Semantics-Enabled Information Retrieval in Requirements Tracing: An Ongoing Experimental Investigation. In *Proc. of the 34th IEEE International Computer Software and Applications Conference*, pages 246–247, 2010. doi: 10.1109/COMPSAC.2010.29.

[82] Anas Mahmoud and Nan Niu. TraCter: A tool for candidate traceability link clustering. In *Proc. of the 19th IEEE International Requirements Engineering Conference*, pages 335–336, 2011. doi: 10.1109/RE.2011.6051663.

[83] John Maindonald and W John Braun. *Data Analysis and Graphics Using R: An Example-Based Approach*, volume 10. Cambridge University Press, 2010. ISBN 978-0-521-76293-9.

[84] A. Marcus and J.I. Maletic. Recovering documentation-to-source-code traceability links using latent semantic indexing. In *Proceedings of the 25th International Conference on Software Engineering*, pages 125–135. IEEE, 2003. doi: 10.1109/ICSE.2003.1201194.

[85] Thorsten Merten, Daniel Krämer, Bastian Mager, Paul Schell, Simone Bürsner, and Barbara Paech. Do Information Retrieval Algorithms for Automated Traceability Perform Effectively on Issue Tracking System Data? In *Proc. of the 22nd International Working Conference Requirements Engineering: Foundation for Software Quality*, pages 45–62, 2016. doi: 10.1007/978-3-319-30282-9_4.

[86] MongoDB. mongoDB: An open-source, document database designed for ease of development and scaling [Online]. http://www.mongodb.org, 2015.

[87] Nachiappan Nagappan, Thomas Ball, and Andreas Zeller. Mining metrics to predict component failures. In *Proc. of the 28th International Conference on Software Engineering*, pages 452–461. ACM, 2006. doi: 10.1145/1134285.1134349.

[88] Nachiappan Nagappan, Brendan Murphy, and Victor Basili. The Influence of Organizational Structure on Software Quality: An Empirical Case Study. In *Proc. of the 30th International Conference on Software Engineering*, pages 521–530. ACM, 2008. doi: 10.1145/1368088.1368160.

[89] Shiva Nejati, Mehrdad Sabetzadeh, Davide Falessi, Lionel Briand, and Thierry Coq. A SysML-based approach to traceability management and design slic-

ing in support of safety certification: Framework, tool support, and case studies. *Information and Software Technology*, 54(6):569–590, 2012. doi: 10.1016/j.infsof.2012.01.005.

[90] Nan Niu, Tanmay Bhowmik, Hui Liu, and Zhendong Niu. Traceability-enabled refactoring for managing just-in-time requirements. In *Proc. of the 22nd IEEE International Requirements Engineering Conference*, pages 133–142, 2014. doi: 10.1109/RE.2014.6912255.

[91] Rajwinder Kaur Panesar-Walawege, Mehrdad Sabetzadeh, Lionel Briand, and Thierry Coq. Characterizing the Chain of Evidence for Software Safety Cases: A Conceptual Model Based on the IEC 61508 Standard. In *Proc. of the Third International Conference on Software Testing, Verification and Validation (ICST)*, pages 335–344. IEEE, 2010. doi: 10.1109/ICST.2010.12.

[92] Rajwinder Kaur Panesar-Walawege, Mehrdad Sabetzadeh, and Lionel Briand. A Model-Driven Engineering Approach to Support the Verification of Compliance to Safety Standards. In *Proc. of the 22nd International Symposium on Software Reliability Engineering*, pages 30–39. IEEE, November 2011. doi: 10.1109/ISSRE.2011.11.

[93] A. Panichella, C. McMillan, E. Moritz, D. Palmieri, R. Oliveto, D. Poshyvanyk, and A. De Lucia. When and How Using Structural Information to Improve IR-Based Traceability Recovery. In *Proc. of the 17th European Conference on Software Maintenance and Reengineering*, pages 199–208. IEEE, March 2013. doi: 10.1109/CSMR.2013.29.

[94] S.L. Pfleeger and S.A. Bohner. A framework for software maintenance metrics. In *Proceedings of the Conference on Software Maintenance*, pages 320–327. IEEE, 1990. doi: 10.1109/ICSM.1990.131381.

[95] Francisco A. C. Pinheiro. Requirements Traceability. In Julio Cesar Sampaio Prado Leite and Jorge Horacio Doorn, editors, *Perspectives on Software Requirements*, pages 91–113. Springer, 2004. ISBN 978-1-4613-5090-3 978-1-4615-0465-8.

[96] K. Pohl. PRO-ART: Enabling requirements pre-traceability. In *Proceedings of the 2nd International Conference on Requirements Engineering*, pages 76–84. IEEE, 1996. doi: 10.1109/ICRE.1996.491432.

[97] Klaus Pohl. *Requirements Engineering: Fundamentals, Principles, and Techniques*. Springer, 2010. ISBN 978-3-642-12577-5.

[98] Václav T Rajlich and Keith H Bennett. A staged model for the software life cycle. *Computer*, 33(7):66–71, 2000. doi: 10.1109/2.869374.

[99] B. Ramesh, T. Powers, C. Stubbs, and M. Edwards. Implementing requirements traceability: A case study. In *Proceedings of the 2nd IEEE International Symposium on Requirements Engineering*, pages 89–95. IEEE, 1995. doi: 10.1109/ISRE.1995.512549.

[100] Balasubramaniam Ramesh. Factors influencing requirements traceability practice. *Communications of the ACM*, 41(12):37–44, 1998. doi: 10.1145/290133.290147.

[101] Gilbert Regan, Fergal McCaffery, Kevin McDaid, and Derek Flood. The Barriers to Traceability and their Potential Solutions: Towards a Reference Framework. In *Proceedings of the 38th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 319–322. IEEE, 2012. doi: 10.1109/SEAA.2012.80.

[102] Gilbert Regan, Miklos Biro, Fergal Mc Caffery, Kevin Mc Daid, and Derek Flood. A Traceability Process Assessment Model for the Medical Device Domain. In *Systems, Software and Services Process Improvement*, volume 425, pages 206–216. Springer, 2014. ISBN 978-3-662-43895-4 978-3-662-43896-1.

[103] Patrick Rempel and Patrick Mäder. A quality model for the systematic assessment of requirements traceability. In *Proceedings of the 23rd IEEE International Requirements Engineering Conference*, pages 176–185. IEEE, 2015. doi: 10.1109/RE.2015.7320420.

[104] Patrick Rempel and Patrick Mäder. Estimating the Implementation Risk of Requirements in Agile Software Development Projects with Traceability Metrics. In Samuel A. Fricker and Kurt Schneider, editors, *Requirements Engineering: Foundation for Software Quality*, volume 9013, pages 81–97. Springer, 2015. ISBN 978-3-319-16100-6 978-3-319-16101-3.

[105] Patrick Rempel and Patrick Mäder. Replication data for: Preventing Defects: The Impact of Requirements Traceability Completeness on Software Quality. *Harvard Dataverse*, 2016. doi: 10.7910/DVN/PLCBZV.

[106] Patrick Rempel, Patrick Mäder, and Tobias Kuschke. An empirical study on project-specific traceability strategies. In *Proceedings of the 21st IEEE International Requirements Engineering Conference*, pages 195–204. IEEE, 2013. doi: 10.1109/RE.2013.6636719.

[107] Patrick Rempel, Patrick Mäder, and Tobias Kuschke. Towards feature-aware retrieval of refinement traces. In *Proc. of the 7th International Workshop on Traceability in Emerging Forms of Software Engineering*, pages 100–104. IEEE, 2013. doi: 10.1109/TEFSE.2013.6620163.

[108] Patrick Rempel, Patrick Mäder, Tobias Kuschke, and Ilka Philippow. Requirements Traceability across Organizational Boundaries - A Survey and Taxonomy. In Joerg Doerr and Andreas L. Opdahl, editors, *Requirements Engineering: Foundation for Software Quality*, volume 7830, pages 125–140. Springer, 2013. ISBN 978-3-642-37421-0 978-3-642-37422-7.

[109] Patrick Rempel, Patrick Mäder, Tobias Kuschke, and Jane Cleland-Huang. Mind the gap: Assessing the conformance of software traceability to relevant guidelines. In *Proceedings of the 36th International Conference on Software Engineering ICSE*, pages 943–954. ACM Press, 2014. doi: 10.1145/2568225.2568290.

[110] Leanna Rierson. *Developing Safety-Critical Software: A Practical Guide for Aviation Software and DO-178C Compliance*. CRC Press, 2013. ISBN 978-1-4398-1368-3.

[111] RTCA. DO-178C: Software Considerations in Airborne Systems and Equipment Certification. Guideline, RTCA, 2011.

[112] Per Runeson and Martin Höst. Guidelines for conducting and reporting case study research in software engineering. *Empirical Software Engineering*, 14(2):131–164, 2009. doi: 10.1007/s10664-008-9102-8.

[113] Seam2 Team. The Seam2 Framework: A powerful open source development platform for building rich Internet applications in Java [Online]. http://seamframework.org, 2015.

[114] Smooks Team. Smooks: An extensible framework for building applications for processing XML and non XML data (CSV, EDI, Java etc) using Java [Online]. http://www.smooks.org/, 2015.

[115] H.M. Sneed. Estimating the costs of software maintenance tasks. In *Proc. of the International Conference on Software Maintenance*, pages 168–181, 1995. doi: 10.1109/ICSM.1995.526539.

[116] SonarSource. SonarQube: An open platform to manage code quality [Online]. http://www.sonarqube.org/, 2015.

[117] SonarSource. SonarQube C# Plugin [Online]. http://docs.codehaus.org/display/SONAR/C#Plugin, 2015.

[118] SourceForge Community. SourceForge - Download, Develop and Publish Free Open Source Software [Online]. http://www.sourceforge.net/, 2015.

[119] Ramanath Subramanyam and Mayuram S. Krishnan. Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *IEEE Transactions on Software Engineering*, 29(4):297–310, 2003. doi: 10.1109/TSE.2003.1191795.

[120] The Apache Software Foundation. The Apache Software Foundation [Online]. http://www.apache.org, 2015.

[121] The Apache Software Foundation. Apache Archiva: An extensible repository management software that helps taking care of your own personal or enterprise-wide build artifact repository [Online]. http://archiva.apache.org, 2015.

[122] The Apache Software Foundation. Apache Axis2 - The Web Services Engine [Online]. http://axis.apache.org/axis2, 2015.

[123] The Apache Software Foundation. Apache Derby: An open source relational database implemented entirely in Java [Online]. http://db.apache.org/derby/, 2015.

[124] The Apache Software Foundation. The Apache Hadoop software library: A framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models [Online]. http://hadoop.apache.org, 2015.

[125] The Apache Software Foundation. Apache Lucene - Java-based indexing and search technology, as well as spellchecking, hit highlighting and advanced analysis/tokenization capabilities [Online]. http://lucene.apache.org/, 2015.

[126] The Apache Software Foundation. Apache Pig: A platform for analyzing large data sets that consists of a high-level language for expressing data analysis programs, coupled with infrastructure for evaluating these programs [Online]. http://pig.apache.org, 2015.

[127] The Apache Software Foundation. Apache Subversion: An open source version control system [Online]. http://subversion.apache.org/, 2015.

[128] The Apache Software Foundation. Committing patches to the Derby Subversion Repository [Online]. http://wiki.apache.org/db-derby/DerbyCommitHowTo, 2016.

[129] The Apache Software Foundation. Guide for Hadoop Core Committers [Online]. https://wiki.apache.org/hadoop/HowToCommit, 2016.

[130] The R Foundation. The R Project for Statistical Computing: A free software environment for statistical computing and graphics. It compiles and runs on a wide variety of UNIX platforms, Windows and MacOS [Online]. http://www.r-project.org, 2015.

[131] A. von Knethen, B. Paech, F. Kiedaisch, and F. Houdek. Systematic requirements recycling through abstraction and traceability. In *Proceedings of the IEEE Joint International Conference on Requirements Engineering*, pages 273–281. IEEE, 2002. doi: 10.1109/ICRE.2002.1048538.

[132] Robert K. Yin. *Case Study Research: Design and Methods*. Applied social research methods. Sage Publications, 4th edition, 2009. ISBN 978-1-4129-6099-1.